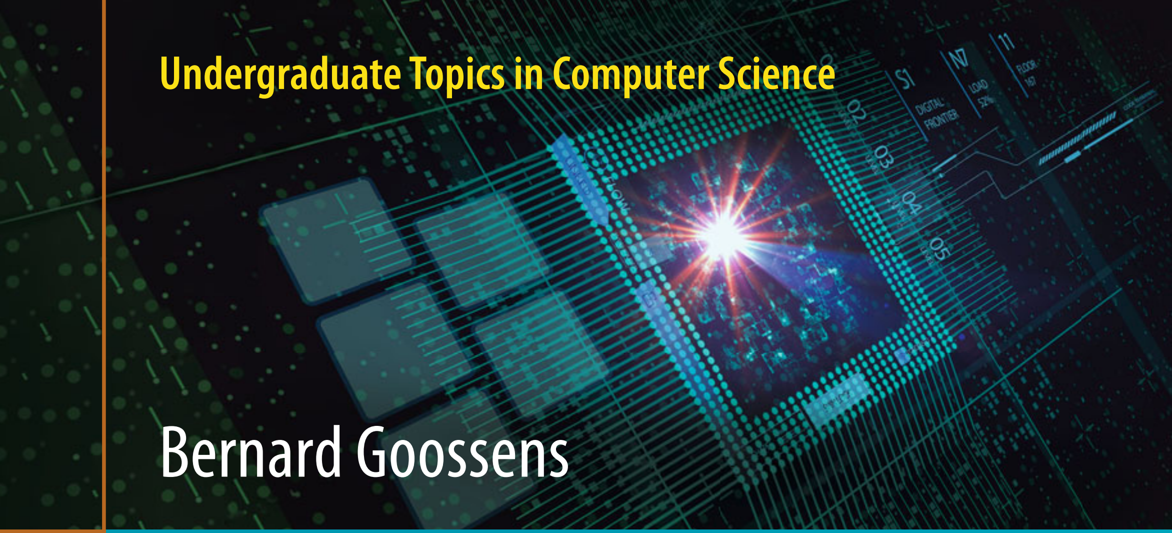


Undergraduate Topics in Computer Science

Bernard Goossens

Guide to Computer Processor Architecture

A RISC-V Approach, with High-Level Synthesis

UTiCS

Springer

Undergraduate Topics in Computer Science

‘Undergraduate Topics in Computer Science’ (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems, many of which include fully worked solutions.

The UTiCS concept relies on high-quality, concise books in softback format, and generally a maximum of 275–300 pages. For undergraduate textbooks that are likely to be longer, more expository, Springer continues to offer the highly regarded Texts in Computer Science series, to which we refer potential authors.

Bernard Goossens

Guide to Computer Processor Architecture

A RISC-V Approach, with High-Level Synthesis

Springer

Bernard Goossens
Université de Perpignan
Perpignan, France

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-031-18022-4 ISBN 978-3-031-18023-1 (eBook)
https://doi.org/10.1007/978-3-031-18023-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Processor Architecture: A Do-It-Yourself Approach

This book is a new textbook on processor architecture. What is new is not the topic, even though actual multicore and multithreaded designs are depicted, but the way processor architecture is presented.

This book can be related to the famous Douglas Comer textbook on Operating System (OS) [1, 2]. As Douglas Comer did to present the design of an OS, I use a DIY approach to present processor designs.

In his book, Douglas Comer builds a full OS from scratch, with C source code. In the present book, I aim to make you build your own processors, also from scratch and also with C source code.

All you need is a computer, an optional development board, and a set of freely available softwares to transform C programs into equivalent FPGA implementations (field-programmable gate array).

If you don't have a development board, you can still *simulate* the processors presented in the book though.

In the 70s (of the twentieth century of course), it became possible for a single person to build a full OS (Kenneth L. Thompson created Unix in 1970), and better than that, to write a *How-To* book giving a complete recipe to implement a Unix-like OS (Douglas Comer published Xinu in 1984).

Two improvements in computer hardware and software made it eventually feasible: the availability of personal computers and the C programming language. They give the full access to the hardware.

Nowadays, an FPGA plays the role of the personal computer of the 70s: It gives access to the logic gates. The High-Level Synthesis tool (HLS) plays the role of the C compiler of the 70s: It gives access to the FPGA through a high-level language.

RISC-V Open-Source Processor Designs

The Douglas Comer book explained how to build an OS using a self-made example named *Xinu*. Even though *Xinu* was claimed not to be Unix (*Xinu* is a recursive acronym meaning "Xinu Is Not Unix"), it would behave very likely, giving the reader and implementer the opportunity to compare his/her own realization to the Unix reference.

In the same idea, I have chosen a reference processor to be able to compare the FPGA-based processor proposed in the book to real RISC-V industrial products.

RISC-V is an *open-source* Instruction Set Architecture (ISA), which means that you can build a RISC-V processor, use it, and even sell it, without the permission of any computer constructor. It would not be so for Intel's X86 or ARM's v7 or v8.

Moreover, RISC-V defines multiple gigogne ISA subsets. A processor may implement any level of the *matriochka* organization of the ISA.

In this book, you will implement one of the most basic subsets, namely RV32I (a set of machine instructions to compute on 32-bits integer words). But you will know enough to be able to expand your processor to 64-bits words, to add a floating-point computation subset and many more, according to the RISC-V specification [3].

Moreover, the subset you will be implementing is enough to boot an OS like Linux (which is not part of this book though).

A Very Practical Introduction to Computer Architecture for Undergraduate Students

This book is also a very practical introduction to computer architecture. It can be seen as an application of more complete reference books of the domain (see for example the most recent editions of the famous books on computer architecture by J. L. Hennessy and D. A. Patterson [4, 5]).

Along the chapters, you will implement different processor organizations:

- Basic (e.g., Intel 4004. The Intel 4004 was the first microprocessor introduced in 1971, i.e., the first processor to fit in a single integrated circuit.) [6, 7].
- Slightly pipelined (e.g., RISC-I. The RISC-I processor was the first pipelined microprocessor, introducing the Reduced Instruction Set Computer or RISC concept in 1980.) [8].
- Moderately pipelined (e.g., MIPS. The MIPS or microprocessor without Interlocked Pipeline Stages was a concurrent of RISC-I, adding techniques to better fill the pipeline.) [9].
- Pipeline for multicycle operations (or multicycle pipeline). This is an enhancement of the pipeline to handle multiple cycle memory accesses, integer multiplication and division, or floating-point operations.

- Multiple threads (e.g., SMT. The SMT or simultaneous multithreading is an enhancement to share the pipeline between multiple threads and so, fill it better. This technique is also known as Hyper Threading, as named by Intel.) [10].
- Multiple cores (e.g., IBM Power-4. The Power-4 was the first multicore processor, introduced in 2001. It had two cores.) [11].

Even though you stay at the simulation level because you do not have a development board, the designs have been tested on an FPGA and they respect hardware constraints.

A Teaching Tool for Instructors With a GitHub Support

All the processors designed in this book are provided as open-source projects (either to build the simulation only version or the full FPGA-based projects to be tested on a Xilinx-based development board) available in the goossens-book-ip-projects repository of the https://github.com/goossens-springer github.

A full chapter of the book is devoted to the installation of the RISC-V tools (gnu toolchain including the RISC-V cross-compiler, spike simulator, gdb debugger, and the RISC-V-tests official test and benchmarks suite provided by the RISC-V international organization (https://riscv.org/).

The implementations proposed in the book are compared from a performance perspective, applying the famous Hennessy–Patterson "quantitative approach" [4]. The book presents an adaptation of a benchmark suite to the development board no-OS environment.

The same benchmark suite is used throughout the book to test and compare the performance of the successive designs. These comparisons highlight the cycle per instruction (CPI) term in the processor performance equation.

Such comparisons based on really implemented softcores are more convincing for students than similar evaluations relying on simulation with no real hardware constraints.

The different microarchitectures described in the book introduce the general concepts related to pipelining: branch delay and cancelation, bypassing in-flight values, load delay, multicycle operators, and more generally, filling the pipeline stages.

The chapter devoted to the RISC-V RV32I is also an introduction to assembly programming. RISC-V codes are obtained from the compilation of C patterns (expressions, tests, loops, and functions) and analyzed.

Throughout the book, some exercises are proposed which can serve as semester projects, like extending the given implementations to the RISC-V M or F ISA subsets.

Experimentation

Such a box signals some experimentation the reader can do from the resources available in the goossens-book-ip-projects/2022.1 folder in the https://github.com/goossens-springer github.

The Vitis_HLS projects are pre-built (you just need to select the testbench file you want to use for your IP simulation) (IP means Intellectual Property, i.e., your component).

The Vivado projects are also pre-built with drivers to directly test your IPs on the development board. The expected results are in the book.

A Practical and Detailed Introduction to High-Level Synthesis and to RISC-V for FPGA Engineers

The book is a very detailed introduction to High-Level Synthesis. HLS will certainly become the standard way to produce RTL, progressively replacing Verilog/VHDL, as in the 50s and 60s, high-level languages progressively replaced the assembly language.

Chapter 2 of the book presents the Xilinx HLS environment in the Vitis tool suite. Based on an IP implementation example, it travels through all the steps from HLS to the Xilinx IP integrator Vivado and the Xilinx Vitis IDE (Integrated Design Environment) to upload the bitstream on the FPGA.

The book explains how to implement, simulate, synthesize, run on FPGA, and even debug HLS softcore projects without the need to go down to Verilog/VHDL or chronograms. HLS is today a matured tool which gives engineers the ability to quickly develop FPGA prototypes. The development of a RISC-V processor in HLS is a one engineer-month duty, when implementing an ARM processor in VHDL was more like a year job, thanks to HLS and thanks to the simplicity of the RV32I instruction nucleus in the RISC-V ISA.

The book explains the major pragmas used by the HLS synthesizer (ARRAY PARTITION, DEPENDENCE, INTERFACE, INLINE, LATENCY, PIPELINE, UNROLL).

Part I of the book concerns individual IPs, and Part II is devoted to System-on-Chips built from multiples core and memory IPs interconnected by an AXI interconnection component.

The book is also an introduction to RISC-V. A full chapter is devoted to the presentation of the RV32I ISA.

The market growth of RISC-V processors is already impressive in the domain of embedded computing. The future of RISC-V might be the same as what was the progression of Unix in the domain of operating systems. At least, the first steps are comparable, with a no-constructor and open-source philosophy.

What the Book Does Not Contain

However, this book does not contain any implementation of the techniques found in the most advanced processors, like superscalar execution, out-of-order execution, speculation, branch prediction, or value prediction (however, these concepts are at least defined).

The reason is that these microarchitectural features are too complex to fit in a small FPGA like the one I used. For example, a cost-effective out-of-order design requires a superscalar pipeline, an advanced branch predictor, and a hierarchical memory altogether.

There is no implementation of advanced parallel management units like a shared memory management (i.e., cache coherency) for the same reason.

The book does not include any implementation of caches or complex arithmetic operators (multiplication, division, or floating-point unit). They can fit on the FPGA (at least in a single core and single thread processor). They are left as an exercise for the reader.

An Organization in Two Parts: Single Core Designs, Multiple Core Designs

The book is divided into two parts and 14 chapters, including an introduction and a conclusion. Part I, from Chaps. 1 to 10, is devoted to single core processors. Part II, from Chaps. 11 to 14, presents some multicore implementations.

Chapter 1 is the introduction. It presents what an FPGA is and how HLS works to transform a C program into a bitstream to configure the FPGA.

The two following chapters give the necessary indications to build the full environment used in the book to develop the RISC-V processors.

Chapter 2 is related to the Xilinx Vitis FPGA tools (the **Vitis_HLS** FPGA synthesizer, the **Vivado** FPGA integrator, and the **Vitis IDE** FPGA programmer).

Chapter 3 presents the RISC-V tools (the Gnu toolchain, the **Spike** simulator, and the **OpenOCD/gdb** debugger), their installation, and the way to use them.

Chapter 4 presents the RISC-V architecture (more precisely, the RV32I ISA) and the assembly language programming.

Chapter 5 shows the three main steps in building a processor: fetching, decoding, and executing. The construction is incremental.

The general principles of HLS programming, in contrast to classic programming, are explained in the first section of Chap. 5.

Chapter 6 completes chapter five with the addition of a data memory to fulfill the first RISC-V processor IP. The implemented microarchitecture has the most simple non-pipelined organization.

Chapter 7 explains how you should test your processor IPs, using small RISC-V codes to check each instruction format individually, also using the official RISC-V-tests pieces of codes provided by the RISC-V organization. Eventually, you should run some benchmarks to test your IP behavior on real applications.

I have combined the RISC-V-tests and the mibench benchmarks [12] to form a suite which is used both to test and to compare the different implementations throughout the book.

At the end of Chap. 7, you will find many hints on how to debug HLS codes and IPs on the FPGA.

Chapter 8 describes pipelined microarchitectures, starting with a two-stage pipeline and ending with a four-stage pipeline.

Chapter 9 pushes pipelining a step further to handle multicycle instructions. The multicycle pipeline RISC-V IP has six stages. It is a necessary improvement in the pipeline organization to run RISC-V multicycle instructions like the ones found in the F and D floating-point extensions, or to implement cache levels building hierarchized memories.

Chapter 10 presents a multiple *hart* IP (a *hart* is a HARdware Thread). Multithreading is a technique to help filling the pipeline and improve the processor throughput. The implemented IP is able to run from two to eight threads simultaneously.

Chapter 11 starts the second part. It describes the AXI interconnection system and how multiple IPs can be connected together in Vivado and exchange data on the FPGA.

Chapter 12 presents a multicore IP based on the multicycle six-stage pipeline. The IP can host from two to eight cores running either independent applications or parallelized ones.

Chapter 13 shows a multicore multihart IP. The IP can host two cores with four harts each or four cores with two harts each.

Chapter 14 concludes by showing how you can use your RISC-V processor implementations to play with your development board, lighting LEDs (Light-Emitting Diode) as push buttons are pressed.

An acronym section is added in the Frontmatter to give the meaning of the abbreviations used in the book.

A few exercises are proposed in the book (with no given solution) which should be viewed by professors as project or laboratory suggestions.

Perpignan, France   Bernard Goossens

References

1. Comer, D.: Operating System Design: The Xinu Approach. Prentice Hall International, Englewood Cliffs, New Jersey (1984)
2. Comer, D.: Operating System Design: The Xinu Approach, Second Edition. Chapman and Hall, CRC Press (2015)

3. https://riscv.org/specifications/isa-spec-pdf/
4. Hennessy, J.L., Patterson, D.A.: Computer Architecture, A quantitative Approach, 6th edition, Morgan Kaufmann (2017)
5. Hennessy, J.L., Patterson, D.A.: Computer Organization and Design: The Hardware/Software Interface, 6th edition, Morgan Kaufmann (2020)
6. Faggin, F.: The Birth of the Microprocessor. Byte, Vol.17, No.3, pp. 145–150 (1992)
7. Faggin, F.: The Making of the First Microprocessor. IEEE Solid-State Circuits Magazine, (2009) https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4776530
8. Patterson, D., Ditzel, D.: The Case for the Reduced Instruction Set Computer. ACM SIGARCH Computer Architecture News, Vol.8, No.6, pp. 5–33 (1980)
9. Chow, P., Horowitz, M.: Architectural tradeoffs in the design of MIPS-X, ISCA'87 (1987)
10. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: Maximizing on-chip parallelism. 22nd Annual International Symposium on Computer Architecture. IEEE. pp. 392–403 (1995).
11. Tendler, J. M., Dodson, J. S., Fields Jr., J. S., Le, H., Sinharoy, B.: POWER4 system microarchitecture, IBM Journal of Research and Development, Vol.46, No 1, pp. 5–26 (2002)
12. https://vhosts.eecs.umich.edu/mibench/

Acknowledgements

I want to thank my reviewers for the invaluable selfless work they have done, reading and correcting an always changing manuscript. They were all great!

So, the only order of presentation which is acceptable is the alphabetical one (based on the last name). If you find some mistakes in the textbook, they are all mine of course.

Yves Benhamou is a computer engineer in a domain not at all related to processor architecture. I asked Yves to be the neophyte reader, going through the steps of the first constructions like installing the softwares and make the first example run on the Pynq-Z2 board I had given him. Moreover, Yves mainly works on Windows and he had to start with a fresh new Ubuntu installation, so he could give me a lot of very important remarks for the reader in the same situation, with missing commands, files, and needed environment preparations. Yves, I hope you had fun discovering FPGA, softcore designs, and HLS! On my side, I was very happy to see that again you were eager to make it run!

Johannes Schoder is a Ph.D. student at Friedrich Schiller University, Jena, Germany. He contacted me in June 2021 to ask for a free access to my "Out-of-Order RISC-V Core Developed with HLS" which I had presented at the second RISCV-week in Paris in October 2019. I was already writing the book, but it was still at a very preliminary stage. Later, after Johannes had been through the code of what is in the book the "multicycle_pipeline_ip", I proposed him to review a part of the textbook. He accepted enthusiastically. Johannes, you did a great job!

Arnaud Tisserand is "Directeur de Recherche" in the French CNRS (Centre National de la Recherche Scientifique). He is an expert in architecture, focusing his research work on chip accelerators in arithmetic and cryptography. I asked him to review the FPGA part of the textbook. It turns out that Arnaud wants to use HLS more intensely for his future designs (mainly because HLS produces prototypes very quickly; this speed in production is invaluable for PhD students), so he proposed to review a bit more stuffs and even try some of the proposed implementations. Thank you so much Arnaud for your help!

I also want to thank the XUP (Xilinx University Program) and more specifically, his manager Cathal Mac Cabe. Cathal is an engineer at Xilinx/AMD. I have been in contact with him through mail, as all the members of the XUP are, asking him many questions on HLS and Vitis. Cathal does a great job at XUP. He is very helpful to all the worldwide academic community of Xilinx product users. The XUP program is crucial for many little research teams like mine. When a professor has the project to teach FPGAs or HLS to undergraduate students, he/she must convince his/her colleagues that devoting a few thousand of euros on the yearly budget to buy development boards is worth the spending. This is where XUP plays a major role. By providing free boards, it helps the pedagogical teams to set preliminary experimentations which can serve as persuasive arguments for the faculty members who are not in the computer science domain. So, thank you again Cathal for the great supporting job (and for the free boards)!

There are many more people who directly or indirectly influenced the textbook. Of course, my colleagues of the LIRMM laboratory (Laboratoire d'Informatique, Robotique et Microélectronique de Montpellier) and more particularly the actual members of the DALI team (Digits, Architecture et Logiciels Informatiques) in Perpignan: Dushan Bikov, Matthieu Carrère, Youssef Fakhreddine, Philippe Langlois, Kenelm Louetsi, Christophe Nègre, David Parello, Guillaume Révy, and Vincent Zucca.

Last but not least, I thank my wife, Urszula, for her every day support. It is not easy for the family to accept sacrificing so much time, including weekends and holidays, to this kind of long ongoing project. Urszula, you took a so important part in the realization of this book!

Contents

Part I Single Core Processors

Part II Multiple Core Processors

Acronyms

The given definitions are all taken from online Wikipedia. The “In short” versions are all mine. They intend to be less general but a bit more practical.

ABI Application Binary Interface: an interface between two binary program modules. In short, an ABI fixes a general frame to build applications from the processor architecture.

ALU Arithmetic and Logic Unit: A combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers.

AXI Advanced eXtensible Interface: a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface, mainly designed for on-chip communication. In short, an IP interconnection system.

CLB Configurable Logic Block: A fundamental building block of field-programmable gate array (FPGA) technology. Logic blocks can be configured by the engineer to provide reconfigurable logic gates. In short, the elementary building structure in FPGAs.

CPU Central Processing Unit: The electronic circuitry within a computer that executes instructions that make up a computer program. In short, the processor core.

ELF Executable and Linkable Format: It is a common standard file format for executable files, object code, shared libraries, and core dumps. In 1999, it was chosen as the standard binary file format for Unix and Unix-like systems on x86 processors by the 86open project. By design, the ELF format is flexible, extensible, and cross-platform. For instance, it supports different endiannesses and address sizes, so it does not exclude any particular central processing unit (CPU) or Instruction Set Architecture. This has allowed it to be adopted by many different operating systems on many different hardware platforms. In short, the loadable format of all the executable files in Linux or MacOS systems.

FPGA Field-Programmable Gate Array: An integrated circuit designed to be configured by a customer or a designer after manufacturing. In short, it is a programmable chip.

GUI Graphical User Interface: A form of user interface that allows users to interact with electronic devices through graphical icons and audio indicator such as primary notation, instead of text-based user interfaces, typed command labels or text navigation.

HDL Hardware Description Language: A specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits. In short: an HDL is to integrated circuits what a programming language is to algorithms.

HLS High-Level Synthesis: An automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. In short, implementing hardware with a program written in a high-level language like C or C++.

IP Intellectual Property: A category of property that includes intangible creations of the human intellect. In short, a component.

ISA Instruction Set Architecture: An abstract model of a computer. It is also referred to as architecture or computer architecture. A realization of an ISA, such as a central processing unit (CPU), is called an implementation. In short, a processor architecture is defined by an ISA, i.e., a machine language (or assembly language).

LAB Logic Array Block: see the CLB entry. The LAB is for the Altera FPGA constructor what the CLB is for the Xilinx FPGA constructor.

LUT Lookup Table: An array that replaces runtime computation with a simpler array indexing operation. An n-bit lookup table implements any of the 2^{2^n} Boolean functions of n variables. In an FPGA, a 6-bit LUT is a 64-bits addressable table, which is addressed with a 6-bit word built from the Boolean values of six variables. The addressed bit gives the Boolean value of the function for the input combination forming the address.

OoO Out-of-Order: A paradigm used in most high-performance central processing units to make use of instruction cycles that would otherwise be wasted. In short, a hardware organization to run instructions in their producer to consumer dependencies.

OS Operating System: A system software that manages computer hardware, software resources, and provides common services for computer programs. In short, Linux, Windows, or MacOS.

PCB Printed Circuit Board: A printed circuit board (PCB) mechanically supports and electrically connects electrical or electronic components using conductive tracks, pads, and other features etched from one or more sheet layers of copper laminated onto and/or between sheet layers of a non-conductive substrate. In short, your development board.

RAM Random Access Memory: A form of computer memory that can be read and changed in any order, typically used to store working data and machine code. In short, the processor's main memory.

RAW Read After Write dependency (or true dependency): An instruction refers to a result that has not yet been calculated or retrieved.

RTL Register Transfer Level: A design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. In short, it is the description of the behavior of a circuit in terms of gates or VHDL/Verilog program.

USB Universal Serial Bus: An industry standard that establishes specifications for cables, connectors, and protocols for connection, communication, and power supply (interfacing) between computers, peripherals, and other computers. In short: an interface to connect low- or medium-speed peripherals to the computer.

Verilog Verilog, standardized as IEEE 1364: A hardware description language (HDL) used to model electronic systems. It is most commonly used in the design and verification of digital circuits at the register-transfer level of abstraction. In short: one of the two mostly used HDL, with VHDL.

VHDL VHSIC Hardware Description Language: A hardware description language (HDL) that can model the behavior and structure of digital systems at multiple levels of abstraction, ranging from the system level down to that of logic gates, for design entry, documentation, and verification purposes. In short: VHDL is to integrated circuits what C is to algorithms.

VHSIC Very High-Speed Integrated Circuit Program: A United States Department of Defense (DOD) research program that ran from 1980 to 1990. Its mission was to research and develop very high-speed integrated circuits for the United States Armed Forces.

Part I
Single Core Processors

In this first part, I present a set of four implementations of the RV32I RISC-V ISA: non-pipelined, pipelined, multicycle, and multihart (i.e., multithreaded). Each defines a single core IP which has been simulated and synthesized with the Vitis HLS tool, placed and routed by the Vivado tool, and tested on the Xilinx FPGA available on a Pynq-Z1/Pynq-Z2 development board.

1 Introduction: What Is an FPGA, What Is High-Level Synthesis or HLS?

Abstract

This chapter shows what an FPGA is and how it is structured from Configurable Logic Blocks or CLB (in the Xilinx terminology, or LAB, i.e. Logic Array Blocks in Altera FPGAs). It also shows how a hardware is mapped on the CLB resources and how a C program can be used to describe a circuit. An HLS tool transforms the C source code into an intermediate code in VHDL or Verilog and a placement and routing tool builds the bitstream to be sent to configure the FPGA.

1.1 What Hardware to Put in an FPGA?

A processor is a hardware device (an *electronic circuit*, or a *component*), able to calculate everything which is *computable*, i.e. any calculation for which there is an *algorithm*. It is enough to transform this algorithm into a program and apply the processor to this program with data to obtain the result of the calculation.

For example, a processor can run a program turning it into a tremendous calculator, capable of carrying out all the mathematical operations. For this, the processor contains a central component, the Arithmetic and Logic Unit (ALU). As its name suggests, this electronic circuit unit within the processor performs basic arithmetic and logic operations. In the ALU, the calculation is the propagation of signals passing through *transistors* (a transistor is a kind of small switch or tap which you open or close to give way or block the current; a transistor has three ends: the input, the output, and the open/close command).

The ALU itself contains an *adder*, i.e. a circuit to do binary additions.

How can transistors be assembled in a way which, by supplying the 0s and 1s of the binary representation of two integers as input, obtain the 0s and 1s of the binary representation of the result at the output of the transistor network?

When the solution to the problem is not directly apparent, we must break it down to bring us back to a simpler problem. In school, we learned that to add two decimal

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_1

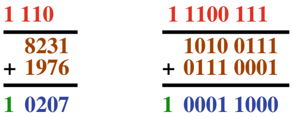


Fig. 1.1 Decimal addition (left), binary addition (right). The two additions are applied to different numbers

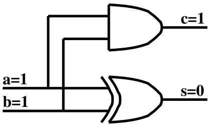


Fig. 1.2 A pair of gates representing a binary adder circuit

numbers is to go from right to left—from units to tens, hundreds, etc.—by adding the digits of the same rank and by propagating the carry to the left. In doing so, two digits are produced at each step (each rank). One is the sum modulo 10 and the other is the carry (see the left part of Fig. 1.1; the line in red is the carries one; the two numbers to be added are in brown; the result is composed of the sum in blue and the final carry in green).

We thus reduced our general problem of the addition of two numbers to the simpler problem of the addition of two digits.

If the digits are binary ones (these are *bits*, abbreviation of *binary digits*), the addition process is identical. The sum is calculated modulo 2 and there is a carry as soon as at least two of the three input bits are 1s (see the right part of Fig. 1.1).

The modulo 2 sum of two bits and their carry can be defined as Boolean operators.

The modulo 2 sum is the exclusive OR (or XOR) of the two bits (the symbol $\oplus$) and the carry is the AND (the symbol $\wedge$).

Notice that $a \oplus b$ is 1 if and only if a and b are different (one of the two is a 1 and the other is a 0). Similarly, $a \wedge b$ is 1 if and only if a and b are both 1s.

The transition to Boolean operators is a decisive step towards a hardware representation of the binary addition because we know how to build, based on transistors, small constructions which form logic *gates*, i.e. Boolean operators. Thus, we find NOT, AND, OR, XOR gates, but also NAND, NOR, and all the Boolean functions of two variables.

Our binary adder becomes the pair of gates in Fig. 1.2. The top gate is an AND and the bottom gate is an XOR. The figure shows the operation of the circuit when two inputs are provided with the same value 1.

To build a circuit, we must draw the gates which compose it or write a program in a hardware description language (HDL) like VHDL or Verilog. The code in Listing 1.1 is the VHDL interface and implementation of the adder shown in Fig. 1.2.

Listing 1.1 A VHDL function defining a 1-bit adder

```
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity BIT_ADDER is
  port(a, b: in  STD_LOGIC;
       s, c: out STD_LOGIC);
end BIT_ADDER;
architecture BHV of BIT_ADDER is
begin
```

```
  s <= a xor b;
  c <= a and b;
end BHV;
```

A succession of softwares transform these gates or this program into transistors then organize them on a *layout*. Finally, a founder physically builds the circuit.

Instead of relying on an electronic foundry, we can use an FPGA. Its name indicates that it is more or less an array of gates which can be *programmed*. The programming consists of linking these gates into subsets forming small circuits within the large circuit which houses them.

1.2 Look-Up Table (LUT): A Piece of Hardware to Store a Truth Table

We want to add two 1-bit words, i.e. $s = a + b$, where a and b are Boolean variables. The sum s is a 2-bit word, composed of the modulo 2 sum and the carry bit.

For example, if the pair (a, b) is $(1, 1)$, their sum s in binary is 10, which is the concatenation of the carry bit (1) and the modulo 2 sum (0).

Let us first concentrate on the modulo 2 sum.

We can define the modulo 2 sum as a Boolean function of two variables, which truth table is presented as Table 1.1, where the arguments of the function are in blue and the values of the function are in red.

For example, the last line of the table says that if $a = b = 1$ then $s = 0$, i.e. $s(1, 1) = 0$.

A LUT (acronym of Look-Up Table) is a hardware device which is similar to a memory. This memory is filled with the truth table of a Boolean function. For example, a 4-bit LUT (or a LUT-2), can store the truth table of a Boolean function of two variables (the red values in Table 1.1).

More generally, a 2^n-bit LUT (or a LUT-n), can contain the truth table of a Boolean function of n variables (where you can fit 2^n truth values).

For example, NOT is a single-variable Boolean function. It is represented by a two line truth table and a LUT-1, i.e. two truth values (NOT(0)=**1** and NOT(1)=**0**).

The two-variable Boolean function AND can be extended to three variables (a AND b AND c). The truth table has eight lines, as shown in Table 1.2.

Table 1.1 The truth table of the modulo 2 sum of two 1-bit words

a	b	s
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.2 The truth table of the operator *a* AND *b* AND *c*

a	b	c	s
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Table 1.3 The truth table of the operator "IF *a* THEN *b* ELSE *c*"

a	b	c	s
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

The operator "IF *a* THEN *b* ELSE *c*" is another example of a three-variable Boolean function. It is represented by the eight truth values in Table 1.3. The function "IF *a* THEN *b* ELSE *c*" is *c* if *a* is 0 (the values in cyan), *b* otherwise (the values in orange).

The LUT, like the truth table, is a kind of universal Boolean function. By filling it, you define the Boolean function it contains.

In an FPGA, a LUT is represented by a memory, i.e. an addressable hardware device. By addressing the memory, we obtain what it contains at the provided address.

Figure 1.3 shows how, from $a = 1$ and $b = 1$ inputs, meaning the LUT address 3 (**11** in binary), the bit contained in the memory cell at the bottom right is accessed. The address is split into two halves. One half (*a* input) serves as a row selector and the other half (*b* input) as a column selector.

In Fig. 1.3, the memory is addressed with $a = 1$, which selects the line framed in red. It is also addressed with $b = 1$, which selects the column framed in green. The intersection of the chosen row and column provides the output *s* of the LUT.

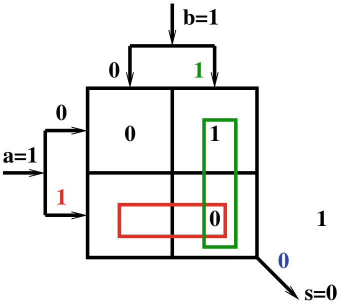


Fig. 1.3 Accessing the address 3 cell (**11** in binary) of the LUT to compute $s = (1 + 1)$ modulo 2

1.3 Combining LUTs

Let us continue our construction of an adder. This time, let us try to build a *full adder*, that is, a hardware cell calculating the modulo 2 sum of two bits and an input carry. The *carry* is a new input variable which extends the truth table from four to eight rows. The modulo 2 sum of the three input bits a, b, and c_i is their XOR ($a \oplus b \oplus c_i$).

Rather than being built with a LUT-3, the full adder is built with two LUT-2. It calculates not only the modulo 2 sum of its three inputs but also their carry.

For example, if $a = 0$ and $b = 1$, the modulo 2 sum is $s = 1$. But if there is a carry in, say $c_i = 1$, then the modulo 2 sum is $s = 0$ and there is a carry out $c_o = 1$.

A first LUT-2 is used to store the truth table of the Boolean function *generate*. As its name suggests, the *generate* function value is 1 when the sum of the two sources a and b generates a carry, i.e. when $a = b = 1$, meaning $a \wedge b$ is true (a AND b).

A second LUT-2 stores the truth table of the Boolean function *propagate*. The *propagate* function value is 1 when both sources a and b can propagate an inbound carry but cannot generate one, which happens when either of the two sources is a 1 but not both simultaneously. The *propagate* function is the XOR ($a \oplus b$).

We link the two tables as shown in Fig. 1.4. In the figure, the addressed table entries are shown in red. The values out of the LUTs are propagated to the *mux* box and to the XOR gate on the right.

The box labeled *mux* is a *multiplexer*, that is, equivalent to the Boolean function “IF x THEN y ELSE z”. The input coming from the left side of the box is the x selector. The entries coming from the lower edge of the box are the choices y (the right input) and z (the left input). If the selector x is a 0, the choice on the left (z) is found at the output. Otherwise, it is the right choice which crosses the multiplexer (y).

Thus, if $a = 0$ and $b = 1$, the *propagate* function value is 1 and the *generate* function value is 0 (these values are in red in the figure). If the incoming carry c_i is 1, the multiplexer selector (1) passes the right choice c_i and the outgoing carry is $c_o = 1$.

The structure of the prefabricated circuit composed of the two LUTs and the two gates brings out the *propagation* of the input carry c_i towards the output c_o when the multiplexer chooses its right input, i.e. when *propagate* is 1. This carry propagation

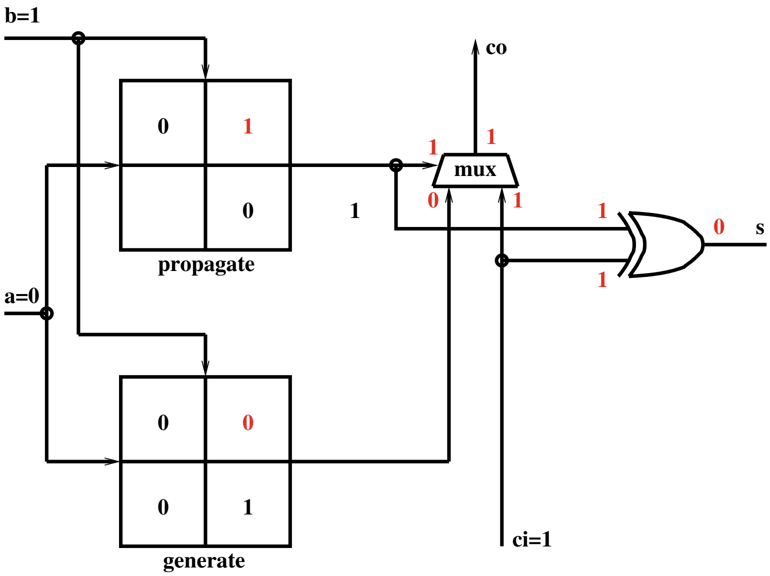


Fig. 1.4 A full adder with two LUT-2

mode is very efficient because the input signal c_i is found at the output after a single gate crossing.

The gate on the right of the figure is an XOR. It produces the modulo 2 sum of *propagate* and c_i. Since *propagate* is itself an XOR, the output s is the modulo 2 sum of the three inputs a, b, and c_i $(a \oplus b \oplus c_i)$.

In the example presented, the combination of the two LUTs, the multiplexer and the XOR gate calculates two bits, one representing the modulo 2 sum of the three inputs and the other being the outgoing carry. By sticking these two bits together, we form the 2-bit sum of the three inputs ($0 + 1 + 1 = 10$ in binary).

The organization of the full adder proposed above corresponds to what is found in a CLB, i.e. a Configurable Logic Block, which is the basic building block of the FPGA (in [1], pages 19 and 20, you have the exact description of the CLBs you find in the FPGA used in this book).

A CLB combines a LUT with a fast carry propagation mechanism. It is a kind of Swiss army knife, which computes logic functions with the LUT and arithmetic functions with the carry propagation.

The programming or *configuration* of the CLB is the filling of the LUTs with the truth values of the desired Boolean functions (in the example, the *propagate* and *generate* functions; the Swiss army knife may divide the LUT in two halves to install two Boolean functions).

1.4 The Structure of an FPGA

Let us try to extend our adder from a 1-bit word adder to a 2-bit word adder.

Let $A = a_1a_0$ and $B = b_1b_0$, for example $A = 10$, with $a_1 = 1$ and $a_0 = 0$ (i.e. $1 * 2^1 + 0 * 2^0$, or 2 in decimal), and $B = 01$ (i.e. $0 * 2^1 + 1 * 2^0$, or 1 in decimal). The sum $A + B + c_i$ is the 3-bit word $c_os_1s_0$ (for example $10 + 01 + 1 = 100$ or in decimal $2 + 1 + 1 = 4$).

By combining two CLBs configured as a full adder, with the output of the first (c_o0 in Fig. 1.5) connected to the input of the second (c_i1 in Fig. 1.5) and inputs a_0 and b_0 for the first and a_1 and b_1 for the second, three bits are output, forming the $c_os_1s_0$ sum of the two 2-bit words $A = a_1a_0$ and $B = b_1b_0$ and an incoming carry c_i.

Figure 1.5 shows this 2-bit adder.

An FPGA contains a matrix of CLBs (see the left part of Fig. 1.6).

For example, the Zynq XC7Z020 from Xilinx is a SoC (System-on-Chip, i.e. a circuit containing several components: processors, memories, USB and Ethernet interfaces, and an FPGA of course) whose programmable part (the FPGA) contains 6650 CLBs.

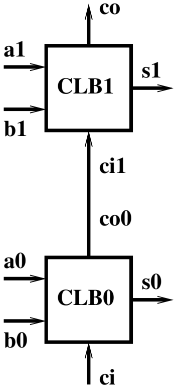
One can imagine that the CLBs are organized in a square of more or less 80 columns and 80 rows (the exact geometry is not described). For a detailed presentation of FPGAs (including their history), you can refer to the Hideharu Amano book [2].

Each CLB contains two identical, parallel and independent SLICEs (centre part of Fig. 1.6). Each slice mainly consists of four LUT-6 and eight flip-flops (the red squares labeled FF—for Flip-Flop—in the centre and right part of Fig. 1.6). Each flip-flop is a 1-bit clocked memory point, which collects the output of the LUT.

Each LUT-6 can represent a six-variable Boolean function or can be split into two LUT-5, each representing a five-variable Boolean function.

The LUTs of the same slice are linked together by a carry propagation chain identical to that of Figs. 1.4 and 1.5 (including a multiplexer and an XOR gate).

Fig. 1.5 A 2-bit adder built from two CLBs



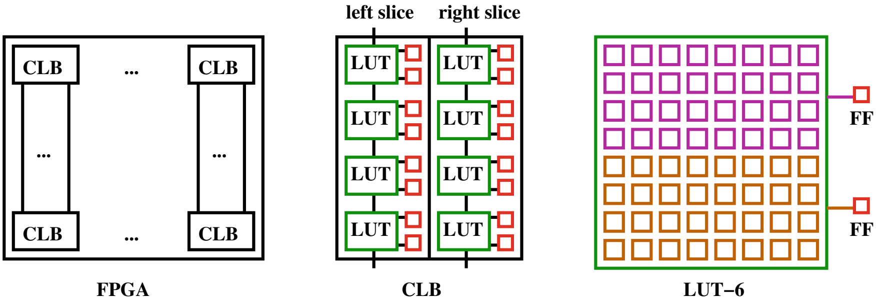
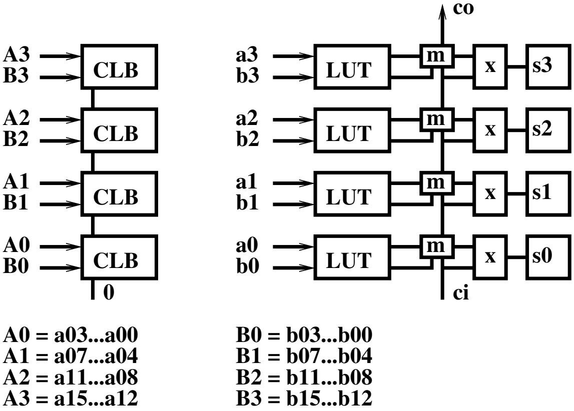


Fig. 1.6 The structure of an FPGA

Fig. 1.7 A 16-bit adder built with four CLBs



A LUT can be partially filled. It may contain only four useful bits out of the 64 available to represent a Boolean function with two variables. But it is better not to waste this resource.

By continuing to extend the 2-bit adder, one can build an adder of any size.

A 16-bit adder links four CLBs of a single column, using 16 LUTs in series, each LUT containing the two tables in Fig. 1.4 (notice that the adder uses only one of the two available slices in each CLB).

The left part of Fig. 1.7 shows how the two 16-bit words to be added are distributed by nibbles in the four CLBs (for exemple the $A_3 = a_{15}a_{14}a_{13}a_{12}$ nibble inputs the highest CLB in the figure). The first CLB in the chain (the lowest in the figure) receives an input carry entry $c_i = 0$.

The right part of the figure shows the addition of the first nibbles ($a_3a_2a_1a_0 + b_3b_2b_1b_0$) in the LUTs (each LUT is split into two half LUTs, the first containing the *generate* function and the second containing the *propagate* function). The multiplexers (boxes labeled m) propagate the carry from the input c_i to the output c_o. The XOR gates (boxes labeled x) provide the modulo 2 sum bits, which may be stored in the flip-flops (rightmost boxes, labeled s_0 through s_3).

1.5 Programming an FPGA

An FPGA is a *programmable* circuit. How do you program a circuit?

A circuit is programmable when it is given two successive operating modes.

The first mode is the initialization. Storing structures are filled with initial values. Once this phase has been completed, the second operating mode starts. It is the computation, which uses the values installed during the initialization.

The initialization phase of an FPGA fills the LUTs with the truth values of the Boolean functions which they represent.

It is also necessary to link the CLBs participating in the same calculation (for example, clear the c_i input of the first CLB of the adder, and link its c_o output to the next CLB c_i input).

This link phase is done with multiplexers (for example to choose c_i between 0 and c_o). The initialization of the links sets the selection bit of the multiplexer.

All the resources of the FPGA (LUTs and inter CLB links) are thus initialized from a sequence of bits which constitutes a *bitstream*.

This sequence of bits is sent from a programming station (i.e. your computer). It comes in the FPGA bit by bit (known as *serial transmission*). Each bit takes its corresponding place in the FPGA.

Once this phase is completed, the FPGA enters the computation phase which performs the function assigned to it by the configuration phase.

In practice, the programming phase lasts a few seconds.

One question remains: how do you go from a function to its hardware implementation on the FPGA?

In the early days of FPGAs (i.e. in the mid-80s), gates were drawn as in Fig. 1.2. A translator was in charge of placing these gates in LUTs.

Soon enough, FPGAs became big and complex enough to implement units which became difficult (and unsafe) to define in schematics.

Hardware description languages (HDL) were proposed (VHDL, Very high speed integrated circuits HDL, and Verilog are the two most used languages).

Rather than drawing gates, we describe a circuit behaviour with a hardware-specific language and a compiler transforms the source description into a bitstream.

An HDL program exactly says how a circuit should behave, from its inputs to its outputs, through binary operations or subroutine calls. In addition to the calculation, the program also expresses the temporality of the signals—i.e. the variables of the program—related to their propagation time in the circuit.

In the mid-90s, a higher level method was proposed, based on classic programming languages (like C or C++). The temporality is left as a job for a translator. The idea was to define a circuit by a program and let the translator implement the corresponding component with CLBs. This method is HLS (High-Level Synthesis; *synthesis* is the name given to the construction of the circuit from a transformation of the source program; the translator is a *synthesizer*).

For example, the C function shown in Listing 1.2 builds a 32-bit adder.

Listing 1.2 A function defining a 32-bit adder

```
void adder_ip(unsigned int  a,
              unsigned int  b,
              unsigned int *c){
  *c = a + b;
}
```

HLS transforms this C code into an intermediate representation (one of them is the RTL or Register Transfer Level representation, which is used in HDL programs; from the RTL, a VHDL or a Verilog program can be built).

A placement and routing software maps the RTL on CLBs (placement phase), then associates these CLBs with those of the FPGA by minimizing the propagation times and sets up the necessary links (routing phase).

The placement and routing leads to the constitution of the bitstream which is lastly transmitted to the FPGA through a USB link (Universal Serial Bus).

The next chapter is devoted to the installation of the HLS, placement and routing softwares, and their application to the adder example.

References

1. https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf
2. H. Amano, *Principles and Structures of FPGAs* (Springer, 2018)

Setting up and Using the Vitis_HLS, Vivado, and Vitis IDE Tools

2

Abstract

This chapter gives you the basic instructions to setup the Xilinx tools to implement some circuit on an FPGA and to test it on a development board. It is presented as a lab that you should carry out. The aim is to learn how to use the Vitis/Vivado tools to design, implement, and run an IP.

2.1 Getting the Hardware

You should first order your development board on which you will later upload your RISC-V processor design.

Any development board with an FPGA and an USB connection can fit.

I use a Pynq-Z1 board from Digilent equipped with a Xilinx Zynq XC7Z020 FPGA [1]. The FPGA is large enough to host the RV32I ISA subset. An equivalent Pynq-Z2 board (from TUL [2]) with the same FPGA would also be fine and very close to my Pynq-Z1.

The Basys3 from Digilent [3] has a Xilinx Artix-7 XC7A35T FPGA. It is also suited to the book goals.

Older boards like the Zybo (Zynq XC7Z020 FPGA), the Zedboard (Zynq XC7Z020 FPGA), or the Nexys4 (Artix-7 XC7A100T FPGA) are also suitable.

More expensive boards like the ZCU 102/104/106 are very large. They can host more ambitious IPs than the ones proposed in the book (for example, more than eight cores or harts in a multicore or multihart processor).

More generally, any board embedding an FPGA with at least 10K LUTs is (more than) large enough to host an RV32I RISC-V core (the more the LUTs, the larger the FPGA). However, to implement the multicore designs presented in the second part of the book, a larger FPGA with at least 30K LUTs is needed.

In the Xilinx Zynq-7000 FPGA family, the XC7Z010 FPGA has 18K LUTs. The XC7Z020 FPGA has 53K LUTs.

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_2

In the Xilinx Artix-7 FPGA family, the XC7A35T has 33K LUTs. The XC7A100T has 101K LUTs.

In the Xilinx UltraScale+ FPGA family, the XCZU7EV has 230K LUTs (ZCU104/106 boards). The XCZU9EG has 274K LUTs (ZCU 102 board).

I have tested two types of development boards: the ones embedding an Artix-7 series FPGA (e.g. Nexys4 and Basys3) and the ones embedding a Zynq-7000 series FPGA (Pynq-Z1, Pynq-Z2, Zybo, and Zedboard).

The difference comes from the way they are interfaced.

The Artix-7 based boards give access to the programmable part of the FPGA through a *microblaze* processor (the microblaze is a Xilinx softcore processor featuring a MIPS-like CPU).

The Zynq based boards are interfaced through a Zynq7 Processing System IP, placed between an embedded Cortex-A9 ARM processor and the programmable part of the FPGA (you can find a very good description of what is inside a Zynq and what you can do with it, for example on a Zedboard, in the Zynq Book [4]).

It does not make much difference in the programming because both processors, microblaze or ARM, run programs built from C codes. But there are differences in the way the IP you will develop should be connected to the interface system IP, either microblaze or Zynq.

If you are a university professor, you may ask for a free board (you will receive a Pynq-Z2 board) through the XUP Xilinx University Program [5].

Otherwise, the Pynq-Z2 board costs around 200€ (210$ or 170£; these are the 2022 Q2 prices) (the Pynq-Z1 board is no more sold). The Basys3 has a smaller FPGA but still large enough to host all the RV32I based RISC-V processors presented in this book. It is sold at 130€ (140$ or 110£).

The development board is the only element you will have to purchase (and this book; but you already have it). Everything else is for free.

If you request XUP, it will probably take a few weeks before you receive your board at home. This is why I started by this step. But meanwhile, you have plenty of duties you can carry out (all the sections in this chapter except Sect. 2.7).

2.2 Getting the Software: The Xilinx Vitis Tool

(If you already know how to use Vitis/Vivado, i.e. Vitis_HLS, Vivado, and Vitis IDE, and if you have already installed Vitis on your computer, you can jump to Chap. 3 to install the RISC-V tools.)

First of all, the following is an important notice concerning the compatibility of the different softwares with the Operating System.

In this book, I assume Linux, Ubuntu (any version from the 16.04 should be compatible with Vitis; I use Ubuntu 22.04 LTS ‘Jammy Jellyfish’ in this book). I also assume Vitis 2022.1 or a later version (if you have an older version, some of my HLS codes might not be synthesizable, but they surely can be simulated within the Vitis HLS tool).

If you are using Windows, you will have to find software installation procedures with your preferred browser.

For the RISC-V simulator, the standard tool spike is not available for Windows. As far as I know, people run spike within a Linux virtual machine inside Windows. Maybe you should consider this option for the RISC-V simulations (and learn a bit of Linux from the commands used in this book).

If you are using MacOS X, you have to install the Xilinx tools through a Linux virtual machine.

If you are using another Linux distribution (e.g. Debian), the given explanations should more or less work.

The Xilinx Vitis software is available for Windows and Linux. If you use Windows, you should find a few differences on how to start the Xilinx tools. Once inside the Vitis software, there is no difference between Linux and Windows.

The Vitis software [6] is freely downloadable from the Xilinx site at the URL shown in Listing 2.1 (you will have to register at Xilinx to download).

Listing 2.1 Xilinx URL from where to download the Vitis software

```
https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/
    vitis.html
```

On Ubuntu 20.04 and 22.04, before installing Vitis, you must install the libtinfo.so.5 library ("sudo apt-get install libtinfo5") (otherwise, the installer hangs as mentioned at https://support.xilinx.com/s/article/76616?language=en_US).

I assume you will install the software in the /opt/Xilinx folder.

If you are working on a Linux computer, you downloaded a file named Xilinx_Unified_2022.1_0420_0327_Lin64.bin (your version name might differ from mine, especially if you get a later one). You must set it to executable and run it with the commands in Listing 2.2 (in sudo mode to install in the /opt/Xilinx folder) (the commands are available in the install_vitis.txt file in the goossens-book-ip-projects/2022.1/chapter_2 folder).

Listing 2.2 Run the Vitis installation software

```
$ cd $HOME/Downloads
$ chmod u+x Xilinx_Unified_2022.1_0420_0327_Lin64.bin
$ sudo ./Xilinx_Unified_2022.1_0420_0327_Lin64.bin
...
$
```

Within the installer, you should choose to install Vitis (first choice in the **Select Product to Install** page).

The installation is a rather long process (i.e. several hours, but mainly depending on the speed of your internet connection; on my computer with a Gb speed connection, it took 12h45 to download 65.77 GB, i.e. 1.43 MB per second; the installation itself took 30 min).

The installation requires a lot of disk space (272 GB), which you can reduce a bit by deselecting some design tools (you really need Vitis, Vivado, and Vitis HLS) and some devices (for the PynqZ1/Z2 boards you need the SoCs devices and for the Basys3 board, you need the 7 Series ones).

Since april 2022, git has been upgraded to face a security vulnerability. If you have not freshly installed or recently upgraded your version of git, you should do it (your version should be superior or equal to 2.35.2; check with "git --version"). Run the commands in Listing 2.3 (they are available in the install_git.txt file in the chapter_2 folder).

Listing 2.3 upgrading git

```
$ sudo add-apt-repository -y ppa:git-core/ppa
$ sudo apt-get update
$ sudo apt-get install git -y
$
```

2.3 Installing the Development Board Definition in the Vitis Software

From the github sites given in Listings 2.4–2.6, download either of these three files: pynq-z1.zip, pynq-z2.zip, master.zip (the first one if you have a Pynq-Z1 board, the second one if you have a Pynq-Z2 board, the third one if you have a Basys3 board; the master.zip file from the Digilent site also contains the definitions of many other boards among which the Nexys4, Zybo, and Zedboard; if you do not find the zip files, search with your preferred browser for "pynq-z1 pynq-z2 basys3 board file").

Listing 2.4 github from where to download the pynq-z1.zip file

```
https://github.com/cathalmccabe/pynq-z1_board_files
```

Listing 2.5 github from where to download the pynq-z2.zip file

```
https://dpoauwgwqsy2x.cloudfront.net/Download/pynq-z2.zip
```

Listing 2.6 github from where to download the master.zip file

```
https://github.com/Digilent/vivado-boards/archive/master.zip
```

Extract the zip and place the extracted folder (the main folder and its sub-folders) into your Vitis installation in the /opt/Xilinx/Vivado/2022.1/data/boards/board_files directory (create the missing board_files folder).

2.4 Installing the Book Resources

To install the book resources, run the commands in Listing 2.7 (available in install_book_resources.txt in the chapter_2 folder; as you have not cloned the resources yet, you should get the file from the github repository).

Listing 2.7 Cloning the book resource folder

```
$ cd
$ git clone
https://github.com/goossens-springer/goossens-book-ip-projects
...
$
```

The newly created goossens-book-ip-projects folder in your HOME is the place where you will work.

2.5 Using the Software

All the source files and shell command files related to the my_adder_ip can be found in the chapter_2 folder.

The Vitis software is a huge piece of code, with many usages among which you will use just a tiny part. To quickly focus on what is useful for you, you will design a small but complete IP example.

The example IP is the adder presented in the introduction, which receives two 32-bit integers and outputs their sum modulo 2^{32}.

You will write two C pieces of code, one which describes the component and the other to use it.

The C code can be input through any text editing tool but I recommend to use the Vitis_HLS tool Graphical User Interface (GUI).

Before you build your project, make sure that the build-essential package is installed. To check, just try to install it again by running the command in Listing 2.8 (this command is in the install_build-essential.txt file in the chapter_2 folder).

Listing 2.8 Install the build-essential package

```
$ sudo apt-get install build-essential
...
$
```

To start the GUI environment, type in a terminal the commands shown in Listing 2.9 (these commands are in the start_vitis_hls.txt file in the chapter_2 folder).

Listing 2.9 Set the required environment and start Vitis_HLS

```
$ cd /opt/Xilinx/Vitis_HLS/2022.1
$ source settings64.sh
$ cd $HOME/goossens-book-ip-projects/2022.1
$ vitis_hls&
...
$
```

On the Debian distribution of Linux, you may have to update the LD_LIBRARY_PATH environment variable before launching Vitis_HLS (export "LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu:$LD_LIBRARY_PATH").

2.5.1 Creating a Project

Once in the Vitis_HLS tool (see Fig. 2.1), you are in the **Vitis_HLS Welcome Page** where you click on **Project/Create Project**.

This opens a dialog box (**New Vitis_HLS Project/Project Configuration**, see Fig. 2.2) which you fill with the name of your project (my_adder_ip; I systematically name my Vitis_HLS project with the _ip suffix). You click on the **Next** button.

The next dialog box opens (**New Vitis_HLS Project/Add/Remove Design Files**, see Fig. 2.3), where you name the top function (e.g. my_adder_ip; I systematically give the same name to the Vitis_HLS project and its top function). This name will be given to the built IP. You can leave the **Design Files** frame empty (you will add design files later). Click on **Next**.

In the next dialog box (**New Vitis_HLS Project/Add/Remove Testbench Files**, see Fig. 2.4), you can leave the **TestBench Files** frame empty (you will add testbench files later). Click on **Next**.

In the **Solution Configuration** dialog box (see Fig. 2.5), you have to select which development board you are targetting. In the **Part Selection** frame, click on the **"..."** box.

In the **Device Selection** dialog box (see Fig. 2.6), click on the **Boards** button.

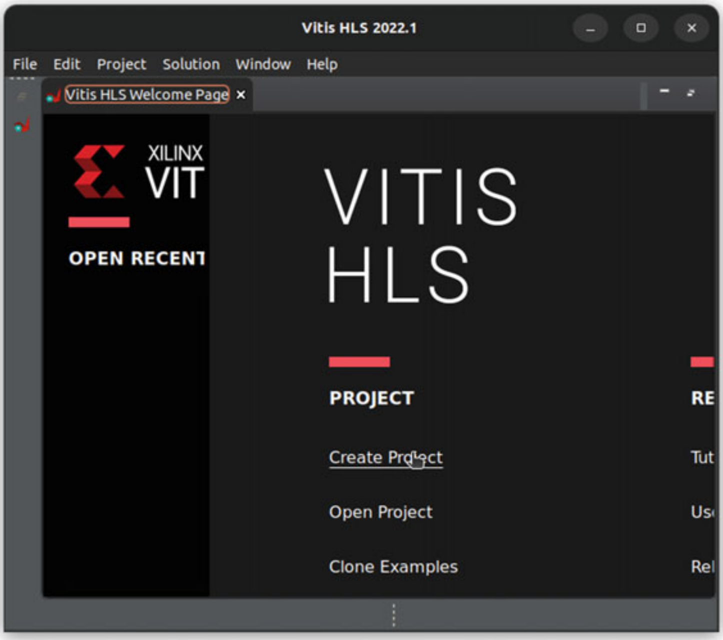
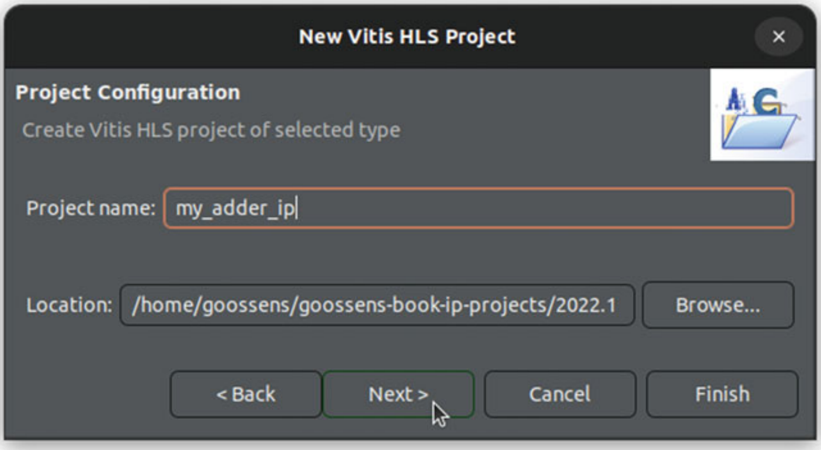


Fig. 2.1 The Vitis_HLS tool welcome page



New Vitis HLS Project

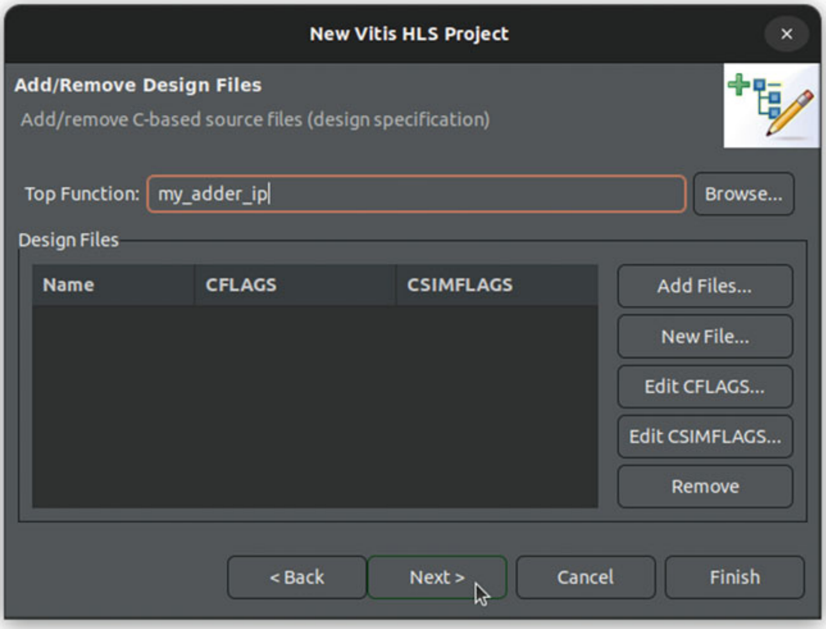
Project Configuration

Create Vitis HLS project of selected type

Project name: my_adder_ip

Location: /home/goossens/goossens-book-ip-projects/2022.1

Fig. 2.2 The **Project Configuration** dialog box



New Vitis HLS Project

Add/Remove Design Files

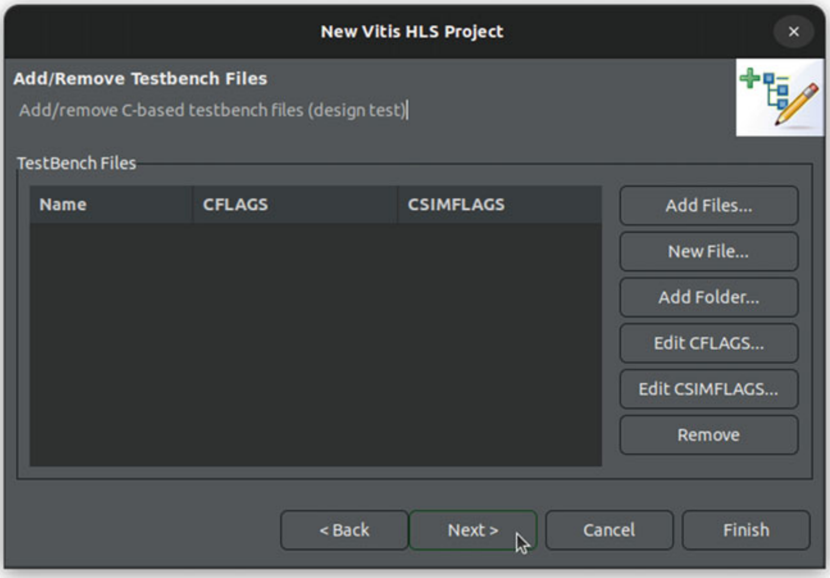
Add/remove C-based source files (design specification)

Top Function: my_adder_ip

Design Files

Name	CFLAGS	CSIMFLAGS

Fig. 2.3 The **Add/Remove Design Files** dialog box to name the top function



New Vitis HLS Project

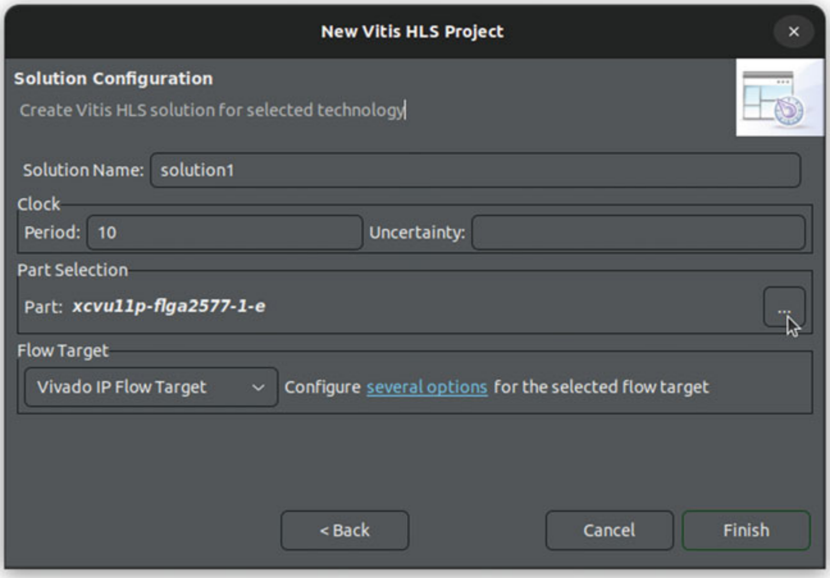
Add/Remove Testbench Files

Add/remove C-based testbench files (design test)

TestBench Files

Name	CFLAGS	CSIMFLAGS

Fig. 2.4 The **Add/Remove Testbench Files** dialog box



New Vitis HLS Project

Solution Configuration

Create Vitis HLS solution for selected technology

Solution Name: solution1

Clock

Period: 10 Uncertainty:

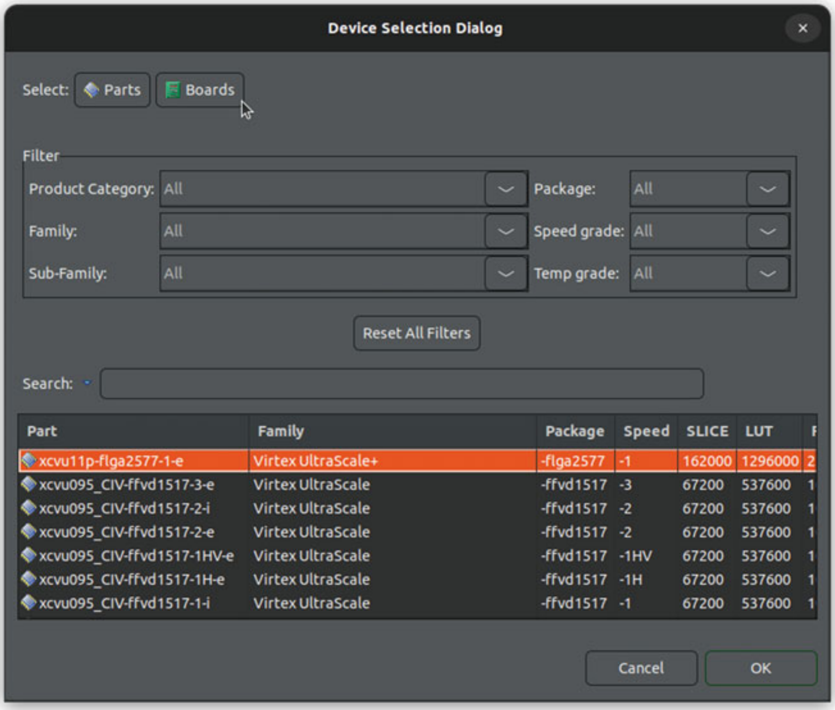
Part Selection

Part: *xcvu11p-flga2577-1-e*

Flow Target

Vivado IP Flow Target Configure several options for the selected flow target

Fig. 2.5 The **Solution Configuration** dialog box to select your development board



Product Category: All
Family: All
Sub-Family: All
Package: All
Speed grade: All
Temp grade: All
Search:

Part	Family	Package	Speed	SLICE	LUT	F
xcvu11p-flga2577-1-e	Virtex UltraScale+	-flga2577	-1	162000	1296000	2
xcvu095_CIV-ffvd1517-3-e	Virtex UltraScale	-ffvd1517	-3	67200	537600	1
xcvu095_CIV-ffvd1517-2-i	Virtex UltraScale	-ffvd1517	-2	67200	537600	1
xcvu095_CIV-ffvd1517-2-e	Virtex UltraScale	-ffvd1517	-2	67200	537600	1
xcvu095_CIV-ffvd1517-1HV-e	Virtex UltraScale	-ffvd1517	-1HV	67200	537600	1
xcvu095_CIV-ffvd1517-1H-e	Virtex UltraScale	-ffvd1517	-1H	67200	537600	1
xcvu095_CIV-ffvd1517-1-i	Virtex UltraScale	-ffvd1517	-1	67200	537600	1

Fig. 2.6 The **Device Selection** dialog box to select your development board

In the **Search** frame (see Fig. 2.7), type z1 or z2 or basys3 (according to your board; even if you are still waiting for it, you can proceed). Select your board (in my case, **Pynq-Z1, xc7z020clg400-1**) and click on **OK** (if you do not see your board in the proposed selection, it means you have not installed the board files properly: go back to 2.3).

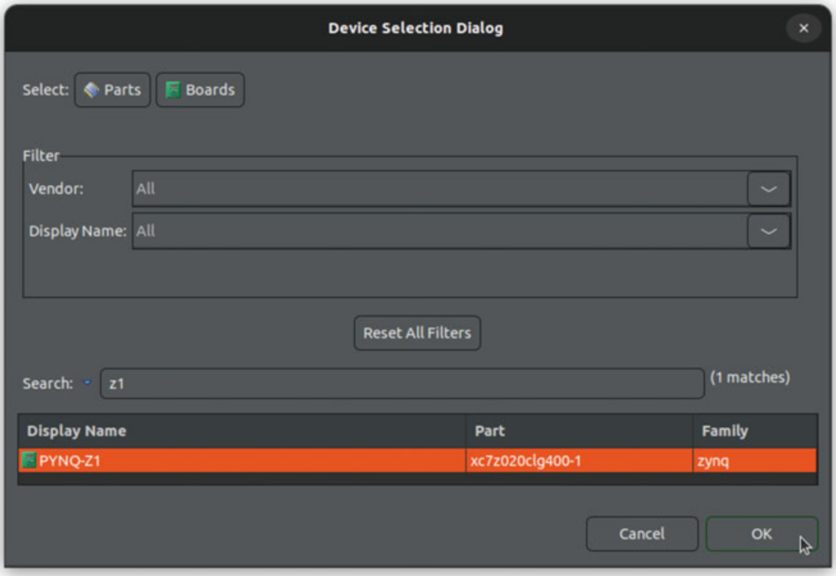
You are back to the **Solution Configuration** box. Click on **Finish** to create your Vitis_HLS project skeleton (see Fig. 2.8).

2.5.2 Creating an IP

The next step after preparing the project is to fill it with some content. You will add two pieces, one which will represent your adder IP and a second devoted to its verification through simulation.

After the project creation, the Vitis_HLS tool opens the **Vitis_HLS 2022.1 - my_adder_ip** eclipse-based window (see Fig. 2.9).

I will not describe the full possibilities. What I will describe now only concerns the editing of the source and testbench files.



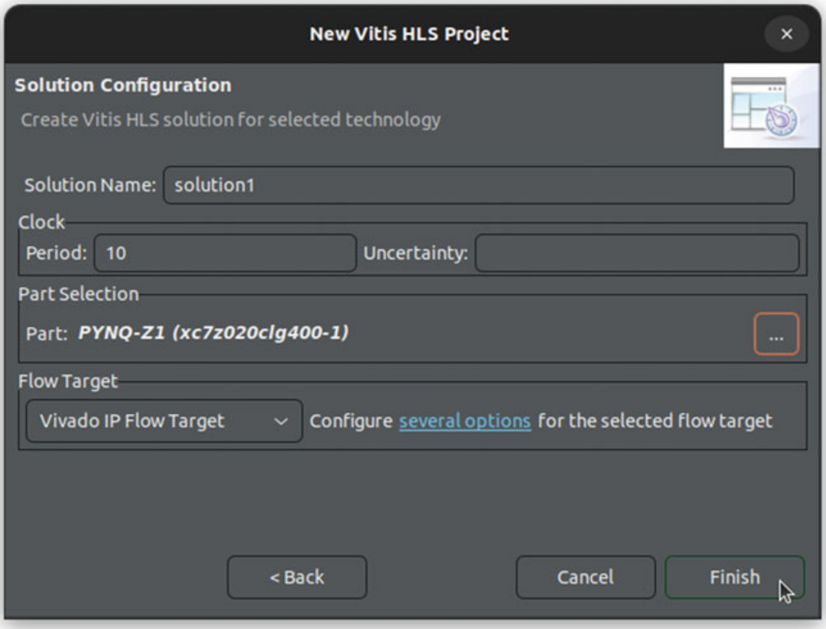
Vendor: All

Display Name: All

Search: z1 (1 matches)

Display Name	Part	Family
PYNQ-Z1	xc7z020clg400-1	zynq

Fig. 2.7 Selecting the Pynq-Z1 board



Solution Configuration

Create Vitis HLS solution for selected technology

Solution Name: solution1

Clock

Period: 10 Uncertainty:

Part Selection

Part: ***PYNQ-Z1 (xc7z020clg400-1)***

Flow Target

Vivado IP Flow Target

Configure several options for the selected flow target

Fig. 2.8 Finishing the Vitis_HLS project creation

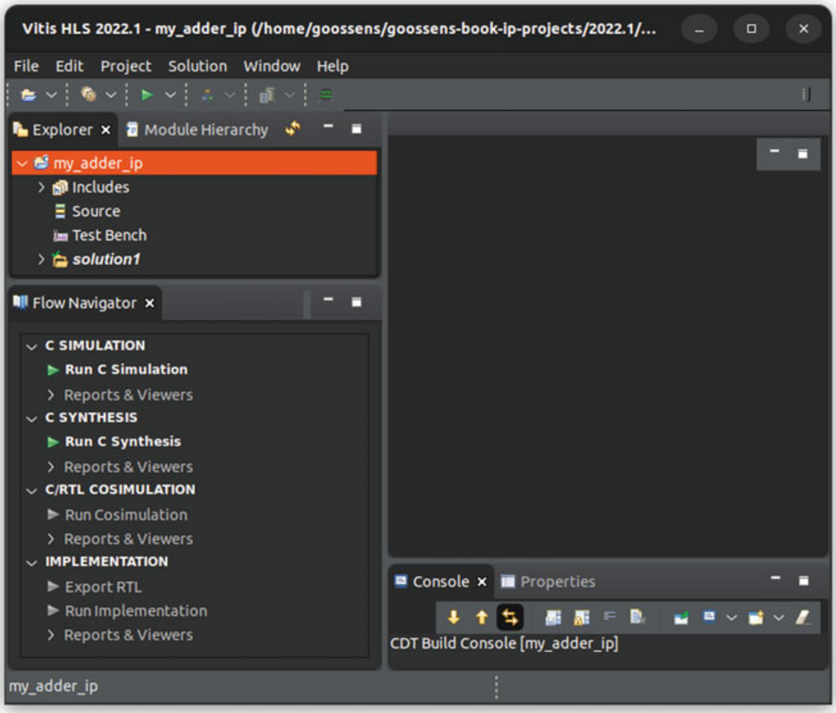


Fig. 2.9 The main Vitis_HLS eclipse-based window

The IP should be designed through a void top function. This top function depicts a component.

A component has a pinout and this pinout is the only way to interact with the component. There is no backdoor. You should not be able to observe nor to modify inside the component, which means that no observer nor modifier function should be provided.

A component may be *combinational* or *sequential*. In the former case, the outputs are a combination of the inputs. In the latter case, the outputs are a combination of the inputs and an internal state which evolves. The internal state is memorized. A sequential component is clocked and at the start of each clock cycle, the internal state is updated.

The internal state is initialized through input pins. It should never be directly manipulated from outside the IP, neither to set it nor to observe its values.

For example, a processor is a sequential component. Its internal state includes the register file. The register file is not visible from outside the processor. The processor definition may include an initialization phase (i.e. a reset phase) to clear the register file and an ending phase to dump it to memory (i.e. a halt phase). But the external world should have no access at all to the register file.

The adder example is a combinational circuit. It has no internal state. However, your processor designs will be sequential IPs with an internal state.

To represent the component and its pinout, you will use a void function prototype. The function arguments are the pinout, with input arguments acting as input pins and output arguments acting as output pins.

In the my_adder_ip example, there are two inputs and one output. The two inputs are the words to be added (32-bit integers) and the output is the sum word (another 32-bit integer). Listing 2.10 shows the my_adder_ip function prototype.

Listing 2.10 The my_adder_ip component prototype or pinout

```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c);
```

The input arguments are values and the output argument is a pointer.

When your component will be simulated, you will use a main function to call the my_adder_ip function. This call will have a hardware signification: apply inputs to the component and let it the time to produce its output and save it into the unsigned int c location.

In the main function, after the call to my_adder_ip, you will print the *c value to mimic the electronic observation of the c pins.

In the FPGA, the a, b and c arguments will be implemented as memory points connected to the my_adder_ip chip inside the programmable part.

Within the Vitis_HLS GUI, right-click on the **Source** button in the **Explorer** frame and choose **New Source File...** (see Fig. 2.10).

In the navigation window, navigate to the my_adder_ip folder, open it, name your new file my_adder_ip.cpp and click on **Save** (see Fig. 2.11).

The new file is added to the sources as shown in Fig. 2.12.

The **my_adder_ip.cpp** tab in the central frame let you edit your top function. Copy/paste the my_adder_no_pragma_ip.cpp file from the chapter_2 folder).

Listing 2.11 The my_adder_ip top-function code

```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c){
  *c = a + b;
}
```

The **my_adder_ip.cpp** tab in your Vitis_HLS window should look like in Fig. 2.13. Save the file.

2.5.3 Simulating an IP

Before you implement the IP on the FPGA, it is better to take the time to test it. You will see that placing and routing a design on an FPGA may be a rather long process (from a few seconds to a few hours, according to the design complexity and your computer efficiency). To avoid repeatedly waits, it is a good practice to debug your HLS code with the Vitis_HLS tool simulation facility. For such a simulation, you

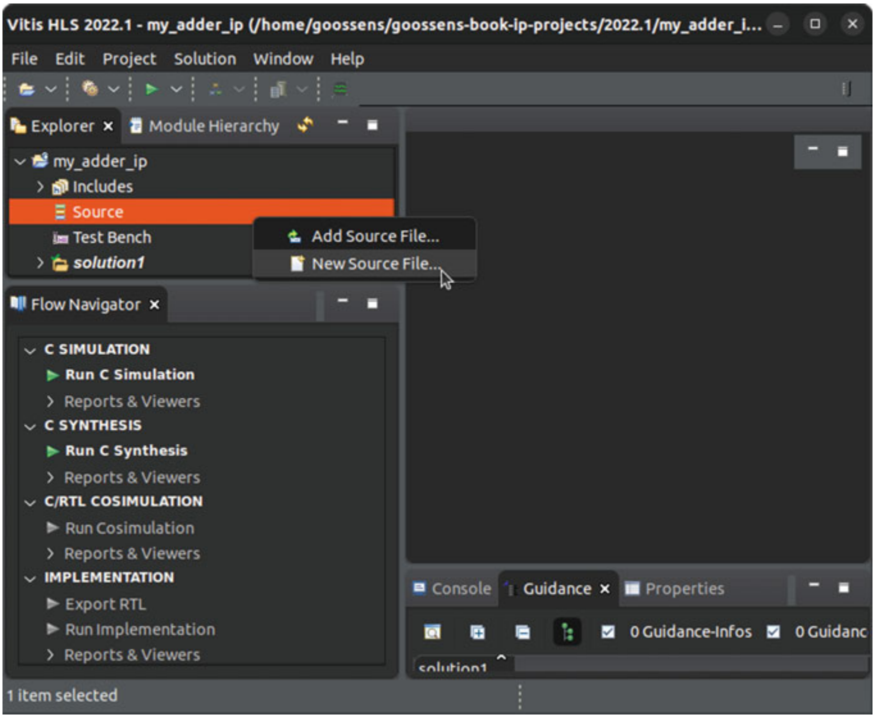
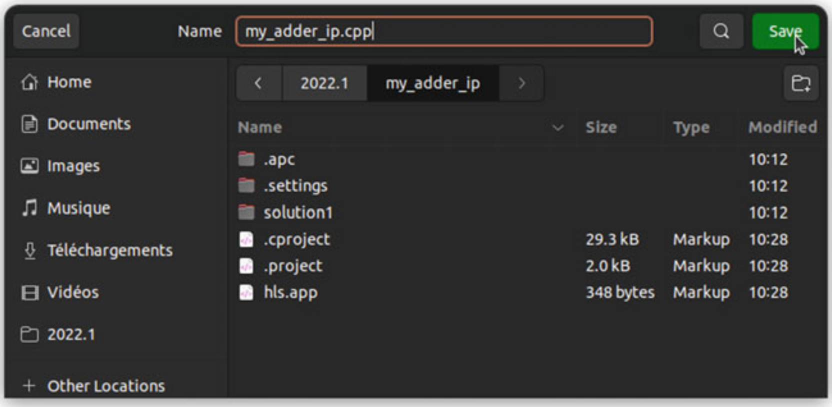


Fig. 2.10 Adding a new source file



Name: my_adder_ip.cpp

Name	Size	Type	Modified
.apc			10:12
.settings			10:12
solution1			10:12
.cproject	29.3 kB	Markup	10:28
.project	2.0 kB	Markup	10:28
hls.app	348 bytes	Markup	10:28

Fig. 2.11 Adding a new my_adder_ip.cpp source file in the my_adder_ip folder

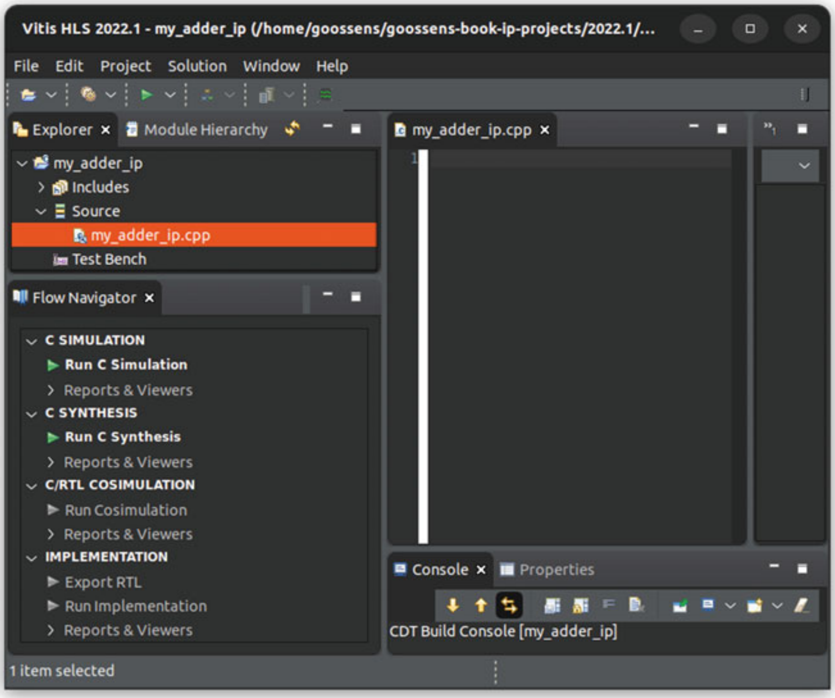


Fig. 2.12 The my_adder_ip.cpp file added to sources

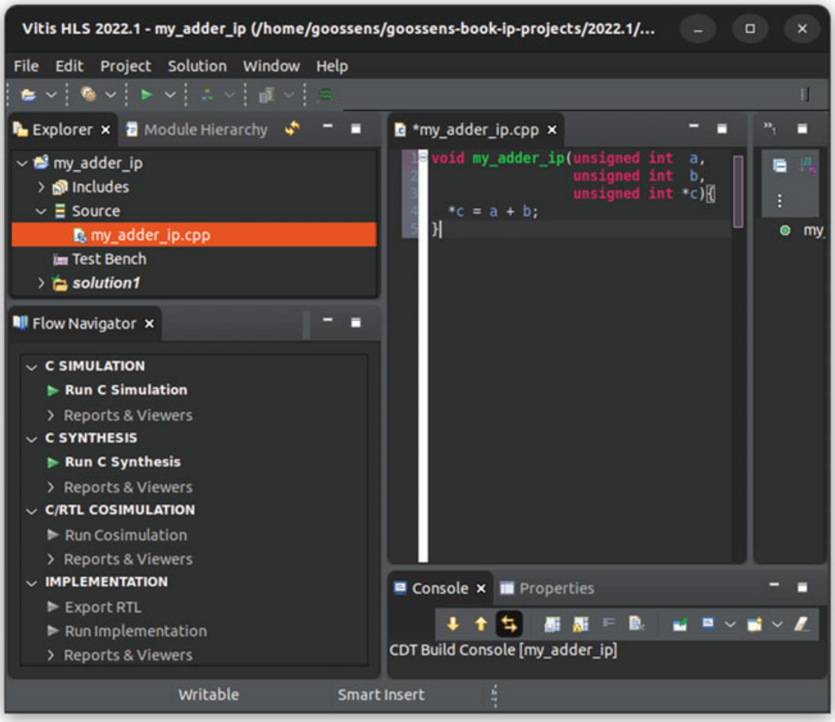
need to provide a main function within a testbench file. The role of this function is to create the component, run it and observe its behaviour.

Right-click on the **Test Bench** button in the **Explorer** frame. Select **New Test Bench File...** (see Fig. 2.14).

In the navigation window, name your new testbench file testbench_my_adder_ip.cpp and click on **Save** (see Fig. 2.15).

The new testbench file is added as shown in Fig. 2.16.

Click on the **testbench_my_adder_ip.cpp** tab in the central frame and fill it with the code in Listing 2.12 (copy/paste the testbench_my_adder_ip.cpp file from the chapter_2 folder).



```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c){
  *c = a + b;
}
```

CDT Build Console [my_adder_ip]

Fig. 2.13 The my_adder_ip top function in the Vitis_HLS window

Listing 2.12 The testbench file and the main function

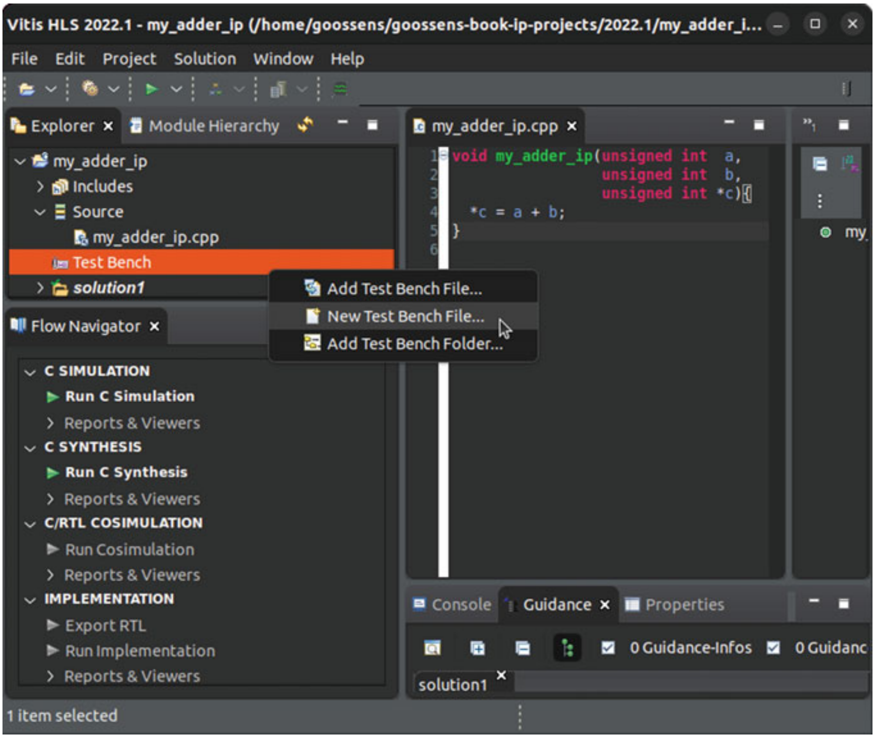
```
#include <stdio.h>
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c);
int main(){
  unsigned int a, b, c;
  a = 10000;
  b = 20000;
  my_adder_ip(a, b, &c);
  printf("%d + %d is %d\n", a, b, c);
  if (c != (a+b)) return 1;
  else return 0;
}
```

The testbench_my_adder_ip.cpp tab in your Vitis_HLS window should look like in Fig. 2.17.

To simulate, click on the **C SIMULATION/Run C Simulation** button in the **Flow Navigator** frame at the bottom left of the window (see Fig. 2.18).

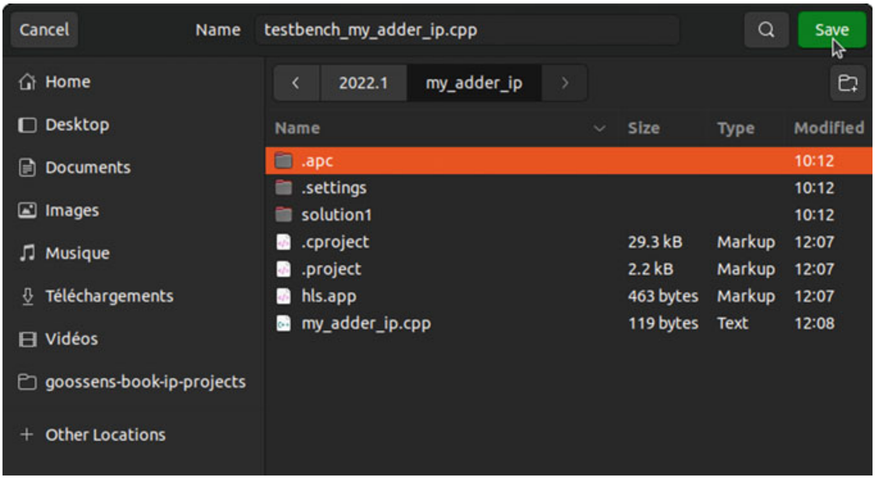
In the **C Simulation** dialog box, just click on the bottom **OK** button (leave all the boxes unchecked; see Fig. 2.19).

You see the result of the compilation and the run (see Fig. 2.20).



```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c){
  *c = a + b;
}
```

Fig. 2.14 Adding a new testbench file to the my_adder_ip folder



Name testbench_my_adder_ip.cpp

Name	Size	Type	Modified
.apc			10:12
.settings			10:12
solution1			10:12
.cproject	29.3 kB	Markup	12:07
.project	2.2 kB	Markup	12:07
hls.app	463 bytes	Markup	12:07
my_adder_ip.cpp	119 bytes	Text	12:08

Fig. 2.15 Adding the testbench_my_adder_ip.cpp file to the my_adder_ip folder

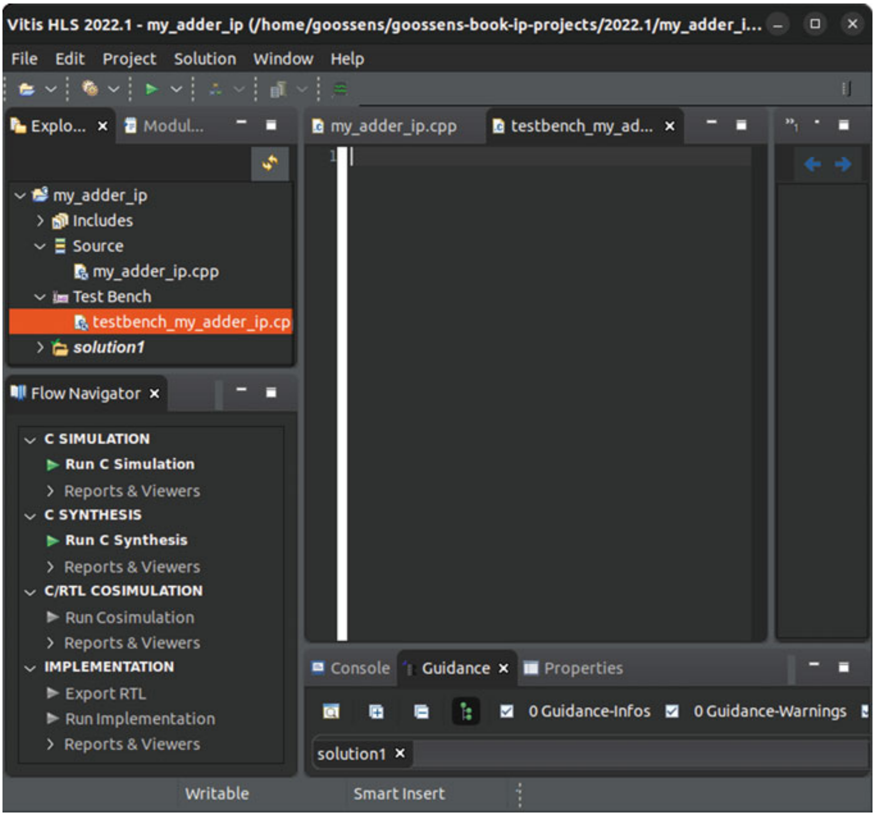


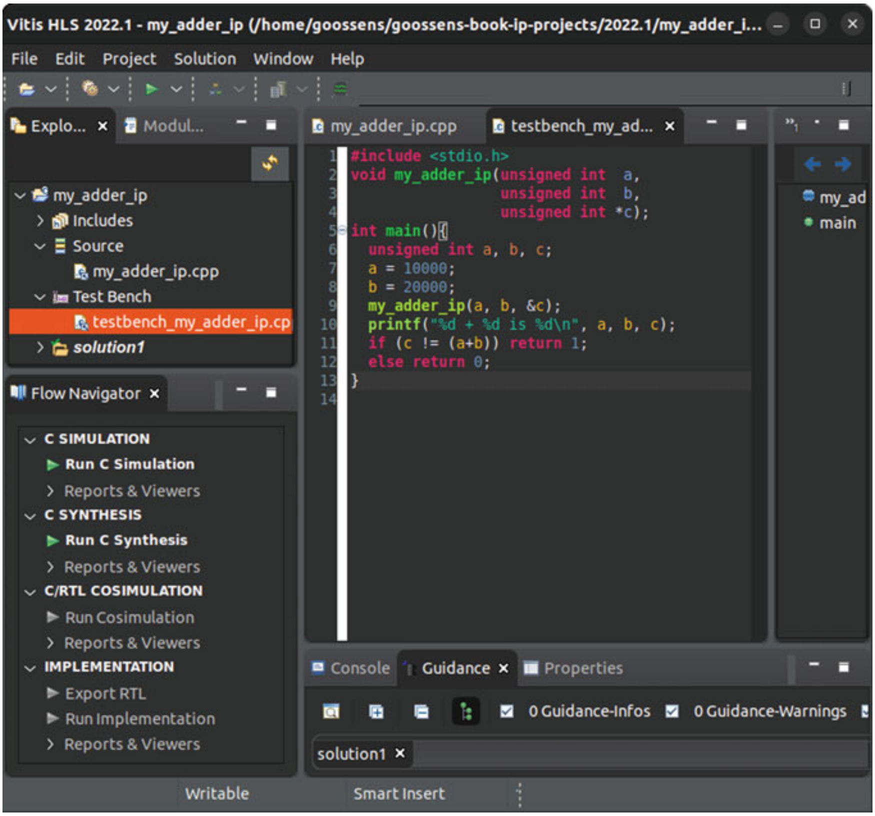
Fig. 2.16 The testbench_my_adder_ip.cpp file added to testbench files

The result of the run (i.e. the print of "10,000 + 20,000 is 30,000") appears in the middle of the output shown in the my_adder_ip_csim.log tab.

On Linux/Ubuntu, if the simulation complains about missing files, it means you have to install some missing libraries. For example, if features.h is the requested missing file, install the g++-multilib library (sudo apt-get install g++-multilib). To know what to install, try to browse with the missing file message.

2.5.4 Synthesizing an IP

To transform your C code into a synthesizable IP, you need to add some indications to the synthesizer. For example, you must specify how the pinout will be mapped on the FPGA. These indications are given through *pragmas*. The Vitis_HLS environment provides many HLS pragmas I will progressively use. For the moment, I will only use the HLS INTERFACE pragma for the pinout.



```
#include <stdio.h>
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c);
int main(){
  unsigned int a, b, c;
  a = 10000;
  b = 20000;
  my_adder_ip(a, b, &c);
  printf("%d + %d is %d\n", a, b, c);
  if (c != (a+b)) return 1;
  else return 0;
}
```

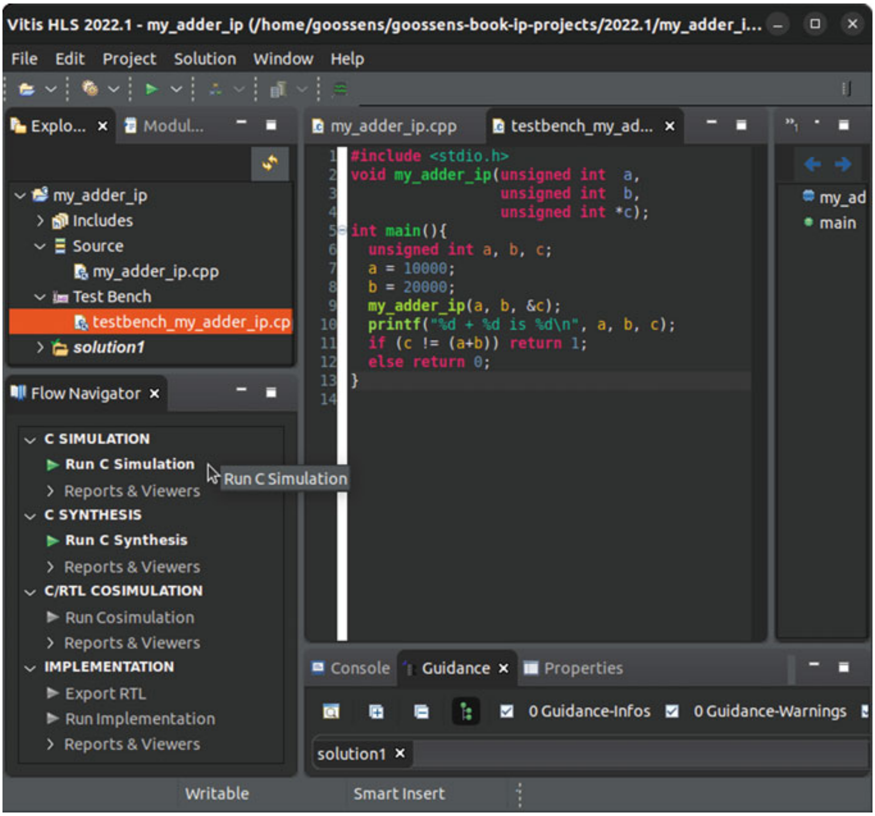
Fig. 2.17 The my_adder_ip testbench function in the Vitis_HLS window

Update the my_adder_ip.cpp file with the pragmas shown in Listing 2.13 (copy-/paste the my_adder_ip.cpp file in the chapter_2 folder) (one HLS INTERFACE pragma for each argument of the my_adder_ip top function and one more named return for the control of the IP start and stop).

Listing 2.13 The HLS INTERFACE pragma

```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c){
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
#pragma HLS INTERFACE s_axilite port=return
  *c = a + b;
}
```

To synthesize your IP, use the **C SYNTHESIS/Run C Synthesis** button in the **Flow Navigator** frame (see Fig. 2.21).



```
#include <stdio.h>
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c);
int main(){
  unsigned int a, b, c;
  a = 10000;
  b = 20000;
  my_adder_ip(a, b, &c);
  printf("%d + %d is %d\n", a, b, c);
  if (c != (a+b)) return 1;
  else return 0;
}
```

Fig. 2.18 Launching the C simulation

A **C Synthesis - Active Solution** window opens, which recapitulates the synthesis parameters (mainly the board used, i.e. the Pynq-Z1 for me). Click on **OK** (see Fig. 2.22).

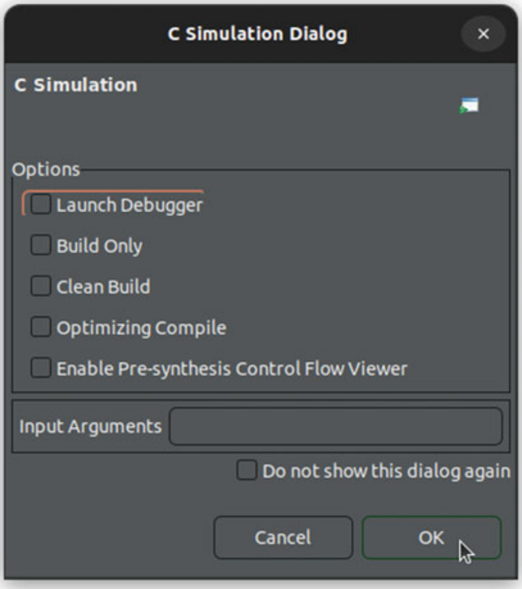
The Synthesis outputs a report in a tab labeled **Synthesis Summary(solution1)** (see Fig. 2.23).

The report mentions that the estimated delay of the adder IP is 4.552 ns with an uncertainty of 2.70 ns (your values could differ if you are working on a different model of FPGA or a different version of Vitis_HLS).

The resource usage estimation is 271 LUTs out of 53,200 available in the FPGA and 150 FF (Flip-Flops) out of 106,400. The real resource utilisation will be given later, in the Vivado tool, after the design placement and routing.

In the **Flow Navigator** frame, click on the **C SYNTHESIS/Reports & Viewers/Schedule Viewer** (see Fig. 2.24).

A **Schedule Viewer** tab opens in which you can vizualize the progression of the computation (see Fig. 2.25).



C Simulation Dialog

C Simulation

Options

□ Launch Debugger

□ Build Only

□ Clean Build

□ Optimizing Compile

□ Enable Pre-synthesis Control Flow Viewer

Input Arguments

□ Do not show this dialog again

Fig. 2.19 Starting the C simulation

In parallel, *a* and *b* are read on the input pins (b_read(read) and a_read(read)), then the addition is done (labeled add_ln8(+), referring to line 8 in the my_adder_ip.cpp file) and last, the result is copied on the *c* output pin (c_write_ln8(write)).

The duration is one FPGA cycle (cycle 0), i.e. 10 ns. The schedule shows that the result is computed before the end of the cycle.

If you click on the add_ln8(+) line, you will highlight blue arrows showing the dependencies between the inputs, the computation and the output (see Fig. 2.26).

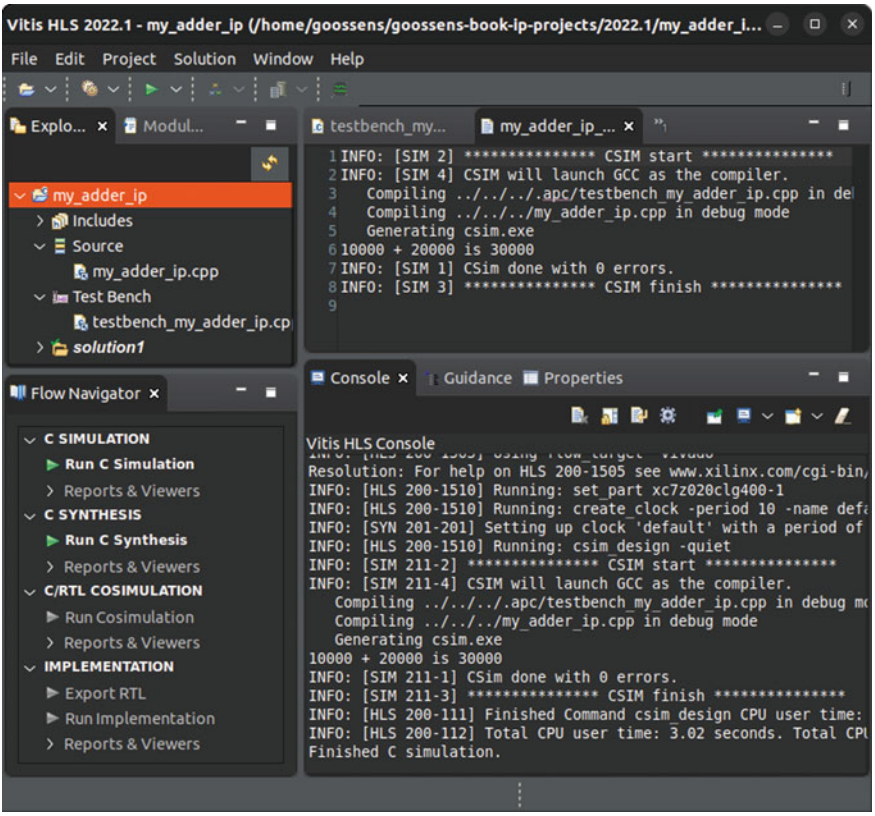
If you click on the **Properties** tab in the lowest frame, you display properties of the output signals involved in the addition (see Fig. 2.27): bit width of the sum (32 bits), delay (2.55 ns).

If you right-click on the add_ln8(+) line and the proposed **Goto Source** (see Fig. 2.28), you display the source code in the **C Source** tab in the lowest frame.

This is the code which the add_ln8(+) comes from (see Fig. 2.29). The involved line is highlighted with a blue background.

After you have successfully synthesized your adder you can try to cosimulate it.

Cosimulation runs two simulations. One simulation is the former C simulation we have already done, which produces the expected result. The other simulation is a logical simulation from the HDL description of the circuit and a simplified library model of the FPGA. Its output is compared to the C simulation one. If they match, it means that the signal propagation in the FPGA produced the same final value as the one computed by the C simulation.



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../.apc/testbench_my_adder_ip.cpp in de
   Compiling ../../../my_adder_ip.cpp in debug mode
   Generating csim.exe
10000 + 20000 is 30000
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

```
INFO: [HLS 200-1505] Using flow_target 'vivado'
Resolution: For help on HLS 200-1505 see www.xilinx.com/cgi-bin,
INFO: [HLS 200-1510] Running: set_part xc7z020clg400-1
INFO: [HLS 200-1510] Running: create_clock -period 10 -name defa
INFO: [SYN 201-201] Setting up clock 'default' with a period of
INFO: [HLS 200-1510] Running: csim_design -quiet
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
   Compiling ../../../.apc/testbench_my_adder_ip.cpp in debug mc
   Compiling ../../../my_adder_ip.cpp in debug mode
   Generating csim.exe
10000 + 20000 is 30000
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
INFO: [HLS 200-111] Finished Command csim_design CPU user time:
INFO: [HLS 200-112] Total CPU user time: 3.02 seconds. Total CPL
Finished C simulation.
```

Fig. 2.20 The C simulation log report

To run the cosimulation, in the **Flow Navigator** frame, click on **C/RTL COSIMULATION/Run Cosimulation** (see Fig. 2.30).

In the dialog box (see Fig. 2.31), just click on the bottom **OK** button.

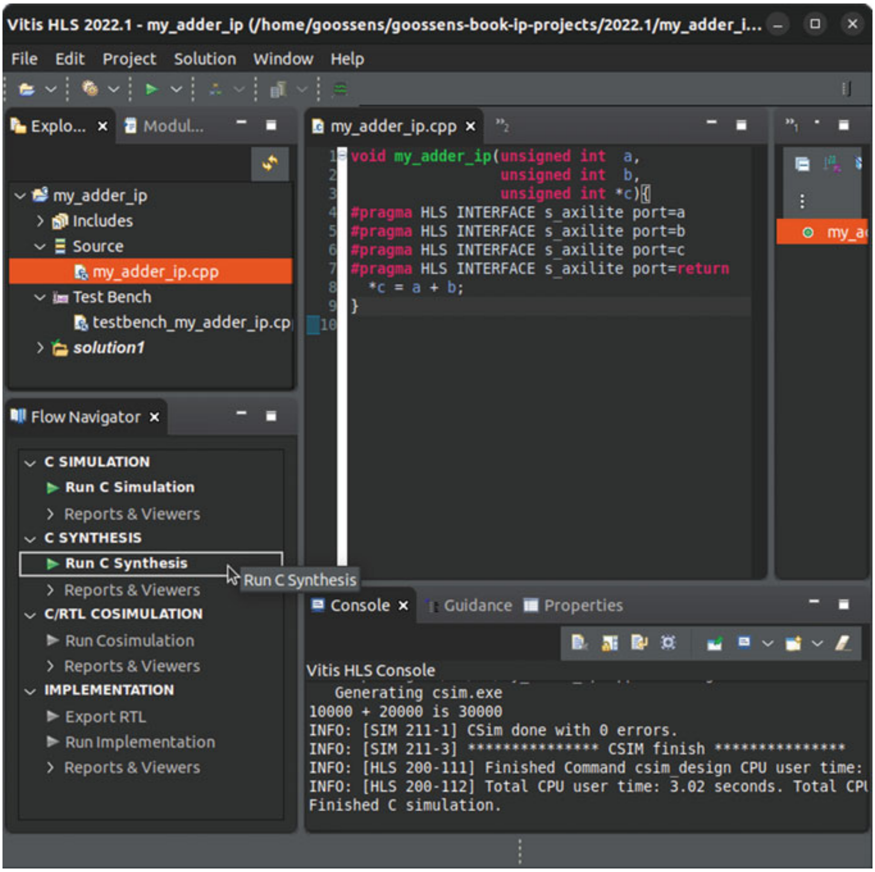
The **Cosimulation Report** for your my_adder_ip should show Pass in the **Status** entry of the displayed **General Information** (see Fig. 2.32).

The IP can be exported to the library of available components, usable in the Vivado tool.

To export the RTL file built by the synthesis, click on **IMPLEMENTATION/Export RTL** (see Fig. 2.33).

In the dialog box click on **OK** (see Fig. 2.34).

Click on the **Console** tab. The **Console** frame (see Fig. 2.35) should report that the export has finished (which means successfully).



```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c){
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
#pragma HLS INTERFACE s_axilite port=return
  *c = a + b;
}
```

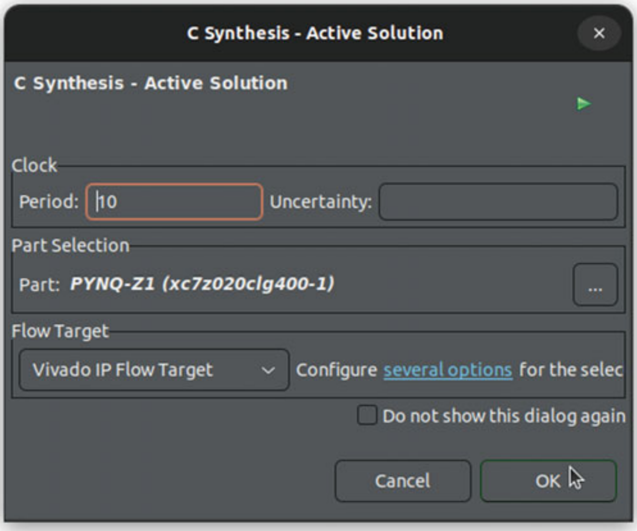
```
   Generating csim.exe
10000 + 20000 is 30000
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
INFO: [HLS 200-111] Finished Command csim_design CPU user time:
INFO: [HLS 200-112] Total CPU user time: 3.02 seconds. Total CPU
Finished C simulation.
```

Fig. 2.21 Running the C synthesis

The last step before you test your design on the board FPGA is to run the exported RTL. Click on **IMPLEMENTATION/Run Implementation** (see Fig. 2.36).

In the **Run Implementation** dialog box, change the default selection by checking the **RTL Synthesis, Place & Route** button and click on **OK** (see Fig. 2.37).

The report shows the final resource usage and IP timing as shown in Fig. 2.38 (these are still estimations; they may differ from the real implementation done in Vivado; moreover, you may have different values if you are working on a different model of FPGA or a different version of Vitis_HLS). The design uses 128 LUTs and 150 FFs (Flip-Flops). No RAM block is used (BRAM).



Period: 10 Uncertainty:

Part: PYNQ-Z1 (xc7z020clg400-1)

Vivado IP Flow Target Configure several options for the selec

□ Do not show this dialog again

Fig. 2.22 Confirming the active solution parameters

2.6 Creating a Design with Vivado

(If you already know Vivado and either Vitis IDE or Vivado SDK, you can jump to Chap. 3 to install the RISC-V tools.)

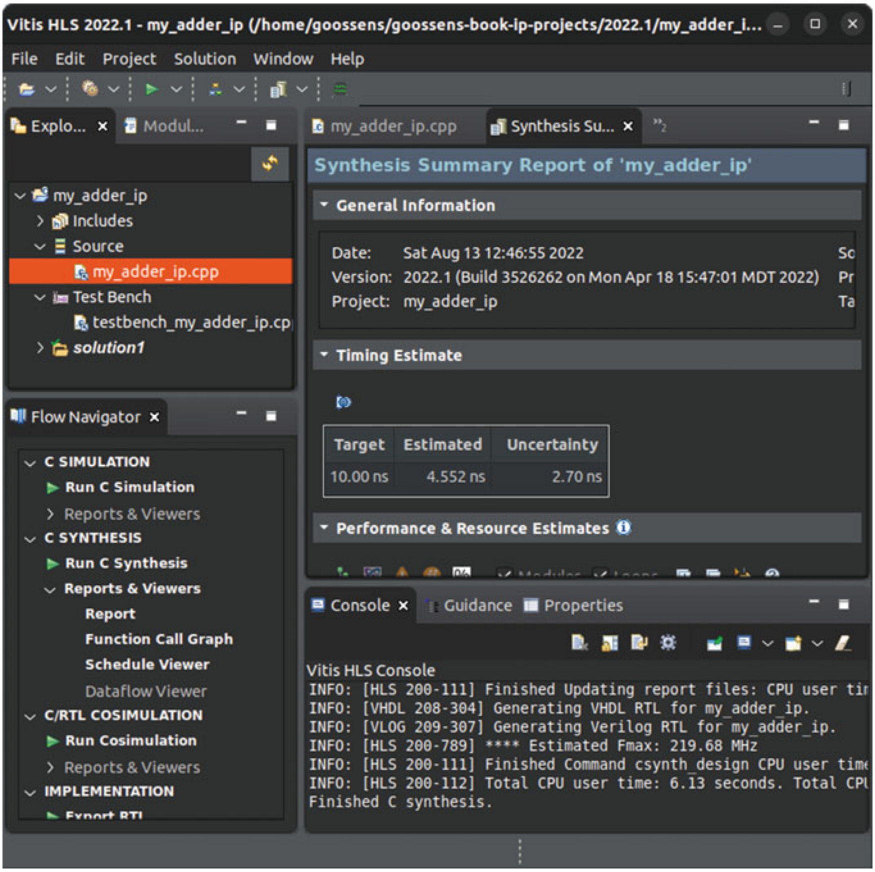
To program your FPGA, you need to generate a bitstream file from your exported RTL. This should be done in another tool named Vivado.

To start the Vivado GUI, type the commands shown in Listing 2.14 in a terminal (they can be found in the start_vivado.txt file in the chapter_2 folder).

Listing 2.14 Start the Vivado GUI tool

```
$ cd /opt/Xilinx/Vitis_HLS/2022.1
$ source settings64.sh
$ cd $HOME/goossens-book-ip-projects/2022.1
$ vivado&
...
$
```

A new window opens (see Fig. 2.39), in which you select **Quick Start/Create Project**.



Synthesis Summary Report of 'my_adder_ip'

General Information

Date: Sat Aug 13 12:46:55 2022
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: my_adder_ip

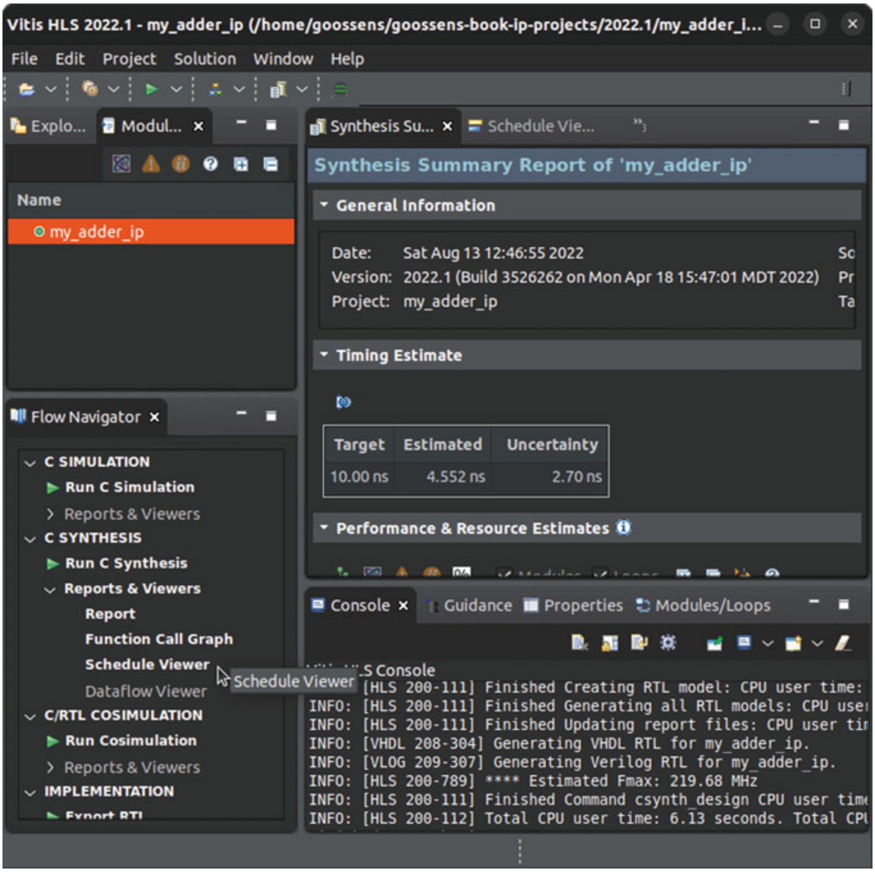
Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	4.552 ns	2.70 ns

Performance & Resource Estimates

```
Vitis HLS Console
INFO: [HLS 200-111] Finished Updating report files: CPU user ti
INFO: [VHDL 208-304] Generating VHDL RTL for my_adder_ip.
INFO: [VLOG 209-307] Generating Verilog RTL for my_adder_ip.
INFO: [HLS 200-789] **** Estimated Fmax: 219.68 MHz
INFO: [HLS 200-111] Finished Command csynth_design CPU user time
INFO: [HLS 200-112] Total CPU user time: 6.13 seconds. Total CP
Finished C synthesis.
```

Fig. 2.23 The C synthesis report



Synthesis Summary Report of 'my_adder_ip'

General Information

Date: Sat Aug 13 12:46:55 2022
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: my_adder_ip

Timing Estimate

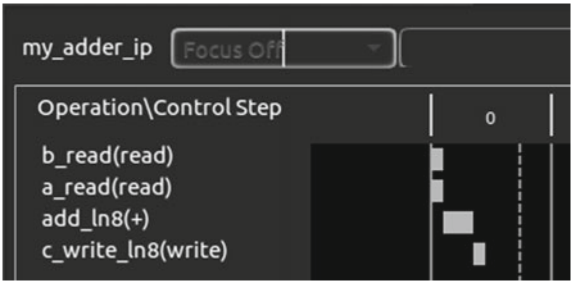
Target	Estimated	Uncertainty
10.00 ns	4.552 ns	2.70 ns

Performance & Resource Estimates

```
Vitis HLS Console
       [HLS 200-111] Finished Creating RTL model: CPU user time:
INFO: [HLS 200-111] Finished Generating all RTL models: CPU use
INFO: [HLS 200-111] Finished Updating report files: CPU user ti
INFO: [VHDL 208-304] Generating VHDL RTL for my_adder_ip.
INFO: [VLOG 209-307] Generating Verilog RTL for my_adder_ip.
INFO: [HLS 200-789] **** Estimated Fmax: 219.68 MHz
INFO: [HLS 200-111] Finished Command csynth_design CPU user time
INFO: [HLS 200-112] Total CPU user time: 6.13 seconds. Total CPU
```

Fig. 2.24 Opening the schedule viewer

Fig. 2.25 Adder IP computation progression



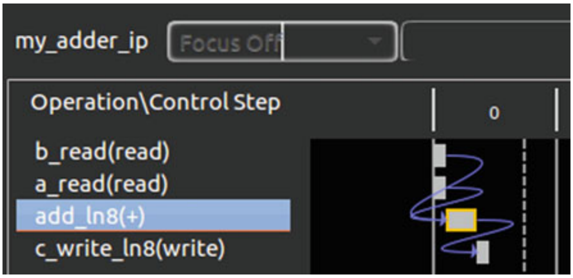
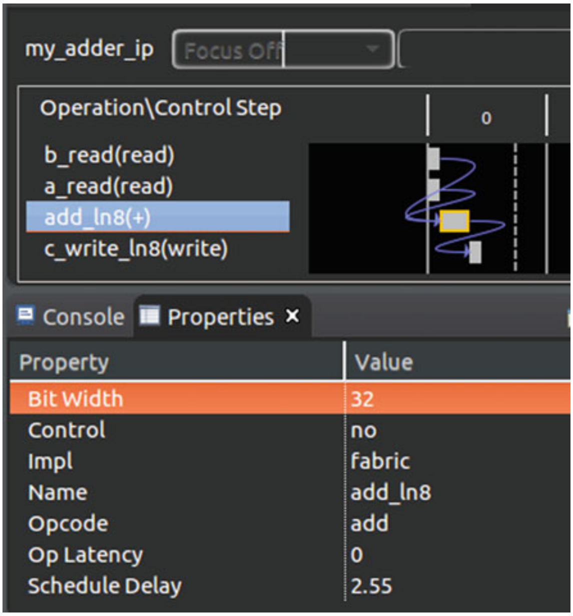
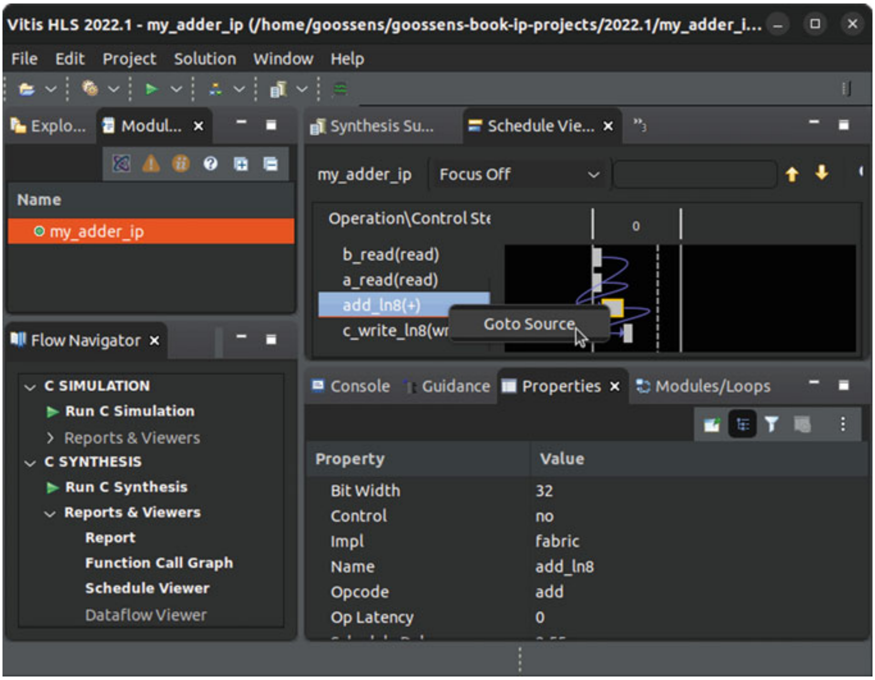


Fig. 2.26 Dependencies



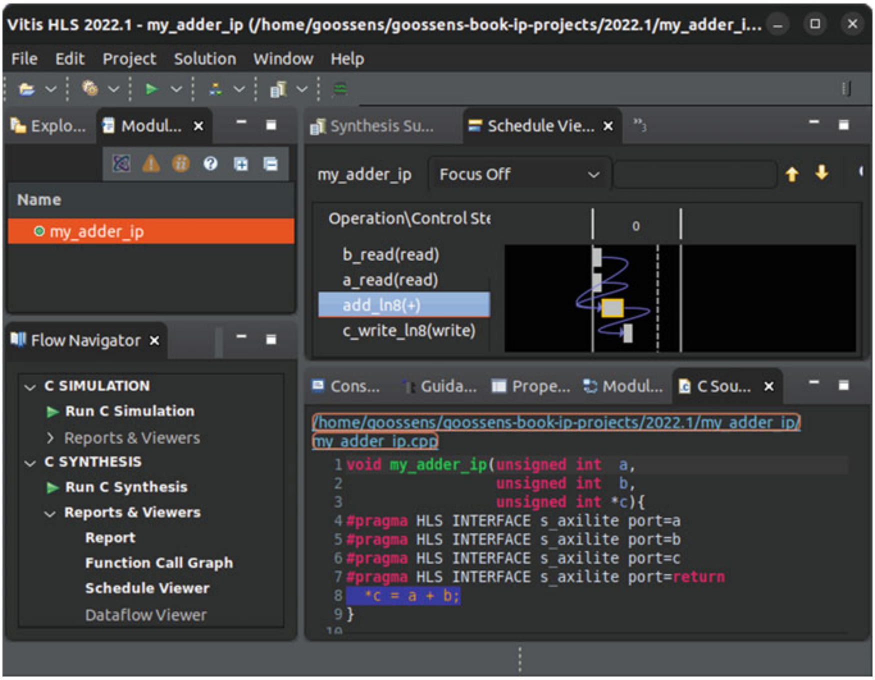
Property	Value
Bit Width	32
Control	no
Impl	fabric
Name	add_ln8
Opcode	add
Op Latency	0
Schedule Delay	2.55

Fig. 2.27 Signal properties



Property	Value
Bit Width	32
Control	no
Impl	fabric
Name	add_ln8
Opcode	add
Op Latency	0

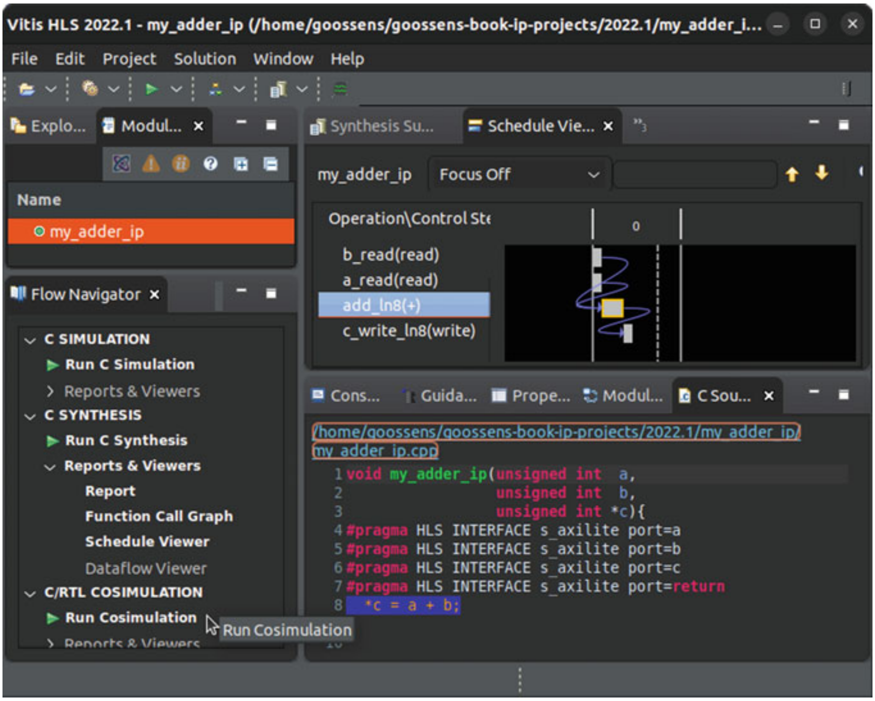
Fig. 2.28 Selecting **Goto Source**



/home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip/my_adder_ip.cpp

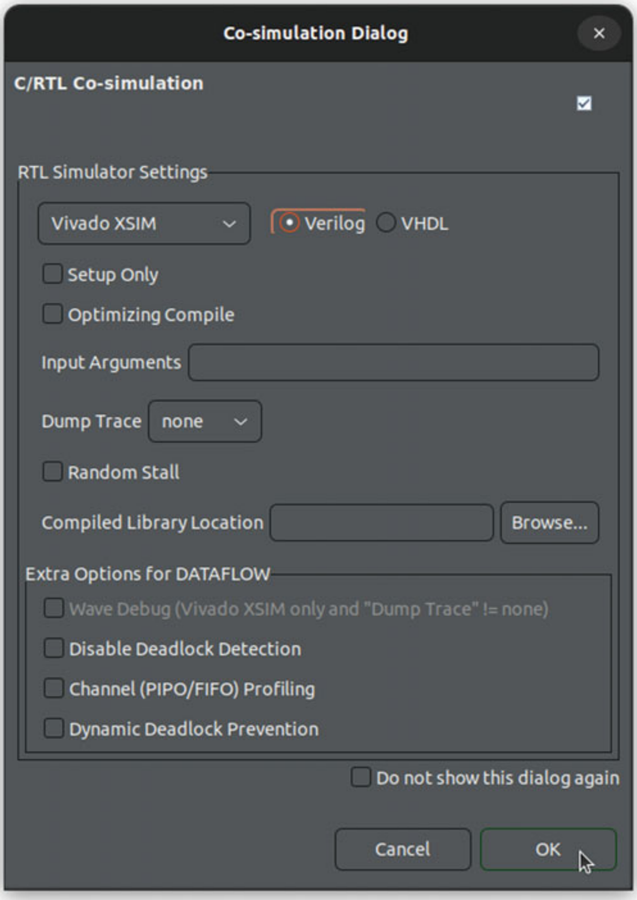
```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c){
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
#pragma HLS INTERFACE s_axilite port=return
  *c = a + b;
}
```

Fig. 2.29 The C source file which the add_ln8(+) comes from



```
void my_adder_ip(unsigned int  a,
                 unsigned int  b,
                 unsigned int *c){
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
#pragma HLS INTERFACE s_axilite port=return
  *c = a + b;
```

Fig. 2.30 Starting the cosimulation



Co-simulation Dialog

C/RTL Co-simulation ☑

RTL Simulator Settings

Vivado XSIM ● Verilog ○ VHDL

□ Setup Only

□ Optimizing Compile

Input Arguments

Dump Trace: none

□ Random Stall

Compiled Library Location

Extra Options for DATAFLOW

□ Wave Debug (Vivado XSIM only and "Dump Trace" != none)

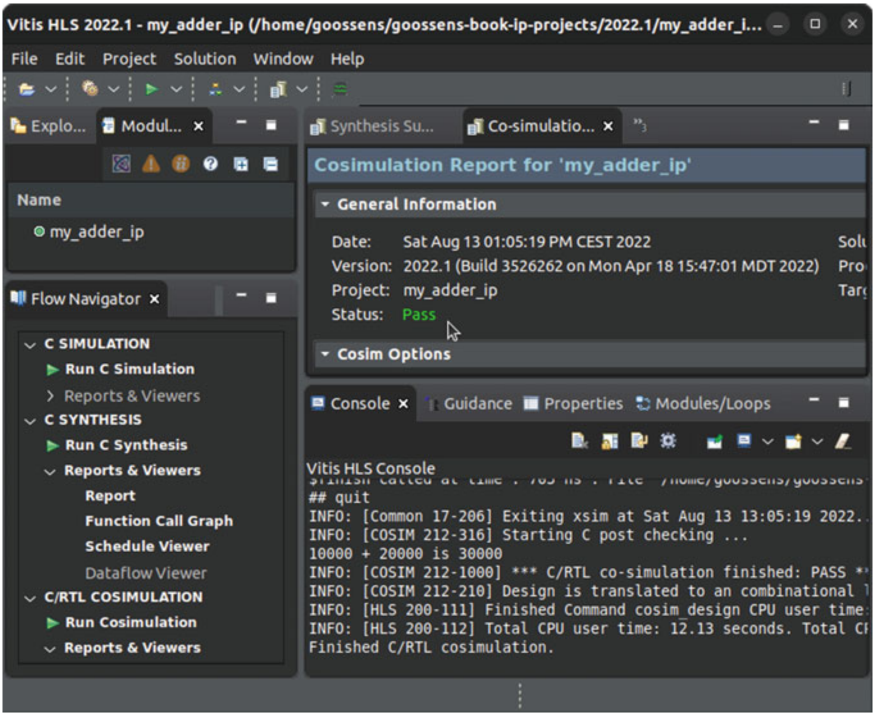
□ Disable Deadlock Detection

□ Channel (PIPO/FIFO) Profiling

□ Dynamic Deadlock Prevention

□ Do not show this dialog again

Fig. 2.31 Co-simulation C/RTL dialog box



Cosimulation Report for 'my_adder_ip'

General Information

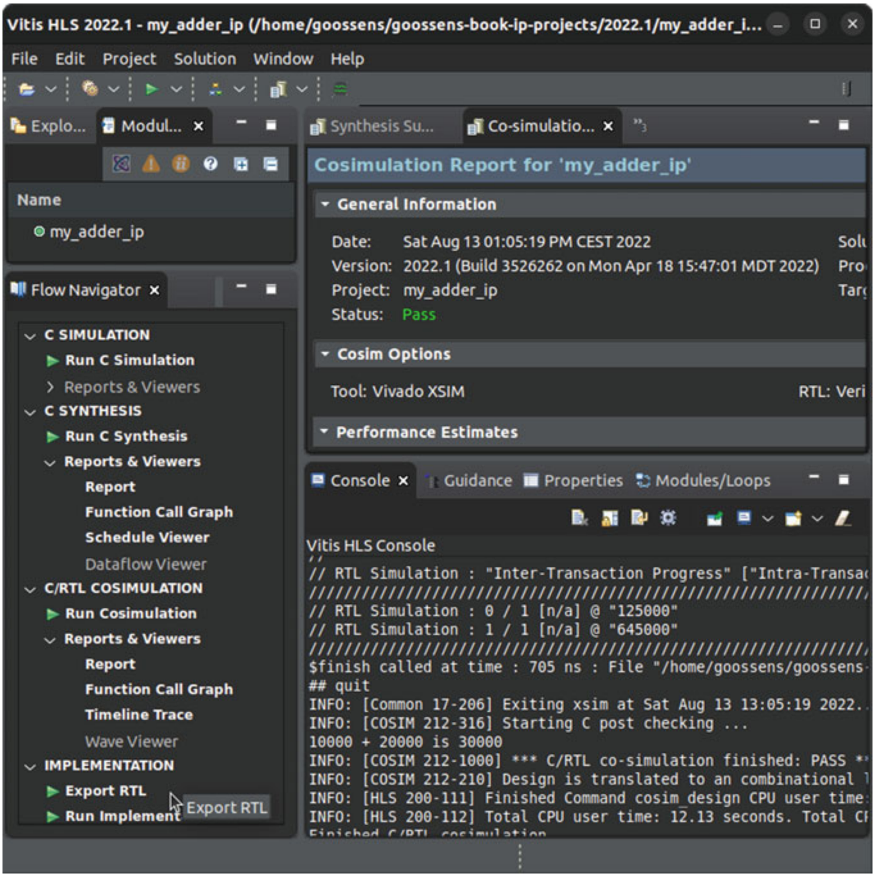
Date: Sat Aug 13 01:05:19 PM CEST 2022
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: my_adder_ip
Status: Pass

Solu
Pro
Tar

Cosim Options

```
$finish called at time : 705 ns : File "/home/goossens/goossens
## quit
INFO: [Common 17-206] Exiting xsim at Sat Aug 13 13:05:19 2022..
INFO: [COSIM 212-316] Starting C post checking ...
10000 + 20000 is 30000
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS **
INFO: [COSIM 212-210] Design is translated to an combinational l
INFO: [HLS 200-111] Finished Command cosim_design CPU user time:
INFO: [HLS 200-112] Total CPU user time: 12.13 seconds. Total CP
Finished C/RTL cosimulation.
```

Fig. 2.32 Co-simulation report



Cosimulation Report for 'my_adder_ip'

General Information

Date: Sat Aug 13 01:05:19 PM CEST 2022
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: my_adder_ip
Status: Pass

Solu
Pro
Tar

Cosim Options

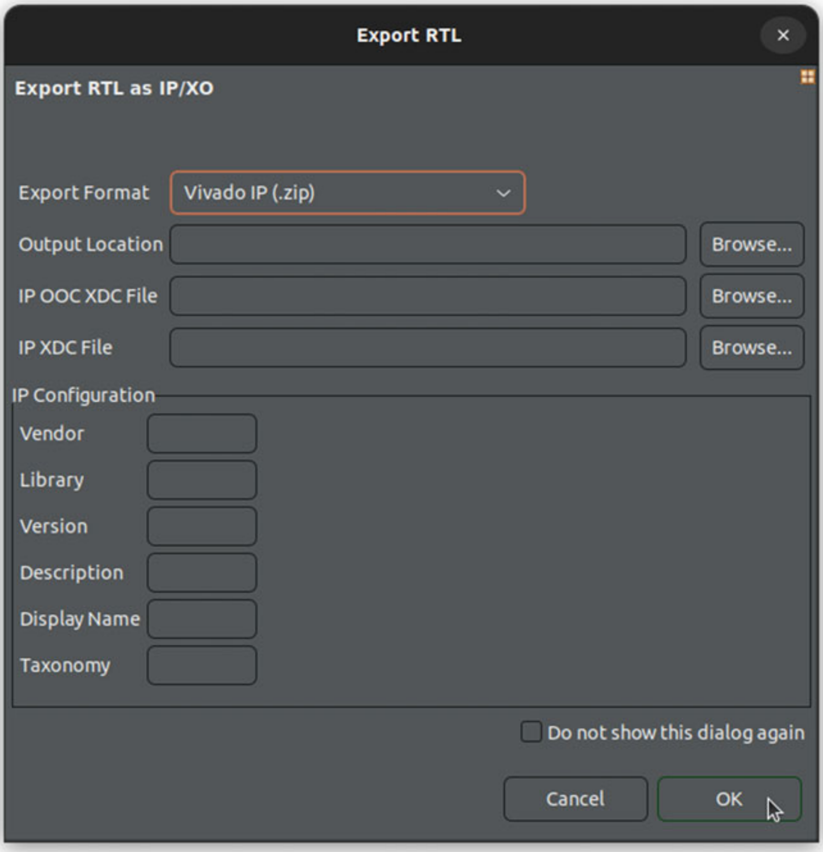
Tool: Vivado XSIM
RTL: Veri

Performance Estimates

Vitis HLS Console

```
//
// RTL Simulation : "Inter-Transaction Progress" ["Intra-Transac
/////////////////////////////////////////////////////////////////
// RTL Simulation : 0 / 1 [n/a] @ "125000"
// RTL Simulation : 1 / 1 [n/a] @ "645000"
/////////////////////////////////////////////////////////////////
$finish called at time : 705 ns : File "/home/goossens/goossens-
## quit
INFO: [Common 17-206] Exiting xsim at Sat Aug 13 13:05:19 2022..
INFO: [COSIM 212-316] Starting C post checking ...
10000 + 20000 is 30000
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS **
INFO: [COSIM 212-210] Design is translated to an combinational l
INFO: [HLS 200-111] Finished Command cosim_design CPU user time:
INFO: [HLS 200-112] Total CPU user time: 12.13 seconds. Total CP
Finished C/RTL cosimulation
```

Fig. 2.33 Start export



Export RTL

Export RTL as IP/XO

Export Format: Vivado IP (.zip)

Output Location:

IP OOC XDC File:

IP XDC File:

IP Configuration

Vendor:

Library:

Version:

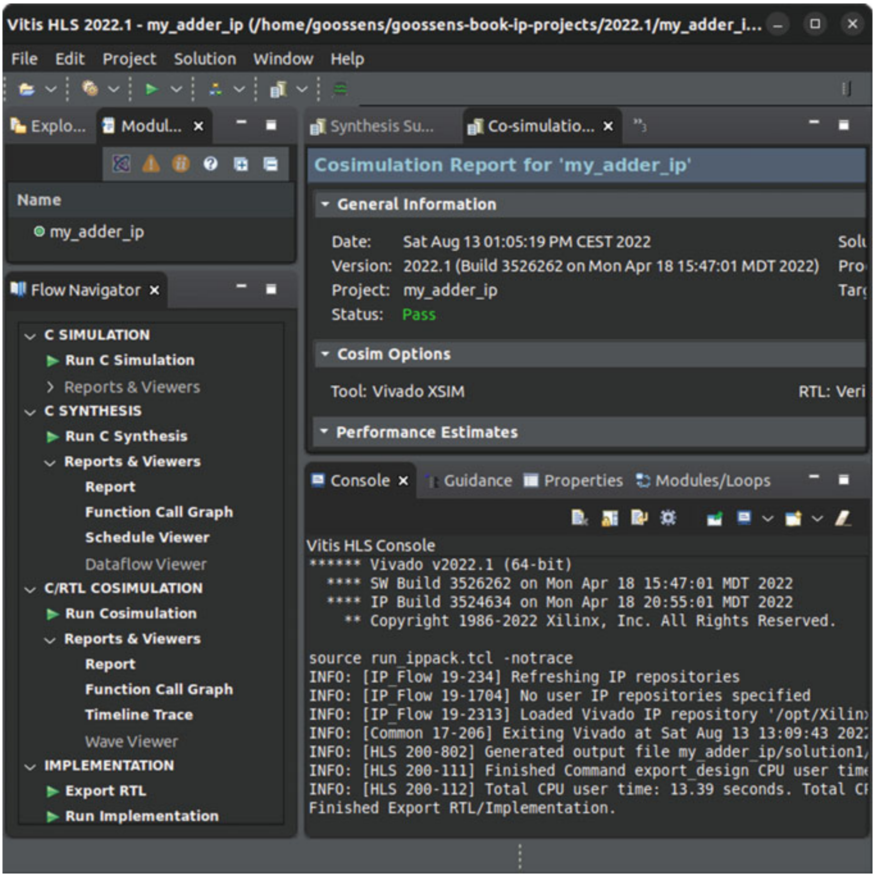
Description:

Display Name:

Taxonomy:

□ Do not show this dialog again

Fig. 2.34 Export RTL dialog box



Cosimulation Report for 'my_adder_ip'

General Information

Date: Sat Aug 13 01:05:19 PM CEST 2022 Solu

Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022) Pro

Project: my_adder_ip Targ

Status: Pass

Cosim Options

Tool: Vivado XSIM RTL: Veri

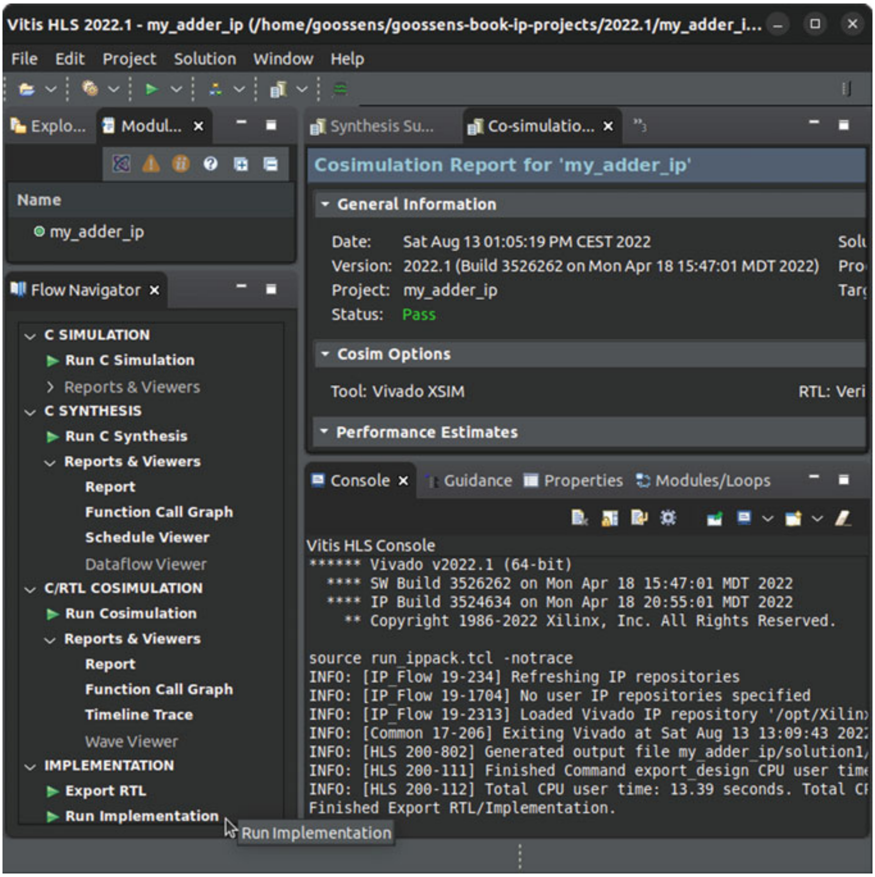
Performance Estimates

Vitis HLS Console

```
****** Vivado v2022.1 (64-bit)
  **** SW Build 3526262 on Mon Apr 18 15:47:01 MDT 2022
  **** IP Build 3524634 on Mon Apr 18 20:55:01 MDT 2022
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/opt/Xilinx
INFO: [Common 17-206] Exiting Vivado at Sat Aug 13 13:09:43 2022
INFO: [HLS 200-802] Generated output file my_adder_ip/solution1/
INFO: [HLS 200-111] Finished Command export_design CPU user time
INFO: [HLS 200-112] Total CPU user time: 13.39 seconds. Total CF
Finished Export RTL/Implementation.
```

Fig. 2.35 Console output after export



Cosimulation Report for 'my_adder_ip'

General Information

Date: Sat Aug 13 01:05:19 PM CEST 2022 Solı

Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022) Pro

Project: my_adder_ip Tar

Status: Pass

Cosim Options

Tool: Vivado XSIM RTL: Veri

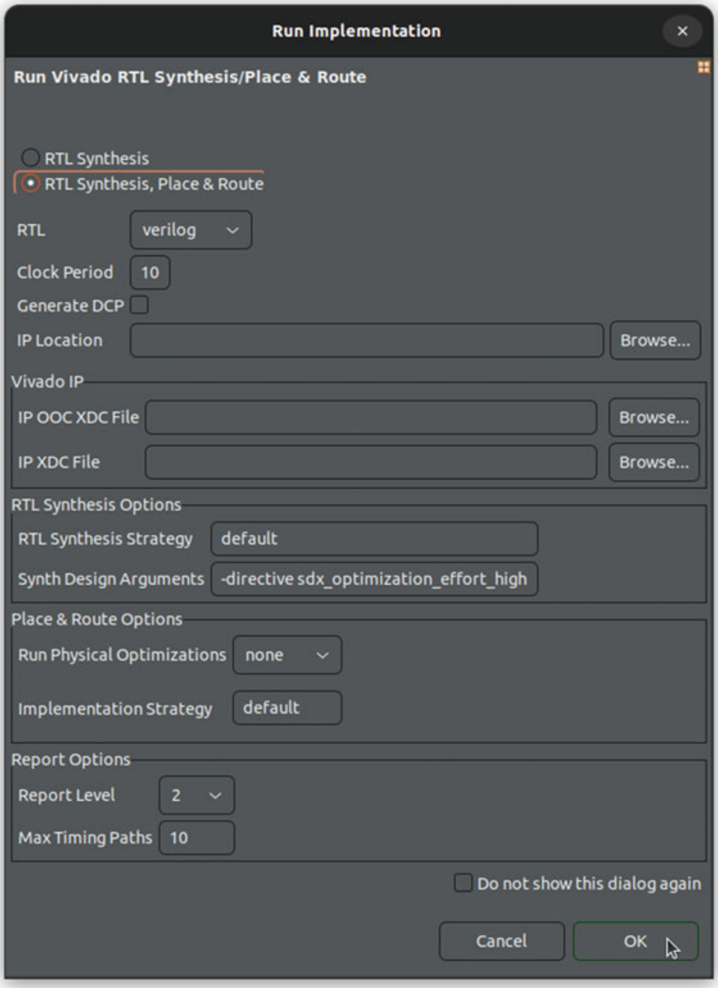
Performance Estimates

Vitis HLS Console

```
****** Vivado v2022.1 (64-bit)
  **** SW Build 3526262 on Mon Apr 18 15:47:01 MDT 2022
  **** IP Build 3524634 on Mon Apr 18 20:55:01 MDT 2022
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/opt/Xilinx
INFO: [Common 17-206] Exiting Vivado at Sat Aug 13 13:09:43 2022
INFO: [HLS 200-802] Generated output file my_adder_ip/solution1/
INFO: [HLS 200-111] Finished Command export_design CPU user time
INFO: [HLS 200-112] Total CPU user time: 13.39 seconds. Total CP
Finished Export RTL/Implementation.
```

Fig. 2.36 Running the exported RTL



Run Implementation

Run Vivado RTL Synthesis/Place & Route

○ RTL Synthesis

● RTL Synthesis, Place & Route

RTL: verilog

Clock Period: 10

Generate DCP □

IP Location:

Vivado IP

IP OOC XDC File:

IP XDC File:

RTL Synthesis Options

RTL Synthesis Strategy: default

Synth Design Arguments: -directive sdx_optimization_effort_high

Place & Route Options

Run Physical Optimizations: none

Implementation Strategy: default

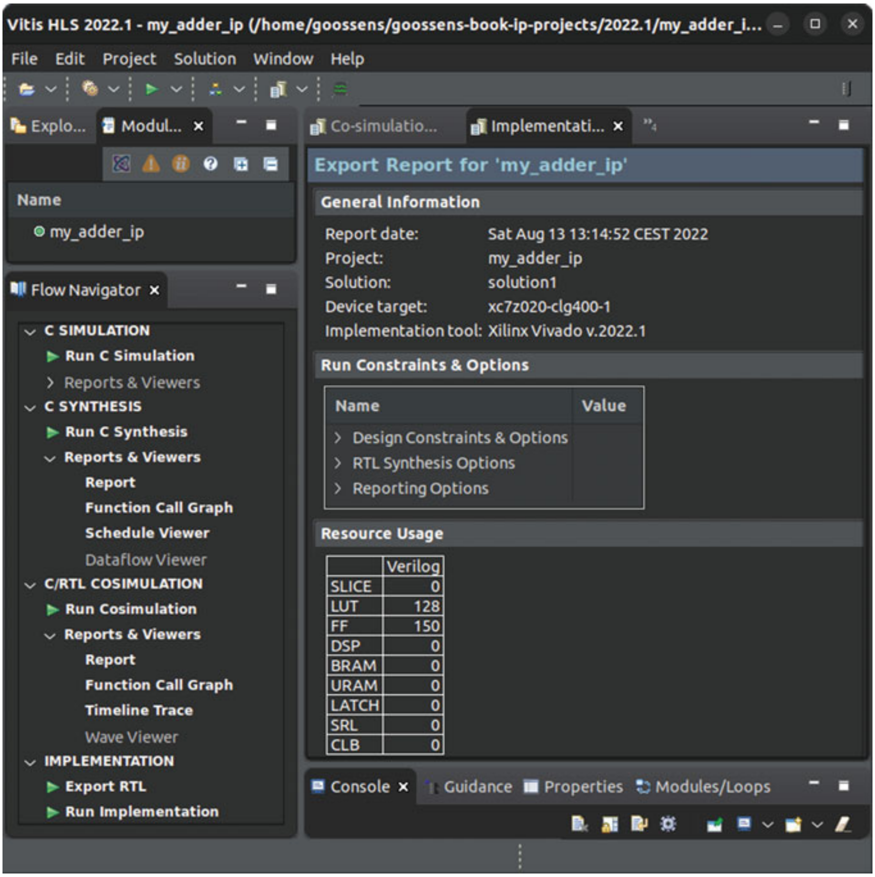
Report Options

Report Level: 2

Max Timing Paths: 10

□ Do not show this dialog again

Fig. 2.37 The implementation run dialog box



Export Report for 'my_adder_ip'

General Information

Report date: Sat Aug 13 13:14:52 CEST 2022
Project: my_adder_ip
Solution: solution1
Device target: xc7z020-clg400-1
Implementation tool: Xilinx Vivado v.2022.1

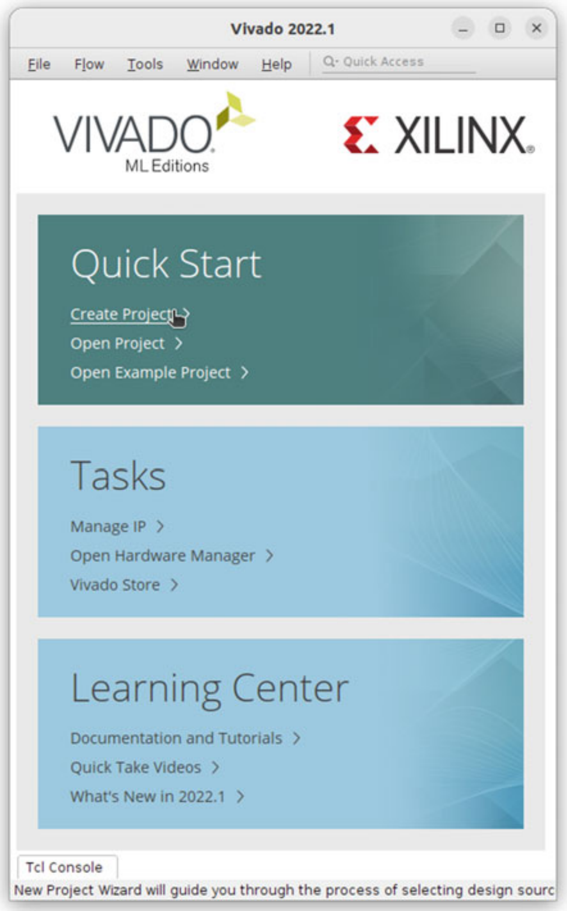
Run Constraints & Options

Name	Value
› Design Constraints & Options	
› RTL Synthesis Options	
› Reporting Options	

Resource Usage

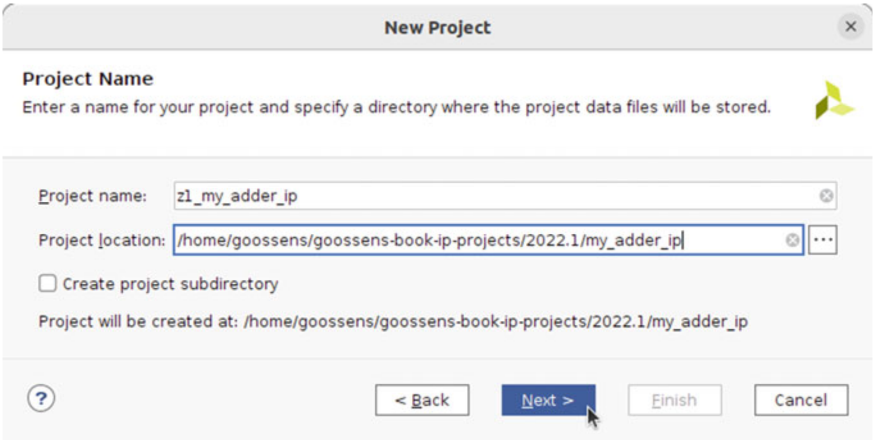
	Verilog
SLICE	0
LUT	128
FF	150
DSP	0
BRAM	0
URAM	0
LATCH	0
SRL	0
CLB	0

Fig. 2.38 The run implementation report



New Project Wizard will guide you through the process of selecting design sourc

Fig. 2.39 The Vivado **Quick Start** page


New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: z1_my_adder_ip

Project location: /home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip

□ Create project subdirectory

Project will be created at: /home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip

Fig. 2.40 The Vivado **Project Name** page (set name and place)

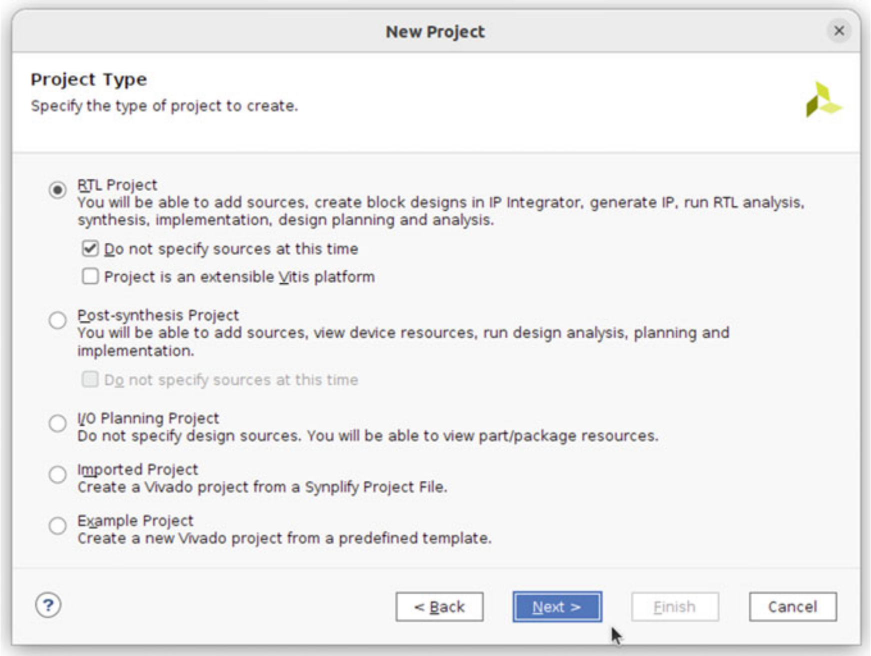
In the **New Project/Create a New Vivado Project** page, click on **Next**. In the **New Project/Project Name** page, name your project (e.g. replace project_1 by z1_my_adder_ip; I always name my Vivado projects with the name of the Vitis_HLS project prefixed with the name of the target development board). Choose my_adder_ip as the hosting folder (see Fig. 2.40). Uncheck the Create project subdirectory box. Click on **Next**.

In the next page (**Project Type**, see Fig. 2.41), make sure the **Do not specify sources at this time** box is checked and click on **Next**.

In the **Default Part** page (see Fig. 2.42), click on the **Boards** tab.

In the **Search** box, type z1 (or z2, or basys3). Select the board (see Fig. 2.43) and click on **Next**.

In the **New Project Summary** page, check that the Pynq-Z1 board (or Pynq-Z2, or Basys3) is mentioned as the **Default Board** (otherwise, return **Back** to the previous page and select the board as described above). Click on **Finish** (see Fig. 2.44).



New Project

Project Type

Specify the type of project to create.

● RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

☑ Do not specify sources at this time

□ Project is an extensible Vitis platform

○ Post-synthesis Project
You will be able to add sources, view device resources, run design analysis, planning and implementation.

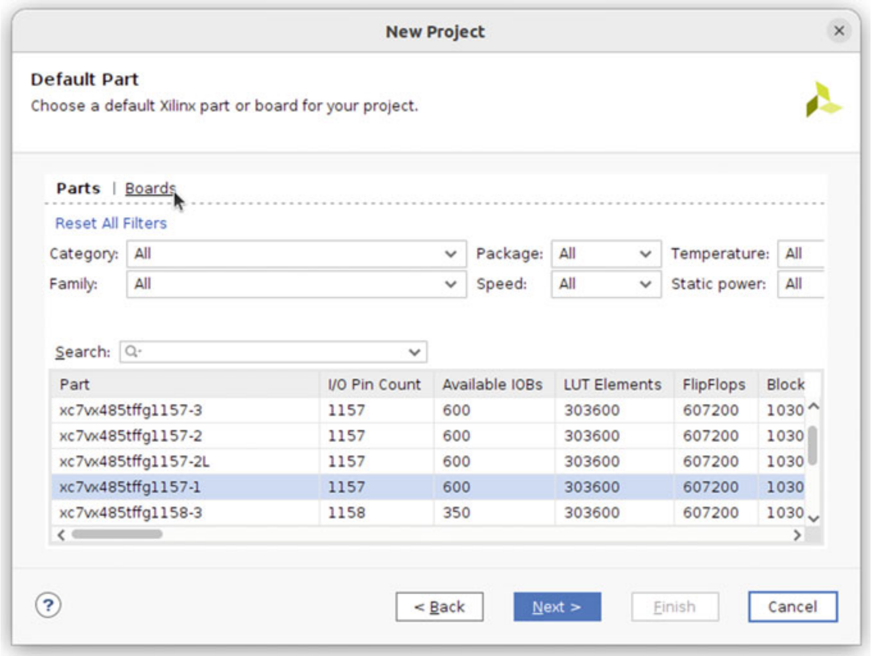
□ Do not specify sources at this time

○ I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

○ Imported Project
Create a Vivado project from a Synplify Project File.

○ Example Project
Create a new Vivado project from a predefined template.

Fig. 2.41 The Vivado **Project Type** page



New Project

Default Part

Choose a default Xilinx part or board for your project.

Parts | Boards

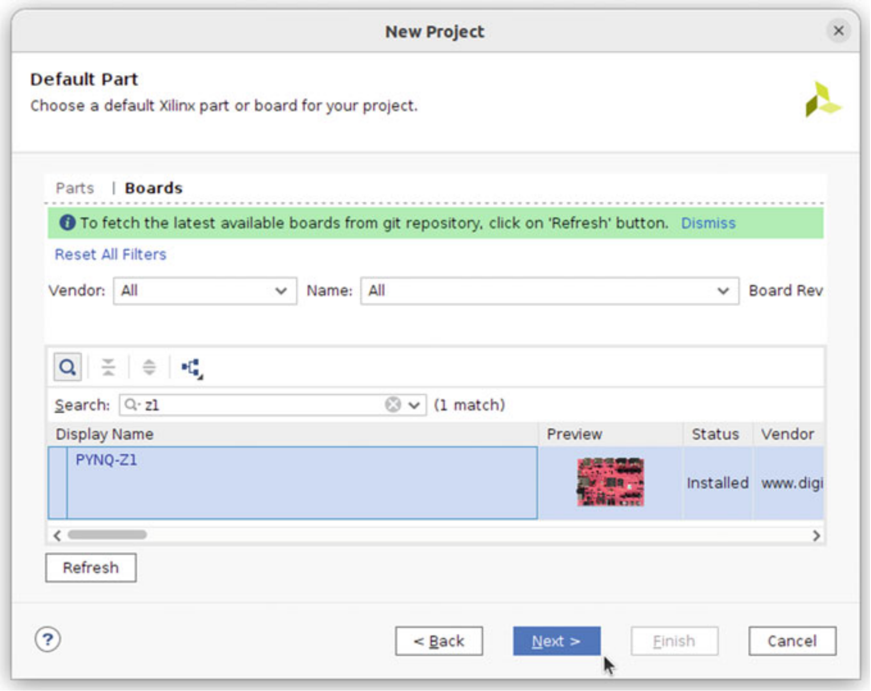
Reset All Filters

Category: All
Package: All
Temperature: All
Family: All
Speed: All
Static power: All

Search:

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block
xc7vx485tffg1157-3	1157	600	303600	607200	1030
xc7vx485tffg1157-2	1157	600	303600	607200	1030
xc7vx485tffg1157-2L	1157	600	303600	607200	1030
xc7vx485tffg1157-1	1157	600	303600	607200	1030
xc7vx485tffg1158-3	1158	350	303600	607200	1030

Fig. 2.42 The Vivado **Default Part** page



New Project

Default Part

Choose a default Xilinx part or board for your project.

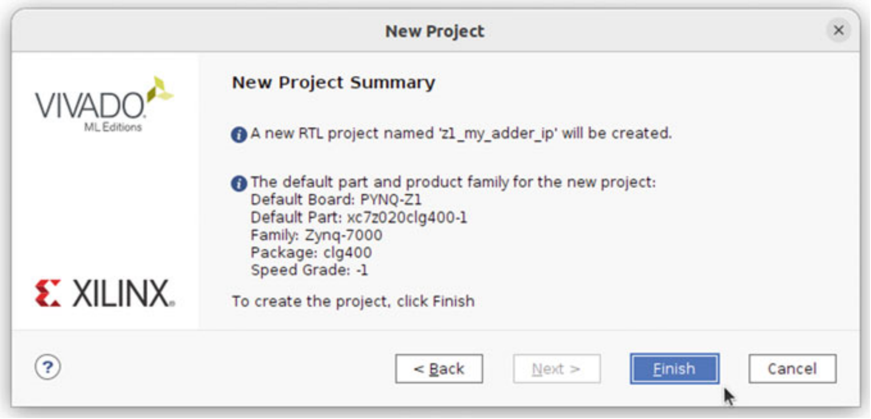
To fetch the latest available boards from git repository, click on 'Refresh' button. Dismiss

Vendor: All   Name: All   Board Rev

Search: z1 (1 match)

Display Name	Preview	Status	Vendor
PYNQ-Z1		Installed	www.digi

Fig. 2.43 The Pynq-Z1 board selection



New Project

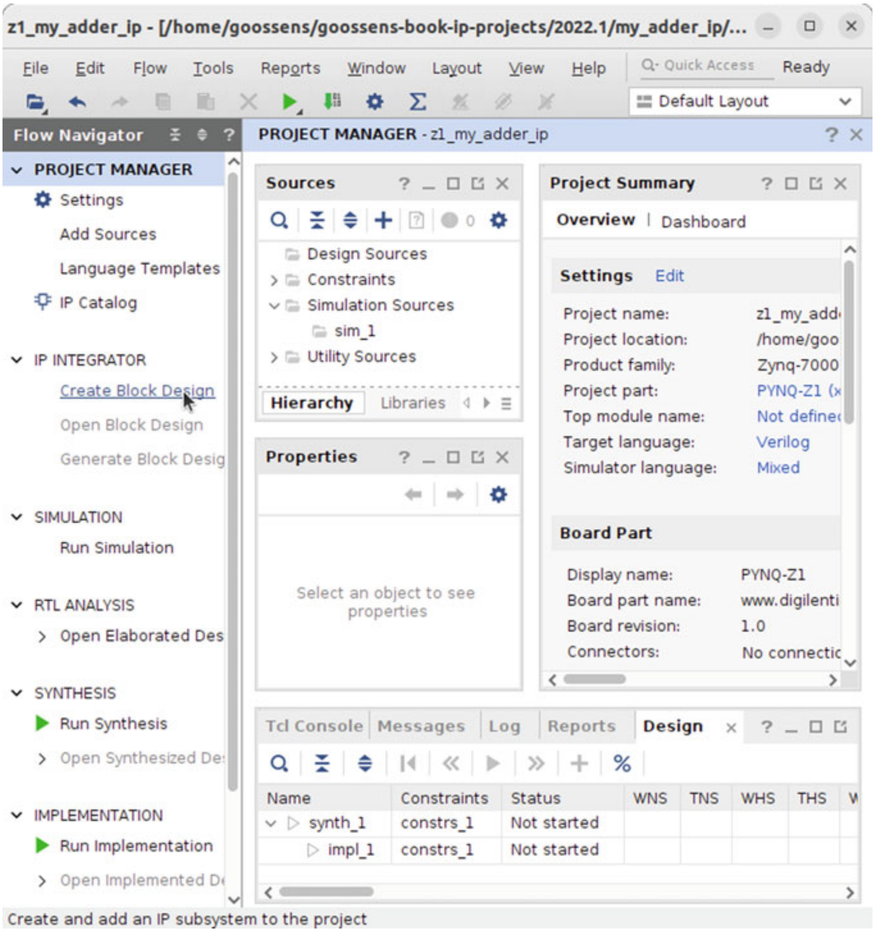
New Project Summary

A new RTL project named 'z1_my_adder_ip' will be created.

The default part and product family for the new project:
Default Board: PYNQ-Z1
Default Part: xc7z020clg400-1
Family: Zynq-7000
Package: clg400
Speed Grade: -1

To create the project, click Finish

Fig. 2.44 The Vivado **New Project Summary** page



z1_my_adder_ip - [/home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip/...

PROJECT MANAGER - z1_my_adder_ip

Settings Edit

Project name:	z1_my_add
Project location:	/home/goo
Product family:	Zynq-7000
Project part:	PYNQ-Z1 (x
Top module name:	Not define
Target language:	Verilog
Simulator language:	Mixed

Board Part

Display name:	PYNQ-Z1
Board part name:	www.digilenti
Board revision:	1.0
Connectors:	No connectic

Name	Constraints	Status	WNS	TNS	WHS	THS	W
synth_1	constrs_1	Not started					
impl_1	constrs_1	Not started					

Create and add an IP subsystem to the project

Fig. 2.45 The Vivado **z1_my_adder_ip** project page

The **z1_my_adder_ip** project page opens (see Fig. 2.45).

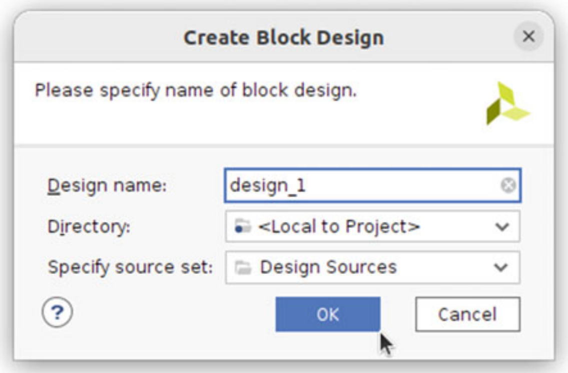
In the left panel, select **Create Block Design**.

In the dialog box shown in Fig. 2.46, name your design (or keep it as **design_1**; this is what I do) and click on **OK**.

An empty block design opens (see Fig. 2.47).

The following (until further notice) applies to the Zynq based boards (e.g. Pynq-Z1, Pynq-Z2, Zedboard, Zybo) (for the Artix-7 based boards (e.g. Basys3 or Nexys4), jump to page 57).

On the right, you can see a **Diagram** frame, with a "+" button (see the position of the pointer in Fig. 2.47). Click on it.



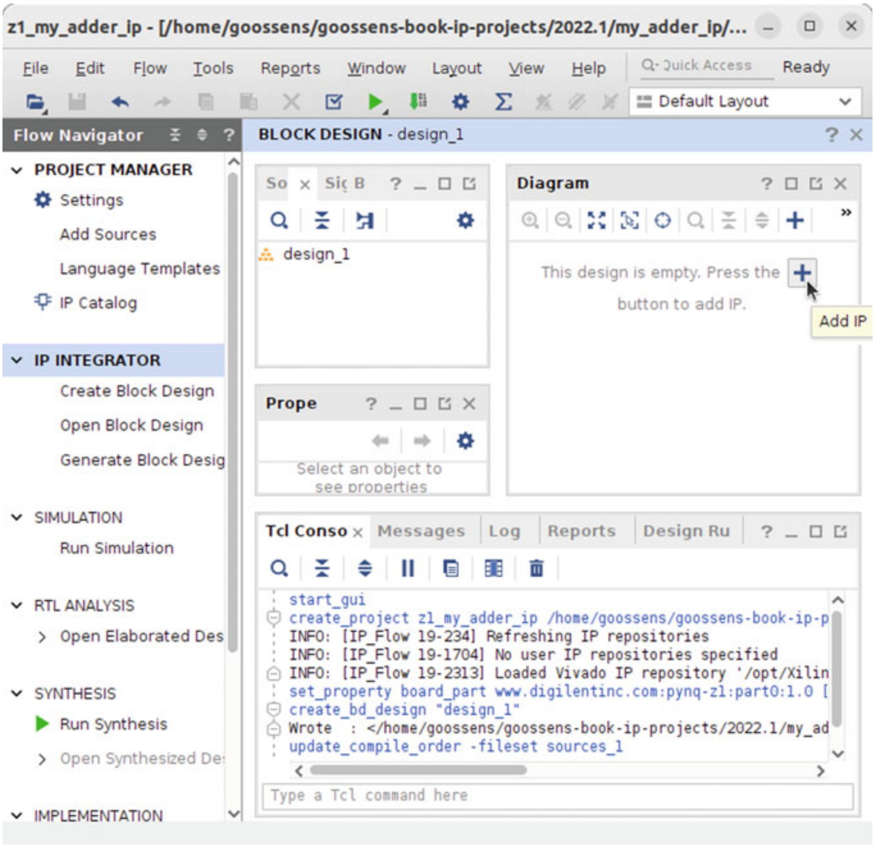
Please specify name of block design.

Design name: design_1

Directory: <Local to Project>

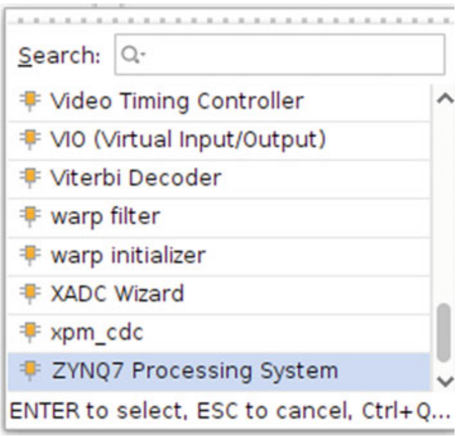
Specify source set: Design Sources

Fig. 2.46 Creating a block design



```
start_gui
create_project z1_my_adder_ip /home/goossens/goossens-book-ip-p
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories specified
INFO: [IP_Flow 19-2313] Loaded Vivado IP repository '/opt/Xilin
set_property board_part www.digilentinc.com:pynq-z1:part0:1.0 [
create_bd_design "design_1"
Wrote  : </home/goossens/goossens-book-ip-projects/2022.1/my_ad
update_compile_order -fileset sources_1
```

Fig. 2.47 An empty block design

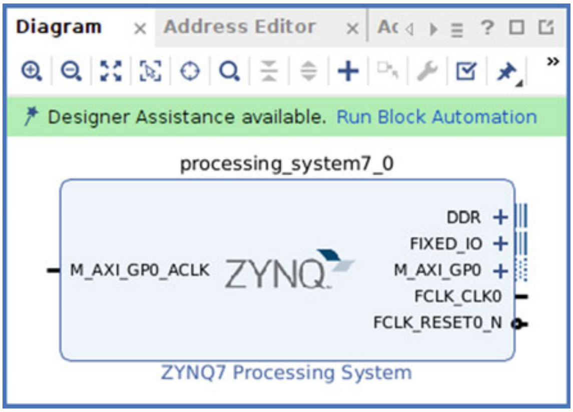


Search:

- Video Timing Controller
- VIO (Virtual Input/Output)
- Viterbi Decoder
- warp filter
- warp initializer
- XADC Wizard
- xpm_cdc
- ZYNQ7 Processing System

ENTER to select, ESC to cancel, Ctrl+Q...

Fig. 2.48 Selecting the ZYNQ7 processing system IP



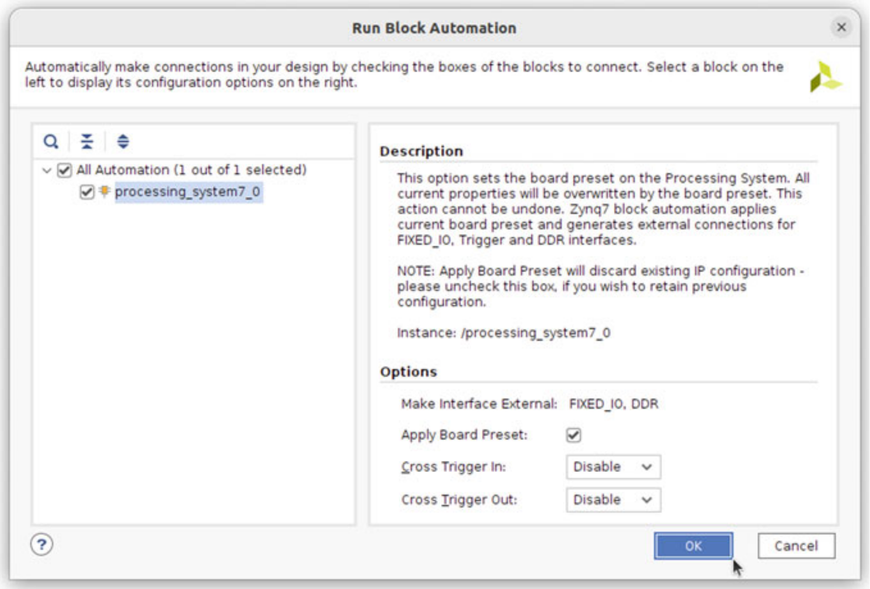
Designer Assistance available. Run Block Automation

Fig. 2.49 Adding the ZYNQ7 Processing System IP to the block design

Scroll all the way down the proposed list and select **ZYNQ7 Processing System** (see Fig. 2.48).

A ZYNQ component is placed in the center of the **Diagram** frame (see Fig. 2.49). This component will match an equivalent feature in the Zynq-based development board FPGA. This component is used to interface your adder IP with the embedded ARM processor (the ARM processor itself interfaces the ZYNQ7 Processing System IP with the host computer).

A green line on top of the **Diagram** frame proposes to **Run Block Automation**. This is to connect your component to the board environment. Click on it. In the dialog box (see Fig. 2.50), keep the settings unchanged and click on **OK**.



Run Block Automation

Automatically make connections in your design by checking the boxes of the blocks to connect. Select a block on the left to display its configuration options on the right.

☑ All Automation (1 out of 1 selected)

☑ processing_system7_0

Description

This option sets the board preset on the Processing System. All current properties will be overwritten by the board preset. This action cannot be undone. Zynq7 block automation applies current board preset and generates external connections for FIXED_IO, Trigger and DDR interfaces.

NOTE: Apply Board Preset will discard existing IP configuration - please uncheck this box, if you wish to retain previous configuration.

Instance: /processing_system7_0

Options

Make Interface External: FIXED_IO, DDR

Apply Board Preset: ☑

Cross Trigger In: Disable

Cross Trigger Out: Disable

Fig. 2.50 Connecting the ZYNQ7 Processing System IP to the development board environment

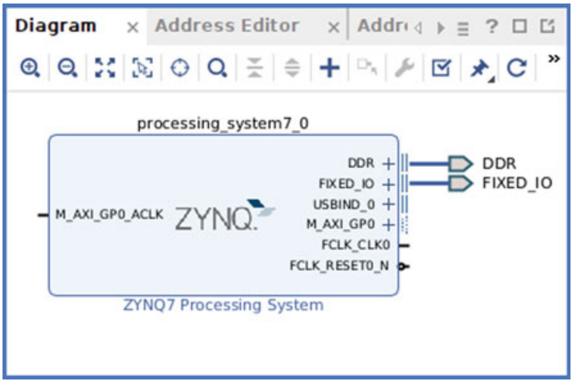
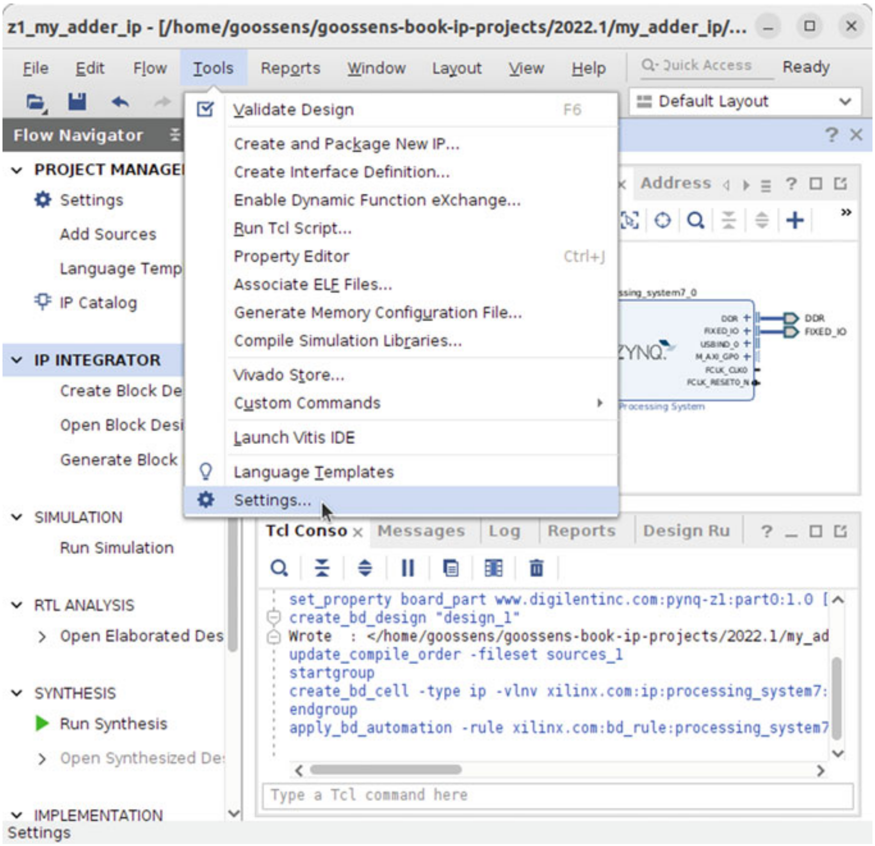


Fig. 2.51 The ZYNQ7 Processing System IP connected to pads of the development board environment



```
set_property board_part www.digilentinc.com:pynq-z1:part0:1.0 [
create_bd_design "design_1"
Wrote  : </home/goossens/goossens-book-ip-projects/2022.1/my_ad
update_compile_order -fileset sources_1
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:
endgroup
apply_bd_automation -rule xilinx.com:bd_rule:processing_system7
```

Fig. 2.52 Selecting the **Settings** tool

The diagram frame shows the Zynq7 Processing System IP connected to some output pads (see Fig. 2.51).

The following applies (until further notice) to the Artix-7 based boards (e.g. Basys3 or Nexys4). For Zynq based boards, jump to the next paragraph.

For the Artix-7 based boards, add the microblaze IP instead of the Zynq7 Processing System IP. After **Run Block Automation**, in **Options/Debug Module**, select **Debug & UART**. After **Run Connection Automation**, select the **diff_clk_rtl** pad in the diagram and delete it. In the **BLOCK DESIGN** frame, **Board** tab, double-click on **System Clock** to have it connected in your design. You can then continue as for the other boards.

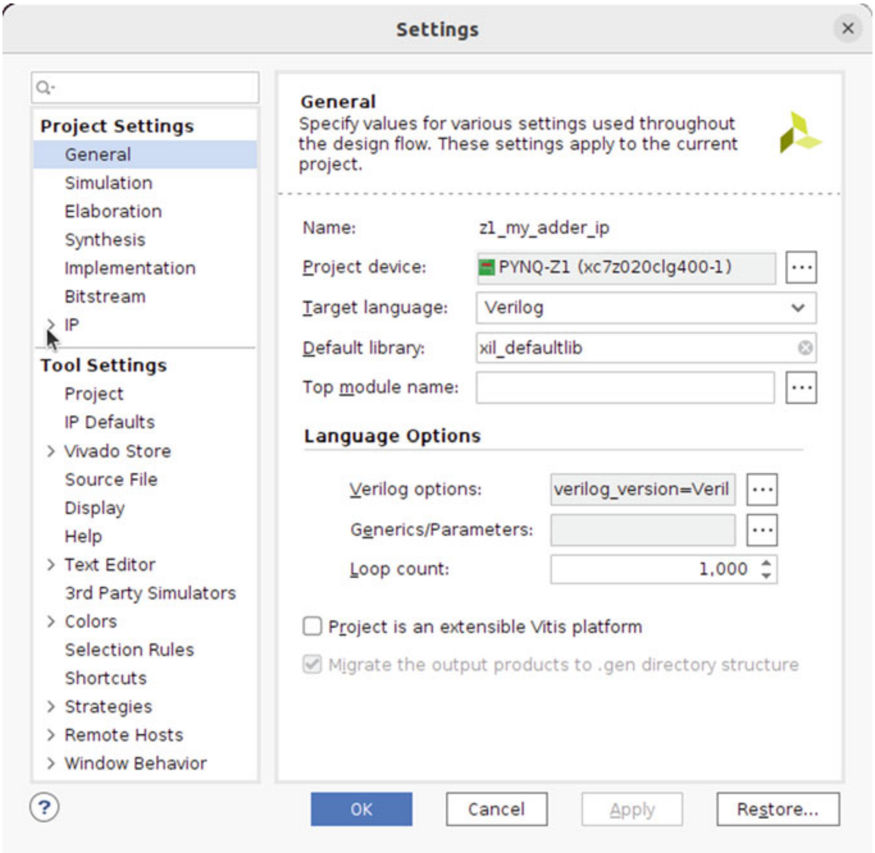
The following applies (until further notice) to all the boards.

In the upper line menu, select **Tools/Settings** (see Fig. 2.52).

In the **Project Settings** frame (see Fig. 2.53), expand the **IP** entry.

Click on **Repository** (see Fig. 2.54).

In the right frame, click on the "+" button (see Figs. 2.54 and 2.55).



General

Specify values for various settings used throughout the design flow. These settings apply to the current project.

Name: z1_my_adder_ip

Project device: PYNQ-Z1 (xc7z020clg400-1)

Target language: Verilog

Default library: xil_defaultlib

Top module name:

Language Options

Verilog options: verilog_version=Veril

Generics/Parameters:

Loop count: 1,000

□ Project is an extensible Vitis platform

☑ Migrate the output products to .gen directory structure

Fig. 2.53 The **Settings** dialog box

Select the proposed folder, i.e. the one in which you saved your Vitis_HLS my_adder_ip project (see Fig. 2.56).

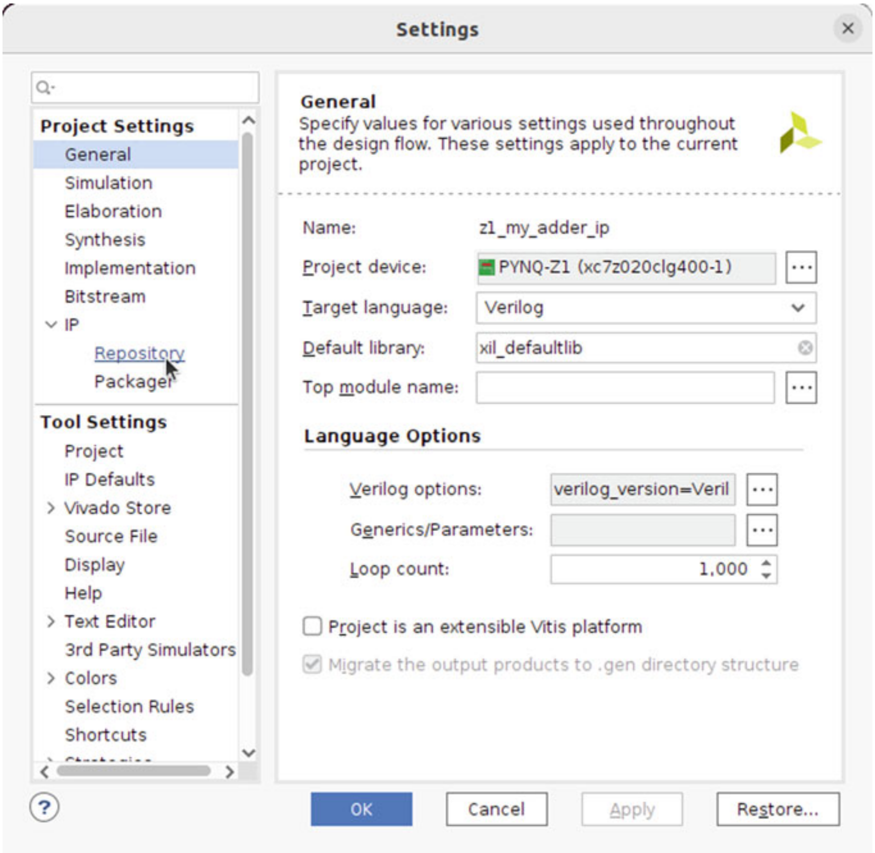
Click on the **Select** button. A message box informs you that an IP repository has been added to the set of IPs available to the Vivado project (see Fig. 2.57).

Click on **OK** and on **OK** again in the **Settings** page.

In the **Diagram** frame, click on the "**+**" button (see Fig. 2.58).

Scroll down and select **My_adder_ip** (this is your adder; see Fig. 2.59).

The **Diagram** frame shows the added My_adder_ip (see Fig. 2.60).



General

Specify values for various settings used throughout the design flow. These settings apply to the current project.

Name: z1_my_adder_ip

Project device: PYNQ-Z1 (xc7z020clg400-1)

Target language: Verilog

Default library: xil_defaultlib

Top module name:

Language Options

Verilog options: verilog_version=Veril

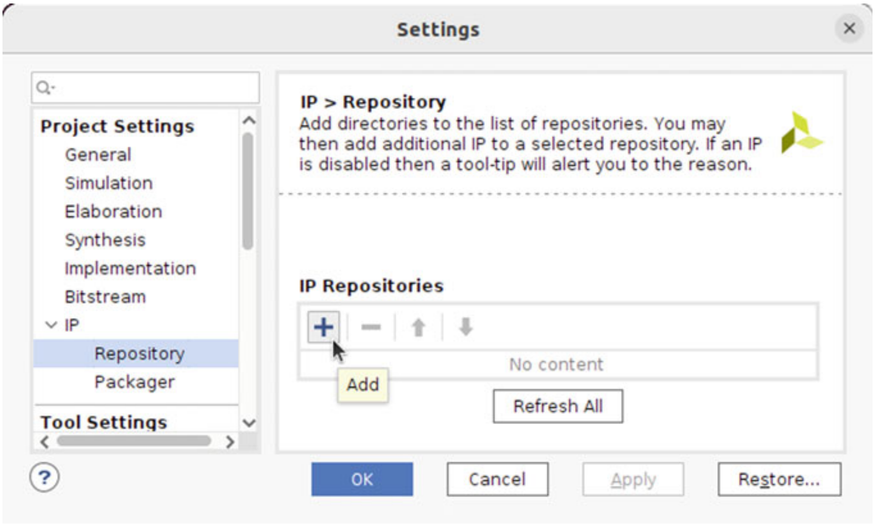
Generics/Parameters:

Loop count: 1,000

☐ Project is an extensible Vitis platform

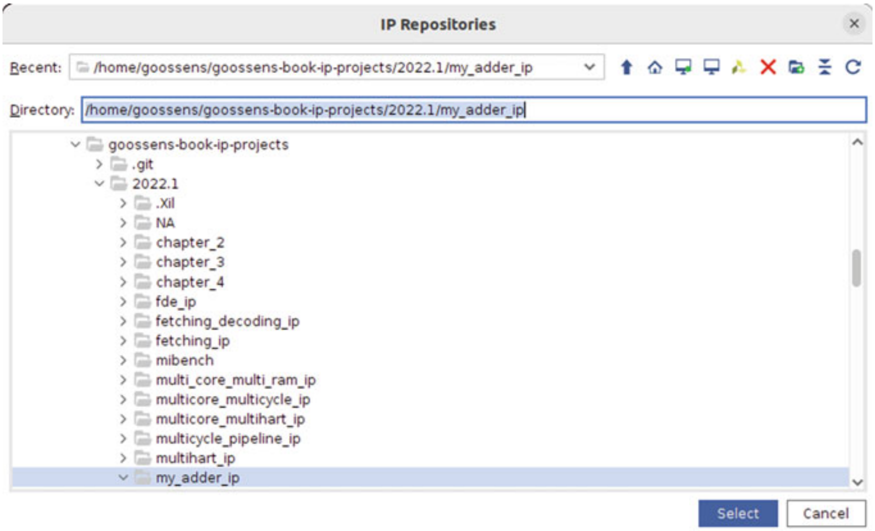
☑ Migrate the output products to .gen directory structure

Fig. 2.54 The **Repository** button



Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

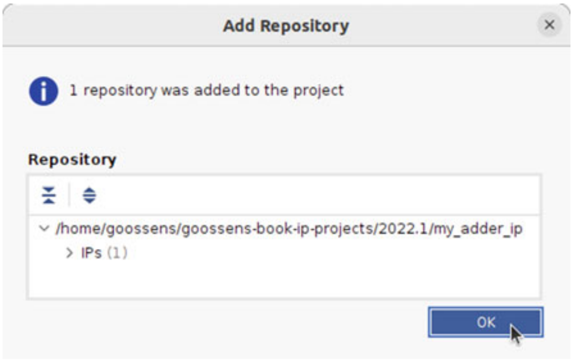
Fig. 2.55 Add a repository



Recent: /home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip

Directory: /home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip

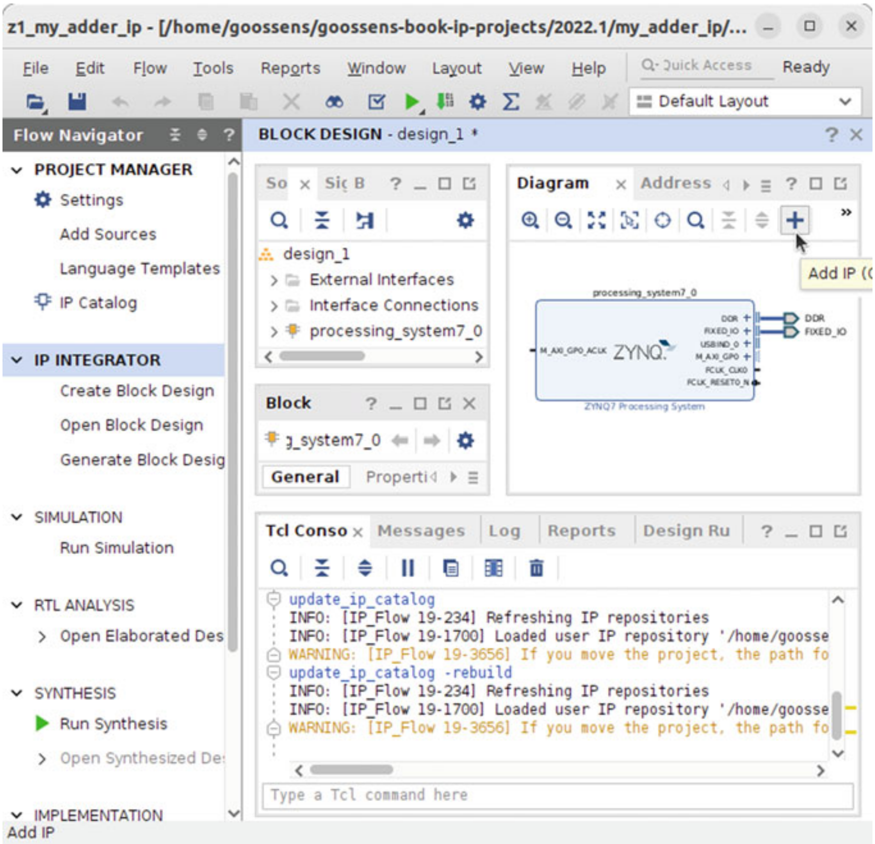
Fig. 2.56 Add the my_adder_ip Vitis_HLS project repository to the IP library



1 repository was added to the project

/home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip
IPs (1)

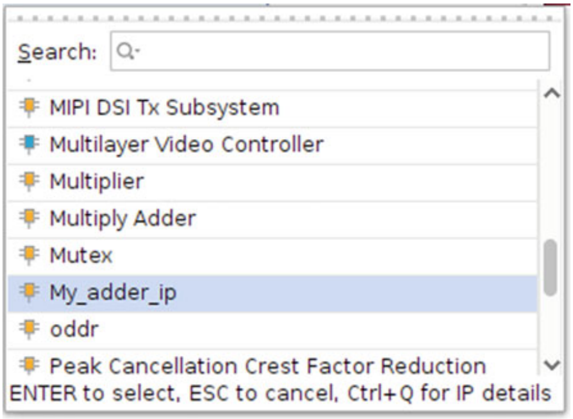
Fig. 2.57 Added repository information message



```
update_ip_catalog
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1700] Loaded user IP repository '/home/goosse
WARNING: [IP_Flow 19-3656] If you move the project, the path fo
update_ip_catalog -rebuild
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1700] Loaded user IP repository '/home/goosse
WARNING: [IP_Flow 19-3656] If you move the project, the path fo
```

Type a Tcl command here

Fig. 2.58 Add an IP to the diagram



Search:

- MIPI DSI Tx Subsystem
- Multilayer Video Controller
- Multiplier
- Multiply Adder
- Mutex
- My_adder_ip
- oddr
- Peak Cancellation Crest Factor Reduction

ENTER to select, ESC to cancel, Ctrl+Q for IP details

Fig. 2.59 Select **My_adder_ip**

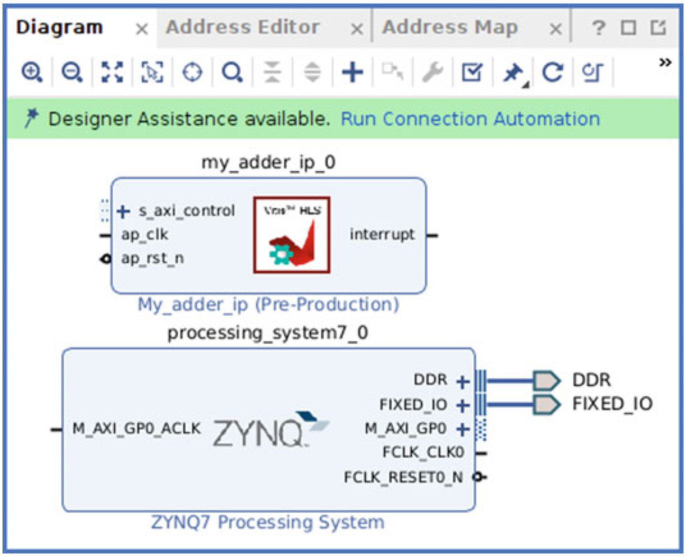


Fig. 2.60 The block design with two IPs

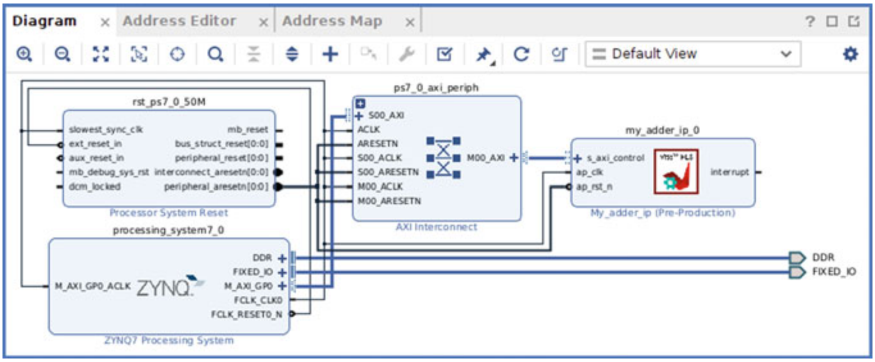
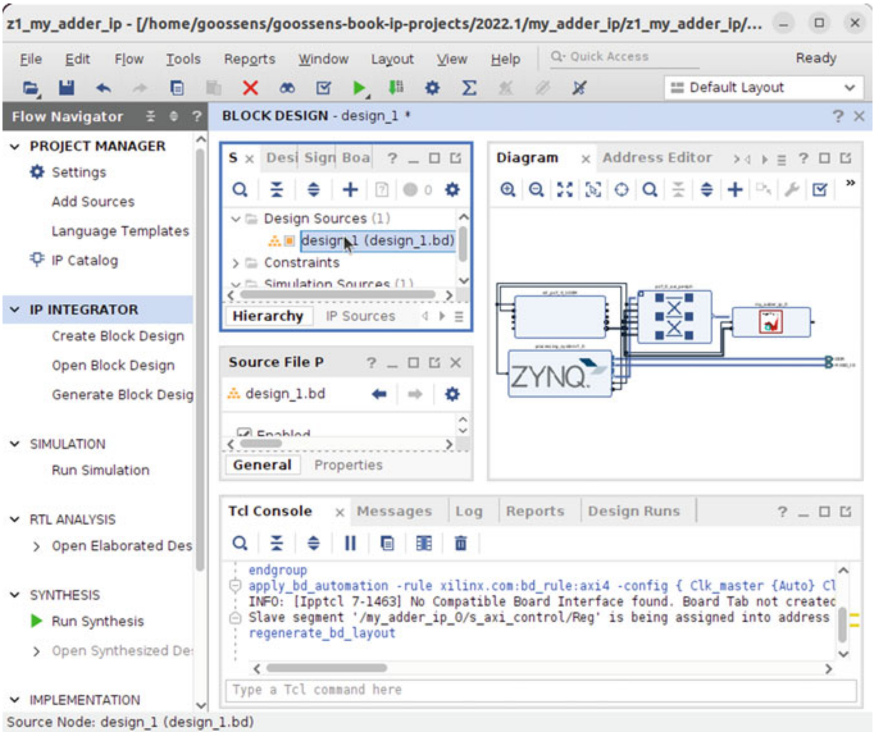


Fig. 2.61 The complete block design after clicking on the **Regenerate Layout** button



```
endgroup
apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config { Clk_master {Auto} Cl
INFO: [Ipptcl 7-1463] No Compatible Board Interface found. Board Tab not createc
Slave segment '/my_adder_ip_0/s_axi_control/Reg' is being assigned into address
regenerate_bd_layout
```

Fig. 2.62 Selecting the design

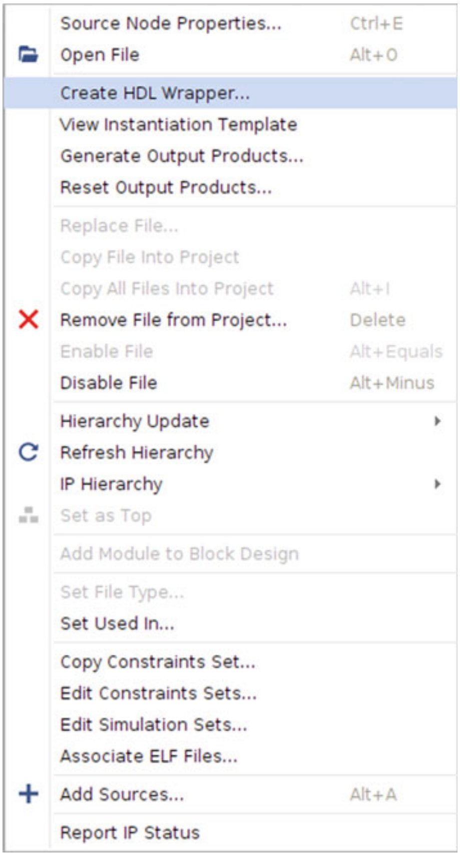


Fig. 2.63 Creating the HDL wrapper

Click on **Run Connection Automation** to let Vivado connect it to the surrounding components. Keep the settings unchanged and click on **OK**.

The **Diagram** frame shows the fully completed block design (see Fig. 2.61 which shows the diagram after clicking on the **Regenerate Layout** button; this button appears at the right of the main button bar of the Diagram frame; it looks like an unclosed loop).

Your design is done.

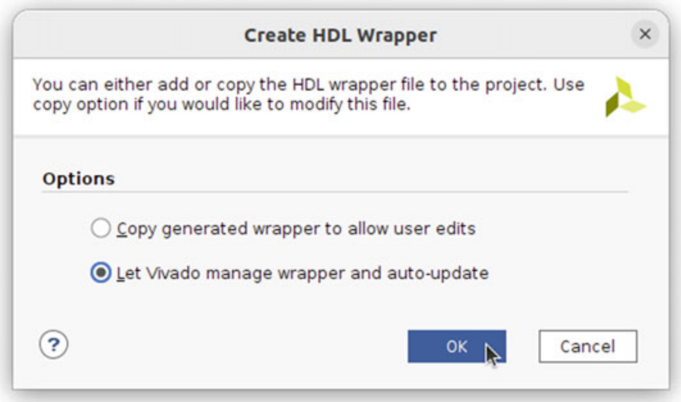
In the central frame, click on the **Sources** tab. Expand **Design Sources** (see Fig. 2.62).

Right-click on the **design_1 (design_1.bd)** line. In the proposed list, choose **Create HDL Wrapper** (see Fig. 2.63).

In the dialog box, click on **OK** (see Fig. 2.64).

Wait until the wrapper appears (**design_1_wrapper** is bolded; see Fig. 2.65).

In the left panel, scroll down to the **Generate Bitstream** button and click on it (see Fig. 2.66).



You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

Options

○ Copy generated wrapper to allow user edits

● Let Vivado manage wrapper and auto-update

Fig. 2.64 Creating the HDL wrapper: confirmation dialog box

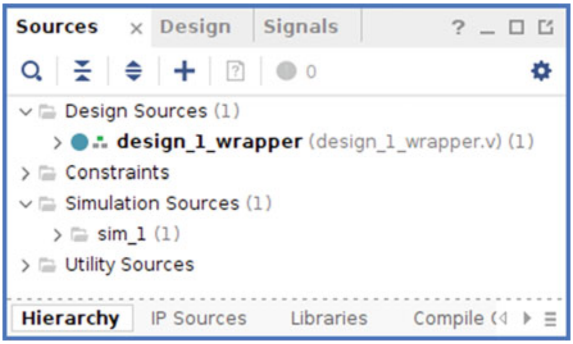


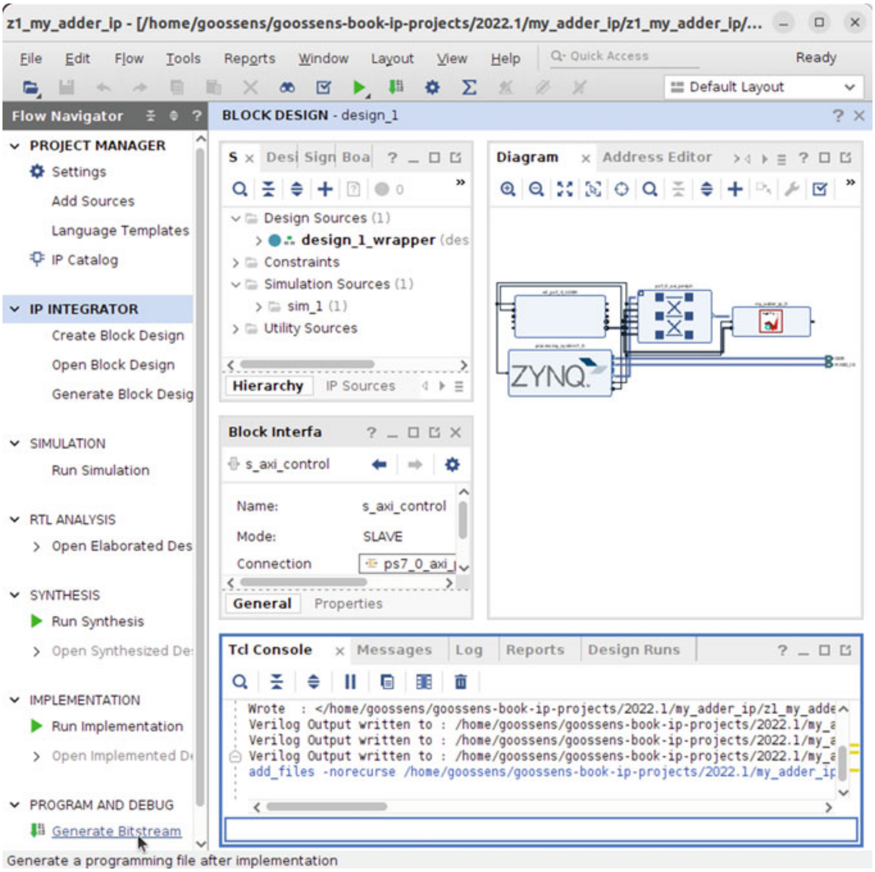
Fig. 2.65 The HDL wrapper

The bitstream generation is a many step process which will successively make the synthesis, implement (i.e. place and route), and generate the bitstream.

The **No Implementation Results Available** warning box pops up (see Fig. 2.67). Click on **Yes**.

The **Launch Runs** window pops up (see Fig. 2.68; the number of jobs may differ: it is related to the number of cores of your computer). Click on **OK**.

You can follow the progression in the upper right corner (see Fig. 2.69).

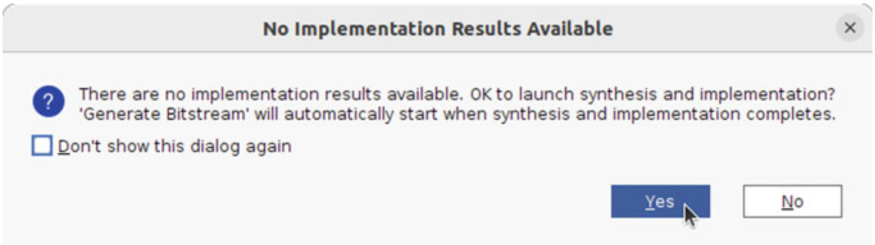


Name: s_axi_control
Mode: SLAVE
Connection ps7_0_axi_

```
Wrote  : </home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip/z1_my_adde
Verilog Output written to : /home/goossens/goossens-book-ip-projects/2022.1/my_a
Verilog Output written to : /home/goossens/goossens-book-ip-projects/2022.1/my_a
Verilog Output written to : /home/goossens/goossens-book-ip-projects/2022.1/my_a
add_files -norecurse /home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip
```

Generate a programming file after implementation

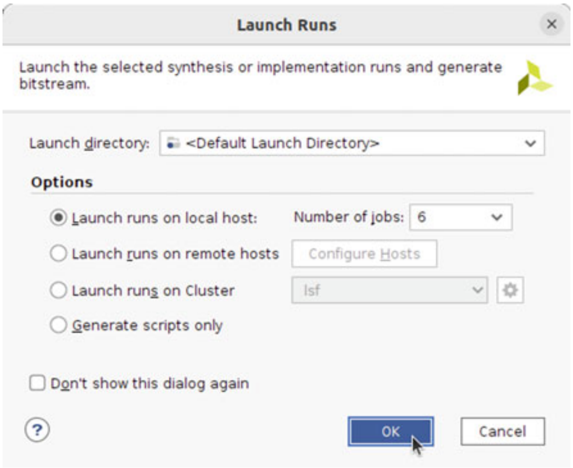
Fig. 2.66 Generating the bitstream


No Implementation Results Available

There are no implementation results available. OK to launch synthesis and implementation? 'Generate Bitstream' will automatically start when synthesis and implementation completes.

□ Don't show this dialog again

Fig. 2.67 The **No Implementation Results Available** warning box



Launch Runs

Launch the selected synthesis or implementation runs and generate bitstream.

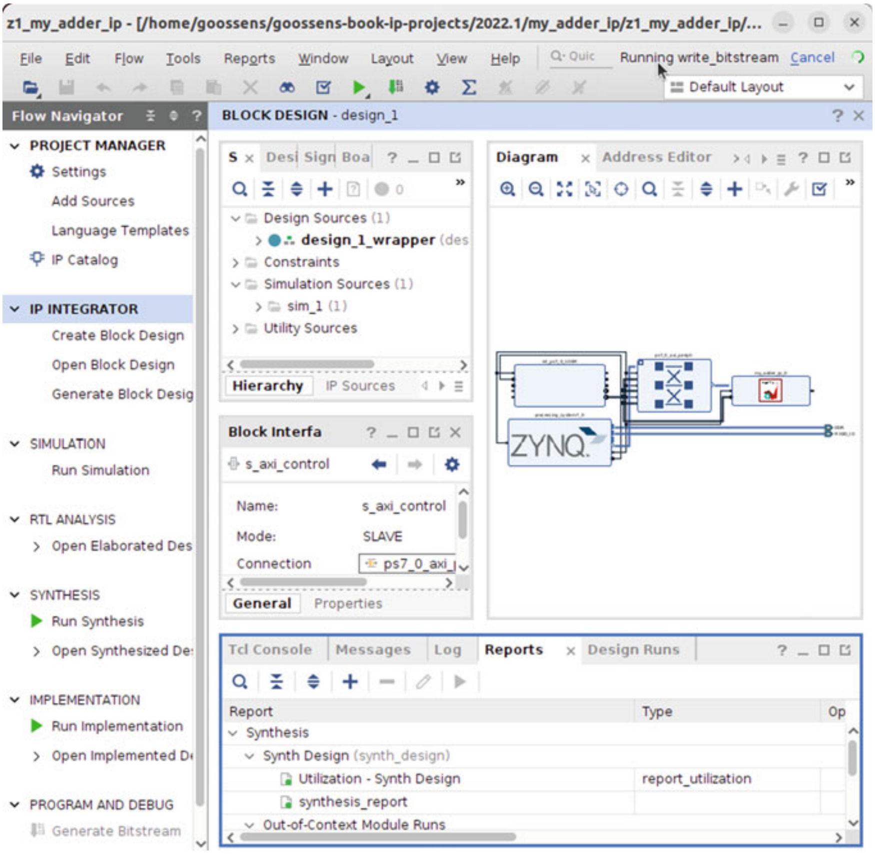
Launch directory: <Default Launch Directory>

Options

- ● Launch runs on local host: Number of jobs: 6
- ○ Launch runs on remote hosts
- ○ Launch runs on Cluster lsf
- ○ Generate scripts only

□ Don't show this dialog again

Fig. 2.68 The **Launch Runs** window



z1_my_adder_ip - [/home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip/z1_my_adder_ip/...

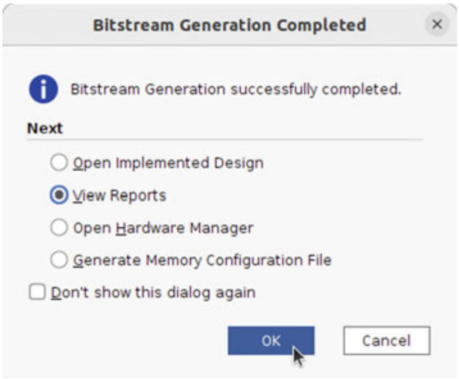
Running write_bitstream Cancel

BLOCK DESIGN - design_1

Name: s_axi_control
Mode: SLAVE
Connection: ps7_0_axi_

Report	Type	Op
Synthesis		
Synth Design (synth_design)		
Utilization - Synth Design	report_utilization	
synthesis_report		
Out-of-Context Module Runs		

Fig. 2.69 Progression of the bitstream generation



Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

○ Open Implemented Design

● View Reports

○ Open Hardware Manager

○ Generate Memory Configuration File

□ Don't show this dialog again

Fig. 2.70 The **Bitstream Generation Completed** information box

Once the generation is done, a dialog box informs you about the success and proposes you a set of possible continuations. Select **View Reports** (see Fig. 2.70).

In the bottom frame, **Reports** tab, scroll down to select the **Implementation/impl_1/Place Design/Utilization - Place Design** line (see Fig. 2.71).

A tab opens in the upper right frame. Scroll down to **1. Slice Logic**. The IP uses 503 LUTs and 443 FFs (see Fig. 2.72; your values may differ if you are using a different model of FPGA or if you are using a different version of Vivado). These are the real values of the implementation cost on the FPGA. They are different from the ones given in the Vitis_HLS tool.

In the upper line menu, select **File/Export/Export Hardware** (see Fig. 2.73). In the pop up window, click on **Next**.

In the next window, check the **Include bitstream** option (the bitstream should be included) and click on **Next** (see Fig. 2.74).

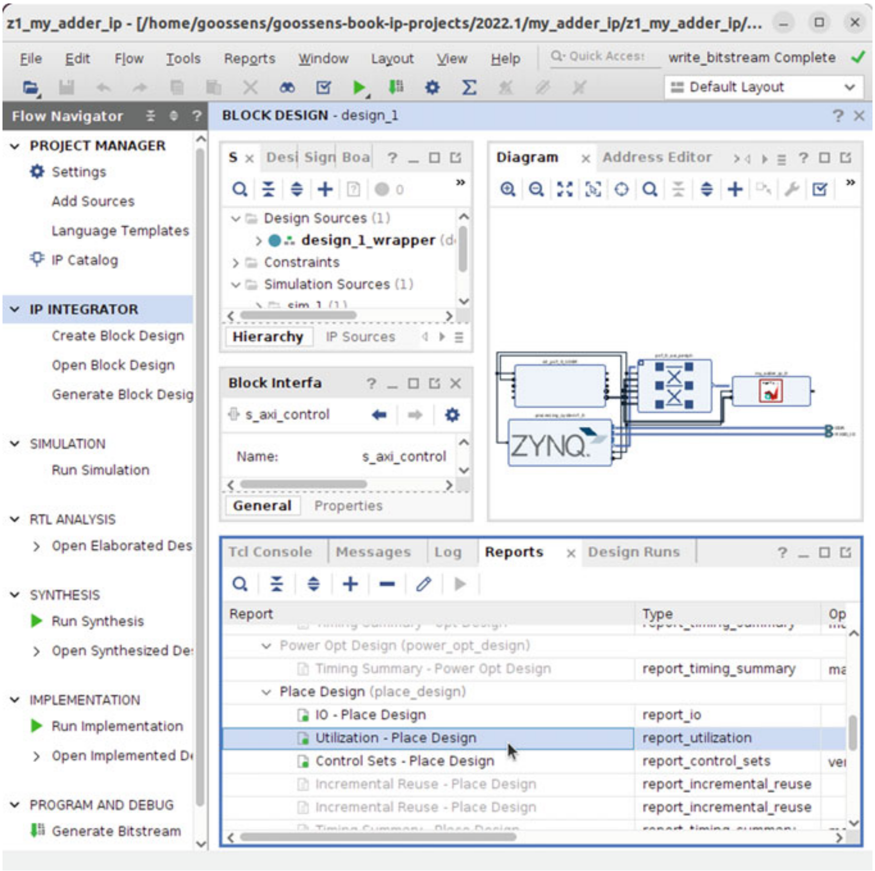
In the next window, click on **Next**. In the last window, click on **Finish** (see Fig. 2.75).

2.7 Loading the IP and Running the FPGA with Vitis

For this section, you need a development board.

You will transfer the bitstream to the FPGA, run it and check the result.

First, you must configure the boot mode of your board as boot from JTAG. A jumper should be configured to connect two pins labeled JTAG. On the Pynq-Z1, the jumper is JP4. It is just above the micro-USB connector. On the Pynq-Z2, the jumper is JP1. It is just under the HDMI IN connector. On the Basys3, the jumper is JP1. It is on the right of the USB connector J2. On the Zybo-Z7-20, the jumper is JP5. It is under the “ZYBOZ7” print.



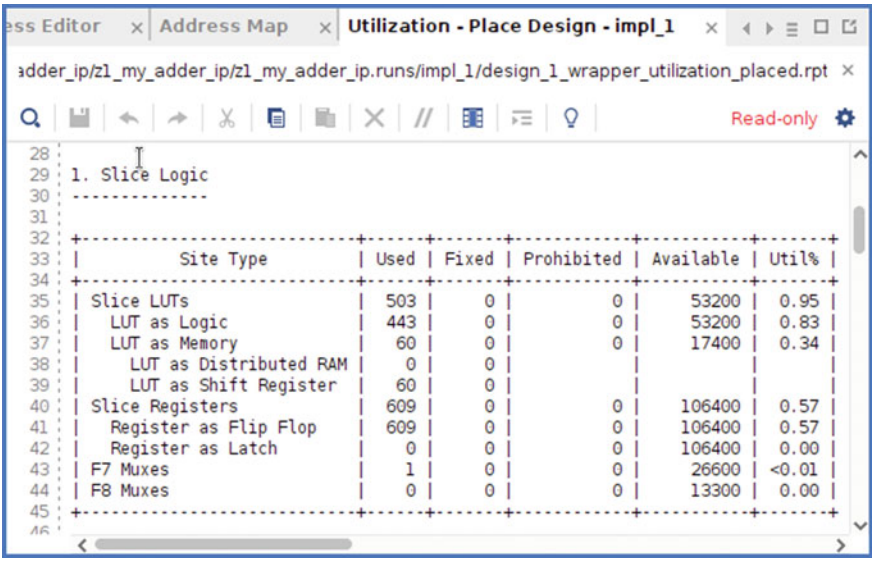
Report	Type	Op
Power Opt Design (power_opt_design)		
Timing Summary - Power Opt Design	report_timing_summary	ma
Place Design (place_design)		
IO - Place Design	report_io	
Utilization - Place Design	report_utilization	
Control Sets - Place Design	report_control_sets	ve
Incremental Reuse - Place Design	report_incremental_reuse	
Incremental Reuse - Place Design	report_incremental_reuse	

Fig. 2.71 Selecting the implementation report after placement

Second, you should configure the power source of your board as coming from the USB link. A jumper should be configured to connect two pins labeled USB. On the Pynq-Z1, the jumper is JP5. It is on the right of the power switch. On the Pynq-Z2, the jumper is J9. It is on the right of the SW1 switch. On the Basys3, the jumper is JP2. It is on the left of the power button. On the Zybo-Z7-20, the jumper is J16. It is on the right of the power switch.

Then, you can plug your board to your computer through a powering USB port. On the board, you should find a micro-USB connector labeled “PROG”. On the Pynq-Z1, the connector is just above the RJ45 ETHERNET connector. On the Pynq-Z2, it is under the RJ45 connector. On the Basys3, it is on the right of the power switch. On the Zybo-Z7-20 board, it is under the power switch.

Switch the power to the **on** position. A red led should light up (otherwise, use another USB port until you find a powering one).


ss Editor
Address Map
Utilization - Place Design - impl_1
adder_ip/z1_my_adder_ip/z1_my_adder_ip.runs/impl_1/design_1_wrapper_utilization_placed.rpt
Read-only

```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|         Site Type          | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 |  503 |     0 |          0 |     53200 |  0.95 |
|   LUT as Logic             |  443 |     0 |          0 |     53200 |  0.83 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            |  609 |     0 |          0 |    106400 |  0.57 |
|   Register as Flip Flop    |  609 |     0 |          0 |    106400 |  0.57 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |    1 |     0 |          0 |     26600 | <0.01 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 2.72 The adder IP resource utilization

You must install the USB cable driver. Run the commands shown in Listing 2.9 (they are in the install_cable_driver.txt file) with the board plugged to the computer and **on** (red led on).

Listing 2.15 Installing the USB cable driver

```
$ cd /opt/Xilinx/Vitis/2022.1/data/xicom
$ cd cable_drivers/lin64/install_script/install_drivers
$ sudo ./install_drivers
$
```

Then, switch the board off, unplug it, replug it and switch it on again.

The following applies (until further notice) to Zynq based boards, not to Artix-7 ones (if you have an Artix-7 based board, e.g. Basys3 or Nexys4, jump to page 74).

On your computer, in a terminal, run the putty serial terminal emulator as shown in Listing 2.16 (in sudo mode; you may have to install putty first from the Linux repositories: "sudo apt-get install putty").

Listing 2.16 Communicating with the board (1)

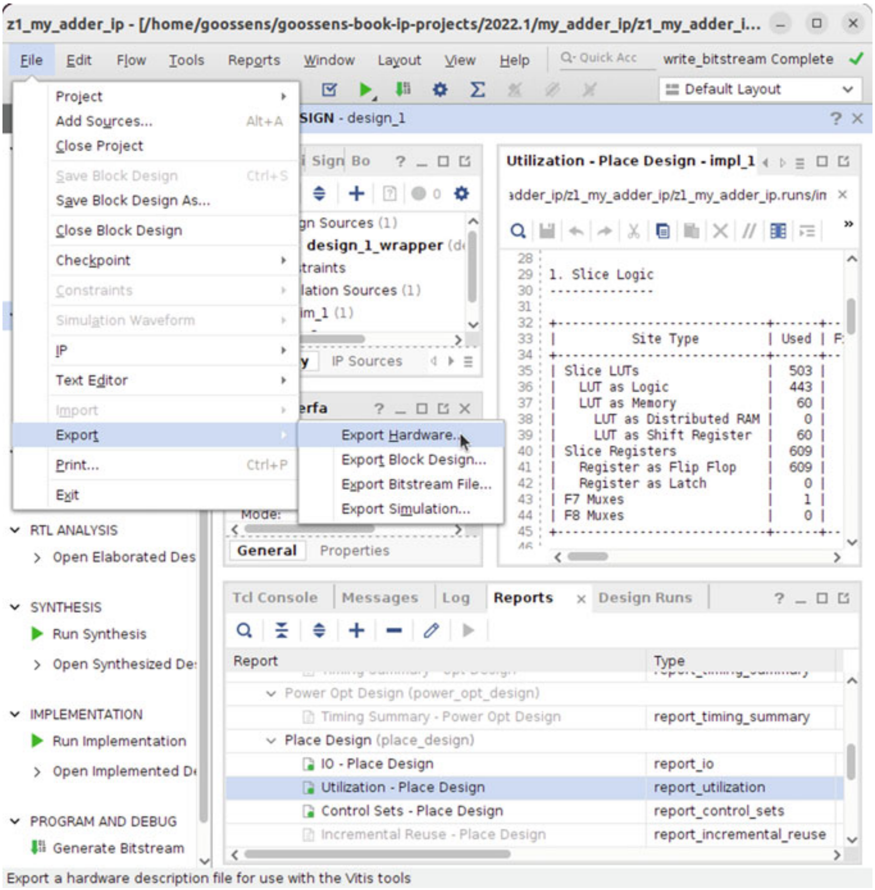
```
$ sudo putty
```

A terminal window pops up.

Make sure there is nothing else plugged on USB connectors on your computer as other devices may interfere with your board.

In the dialog box, check the **Serial** connection type (see Fig. 2.76).

Update the Serial line as /dev/ttyUSB1. Change the Speed to 115200. Click on **Open** (see Fig. 2.77).



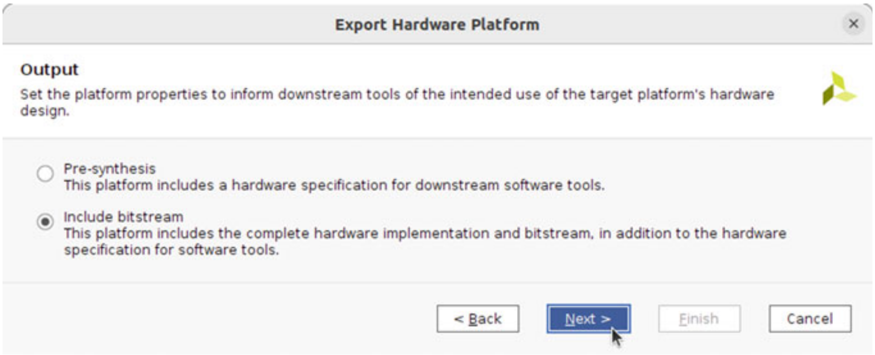
```
1. Slice Logic
--------------

+-----------------------------+------+--
|          Site Type          | Used | F
+-----------------------------+------+--
| Slice LUTs                  |  503 |
|   LUT as Logic              |  443 |
|   LUT as Memory             |   60 |
|     LUT as Distributed RAM  |    0 |
|     LUT as Shift Register   |   60 |
| Slice Registers             |  609 |
|   Register as Flip Flop     |  609 |
|   Register as Latch         |    0 |
| F7 Muxes                    |    1 |
| F8 Muxes                    |    0 |
+-----------------------------+------+--
```

Report	Type
Power Opt Design (power_opt_design)	
Timing Summary - Power Opt Design	report_timing_summary
Place Design (place_design)	
IO - Place Design	report_io
Utilization - Place Design	report_utilization
Control Sets - Place Design	report_control_sets
Incremental Reuse - Place Design	report_incremental_reuse

Export a hardware description file for use with the Vitis tools

Fig. 2.73 Exporting hardware


Export Hardware Platform

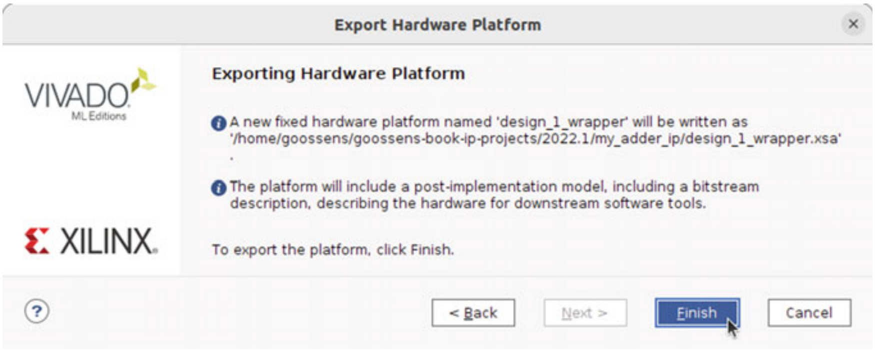
Output

Set the platform properties to inform downstream tools of the intended use of the target platform's hardware design.

○ Pre-synthesis
This platform includes a hardware specification for downstream software tools.

● Include bitstream
This platform includes the complete hardware implementation and bitstream, in addition to the hardware specification for software tools.

Fig. 2.74 Export hardware platform dialog box



Export Hardware Platform

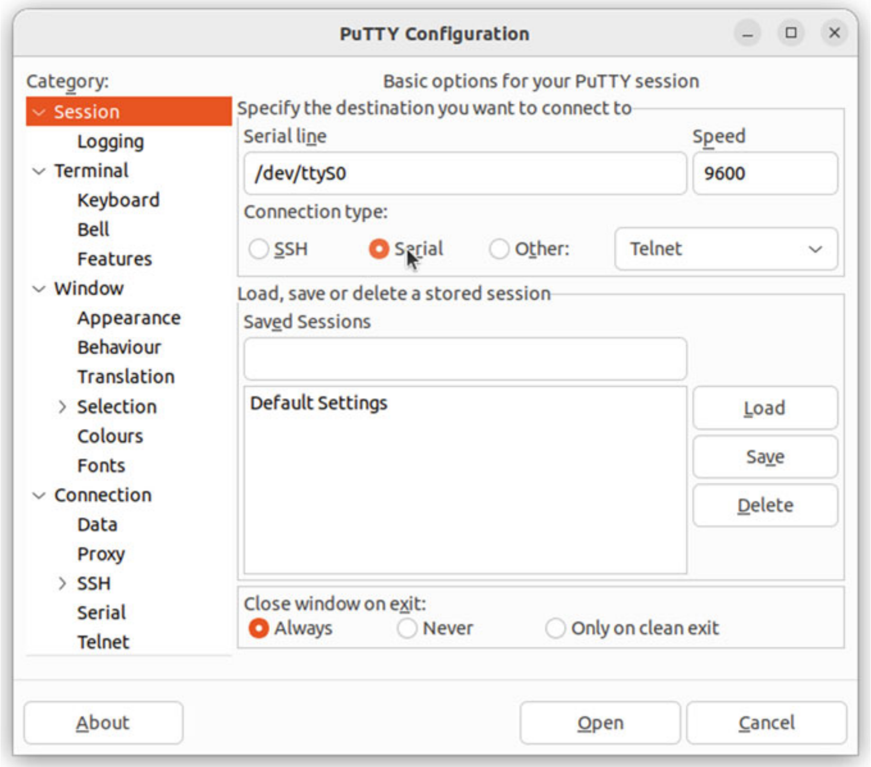
Exporting Hardware Platform

A new fixed hardware platform named 'design_1_wrapper' will be written as '/home/goossens/goossens-book-ip-projects/2022.1/my_adder_ip/design_1_wrapper.xsa' .

The platform will include a post-implementation model, including a bitstream description, describing the hardware for downstream software tools.

To export the platform, click Finish.

Fig. 2.75 Export hardware platform output dialog box



PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Serial line: /dev/ttyS0

Speed: 9600

Connection type: ○ SSH ● Serial ○ Other: Telnet

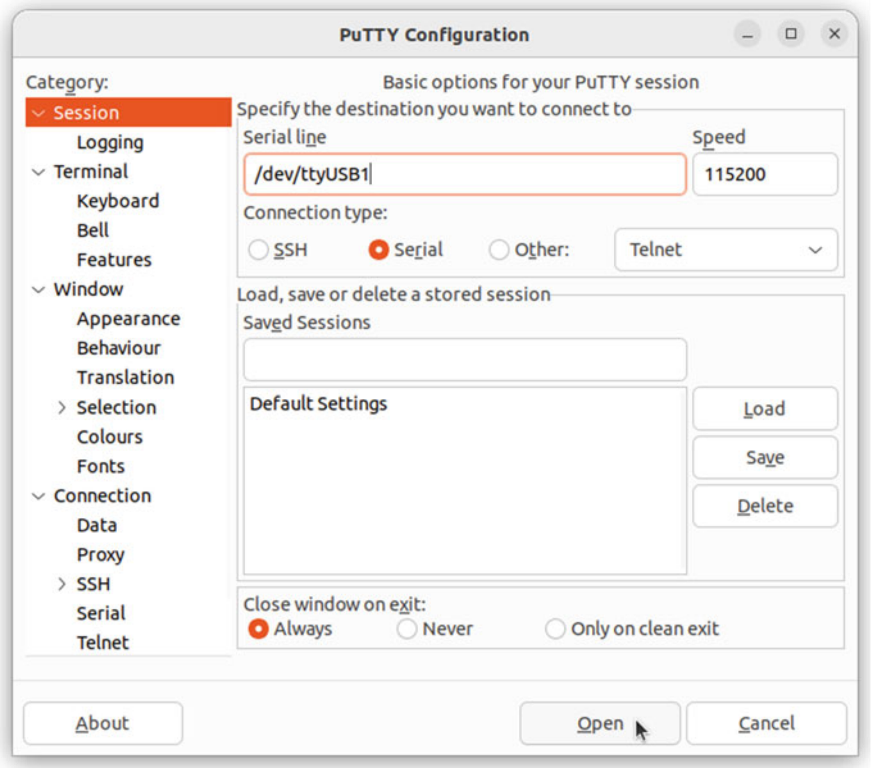
Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit: ● Always ○ Never ○ Only on clean exit

Fig. 2.76 Selecting the serial communication



Serial line: /dev/ttyUSB1

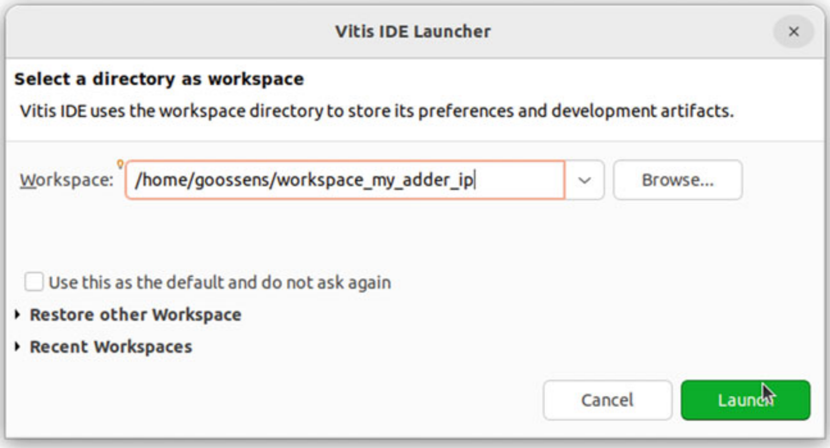
Speed: 115200

Connection type: ○ SSH ● Serial ○ Other: Telnet

Close window on exit: ● Always ○ Never ○ Only on clean exit

Fig. 2.77 Setting the USB serial line and communication speed

Fig. 2.78 The **/dev/ttyUSB1 PuTTY** terminal



Vitis IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: /home/goossens/workspace_my_adder_ip

☐ Use this as the default and do not ask again

Fig. 2.79 Selecting the workspace

A new empty terminal should open, labeled **/dev/ttyUSB1 - PuTTY** (see Fig. 2.78). This is where the board will print its messages while running.

The following applies (until further notice) to all the boards.

In a terminal, run the commands shown in Listing 2.15 (they are in the start_vitis_ide.txt file).

Listing 2.17 Starting Vitis IDE

```
$ cd /opt/Xilinx/Vitis/2022.1
$ source settings64.sh
$ vitis&
...
$
```

A dialog box proposes a workspace location and name (e.g. /your-home-path/workspace). Update the workspace name to reflect the Vivado project it is related to (I use one workspace per project), e.g. workspace_my_adder_ip. Click on **Launch** (see Fig. 2.79).

In the **workspace - Vitis IDE** window, **Welcome** tab, select **Create Application Project**. Click on **Next** (see Fig. 2.80).

In the **New Application Project/Platform** window, select the **Create a new platform** tab (see Fig. 2.81).

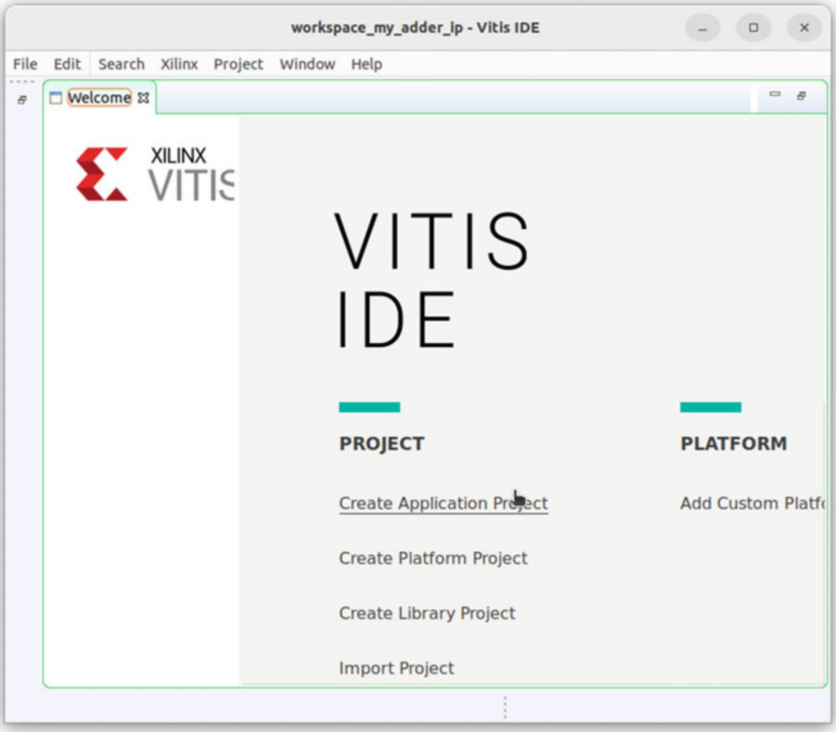
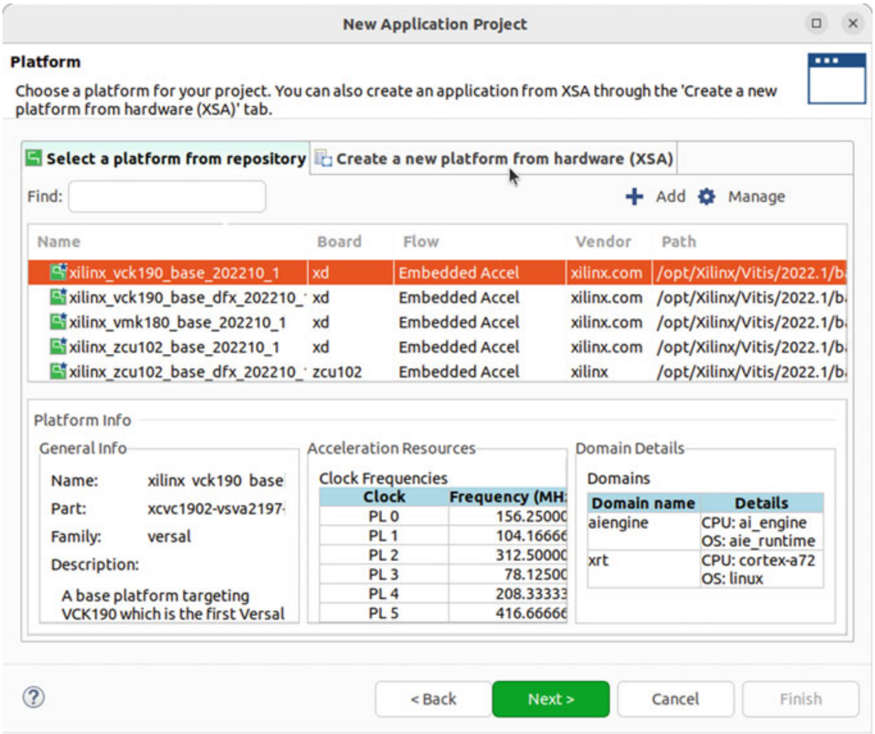


Fig. 2.80 The Vitis IDE welcome page



Platform

Choose a platform for your project. You can also create an application from XSA through the 'Create a new platform from hardware (XSA)' tab.

Find:

Name	Board	Flow	Vendor	Path
xilinx_vck190_base_202210_1	xd	Embedded Accel	xilinx.com	/opt/Xilinx/Vitis/2022.1/b
xilinx_vck190_base_dfx_202210_	xd	Embedded Accel	xilinx.com	/opt/Xilinx/Vitis/2022.1/b
xilinx_vmk180_base_202210_1	xd	Embedded Accel	xilinx.com	/opt/Xilinx/Vitis/2022.1/b
xilinx_zcu102_base_202210_1	xd	Embedded Accel	xilinx.com	/opt/Xilinx/Vitis/2022.1/b
xilinx_zcu102_base_dfx_202210_	zcu102	Embedded Accel	xilinx	/opt/Xilinx/Vitis/2022.1/b

Name: xilinx vck190 base

Part: xcvc1902-vsva2197-

Family: versal

Description:

A base platform targeting VCK190 which is the first Versal

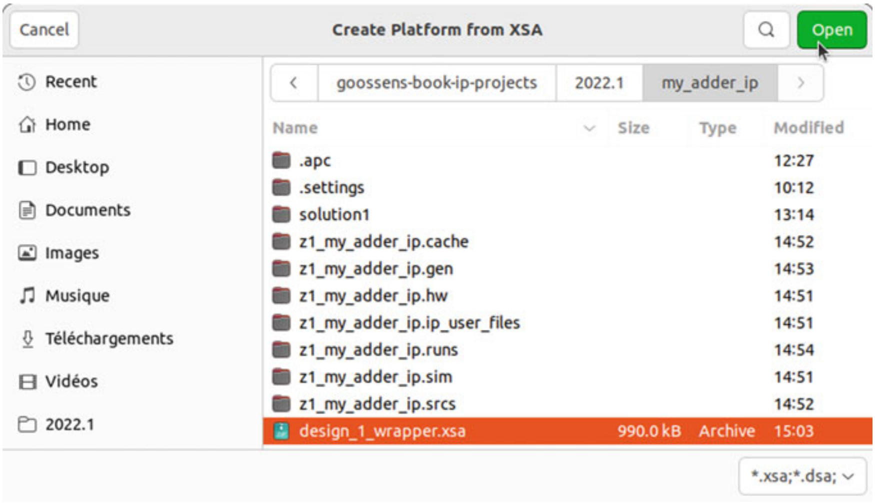
Clock Frequencies

Clock	Frequency (MH
PL 0	156.25000
PL 1	104.16666
PL 2	312.50000
PL 3	78.12500
PL 4	208.33333
PL 5	416.66666

Domains

Domain name	Details
aiengine	CPU: ai_engine OS: aie_runtime
xrt	CPU: cortex-a72 OS: linux

Fig. 2.81 Selecting the **Create a new platform from hardware (XSA)** tab



Name	Size	Type	Modified
.apc			12:27
.settings			10:12
solution1			13:14
z1_my_adder_ip.cache			14:52
z1_my_adder_ip.gen			14:53
z1_my_adder_ip.hw			14:51
z1_my_adder_ip.ip_user_files			14:51
z1_my_adder_ip.runs			14:54
z1_my_adder_ip.sim			14:51
z1_my_adder_ip.srcs			14:52
design_1_wrapper.xsa	990.0 kB	Archive	15:03

Fig. 2.82 Browse to find the XSA file

Browse to find your bitstream file (XSA file in the my_adder_ip folder) and click on **Open** (see Fig. 2.82).

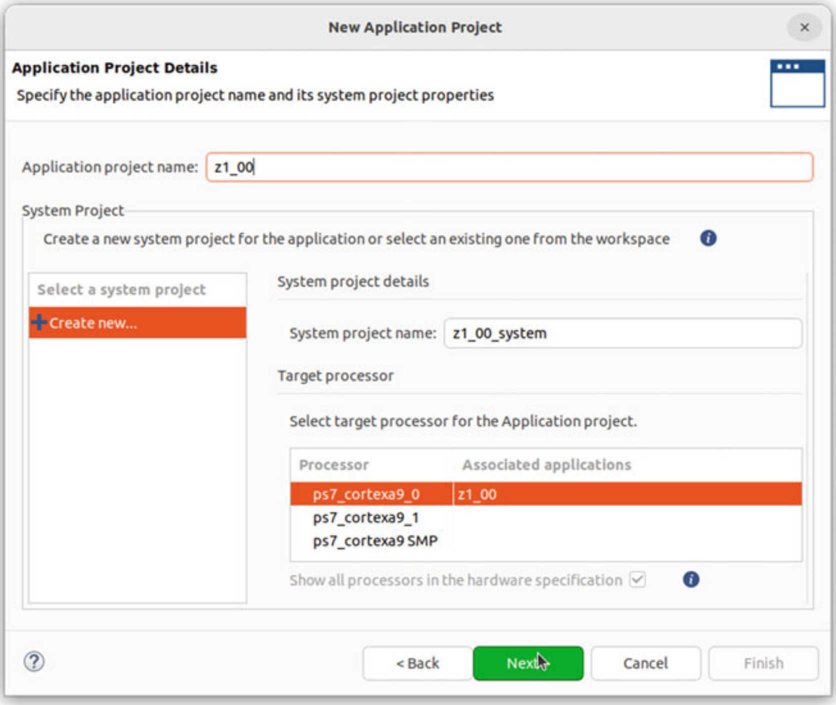
In the **New Application Project/Platform** page, leave the **Platform name** box unchanged (design_1_wrapper). Click on **Next**.

In the **New Application Project/Application Project Details** page, fill the **Application project name** (e.g. z1_00; I name my application projects with the development board name prefix, e.g. z1_, followed by an increasing counter starting from 00 and incrementing for each new version). Click on **Next** (see Fig. 2.83).

In the **Domain** page, click on **Next** (see Fig. 2.84).

In the **Templates** page, select the **Hello World** template and click on **Finish** (see Fig. 2.85).

A new window labeled **workspace_my_adder_ip - z1_00/src/helloworld.c - Vitis IDE** opens. In the left panel, **Explorer** tab, expand src and open file helloworld.c (see Fig. 2.86).



New Application Project

Application Project Details

Specify the application project name and its system project properties

Application project name: z1_00

System Project

Create a new system project for the application or select an existing one from the workspace

Select a system project

+Create new...

System project details

System project name: z1_00_system

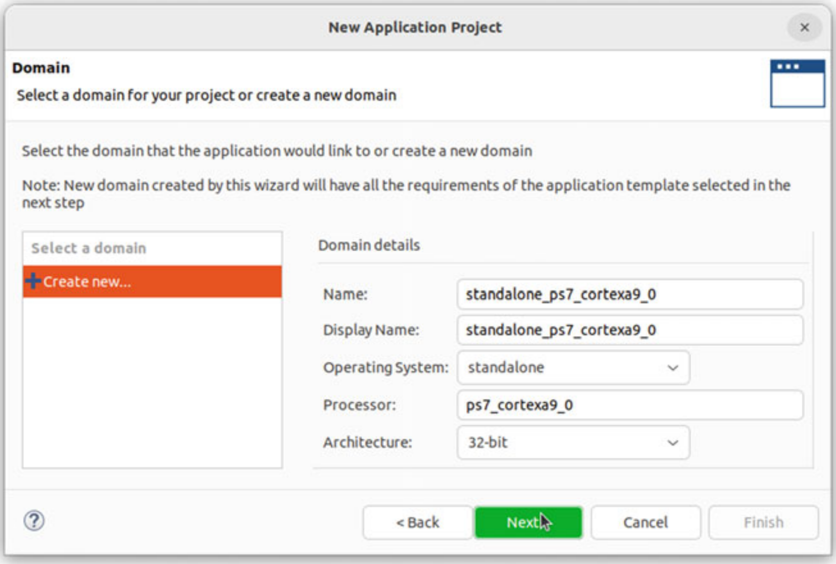
Target processor

Select target processor for the Application project.

Processor	Associated applications
ps7_cortexa9_0	z1_00
ps7_cortexa9_1	
ps7_cortexa9 SMP	

Show all processors in the hardware specification ☑

Fig. 2.83 Setting the name of the application project



New Application Project

Domain

Select a domain for your project or create a new domain

Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

Select a domain

+Create new...

Domain details

Name: standalone_ps7_cortexa9_0

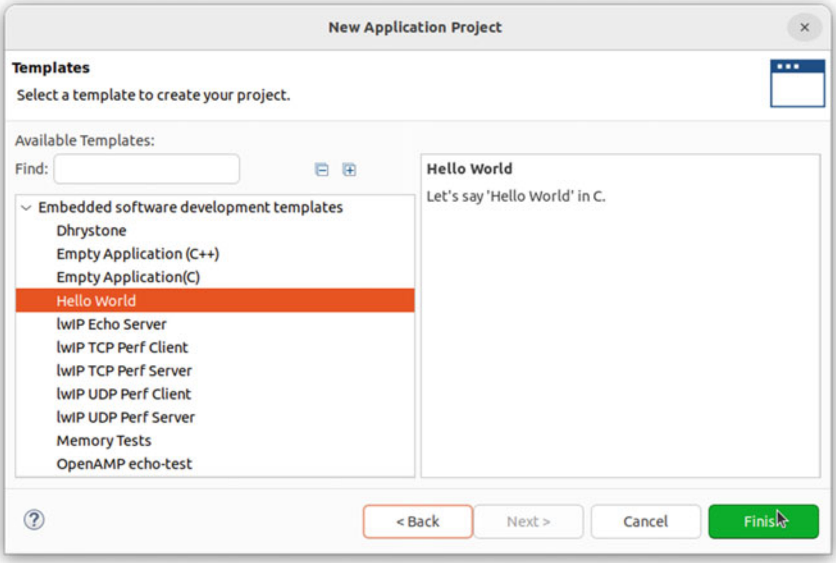
Display Name: standalone_ps7_cortexa9_0

Operating System: standalone

Processor: ps7_cortexa9_0

Architecture: 32-bit

Fig. 2.84 Selecting a domain



Templates

Select a template to create your project.

Available Templates:

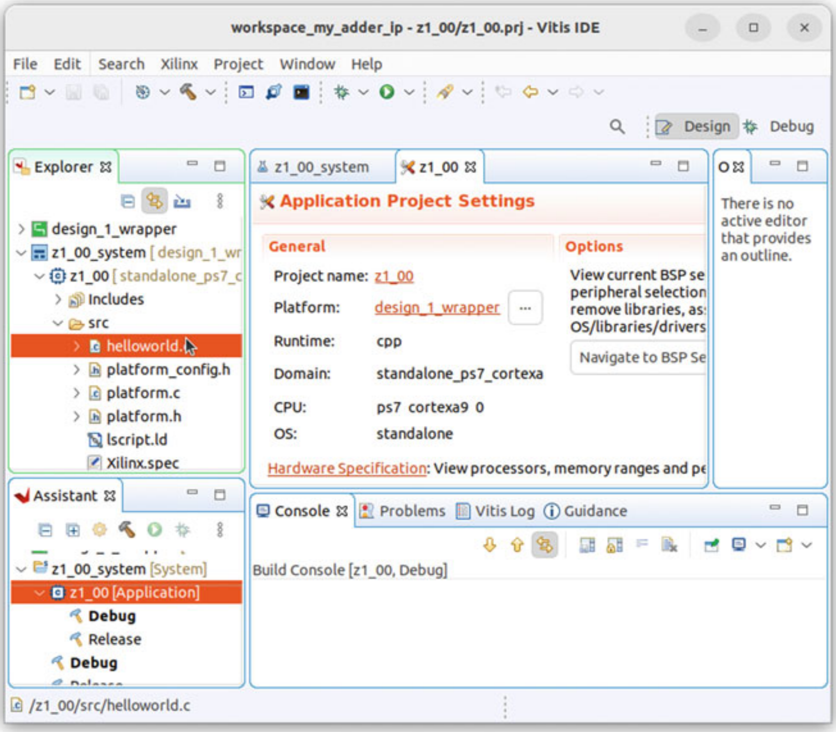
Find:

Embedded software development templates
- Dhrystone
- Empty Application (C++)
- Empty Application(C)
- Hello World
- lwIP Echo Server
- lwIP TCP Perf Client
- lwIP TCP Perf Server
- lwIP UDP Perf Client
- lwIP UDP Perf Server
- Memory Tests
- OpenAMP echo-test

Hello World

Let's say 'Hello World' in C.

Fig. 2.85 Selecting the **Hello World** template



design_1_wrapper
z1_00_system [design_1_wr
z1_00 [standalone_ps7_c
Includes
src
helloworld.
platform_config.h
platform.c
platform.h
lscript.ld
Xilinx.spec

Application Project Settings

General

Project name: z1_00

Platform: design_1_wrapper

Runtime: cpp

Domain: standalone_ps7_cortexa

CPU: ps7_cortexa9_0

OS: standalone

Options

View current BSP se peripheral selection remove libraries, as: OS/libraries/drivers

Hardware Specification: View processors, memory ranges and pe

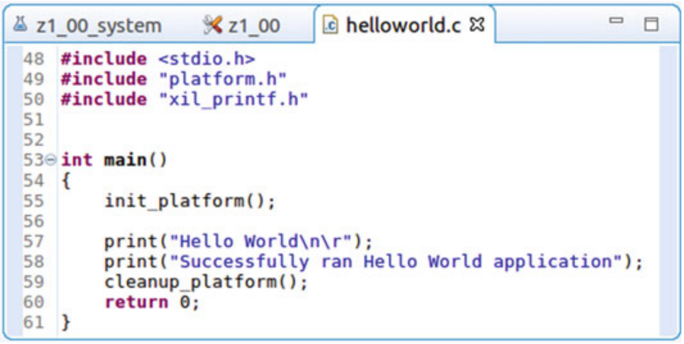
There is no active editor that provides an outline.

z1_00_system [System]
z1_00 [Application]
Debug
Release
Debug

Build Console [z1_00, Debug]

/z1_00/src/helloworld.c

Fig. 2.86 Opening the helloworld.c file

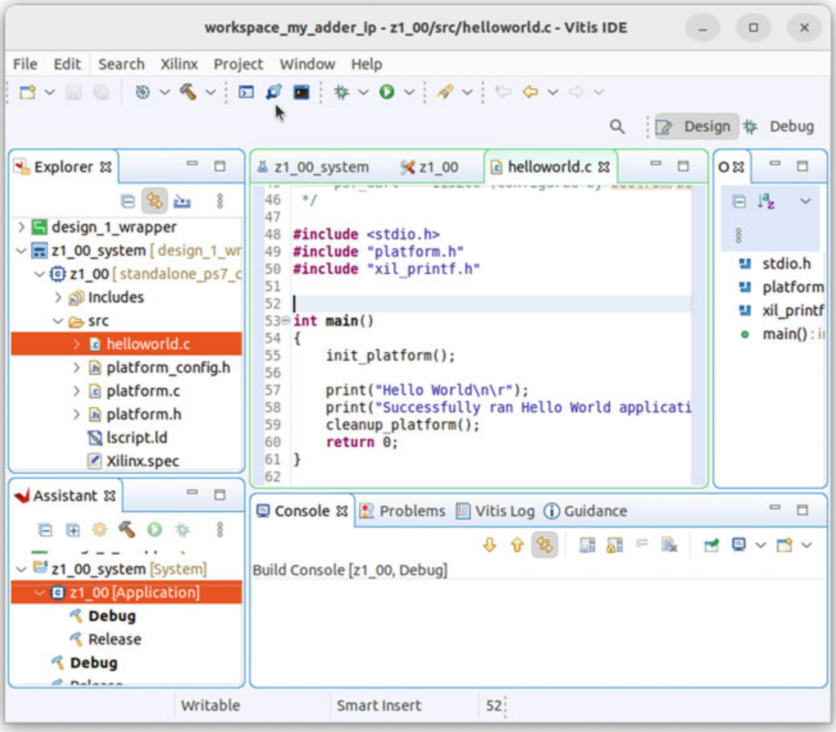


```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    cleanup_platform();
    return 0;
}
```

Fig. 2.87 The helloworld.c code



```
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"


int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World applicati
    cleanup_platform();
    return 0;
}

```

Fig. 2.88 The Launch Target Connection button

This is a basic driver to be downloaded on the FPGA and run (see Fig. 2.87). It initializes the platform (init_platform call) and prints the Hello World message in the **/dev/ttyUSB1 - PuTTY** communication window (with an Artix-7 based board (e.g. Basys3), the message prints in the Vitis IDE window, **Console** frame).

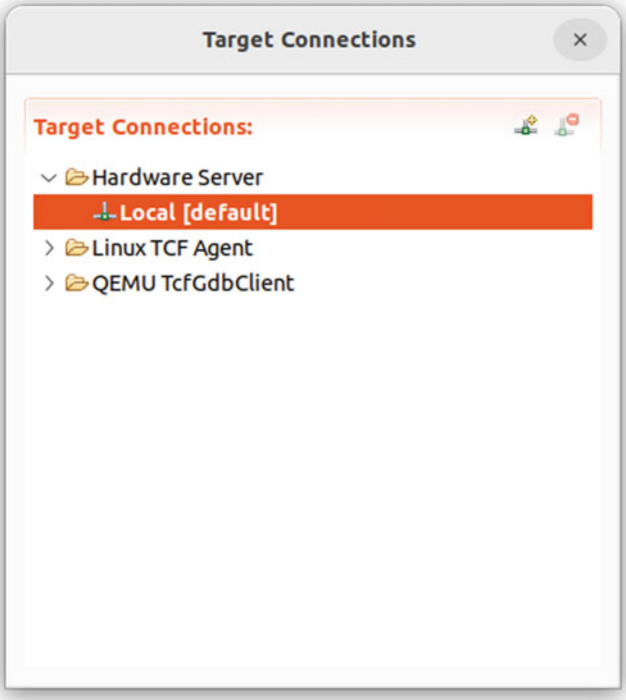


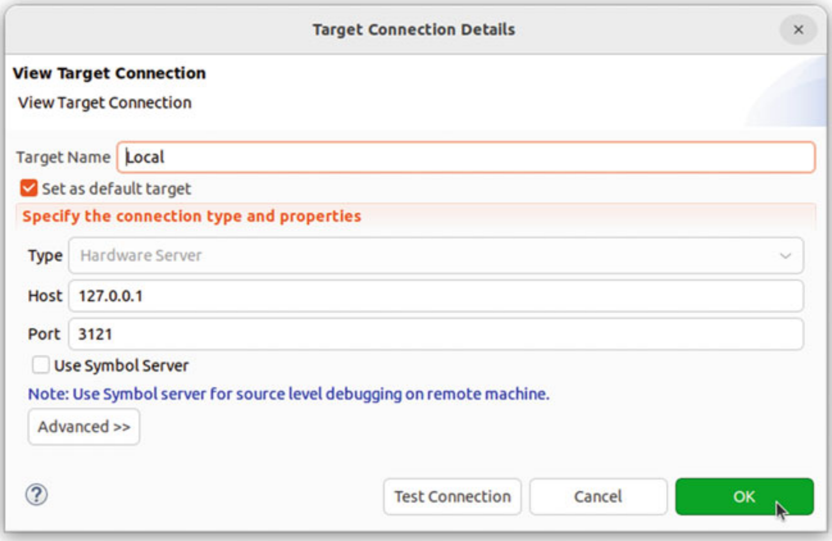
Fig. 2.89 The Local [default] entry

Later, we will update this program to make it print our adder result.

Click on the Launch Target Connection button (see Fig. 2.88; the button is on the top part of the window, under the main menu).

In the Target Connections dialog box, expand the Hardware Server entry. Double-click on the Local [default] entry (see Fig. 2.89).

In the Target Connection Details window, click on OK (see Fig. 2.90). Close the Target Connections dialog box.



Target Connection Details

View Target Connection

View Target Connection

Target Name: Local

☑ Set as default target

Specify the connection type and properties

Type: Hardware Server

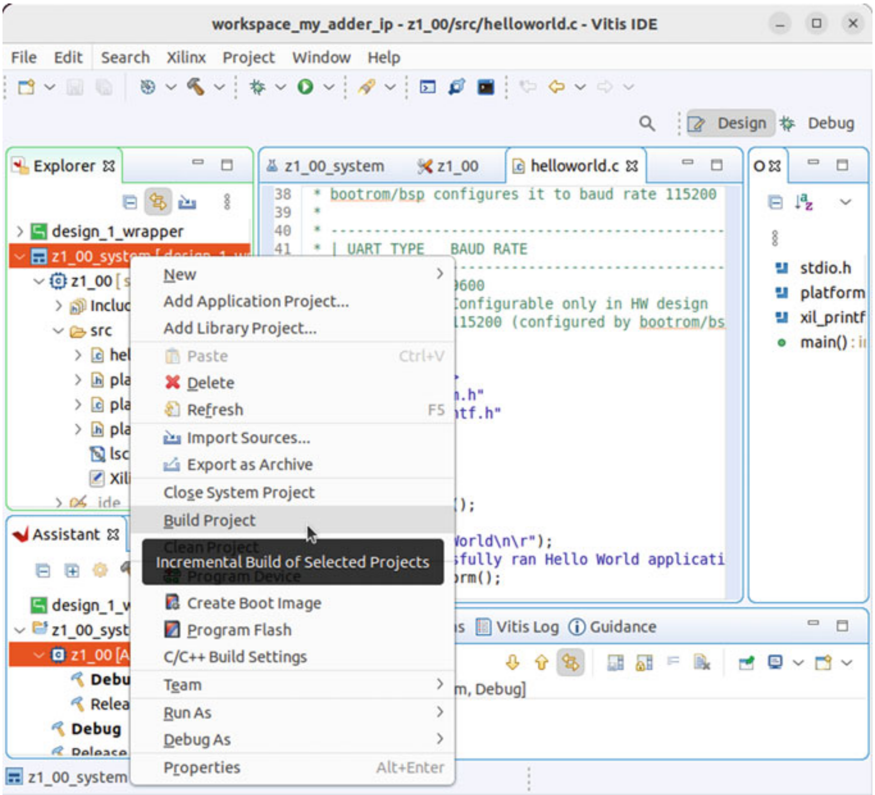
Host: 127.0.0.1

Port: 3121

□ Use Symbol Server

Note: Use Symbol server for source level debugging on remote machine.

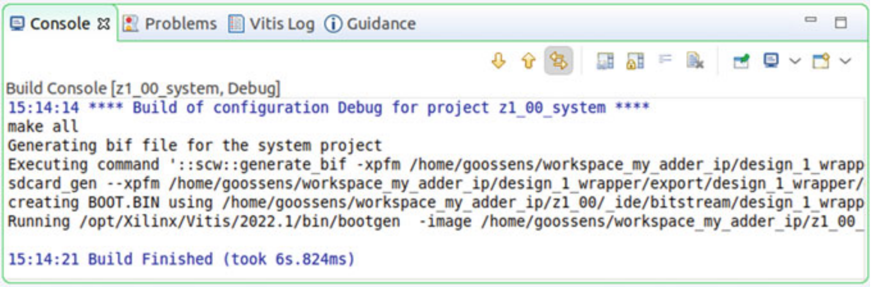
Fig. 2.90 The Target Connection Details window



workspace_my_adder_ip - z1_00/src/helloworld.c - Vitis IDE

```
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE    BAUD RATE
```

Fig. 2.91 Building the z1_00_system



```
Build Console [z1_00_system, Debug]
15:14:14 **** Build of configuration Debug for project z1_00_system ****
make all
Generating bif file for the system project
Executing command '::scw::generate_bif -xpfm /home/goossens/workspace_my_adder_ip/design_1_wrapp
sdcard_gen --xpfm /home/goossens/workspace_my_adder_ip/design_1_wrapper/export/design_1_wrapper/
creating BOOT.BIN using /home/goossens/workspace_my_adder_ip/z1_00/_ide/bitstream/design_1_wrapp
Running /opt/Xilinx/Vitis/2022.1/bin/bootgen  -image /home/goossens/workspace_my_adder_ip/z1_00_

15:14:21 Build Finished (took 6s.824ms)
```

Fig. 2.92 Result of the z1_00_system building

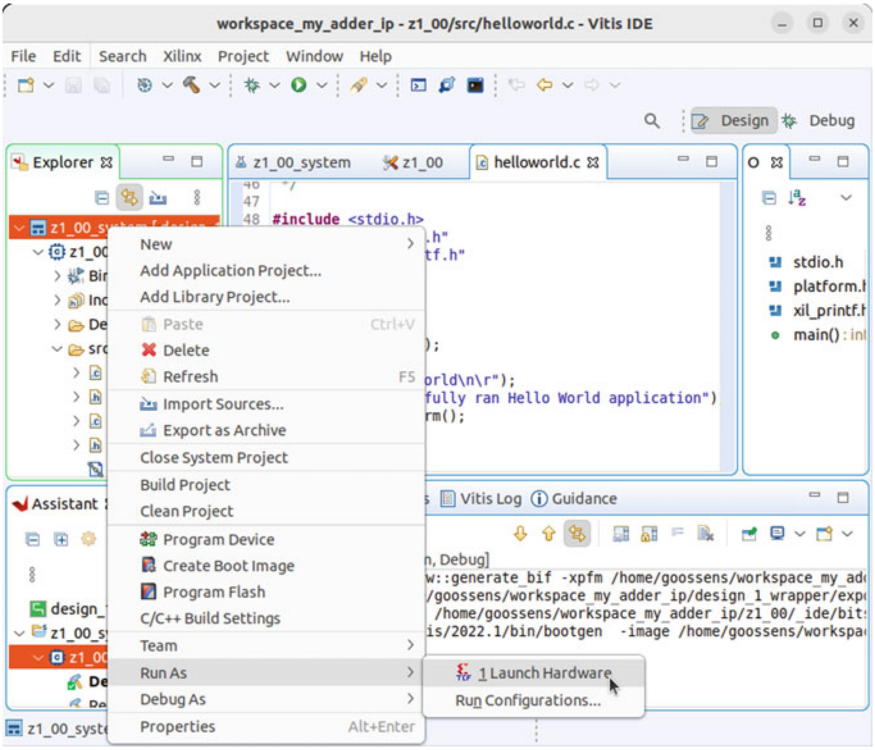
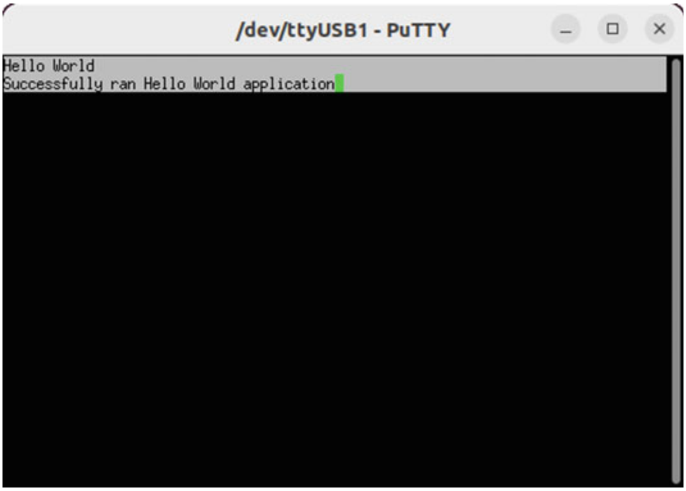


Fig. 2.93 Running the adder IP

Let us experiment running the printing of the Hello World message.

In the left panel, **Explorer** tab, right-click on your system folder name (z1_00_system). Scroll down to the **Build Project** entry and click on it (see Fig. 2.91).

The **Console** frame shows the progress of the compilation until the **Build Finished** message (see Fig. 2.92).



```
Hello World
Successfully ran Hello World application
```

Fig. 2.94 The helloworld print on the /dev/ttyUSB1 terminal

In the left panel, **Explorer** tab, right click on your system folder name. Scroll down and select **Run As/1 Launch Hardware** (see Fig. 2.93).

On the board, a green led lights up. In the **/dev/ttyUSB1 - PuTTY** communication window (or in the Vitis IDE window, **Console** frame on an Artix-7 based board like the Basys3), you should read Hello World and on the next line Successfully ran Hello World application (see Fig. 2.94).

You will now update your helloworld application to make it print the adder result, as shown in Listing 2.18 (you can copy/paste the helloworld.c file in the chapter_2 folder).

Listing 2.18 The updated helloworld.c file

```c
#include <stdio.h>
#include "xmy_adder_ip.h"
#include "xparameters.h"
XMy_adder_ip_Config *cfg_ptr;
XMy_adder_ip ip;
int main(){
  cfg_ptr = XMy_adder_ip_LookupConfig(XPAR_XMY_ADDER_IP_0_DEVICE_ID
      );
  XMy_adder_ip_CfgInitialize(&ip, cfg_ptr);
  XMy_adder_ip_Set_a(&ip, 10000);
  XMy_adder_ip_Set_b(&ip, 20000);
  XMy_adder_ip_Start(&ip);
  while (!XMy_adder_ip_IsDone(&ip));
  printf("%d + %d is %d\n",
    (int)XMy_adder_ip_Get_a(&ip),
    (int)XMy_adder_ip_Get_b(&ip),
    (int)XMy_adder_ip_Get_c(&ip));
  return 0;
}
```

File xmy_adder_ip.h contains the driver interface, i.e. a set of function prototypes to drive the IP on the FPGA. This file is automatically built by the Vitis_HLS tool and exported to the Vitis IDE tool.

There are functions to create an IP, to set its inputs, to start its run, to wait for its end and to get its outputs. All the functions are prefixed by X (for Xilinx I guess) and the IP name (e.g. My_adder_ip_).

Functions XMy_adder_ip_LookupConfig and XMy_adder_ip_CfgInitialize serve to allocate and initialize the adder IP.

Functions XMy_adder_ip_Set_a and XMy_adder_ip_Set_b serve to send input values a and b from the Zynq component to the adder IP component through the axilite connection.

Function XMy_adder_ip_Start starts the adder IP.

Function XMy_adder_ip_IsDone is used to wait until the adder IP has finished its job.

Function XMy_adder_ip_Get_c reads the final c value in the adder IP (transmission to the Zynq component through the axilite connection).

If you use an Artix-7 based board like the Basys3, you should replace the printf function by the xil_printf one (this is because the microblaze interface embarks a light version of printf). You should include the xil_printf.h file. The output of the run will not be displayed on the putty window but on the Vitis IDE one, in the **Console** frame.

Once the updated helloworld.c code has been saved, in the **Explorer** tab, right-click on the **z1_00_system** entry and select **Build Project** (as you already did to compile the former helloworld.c code). When the console informs you that **Build Finished**, right-click again on the **z1_00_system** entry and select **Run As/1 Launch Hardware**. You should see "10000 + 20000 is 30000" printed on the **/dev/ttyUSB1 - PuTTY** window.

References

1. https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start
2. https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html
3. https://reference.digilentinc.com/reference/programmable-logic/basys-3/start
4. L.H. Crockett, R.A. Elliot, M.A. Enderwitz, R.W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc* (Strathclyde Academic Media, 2014)
5. https://www.xilinx.com/support/university.html
6. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html

Installing and Using the RISC-V Tools

3

Abstract

This chapter gives you the basic instructions to setup the RISC-V tools, i.e. the RISC-V toolchain and the RISC-V simulator/debugger. The toolchain includes a cross-compiler to produce RISC-V RV32I machine code. The spike simulator/debugger is useful to run RISC-V codes with no RISC-V hardware. The result of a spike simulation is to be compared to the result of a run on an FPGA implementation of a RISC-V processor IP.

3.1 Installing a RISC-V Toolchain and an Emulator/Debugger

When you will have implemented a RISC-V processor IP, you will need to compare its behaviour to the official specification. If you do not want to buy a RISC-V hardware, the simulation is a good option.

For the RISC-V ISA, there are many simulators available. What you need is a tool to run your RISC-V code on your X86 machine. Soon enough, you will also need a way to trace your execution (list the RISC-V instructions run), run step by step and visualize the registers and the memory. This tool is more a *debugger* than a simulator.

Whatever the simulator you choose, you will first need to be able to produce RISC-V binaries from your host computer. Hence, a cross-compiler, and more extensively, a toolchain (compiler, loader, assembler, elf dumper ...) is absolutely crucial. The Gnu Project has developed the riscv-gnu-toolchain.

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_3

3.1.1 Installing the RISC-V Toolchain

The commands to install the riscv-gnu-toolchain can be found in the install_riscv_gnu_toolchain.txt file in the goossens-book-ip-projects/2022.1/chapter_3 folder.

The following indications only apply to the Linux Operating System (and more precisely, details apply to the Ubuntu distribution).

Before installing the RISC-V toolchain, you must install a set of commands. Run the "apt-get install" command in Listing 3.1.

Listing 3.1 Installing needed commands

```
$ sudo apt-get install autoconf automake autotools-dev curl python3
    python3-pip libmpc-dev libmpfr-dev libgmp-dev gawk build-
   essential bison flex texinfo gperf libtool patchutils bc zlib1g
   -dev libexpat-dev libtinfo5 libncurses5 libncurses5-dev
   libncursesw5-dev device-tree-compiler git pkg-config libstdc
   ++6:i386 libgtk2.0-0:i386 dpkg-dev:i386
...
$
```

When the required commands have been installed and git has been upgraded (refer back to 2.2), you can start the installation of the toolchain.

First, you must clone the riscv-gnu-toolchain git project on your computer. The cloning creates a new folder in the current directory. Run the "git clone" command in Listing 3.2.

Listing 3.2 Cloning the riscv-gnu-toolchain code

```
$ cd
$ git clone https://github.com/riscv/riscv-gnu-toolchain
...
$
```

Once this is done, you can build the RISC-V compiler. Configure and make, as shown in Listing 3.3 (it takes more than one hour to build the toolchain; there is a first long silent step to clone files from the git repository; 15 minutes on my computer).

Listing 3.3 Building the riscv-gnu-toolchain

```
$ cd $HOME/riscv-gnu-toolchain
$ ./configure --prefix=/opt/riscv --enable-multilib --with-arch=
   rv32i
...
$ sudo make
...
$
```

To try your tool, you can compile the hello.c source code shown in Listing 3.4 (it is in the chapter_3 folder).

Listing 3.4 A hello.c file

```
#include <stdio.h>
void main() {
  printf("hello world\n");
}
```

To have access to the commands provided by the riscv-gnu-toolchain you must update your PATH variable. Edit the .profile file in your $HOME folder and add the line shown in Listing 3.5 at the end of the file.

Listing 3.5 Setting the PATH variable

```
PATH=$PATH:/opt/riscv/bin
```

Close your session and login again. After login, the PATH variable includes the added path. You can check it by printing the PATH variable ("echo $PATH").

To compile the hello.c file, use the riscv32-unknown-elf-gcc compiling command in Listing 3.6 (the current directory should be the chapter_3 folder) (the compilation command for hello is available in the compile_hello.txt file in the chapter_3 folder).

Listing 3.6 Compiling with riscv32-unknown-elf-gcc

```
$ riscv32-unknown-elf-gcc hello.c -o hello
$
```

The executable file is hello. To produce a dump file of your hello binary, use the riscv32-unknown-elf-objdump command in Listing 3.7 (the > character is a redirection of the output to the hello.dump file rather than the terminal) (the dump command for hello is available in the compile_hello.txt file in the chapter_3 folder).

Listing 3.7 Dumping the executable file with objdump

```
$ riscv32-unknown-elf-objdump -d hello > hello.dump
$
```

The hello.dump file contains the readable version of the executable file (you can edit it). This is a way to produce RISC-V code in hexadecimal, safer than hand encoding.

To run your RISC-V hello, you need a simulator.

3.1.2 The Spike Simulator

3.1.2.1 Installing the Spike Simulator

The commands to install the spike simulator can be found in the install_spike.txt file in the chapter_3 folder.

The first step is to clone the riscv-isa-sim git project on your computer. Run the commands in Listing 3.8.

Listing 3.8 Cloning the riscv-isa-sim code

```
$ cd
$ git clone https://github.com/riscv/riscv-isa-sim
...
$
```

Then you can build the simulator. Run the commands in Listing 3.9 (it takes about five minute to build spike).

Listing 3.9 Building the spike simulator

```
$ export RISCV=/opt/riscv
$ cd $HOME/riscv-isa-sim
$ mkdir build
$ cd build
$ ../configure --prefix=$RISCV --with-isa=rv32i
...
$ make
...
$ sudo make install
...
$
```

3.1.2.2 Installing the Pk Interpreter

Before you can use the simulator you must build the pk interpreter (pk stands for proxy kernel). It serves as a substitute to the OS kernel. The pk interpreter gives access to a few basic system calls (mainly for I/O) and contains a boot loader (bbl for Berkeley Boot Loader) which can serve as a loader for the application to simulate with spike.

In Linux for example, the code to be run (e.g. the hello file produced by the compilation of the hello.c source code just above) is encapsulated in an ELF file (Executable and Linkable Format) composed of sections (a RISC-V RV32I ELF file like hello can be turned into a readable format with the riscv32-unknown-elf-objdump command I have just presented).

The text section in the ELF file contains the main function which is called by a _start function provided by the OS.

The run starts after the code has been loaded into the code memory and the data memory has been initialized with values given in the data section of the ELF file.

The role of the _start function is to set the stack with the main function arguments (i.e. argc, argv, and env), call the main function, and after return, call the exit function to end the run (you can have a look at the _start function code in the hello.dump file).

The commands to install the pk simulator can be found in the install_pk.txt file in the chapter_3 folder.

To install pk, first clone it from the github, as shown in Listing 3.10.

Listing 3.10 Cloning the riscv-pk code

```
$ cd
$ git clone https://github.com/riscv/riscv-pk
...
$
```

Build the pk interpreter. Run the commands in Listing 3.11.

Listing 3.11 Building the pk interpreter

```
$ export RISCV=/opt/riscv
$ cd $HOME/riscv-pk
$ mkdir build
$ cd build
$ ../configure --prefix=$RISCV --host=riscv32-unknown-elf
```

```
...
$ make
...
$ sudo make install
...
$
```

3.1.2.3 Simulating the Hello RISC-V Binary Code

You can simulate your hello program by running the command in Listing 3.12, which prints hello world, as expected (the current directory should be the chapter_3 folder) (the spike command is available in the spike_command.txt file in the chapter_3 folder).

Notice that the pk command is placed in the /opt/riscv/riscv32-unknown-elf/bin folder. Unfortunately, the spike command is unable to take advantage of the PATH variable to find pk, so you have to provide the full /opt/riscv/riscv32-unknown-elf/bin/pk access path.

Listing 3.12 Emulate hello with spike

```
$ spike /opt/riscv/riscv32-unknown-elf/bin/pk hello
bbl loader
hello world
$
```

With the -h option ("spike -h"), you can have a look at the available options. You can run in debug mode (-d) to progress instruction by instruction, print the registers (reg 0 to get the content of all the registers in core 0, reg 0 a0 to get the content of core 0, register a0).

3.1.2.4 Simulating the Test_op_imm.s Code

The test_op_imm simulation commands can be found in the compile_test_op_imm.txt file in the chapter_3 folder.

You can compile and run one of the test programs which will be used throughout the next chapters to test your RISC-V processor designs. The code shown in Listing 3.13 is a RISC-V assembly program to test the RISC-V computing instructions involving a constant (the test_op_imm.s file is available in the chapter_3 folder).

Listing 3.13 The test_op_imm.s file

```
        .globl  main
main:
        li      a1,5
        addi    a2,a1,1
        andi    a3,a2,12
        addi    a4,a3,-1
        ori     a5,a4,5
        xori    a6,a5,12
        sltiu   a7,a6,13
        sltiu   t0,a6,11
        slli    t1,a6,0x1c
        slti    t2,t1,-10
        sltiu   t3,t1,2022
        srli    t4,t1,0x1c
        srai    t5,t1,0x1c
        ret
```

To compile the test_op_imm.s file, run the command in Listing 3.14 (the current directory should be the chapter_3 folder).

Listing 3.14 Compiling test_op_imm.s with gcc

```
$ riscv32-unknown-elf-gcc test_op_imm.s -o test_op_imm
```

Run it step by step with the commands in Listing 3.15 (you must first localize the main function starting address: use "riscv32-unknown-elf-objdump -t" and pipe into grep ("riscv32-unknown-elf-objdump -t test_op_imm | grep main"); on my machine, the address is 0x1018c).

Listing 3.15 Debug test_op_imm with spike

```
$ riscv32-unknown-elf-objdump -t test_op_imm | grep main
0001018c g       .text  00000000 main
$ spike -d /opt/riscv/riscv32-unknown-elf/bin/pk test_op_imm
: untiln pc 0 1018c
...
core   0: 0x00010120 (0x06c000ef) jal     pc + 0x6c
:
core   0: 0x0001018c (0x00500593) li      a1, 5
: reg 0 a1
0x00000005
:
core   0: 0x00010190 (0x00158613) addi    a2, a1, 1
: reg 0 a2
0x00000006
: q
$
```

The first command "untiln pc X Y" runs the program until the pc of core X reaches address Y.

The 0x1018c address is the main entry point (this address may differ on your machine if you use a different compiler version or a different pk version than mine: try "riscv32-unknown-elf-gcc --version"; in the book, I use riscv32-unknown-elf-gcc (GCC) 11.1.0).

The *n* after "until" means *noisy*: the code run is printed (use "until pc X Y" to run silently). The run stops after the execution of the instruction at address 0x10120 and before the run of the instruction at address 0x1018c.

When no command is entered, i.e. the enter key is typed, the current instruction is run and printed. Hence, typing enter repeatedly runs the program step by step.

The "reg C R" command prints the content of register R in core C. The register name can be symbolic. In the example, I asked to successively print registers a1 and a2 (of core 0).

The q command is quit.

The spike simulator printings when you run the program step by step from main until the return are shown in Listing 3.16 (the "reg 0" command prints all the registers; in your execution some register values other than the ones written by test_op_imm may differ).

Listing 3.16 Run test_op_imm until return from main and print the registers

```
$ spike -d /opt/riscv/riscv32-unknown-elf/bin/pk test_op_imm
: untiln pc 0 1018c
...
core   0: 0x00010120 (0x06c000ef) jal     pc + 0x6c
:
core   0: 0x0001018c (0x00500593) li      a1, 5
:
core   0: 0x00010190 (0x00158613) addi    a2, a1, 1
...
core   0: 0x000101bc (0x41c35f13) srai    t5, t1, 28
:
core   0: 0x000101c0 (0x00008067) ret
: reg 0
zero: 0x00000000 ra: 0x00010124 sp :0x7ffffda0 gp :0x00011db0
  tp: 0x00000000 t0: 0x00000000 t1 :0xb0000000 t2 :0x00000001
  s0: 0x00000000 s1: 0x00000000 a0 :0x00000001 a1 :0x00000005
  a2: 0x00000006 a3: 0x00000004 a4 :0x00000003 a5 :0x00000007
  a6: 0x0000000b a7: 0x00000001 s2 :0x00000000 s3 :0x00000000
  s4: 0x00000000 s5: 0x00000000 s6 :0x00000000 s7 :0x00000000
  s8: 0x00000000 s9: 0x00000000 s10:0x00000000 s11:0x00000000
  t3: 0x00000000 t4: 0x0000000b t5 :0xfffffffb t6 :0x00000000
: q
$
```

3.1.3 Building Executable Code For the RISC-V FPGA Based Processors

When you compile with gcc, the linker adds the _start code which calls your main function. Moreover, it places the code at an address compatible with the OS mapping (e.g. 0x1018c in the test_op_imm example).

In contrast, the codes you build for a processor IP implemented on an FPGA are assumed to be running on bare metal (no OS), directly starting with the main function and ending when it returns. The code is to be placed at address 0.

For example, assume you want to compile the test_op_imm.s RISC-V assembly file shown in the preceding section to run it on your processor IP on the FPGA.

The gcc compiler can be requested to link the executable code at any given address with the "-Ttext address" option as shown in Listing 3.17 (the current directory should be the chapter_3 folder).

The warning message is not important.

Listing 3.17 Compile and base the main address at 0

```
$ riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 test_op_imm.s -o
    test_op_imm_0.elf
/opt/riscv/lib/gcc/riscv32-unknown-elf/11.1.0/../../../../riscv32-
    unknown-elf/bin/ld: warning: cannot find entry symbol _start;
    defaulting to 0000000000000000
$
```

Of course, such an executable file is not to be run with spike, only with your processor IP on the FPGA. To be run with spike/pk, the executable file must be obtained with a classic riscv32-unknown-elf-gcc compilation command ("riscv32-unknown-elf-gcc test_op_imm.s -o test_op_imm").

You can check your standalone and based 0 test_op_imm_0.elf executable file (in ELF format) by dumping it with riscv32-unknown-elf-objdump (the Linux cat command –short for concatenate, because it concatenates its output to the standard output– is used to view a file). Run the commands in Listing 3.18.

Listing 3.18 Dump the based 0 executable file test_op_imm_0.elf

```
$ riscv32-unknown-elf-objdump -d test_op_imm_0.elf > test_op_imm_0.
    dump
$ cat test_op_imm_0.dump

test_op_imm_0.elf:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main>:
   0:   00500593          li   a1,5
   4:   00158613          addi    a2,a1,1
  ...
  30:   41c35f13          srai    t5,t1,0x1c
  34:   00008067          ret
$
```

You can transform the ELF file test_op_imm_0.elf into a binary file test_op_imm_0_text.bin (i.e. remove the ELF format around the binary code) with riscv32-unknown-elf-objcopy and dump the binary file with od (octal dump; the leftmost column lists the addresses in octal). To build the test_op_imm_0_text.bin file and have a look at it, run the commands in Listing 3.19.

Listing 3.19 From ELF to binary

```
$ riscv32-unknown-elf-objcopy -O binary test_op_imm_0.elf
    test_op_imm_0_text.bin
$ od -t x4 test_op_imm_0_text.bin
0000000 00500593 00158613 00c67693 fff68713
0000020 00576793 00c7c813 00d83893 00b83293
0000040 01c81313 ff632393 7e633e13 01c35e93
0000060 41c35f13 00008067
0000070
$
```

This binary file can be translated into an hexadecimal file with hexdump. To build the test_op_imm_0_text.hex file, run the commands in Listing 3.20.

Listing 3.20 From binary to hexadecimal

```
$ hexdump -v -e '"0x" /4 "%08x" ",\n"' test_op_imm_0_text.bin >
    test_op_imm_0_text.hex
$ cat test_op_imm_0_text.hex
0x00500593,
0x00158613,
0x00c67693,
0xfff68713,
0x00576793,
0x00c7c813,
0x00d83893,
0x00b83293,
0x01c81313,
0xff632393,
0x7e633e13,
0x01c35e93,
0x41c35f13,
0x00008067,
$
```

This hex file has the appropriate structure to become an array initializer in a C program, listing the values to populate a code RAM array.

3.2 Debugging With Gdb

The commands used in this section can be found in the debugging_with_gdb.txt file in the chapter_3 folder.

The debugging facility in spike is rudimentary.

For example, to check your FPGA implementations, you will need to run some complex RISC-V code on your processor IP and compare the results with the ones produced by the spike simulator. Hence, you will need to dump portions of the data memory, which is not easy with spike in debugging mode.

3.2.1 Installing Gdb

The spike simulator can be connected to the standard gdb debugger. The gdb debugger has all the facilities you will need.

First, you should install gdb with the command in Listing 3.21.

Listing 3.21 Installation of gdb

```
$ sudo apt-get install build-essential gdb
...
$
```

The gdb debugger is mainly an interface between you and the machine running some code you want to debug. The machine can be a real one (i.e. either the host on which gdb is running or an external hardware like your development board) or a simulator like spike.

To debug with gdb, a code must be compiled with the -g option.

If gdb is interfacing the host, it simply accesses the machine on which it is running.

If gdb is interfacing some external hardware, there should be some tool to access the external hardware for gdb. OpenOCD (Open On-Chip Debugger) is such a sort of universal interface between a debugger like gdb and some external processor.

When gdb is interfacing the spike simulator, the same OpenOCD tool is used to connect the debugger to the simulator.

As a result, you will run spike, OpenOCD, and gdb together. The gdb debugger reads the code to be run from the executable file and sends a run request to OpenOCD which forwards it to spike. The spike simulator runs the code it is requested to and updates its simulated machine state. The gdb debugger can read this new state through some new request sent to OpenOCD and forwarded to spike, like reading the registers or the memory.

3.2.2 Installing OpenOCD

Second, after the gdb installation, you should install OpenOCD. Run the commands in Listing 3.22.

Listing 3.22 Installation of OpenOCD

```
$ cd
$ git clone https://git.code.sf.net/p/openocd/code openocd-code
...
$ cd openocd-code
$ ./bootstrap
...
$ ./configure --enable-jtag_vpi --enable-remote-bitbang
...
$ make
...
$ sudo make install
...
$
```

3.2.3 Defining a Linker Script File Compatible With the Spike Simulated Machine

The code to debug should be placed in a memory area compatible with the simulated machine. As spike is run on the host machine, it defines a memory space compatible with the hosting OS, in which the code to be simulated should be placed. The memory address used by spike in Ubuntu for its simulated code is 0x10010000.

The spike.lds file (in the chapter_3 folder) is shown in Listing 3.23. It defines a text section starting at address 0x10010000 and a data section immediately following the text one.

The structure of linker description files is explained in 7.3.1.4.

Listing 3.23 The spike.lds file

```
$ cat spike.lds
OUTPUT_ARCH( "riscv" )

SECTIONS
{
  . = 0x10010000;
  .text : { *(.text) }
  .data : { *(.data) }
}
$
```

3.2.4 Using the Linker Script File to Compile

You can compile the test_op_imm.s file to build an executable which can be simulated with spike (the -g option adds a table of symbols to the executable file which is used by gdb). Run the command in Listing 3.24 (the current directory should be the chapter_3 folder).

Listing 3.24 Using the spike.lds file to link for spike

```
$ riscv32-unknown-elf-gcc -g -nostartfiles -T spike.lds
    test_op_imm.s -o test_op_imm
$
```

3.2.5 Defining a Spike Configuration File for OpenOCD

OpenOCD needs a configuration file related to spike.

The spike.cfg file (in the chapter_3 folder) is shown in Listing 3.25.

Listing 3.25 The spike.cfg file

```
$ cat spike.cfg
interface remote_bitbang
remote_bitbang_host localhost
remote_bitbang_port 9824

set _CHIPNAME riscv
jtag newtap $_CHIPNAME cpu -irlen 5 -expected-id 0x10e31913

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME

gdb_report_data_abort enable

init
halt
$
```

3.2.6 Connecting Spike, OpenOCD, and Gdb

To connect spike, OpenOCD, and gdb, you will need to open three terminals successively.

In a first terminal with the chapter_3 folder as the current directory, start spike with a code to be simulated placed at address 0x10010000 (the same address as the one set in the spike.lds file), with a memory size of 0x10000. Run the command in Listing 3.26.

Listing 3.26 Starting spike

```
$ spike --rbb-port=9824 -m0x0010010000:0x10000 test_op_imm
Listening for remote bitbang connection on port 9824.
warning: tohost and fromhost symbols not in ELF; can't communicate
    with target
```

The spike process waits for requests from OpenOCD.

In a second terminal with the chapter_3 folder as the current directory, start OpenOCD with the spike.cfg configuration file. Run the command in Listing 3.27.

Listing 3.27 Starting OpenOCD

```
$ openocd -f spike.cfg
Open On-Chip Debugger 0.11.0+dev-00550-gd27d66b
...
Info : starting gdb server for riscv.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

The OpenOCD process waits for requests from gdb.

In a third terminal with the chapter_3 folder as the current directory, start gdb. Run the command in Listing 3.28.

Listing 3.28 Start gdb

```
$ riscv32-unknown-elf-gdb
GNU gdb (GDB) 10.1
...
Type "apropos word" to search for commands related to "word".
(gdb)
```

3.2.7 A Debugging Session

To start the debugging session within gdb, you need to connect gdb to OpenOCD. In the terminal where gdb is running, run the "target remote localhost:3333" command in Listing 3.29.

Listing 3.29 Connect gdb to OpenOCD

```
(gdb) target remote localhost:3333
warning: No executable has been specified and target does not
    support
determining executable automatically.  Try using the "file" command
    .
0x00000000 in ?? ()
(gdb)
```

Then, you must specify which executable file is to be debugged (answer "y" to the question). Run the "file test_op_imm" command in Listing 3.30.

Listing 3.30 Debug the test_op_imm executable file

```
(gdb) file test_op_imm
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from test_op_imm...
(gdb)
```

Load the informations of the linker (necessary to initialize pc). Run the "load" command in Listing 3.31.

Listing 3.31 Load the informations of the linker

```
(gdb) load
Loading section .text, size 0x38 lma 0x10010000
Start address 0x10010000, load size 56
Transfer rate: 448 bits in <1 sec, 56 bytes/write.
(gdb)
```

You can list the source code (command "l"; the "l" abbreviation stands for "list"; as a general rule in gdb, you can use the shortest unambiguous prefix of any command; for the "load" command above, you should at least type "lo"). Run the "l" command in Listing 3.32.

Listing 3.32 List the instructions to run

```
(gdb) l
1              .globl   main
2     main:
3              li       a1,5
4              addi     a2,a1,1
5              andi     a3,a2,12
6              addi     a4,a3,-1
7              ori      a5,a4,5
8              xori     a6,a5,12
9              sltiu    a7,a6,13
10             sltiu    t0,a6,11
(gdb)
```

You can visualize pc. Run the "p $pc" command ("p" for "print") in Listing 3.33.

Listing 3.33 Print pc

```
(gdb) p $pc
$1 = (void (*)()) 0x10010000 <main>
(gdb)
```

You can run one instruction. Run the "si" command in Listing 3.34 ("si" stands for step instruction; the instruction run is the one preceding the printed one; line 3 is run, i.e. "li a1,5"; the debugger is stopped on line 4).

Listing 3.34 Run one machine instruction

```
(gdb) si
[riscv.cpu] Found 4 triggers
4              addi     a2,a1,1
(gdb)
```

If you type enter, the last command is repeated (in the example, "si" is repeated and a new instruction is run). Repeat the "si" command, as shown in Listing 3.35.

Listing 3.35 Run another machine instruction

```
(gdb)
5              andi     a3,a2,12
(gdb)
```

You can print registers a1 and a2 (command "info reg"). Run the "info reg" commands in Listing 3.36.

Listing 3.36 Print registers

```
(gdb) info reg a1
a1                0x5   5
(gdb) info reg a2
a2                0x6   6
(gdb)
```

You can place a breakpoint on the last instruction. Run the "b 16" command in Listing 3.37.

Listing 3.37 Place a breakpoint

```
(gdb) l
1             .globl   main
2    main:
3             li       a1,5
4             addi     a2,a1,1
5             andi     a3,a2,12
6             addi     a4,a3,-1
7             ori      a5,a4,5
8             xori     a6,a5,12
9             sltiu    a7,a6,13
10            sltiu    t0,a6,11
(gdb)
11            slli     t1,a6,0x1c
12            slti     t2,t1,-10
13            sltiu    t3,t1,2022
14            srli     t4,t1,0x1c
15            srai     t5,t1,0x1c
16            ret
(gdb) b 16
Breakpoint 1 at 0x10010034: file test_op_imm.s, line 16.
(gdb)
```

You can continue running up to the breakpoint. Run the "c" command in Listing 3.38.

Listing 3.38 Continuing the run until the breakpoint

```
(gdb) c
Continuing.

Breakpoint 1, main () at test_op_imm.s:16
16            ret
(gdb)
```

You end the different runs in the three terminals with ctrl-c (or q for gdb; two successive ctrl-c for spike). You close the terminals with ctrl-d.

3.3 Debugging a Complex Code with Gdb

The commands used in this section can be found in the debugging_with_gdb.txt file in the chapter_3 folder.

When a code calls multiple functions, it uses the stack.

The stack is not initialized when you debug an executable with no start file (-nostartfiles option added to the compile command). It should be set in the gdb session.

The stack should occupy the ending memory area (set sp as the last memory address; the sp register moves backward, from high to low addresses).

The code in Listing 3.39 (in the chapter_3 folder) is an example of a computation using the stack (the stack keeps the local variables, i.e. a1, b1, c1, d1, array x where the results are saved, X, and solutions).

Listing 3.39 The basicmath_simple.c file

```
#include <stdio.h>
#include <math.h>
#define PI (4*atan(1))
void SolveCubic(double a, double b, double c, double d,
                int *solutions, double *x){
  long double a1 = b/a, a2 = c/a, a3 = d/a;
  long double Q = (a1*a1 - 3.0*a2)/9.0;
  long double R = (2.0*a1*a1*a1 - 9.0*a1*a2 + 27.0*a3)/54.0;
  double      R2_Q3 = R*R - Q*Q*Q;
  double      theta;
  if (R2_Q3 <= 0){
   *solutions = 3;
    theta = acos(R/sqrt(Q*Q*Q));
    x[0] = -2.0*sqrt(Q)*cos(theta/3.0) - a1/3.0;
    x[1] = -2.0*sqrt(Q)*cos((theta+2.0*PI)/3.0) - a1/3.0;
    x[2] = -2.0*sqrt(Q)*cos((theta+4.0*PI)/3.0) - a1/3.0;
  }
  else{
   *solutions = 1;
    x[0] = pow(sqrt(R2_Q3)+fabs(R), 1/3.0);
    x[0] += Q/x[0];
    x[0] *= (R < 0.0) ? 1 : -1;
    x[0] -= a1/3.0;
  }
}
void main(){
  double a1 = 1.0, b1 = -10.5, c1 = 32.0, d1 = -30.0;
  double x[3];
  double X;
  int    solutions;
  /* should get 3 solutions: 2, 6 & 2.5 */
  SolveCubic(a1, b1, c1, d1, &solutions, x);
  return;
}
```

Compile the basicmath_simple.c file with the command in Listing 3.40 (the current directory should be the chapter_3 folder).

Listing 3.40 Compile the basicmath_simple.c file

```
$ riscv32-unknown-elf-gcc -nostartfiles -T spike.lds -g -O3
    basicmath_simple.c -o basicmath_simple -lm
$
```

Start spike to simulate basicmath_simple with the command in Listing 3.41.

Listing 3.41 Start spike for the basicmath_simple executable

```
$ spike --rbb-port=9824 -m0x0010010000:0x10000 basicmath_simple
Listening for remote bitbang connection on port 9824.
warning: tohost and fromhost symbols not in ELF; can't communicate
    with target
```

Start OpenOCD in a second terminal with the command in Listing 3.42 (the current directory should be the chapter_3 folder).

Listing 3.42 Start OpenOCD

```
$ openocd -f spike.cfg
Open On-Chip Debugger 0.11.0+dev-00550-gd27d66b
...
Info : starting gdb server for riscv.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

Start gdb in a third terminal with the command in Listing 3.43 (the current directory should be the chapter_3 folder).

Listing 3.43 Start gdb

```
$ riscv32-unknown-elf-gdb
GNU gdb (GDB) 10.1
...
Type "apropos word" to search for commands related to "word".
(gdb)
```

Connect to OpenOCD, specify the executable, and load the linking informations with the commands in Listing 3.44.

Listing 3.44 Continue gdb

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
warning: No executable has been specified and target does not
    support
determining executable automatically.  Try using the "file" command
    .
0x00000000 in ?? ()
(gdb) file basicmath_simple
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from basicmath_simple...
(gdb) load
Loading section .text, size 0x8390 lma 0x10010000
Loading section .text.startup, size 0x58 lma 0x10018390
...
Loading section .sdata._impure_ptr, size 0x4 lma 0x1001ccc8
Start address 0x10010000, load size 52417
Transfer rate: 2 KB/sec, 1310 bytes/write.
(gdb)
```

Set pc and sp with the commands in Listing 3.45 (pc is set at main, i.e. .text.startup; if the value returned by your gdb is not 0x10018390, update the pc initialization accordingly; sp is set at the end of the memory, i.e. 0x10020000).

Listing 3.45 Set pc and sp

```
(gdb) set $pc = 0x10018390
(gdb) set $sp = 0x10020000
(gdb)
```

List the source code. As the -g option has been added to the compile command with basicmath_simple.c as the input file, gdb works on the C source file rather than on the assembly code like it was for test_op_imm. Run the commands in Listing 3.46.

Listing 3.46 List the source code

```
(gdb) l
18          else{
19           *solutions = 1;
20            x[0] = pow(sqrt(R2_Q3)+fabs(R), 1/3.0);
21            x[0] += Q/x[0];
22          x[0] *= (R < 0.0) ? 1 : -1;
23          x[0] -= a1/3.0;
24        }
25      }
26      void main(){
27        double a1 = 1.0, b1 = -10.5, c1 = 32.0, d1 = -30.0;
(gdb)
28        double x[3];
29        double X;
30        int    solutions;
31        /* should get 3 solutions: 2, 6 & 2.5 */
32        SolveCubic(a1, b1, c1, d1, &solutions, x);
33        return;
34      }
(gdb)
```

Place a breakpoint on the return instruction in main, line 33 with the command in Listing 3.47.

Listing 3.47 Place a breakpoint at the end of the run

```
(gdb) b 33
Breakpoint 1 at 0x100183dc: file basicmath_simple.c, line 33.
(gdb)
```

You continue the run until the breakpoint with the gdb c command in Listing 3.48.

Listing 3.48 Continue the run up to the breakpoint

```
(gdb) c
Continuing.

Breakpoint 1, main () at basicmath_simple.c:33
33    return;
(gdb)
```

The result is somewhere on the stack (at the place allocated for array x by the run; x should contain three double values corresponding to 2.0, 6.0, and 2.5, i.e. in hexadecimal: 0x4000000000000000, 0x4018000000000000, and 0x4004000000000001).

To find the result, you can dump the memory used by the stack with the gdb "x" command in Listing 3.49 (the "x/16x 0x1001ffc0" command dumps 16 words in hexadecimal format, starting at address 0x1001ffc0).

Listing 3.49 Dump the stack

```
(gdb) x/16x 0x1001ffc0
0x1001ffc0: 0x1001ffd4  0x1001ffd8  0x00000000  0x00000000
0x1001ffd0: 0x00000000  0x00000003  0x00000000  0x40000000
0x1001ffe0: 0x00000000  0x40180000  0x00000001  0x40040000
0x1001fff0: 0x00000000  0x00000000  0x00000000  0x00000000
(gdb)
```

The three values are located at address 0x1001ffd8-df, 0x1001ffe0-e7 and 0x1001ffe8-ef (low order 32-bit word first, i.e. little endian).

The RISC-V Architecture

4

Abstract

This chapter briefly presents the RISC-V architecture and more precisely, its RV32I instruction set with examples taken from the compiler translations of small C codes.

4.1 The RISC-V Instruction Set Architecture

The RISC-V architecture is built from a nucleus of 32-bit instructions to handle 32-bit integer data (named RV32I) and extensions for different instruction and data sizes and types (e.g. RVC for 16-bit data, RV64I for 64-bit data, M extension for integer multiplication, division, and remainder instructions, or F extension for simple precision floating-point computations [1]).

Even though the most basic RV32I instruction set is tiny, it is enough to implement any application, even an OS like Linux (the computing power of the RV32I ISA is the same as a Turing machine—it is Turing-complete) (The Turing machine is a theoretical model which was proven by Alan Turing to be able to calculate any computable function—computable functions are algorithms; see the 1936 Turing original paper [2]; also have a look at the web site devoted to Alan Turing at URL http://www.alanturing.net).

The extensions are provided to drive added hardware units to accelerate computations (e.g. add an integer multiplier and an integer divider to the processor).

To these user-level instruction sets, the RISC-V specification adds a privileged architecture devoted to OS implementations. An overview of the full ISA is presented in the David Patterson and Andrew Waterman book [3].

In my book, I only deal with the nucleus, i.e. the RV32I instruction set.

As RV32I is tiny, the resulting applications may be slower than if implemented with a larger instruction set. For example, with RV32I, you can implement C codes using the float and double types, even though your hardware is not equipped with a

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_4

Fig. 4.1 The RISC-V integer register file

name	alias	role	save
x0	zero	hardwired 0	no
x1	ra	return address	yes
x2	sp	stack pointer	no
x3	gp	global pointer	no
x4	tp	thread pointer	no
x5–x7	t0–t2	temporary	no
x8–x9	s0–s1	saved	yes
x10–x15	a0–a5	arguments	no
x16–x17	a6–a7	arguments	no
x18–x27	s2–s11	saved	yes
x28–x31	t3–t6	temporary	no

Floating-Point Unit (FPU). The compiler and the libraries provide the necessary functions to simulate the floating-point operations from integer ones. In this book, I will test the proposed processors on benchmarks involving floating-point computations.

4.1.1 The RV32I Registers and the RISC-V Application Binary Interface

Figure 4.1 shows the mapping of the integer register file. There are 32 registers (32 bits wide in the RV32I ISA), named x0 to x31. The 16 first registers are in black in the figure and the 16 last registers are in red. In the RV32I ISA, the 32 registers are all available. In the RVE (or RV32E) ISA (E stands for embedded), the register file is restricted to the registers in black.

In the figure, the *alias* column, the *role* column, and the *save* column define the RISC-V Application Binary Interface (ABI). The ABI is a general frame to build applications from the processor architecture.

Each register has an ABI name (the *alias* column) related to the suggested role of the register (the *role* column).

The *save* column shows which registers should be saved in the stack. When a saved register is used in a function, it should first be copied to the stack. It should also be restored (from the stack copy) before returning from the function. In such a way, its content is preserved across function calls.

Non saved registers either contain global values, valid all along the application, or temporary values, only valid in a local computation within a single function level.

The instruction set does not really distinguish the 32 registers except for the first of them. This means that all the instructions apply to all the registers and manipulate them the same way. But this first register has a special semantic.

Register zero or x0 contains the value 0 and cannot be changed. You cannot write to register zero (if you use this register as a destination, your write operation is simply discarded). Each time you need to mention the 0 special value, you can use register zero. For example, if you want to say something like "if (x==0) goto label", you would write "beq a0, zero, label" (i.e. branch to label if a0 and zero are equal), with register a0 holding the value of variable *x*.

Even though the architecture does not give any particular role to the registers through the ISA, the ABI does impose some common usage for all the registers. The programmer is strongly invited to follow the ABI constraints to keep his/her programs compatible with the other available pieces of softwares (e.g. libraries) and with the different translators like compilers or assemblers.

Register ra or x1 should be used to save the return address, i.e. the destination of call instructions (JAL and JALR). The usage is to write "jal ra, foo" to call the foo function and save the return address into register ra. However, you could as well write "jal t0, foo" and save the return address into the t0 register. You can also write "jal zero, label". In this case, you do not save any return address and the jal instruction is used as a jump to label.

Register sp or x2 should be used as a stack pointer, i.e. it should contain the address of the top of stack and move according to space allocations and liberations (e.g. "addi sp,sp,-16" run at the start of a function allocates 16 bytes on the top of stack; "addi sp,sp,16" run at the end of the function de-allocates the 16 bytes; in the function, the 16 bytes can be used as a *frame* to save and restore values with store and load instructions).

Registers a0 to a7 (or x10 to x17) should be used to hold function arguments (and results for a0 and a1).

Registers t0 to t6 (or x5 to x7 and x28 to x31) should be used to hold temporary values.

Registers s0 to s11 (or x8, x9, and x18 to x27) should be preserved across functions (if one of these registers is used in a function, it should be saved in the function stack frame at the function start and restored at the function end).

Operating systems may use registers x3 or gp as a global pointer (i.e. for the global data of the running process), x4 or tp as a thread pointer (data for the running thread), and x8 or s0 or fp as a frame pointer, i.e. a portion of stack allocated for a function to hold its arguments and locals.

4.1.2 The RV32I Instructions

The instruction set is traditionally partitioned into three subsets: computing instructions, control flow instructions, and memory access instructions.

Figure 4.2 shows examples of RISC-V instructions. Computing instructions are in red, control flow instructions are in brown, and memory accesses are in green.

4.1.2.1 Computing Instructions

In a RISC architecture (Reduced Instruction Set Computer), computing instructions operate on registers and constants (i.e. not on memory locations).

The available operations are the ones present in an Arithmetic and Logic Unit (ALU), i.e. addition, subtraction, left and right shifts, Boolean operators, and comparisons. Multiplication and division are not part of the RV32I ISA. They belong to the M extension.

Fig. 4.2 RISC-V instruction examples

RISC−V instruction		Semantic
add	a0, a1, a2	x10 = x11 + x12
addi	x10, x11, 1	a0 = a1 + 1
beq	a0, a1, .L1	if (a0 == a1) goto .L1
jal	foo	ra = pc + 4; goto foo
jr	x6	goto x6
ret		goto ra
lw	s0, 4(a0)	s0 = ram[a0 + 4 : a0 + 7]
sb	t1, 1(zero)	ram[1] = t1[0 : 7]

Hence, if you use a multiplication or a division in a C program, the compiler translates it with a call to a library function computing the operation with RV32I instructions.

Computing instructions either compute from two register sources (e.g. "add a0, a1, a2" adds registers a1 and a2) or from one register source and a constant. In this case, the constant is always the right term (e.g. "addi a0, a0, -1" decrements register a0).

4.1.2.2 Control Flow Instructions

Control flow instructions can be jumps or branches.

Jumps are unconditional and branches are conditional.

Jumps can have a link (i.e. a function call is linked to the return point which is saved in the ra register).

The jump target can be direct (i.e. a displacement from the jump instruction to the target one is directly encoded in the instruction like in "jal ra, foo").

The jump target can be indirect (i.e. the target address is given by a register like in "jalr ra, a0". This is the case of "jalr zero, ra", alias "ret", which jumps to the return address, i.e. returns from a function call.

Branches combine a condition with a jump (e.g. "beq a0, a1, label", which means "if (a0==a1) goto label").

In RISC-V, the condition is a comparison.

Comparison operators can be equality (beq), non equality (bne), littler (blt), and greater or equal (bge).

For comparators based on the integer numbers ordering, the comparison can be based on the signed or unsigned ordering (blt/bltu and bge/bgeu).

In signed ordering, the 32-bit data ranges from 0x80000000 to 0xffffffff for negative values, i.e. -2^{31} to -1, and from 0x00000000 to 0x7fffffff for positive values, i.e. 0 to $2^{31} - 1$.

In unsigned ordering, the range is 0x00000000 to 0xffffffff, i.e. 0 to $2^{32} - 1$.

For example 0x80000000 is littler than 0x7fffffff for a signed comparison (-2^{31} is littler than $2^{31} - 1$), but greater for an unsigned one (2^{31} is greater than $2^{31} - 1$).

4.1.2.3 Memory Access Instructions

Memory access instructions use a base + displacement addressing mode, i.e. the accessed location is computed from the sum of two terms.

The base term is given through a register and the displacement term is a constant (e.g. "lw a0, 4(a1)" to load a word from address 4+a1).

Load instructions read from memory to register (they have a destination register; e.g. a0 is the destination of the "lw a0, 4(a1)" load).

Store instructions write from register to memory (they have a second source register; e.g. "sw a0, 4(a1)" stores register a0 to memory).

The data movement can concern a byte (lb/lbu for byte loads, sb for byte stores), a pair of contiguous bytes (lh/lhu for half word loads, sh for half word stores), or a full 32-bit word (lw for word loads, sw for word stores).

The load of a byte or a half word value is sign or 0 extended.

The extension is the way the destination register is padded.

If a single byte is loaded, the three most significant bytes of the destination register are cleared if the load instruction is lbu (the loaded byte is unsigned).

If the load instruction is lb (the loaded byte is signed), the 24 most significant bits of the destination register are copies of the loaded byte sign bit (i.e. its most significant bit).

For example, if "lb a0, 4(a1)" loads byte 0x80 (i.e. 0b10000000 where the most significant bit is 1), register a0 is set as 0xffffff80.

If "lb a0, 4(a1)" loads byte 0x70 (i.e. 0b01110000, where the most significant bit is 0), register a0 is set as 0x00000070.

The RISC-V specification leaves the Execution Environment Interface (EEI), i.e. the implementation, free to deal with misaligned addresses (alignment is presented in Sect. 5.3.3.1 and in Sect. 6.3).

4.1.3 The RV32I Instruction Formats

The instructions are themselves data, i.e. words to be placed into memory.

The RV32I ISA is a set of 32-bit instructions. Each instruction is defined as a 32-bit word. All the semantic details concerning an instruction are encoded in these 32 bits. The way the encoding is done is called a *format*.

The RV32I instruction encoding ([1], Chap. 24) defines four formats: Register or R-TYPE, Immediate or I-TYPE, Upper or U-TYPE, and Store or S-TYPE.

Figure 4.3 shows the decomposition of the 32-bit instruction word into the main fields according to the format (the B-TYPE is a variant of the S-TYPE and the J-TYPE is a variant of the U-TYPE).

The two low order bits are set to **11** (there are instructions with their two low order bits different from **11** but not in the RV32I nucleus; they belong to the RVC extension—the *C* stands for *Compressed*—which encodes instructions in 16-bit words).

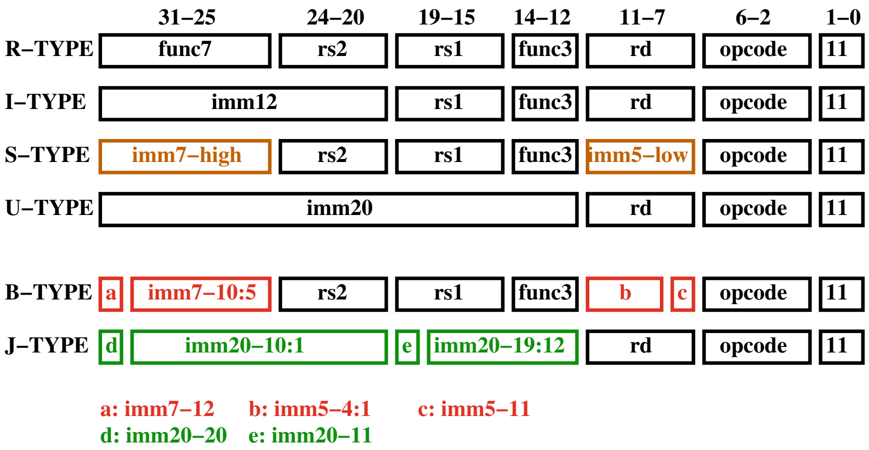


Fig. 4.3 RISC-V instruction formats

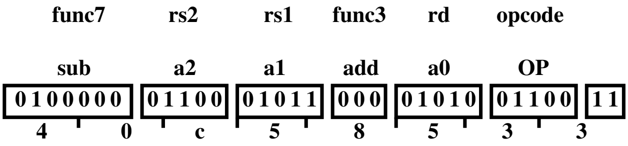


Fig. 4.4 The "sub a0, a1, a2" instruction encoding

4.1.3.1 The R-TYPE Format

The computing instructions using two register sources have the R-TYPE format, i.e. they are a sextuplet (func7, rs2, rs1, func3, rd, opcode).

The rs2 field is the second or right source register, rs1 is the first or left source register, rd is the destination register, and opcode is the major opcode OP.

The major opcode is completed with a minor opcode func3 and an operation specifier func7 to specify the ALU operation.

For example (see Fig. 4.4), "sub a0, a1, a2" is the binary sextuplet [0b0100000, 0b01100, 0b01011, 0b000, 0b01010, 0b0110011] (i.e. 0x40c58533 in hexadecimal).

The registers are 5-bit codes, func3 is a 3-bit code, func7 is a 7-bit code, and opcode is a 5-bit code.

The sub operation is coded by the three terms func7 (sub), func3 (add/sub), and opcode (OP).

Instruction codes ending by 0x33 or 0xb3 are R-TYPE ALU instructions (opcode OP).

4.1.3.2 The I-TYPE Format

The computing instructions using one register source and one constant source have the I-TYPE format, i.e. they are quintuplets (imm12, rs1, func3, rd, opcode).

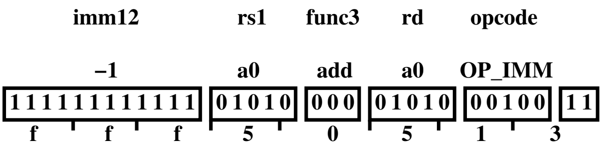


Fig. 4.5 The "addi a0, a0, -1" instruction encoding

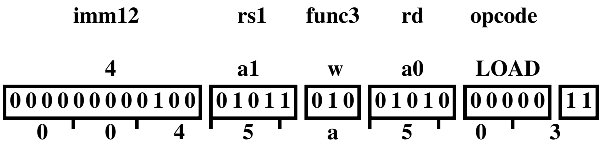


Fig. 4.6 The "lw a0, 4(a1)" instruction encoding

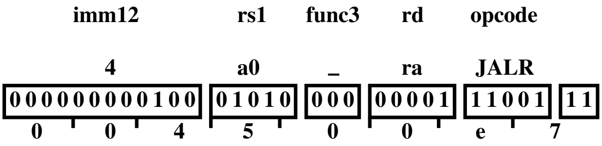


Fig. 4.7 The "jalr ra, 4(a0)" instruction encoding

The four last terms keep the same meaning as for the R-TYPE format (the opcode term is OP_IMM instead of OP).

The imm12 term is the immediate value, i.e. the constant involved in the computation instruction. It is a signed 12-bit field (hence the constant range is from 0x800 to 0x7ff or from -2^{11} to $2^{11} - 1$).

For example (see Fig. 4.5), "addi a0, a0, -1" is the quintuplet [0b111111111111, 0b01010, 0b000, 0b01010, 0b0010011] (i.e. 0xfff50513 in hexadecimal).

Instruction codes ending by 0x13 or 0x93 are I-TYPE ALU instructions (opcode OP_IMM).

The load instructions (see Fig. 4.6) also have the I-TYPE format (e.g. "lw a0, 4(a1)" is the quintuplet [0b000000000100, 0b01011, 0b010, 0b01010, 0b0000011], i.e. 0x0045a503).

Instruction codes ending by 0x03 or 0x83 are I-TYPE load instructions (opcode LOAD).

The indirect jump instructions JALR (see Fig. 4.7) are also of the I-TYPE format (e.g. "jalr ra, 4(a0)" is the quintuplet [0b000000000100, 0b01010, 0b000, 0b00001, 0b1100111], i.e. 0x004500e7).

Instruction codes ending by 0x67 or 0xe7 are I-TYPE jalr instructions (opcode JALR).

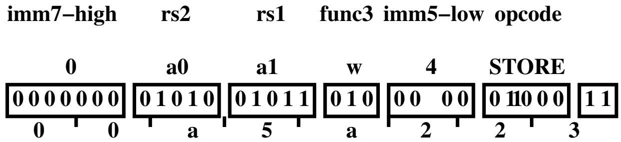


Fig. 4.8 The "sw a0, 4(a1)" instruction encoding

4.1.3.3 The S-TYPE Format

The store instructions (see Fig. 4.8) have the S-TYPE format, i.e. they are sextuplets (imm7-high, rs2, rs1, func3, imm5-low, opcode), where imm7-high and imm5-low are combined to build a signed 12-bit immediate value.

For example, "sw a0, 4(a1)" is the sextuplet [0b0000000, 0b01010, 0b01011, 0b010, 0b00100, 0b0100011] (i.e. 0x00a5a223 in hexadecimal).

Instruction codes ending by 0x23 or 0xa3 are S-TYPE store instructions (opcode STORE).

4.1.3.4 The B-TYPE Format

The branch instructions have the B-TYPE format, which is a variation of the S-TYPE format in the composition of the immediate value.

Instead of being the concatenation of the imm7-high and imm5-low fields, it is the concatenation of four pieces dispatched in the imm7-high and imm5-low fields.

More precisely, the imm7-high field is decomposed in a 1-bit part imm7-12 (bit 12 of the immediate value) (labeled a in Fig. 4.9) and a 6-bit part imm7-10:5 (bits 5 to 10 of the immediate value).

The imm5-low field is separated in a 4-bit part imm5-4:1 (labeled b) and a 1-bit part imm5-11 (labeled c).

A 12-bit constant is built by concatenating a, c, imm7-10:5, and b.

This constant is shifted left (introducing a trailing 0) and sign extended to form a 32-bit immediate value. This value is added to the pc when the condition of the branch is true (i.e. the branch is taken).

A positive constant gives a forward branch and a negative constant gives a backward branch.

For example, "beq a0, a1, pc+12" (see Fig. 4.9) is the octuplet [0b0, 0b000000, 0b01010, 0b01011, 0b000, 0b0110, 0b0, 0b1100011] (i.e. 0x00a58663 in hexadecimal).

In this example, a is 0b0, c is 0b0, imm7-10:5 is 0b000000, and b is 0b0110. Hence, the 12-bit constant is 0b000000000110, i.e. 6. After a one bit left shift and the sign extension, the 32-bit value added to pc is 12 (if a0 is equal to a2, 12 is added to pc, i.e. goto pc+12).

Instruction codes ending by 0x63 or 0xe3 are B-TYPE branch instructions (opcode BRANCH).

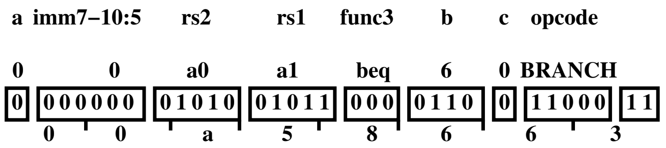


Fig. 4.9 The "beq a0, a1, pc+12" instruction encoding

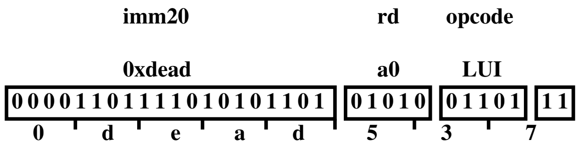


Fig. 4.10 The "lui a0, 0xdead" instruction encoding

4.1.3.5 The U-TYPE Format

The lui and auipc instructions have the U-TYPE format, i.e. they are triplets (imm20, rd, opcode), where imm20 is a 20-bit constant to update the upper part of the destination register (the lui opcode is 0b01101; the auipc opcode is 0b00101).

For example (see Fig. 4.10), "lui a0, 0xdead" is the triplet [0b00001101111010101101, 0b01010, 0b0110111] (0x0dead537 in hexadecimal).

These instructions build a 32-bit word from the 20-bit encoded constant followed by 12 zeros. The lui instruction saves the 32-bit constant to the destination register. The auipc instruction adds the 32-bit constant to pc and saves the result into the destination register.

Instruction codes ending by 0x17 or 0x97 are U-TYPE auipc instructions (opcode AUIPC).

Instruction codes ending by 0x37 or 0xb7 are U-TYPE lui instructions (opcode LUI).

4.1.3.6 The J-TYPE Format

The jal instruction has the J-TYPE, which is a variation of the U-TYPE.

The 20-bit field imm20 is decomposed into four parts: imm20-20 (one bit, labeled d in Fig. 4.11), imm20-10:1 (10 bits), imm20-11 (one bit, labeled e in Fig. 4.11) and imm20-19:12 (eight bits).

The d, imm20-19:12, e, and imm20-10:1 parts are concatenated to form a 20-bit constant.

The 20-bit constant is shifted left (introducing a trailing 0) and sign extended to form a 32-bit value representing a displacement. This displacement is added to pc to form the target address.

For example, "jal pc+12" (see Fig. 4.11) is the sextuplet [0b0, 0b0000000110, 0b0, 0b00000000, 0b00001, 0b1101111], i.e. 0x00e000ef in hexadecimal.

In the example, d is 0b0, imm20-19:12 is 0b00000000, e is 0b0, and imm20-10:1 is 0b0000000110. The 20-bit constant is 0b00000000000000000110, i.e. 6. After the one bit left shift and the sign extension, the 32-bit constant added to pc is 12 (goto pc+12).

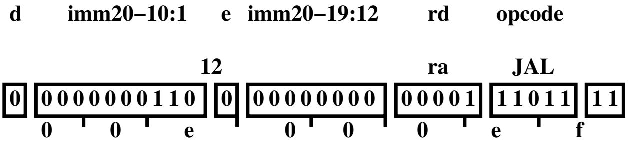


Fig. 4.11 The "jal pc+12" instruction encoding

0x03	LOAD	0x43		0x83	LOAD	0xc3	
0x07		0x47		0x87		0xc7	
0x0b		0x4b		0x8b		0xcb	
0x0f		0x4f		0x8f		0xcf	
0x13	OP_IMM	0x53		0x93	OP_IMM	0xd3	
0x17	AUIPC	0x57		0x97	AUIPC	0xd7	
0x1b		0x5b		0x9b		0xdb	
0x1f		0x5f		0x9f		0xdf	
0x23	STORE	0x63	BRANCH	0xa3	STORE	0xe3	BRANCH
0x27		0x67	JALR	0xa7		0xe7	JALR
0x2b		0x6b		0xab		0xeb	
0x2f		0x6f	JAL	0xaf		0xef	JAL
0x33	OP	0x73		0xb3	OP	0xf3	
0x37	LUI	0x77		0xb7	LUI	0xf7	
0x3b		0x7b		0xbb		0xfb	
0x3f		0x7f		0xbf		0xff	

Fig. 4.12 Decoding RV32I instructions from the least significant byte

Instruction codes ending by 0x6f or 0xef are J-TYPE “jal” instructions (opcode JAL).

4.1.3.7 Decoding RISC-V Instructions

From the hexadecimal representation of the RISC-V instructions encoding, it is easy to decode the instruction as defined by the opcode field.

Figure 4.12 matches the least significant byte value of the instruction encoding with the RV32I opcodes.

For example, an instruction code ending by 0x03 or by 0x83 is the encoding of a LOAD.

The empty fields correspond to opcodes which belong to extensions of the RV32I ISA (e.g. byte 0x07 and byte 0x87 are related to the LOAD-FP opcode defined in the F extension).

4.1.4 The Assembler Syntax

4.1.4.1 The R-TYPE Instructions

For the assembler, the R-TYPE instructions are quadruplets: (operation, destination, left source, right source).

For example, "add a0, a1, a2" means "a0 = a1+a2".

4.1.4.2 The I-TYPE Instructions

The I-TYPE computing instructions are also quadruplets: (operation, destination, left source, constant). The operation is ended by the suffix *i*.

For example, "addi a0, a1, 1" means "a0 = a1+1".

The constant should be in the $[-2^{11}, 2^{11} - 1]$ interval.

The I-TYPE load instructions are quadruplets: (load+size+sign, destination, displacement, base).

For example, "lb a0, 5(a1)" means "a0 =*(((char*)a1)+5)" (single byte load).

The displacement should be in the $[-2^{11}, 2^{11} - 1]$ interval.

The size can be byte (lb/lbu), half word (lh/lhu), or word (lw).

The sign can be empty (meaning signed) or u (meaning unsigned; only for byte (lbu) and half words (lhu)).

The I-TYPE indirect jump instructions (i.e. jalr) are quadruplets: (jalr, destination, displacement, base).

For example, "jalr ra, 4(a0)" means "ra = pc+4; pc = pc+a0+4".

The displacement should be in the $[-2^{11}, 2^{11} - 1]$ interval.

4.1.4.3 The S-TYPE Instructions

The S-TYPE store instructions are quadruplets: (store+size, source, displacement, base).

For example, "sb a0, 5(a1)" means "*(((char*)a1)+5) = a0" (single byte store).

The displacement should be in the $[-2^{11}, 2^{11} - 1]$ interval.

The size can be byte (sb), half word (sh), or word (sw).

4.1.4.4 The B-TYPE Instructions

The B-TYPE branch instructions are quadruplets: (comparison, left source, right source, target label).

For example, "beq a0, zero, .L1" means "if (a0==0) goto .L1".

If you need to compare to a constant different from 0, you should first save this constant into a temporary register (e.g. to compare to constant 10, save 10 into register t0 with instruction "li t0,10" and then compare with register t0).

The target label should be at most $2^{11} - 1$ half words after the branch instruction or 2^{11} half words before (an amplitude of 2^{11} RV32I instructions, half of them for a backward branch).

4.1.4.5 The U-TYPE Instructions

The U-TYPE instructions are triplets: (lui/auipc, destination, constant).

For example, "lui a0, 0xdead" means "a0 = 0xdead<<12" and "auipc a0, 0xdead" means "a0 = pc+(0xdead<<12)".

The constant should be in the $[-2^{19}, 2^{19} - 1]$ interval.

4.1.4.6 The J-TYPE Instructions

The J-TYPE instructions are triplets: (jal, return, target label).

For example "jal ra, foo" means "ra = pc+4; pc = pc+foo".

The target label should be at most $2^{20} - 1$ half words after the branch instruction or 2^{20} half words before (an amplitude of 2^{20} RV32I instructions, half of them for a backward jump).

4.1.4.7 The Pseudo-instructions

The assembler language adds pseudo-instructions to the base ones. These pseudo-instructions are mostly shortcuts (e.g. "ret" which stands for "jalr zero, 0(ra)").

The following list is non exhaustive:

- "nop" (shortcut for "addi, zero, zero, 0"),
- "mv rd, rs" (shortcut for "addi rd, rs, 0"),
- "li rd, imm" (according to the size of imm, the li instruction can expand into multiple basic instructions to fill the rd register with the imm value; the most simple case is when imm is in the $[-2^{11}, 2^{11} - 1]$ interval: the li instruction is turned into an addi one; for example, "li a0,1" is a shortcut for "addi a0, zero, 1"),
- "not rd, rs" (shortcut for "xori rd, rs, -1"),
- "neg rd, rs" ("sub rd, zero, rs").

For branch instructions:

- "beqz rs, label" ("beq rs, zero, label"),
- "bnez rs, label" ("bne rs, zero, label"),
- "blez rs, label" ("bge zero, rs, label"),
- "bgez rs, label" ("bge rs, zero, label"),
- "bltz rs, label" ("blt rs, zero, label"),
- "bgtz rs, label" ("blt zero, rs, label"),
- "bgt rs, rt, label" ("blt rt, rs, label"),
- "ble rs, rt, label" ("bge rt, rs, label"),
- "bgtu rs, rt, label" ("bltu rt, rs, label"),
- "bleu rs, rt, label" ("bgeu rt, rs, label").

For jump instructions:

- "j label" ("jal zero, label"),
- "jal label" ("jal ra, label"),
- "jr rs" ("jalr zero, 0(rs)"),
- "jalr rs" ("jalr ra, 0(rs)"),
- "ret" ("jalr zero, 0(ra)").

4.2 Code Examples

The best way to learn an assembler language is to use the compiler. You write a piece of C which you compile with a moderate optimization level (i.e. -O0 or -O1) and the -S option (to produce the assembly source file). Then, you just have to look at the assembly file and compare it to the C source.

4.2.1 Expressions

Compile the C piece of code shown in Listing 4.1 (all the C source files in this chapter are in the chapter_4 folder; all the compilation commands are in the compile.txt file in the same folder; the C source code to compute *delta* is in the exp.c file; the compilation command for exp.c builds exp.s).

Listing 4.1 Compiling a C expression

```
void main(){
  int a=3, b=5, c=2, delta;
  delta = b*b - 4*a*c;
}
```

The "exp.s" file produced is shown in Listing 4.2 (from the original "exp.s" file, some unimportant details have been removed and comments have been added).

Listing 4.2 An expression in RISC-V assembly language

```
      ...
main: addi sp,sp,-32   /*allocate 32 bytes on the stack*/
      sw   ra,28(sp)   /*save ra on the stack*/
      sw   s0,24(sp)   /*save s0 on the stack*/
      sw   s1,20(sp)   /*save s1 on the stack*/
      addi s0,sp,32    /*copy the stack pointer to s0*/
      li   a5,3        /*a5=3; "a" is initialized*/
      sw   a5,-20(s0)  /*place "a" on the stack*/
      li   a5,5        /*a5=5; "b" is initialized*/
      sw   a5,-24(s0)  /*place "b" on the stack*/
      li   a5,2        /*a5=2; "c" is initialized*/
      sw   a5,-28(s0)  /*place "c" on the stack*/
      lw   a1,-24(s0)  /*load "b" into a1*/
      lw   a0,-24(s0)  /*load "b" into a0*/
      call __mulsi3    /*multiply a0 by a1, result in a0*/
      mv   a5,a0       /*a5=a0 ("b*b")*/
      mv   s1,a5       /*s1=a5 ("b*b")*/
      lw   a1,-28(s0)  /*load "c" into a1*/
      lw   a0,-20(s0)  /*load "a" into a0*/
      call __mulsi3    /*multiply a0 by a1, result in a0*/
      mv   a5,a0       /*a5=a0 ("a*c")*/
      slli a5,a5,2     /*a5=a5<<2; "4*a*c"*/
      sub  a5,s1,a5    /*a5=s1-a5; "b*b-4*a*c"*/
      sw   a5,-32(s0)  /*place "b*b-4*a*c" on the stack*/
      nop              /*no operation*/
      lw   ra,28(sp)   /*restore ra*/
      lw   s0,24(sp)   /*restore s0*/
      lw   s1,20(sp)   /*restore s1*/
      addi sp,sp,32    /*free the 32 bytes from the stack*/
      jr   ra          /*return*/
```

As you can see, it is easy to read, but tedious, mostly because there are so many unnecessary movements with the memory.

This is due to the low level of optimization I have used to compile (-O0). With a higher level (even only -O1), the compiler would have simplified the code by simply doing nothing (as the delta result is not used).

I could trick the compiler to force it to compute delta by printing its value. But even in this case, the compiler would be smarter than me and simply compute delta itself (as it is a few simple operations on constants) and print the resulting constant.

However, this simple example gives a lot of informations: how to write to and read from memory, how to call a function (__mulsi3 which is provided by the standard library; the __mulsi3 function multiplies its two arguments in registers a0 and a1 and returns the product into register a0).

It also shows how to initialize (li) and move (mv) registers, to allocate and free space on the stack ("addi sp, sp, constant") and many other details.

4.2.2 Tests

Compile the C piece of code shown in Listing 4.3 (the C source file is test.c; the compilation command in the compile.txt file builds test.s).

Listing 4.3 Compiling C tests

```
void main(){
  int a=3, b=5, c=2, delta;
  delta = b*b - 4*a*c;
  if (delta<0) printf("no real solution\n");
  else if (delta==0) printf("one solution\n");
  else printf("two solutions\n");
}
```

The "test.s" file produced is shown in Listing 4.4 (I removed the computation of "b*b-4*a*c" which you can see in "exp.s").

Listing 4.4 A few tests in RISC-V assembly language

```
.LC0: .string "no real solution"
      .align  2
.LC1: .string "one solution"
      .align  2
.LC2: .string "two solutions"
      ...
main: addi    sp,sp,-32
      sw      ra,28(sp)
      sw      s0,24(sp)
      sw      s1,20(sp)
      addi    s0,sp,32
      ...                     /*the "b*b-4*a*c" computation*/
      sw      a5,-32(s0)      /*place "b*b-4*a*c" on the stack*/
      lw      a5,-32(s0)      /*load "b*b-4*a*c" into a5*/
      bge     a5,zero,.L2     /*if (b*b-4*a*c>=0) goto .L2*/
      lui     a5,%hi(.LC0)    /*a0=.LC0*/
      addi    a0,a5,%lo(.LC0)
      call    puts            /*puts("no real solution")*/
      j       .L5             /*goto .L5*/
.L2:  lw      a5,-32(s0)      /*load "b*b-4*a*c" into a5*/
```

```
        bne     a5,zero,.L4        /*if (b*b-4*a*c!=0) goto .L4*/
        lui     a5,%hi(.LC1)       /*a0=.LC1*/
        addi    a0,a5,%lo(.LC1)
        call    puts               /*puts("one solution");
        j       .L5                /*goto .L5*/
.L4:    lui     a5,%hi(.LC2)       /*a0=.LC2*/
        addi    a0,a5,%lo(.LC2)
        call    puts               /*puts("two solutions");
.L5:    nop
        lw      ra,28(sp)
        lw      s0,24(sp)
        lw      s1,20(sp)
        addi    sp,sp,32
        jr      ra
```

Apart from the branch and jump instructions, this example shows how to initialize a register with an address in the code (e.g. "a0 = .LC0"). This is done in two steps. First, the register is set with the upper part of the address ("lui a5,%hi(.LC0)"; %hi is a directive to the assembler which extracts the 20 upper bits of the .LC0 address). Second, the lower part is added ("addi a0,a5,%lo(.LC0)"; %lo extracts the 12 lower bits of the .LC0 address).

There are also examples of .align and .string assembler directives. Any word starting with a dot is either a label (if it starts a line) or an assembler directive (if it does not start a line).

Assembler directives are not RISC-V instructions. The assembler builds a text section (i.e. a future code memory) with the hexadecimal values of the instructions. It also builds a data section with the data values found in the text (e.g. what follows the .LC0 label).

The ".align 2" directive aligns the next construction (either a RISC-V instruction or some data) on the next even address (i.e. it moves one byte forward if the current construction ends on an odd address, otherwise it is ineffective).

The .string directive builds a data in the data section from the string which follows.

4.2.3 Loops

The program shown in Listing 4.5 computes the 10th term of the Fibonacci sequence.

Listing 4.5 Compiling C for loop

```
#include <stdio.h>
void main(){
  int i, un, unm1=1, unm2=0;
  for (i=2; i<=10; i++){
    un   = unm1+unm2;
    unm2 = unm1;
    unm1 = un;
  }
  printf("fibonacci(10)=%d\n",un);
}
```

Compile with the -O1 optimization level (the C source file is loop.c; the compilation command in the compile.txt file builds loop.s). The RISC-V assembly code produced by the compilation is shown in Listing 4.6.

Listing 4.6 A for loop in RISC-V assembly language

```
       ...
.LC0:  .string  "fibonacci(10)=%d\n"
       ...
main:  addi  sp,sp,-16      /*allocate*/
       sw    ra,12(sp)      /*save ra in the stack*/
       li    a5,9           /*number of iterations n: 10-2+1=9*/
       li    a3,0           /*unm2=0*/
       li    a4,1           /*unm1=1*/
       j     .L2            /*goto .L2*/
.L3:   mv    a4,a1          /*unm1=un*/
.L2:   add   a1,a4,a3       /*un=unm1+unm2*/
       addi  a5,a5,-1       /*n--*/
       mv    a3,a4          /*unm2=unm1*/
       bne   a5,zero,.L3    /*if (n!=0) goto .L3*/
       lui   a0,%hi(.LC0)   /*a0=.LC0*/
       addi  a0,a0,%lo(.LC0)
       call  printf         /*printf("fibonacci(10)=%d\n",un)*/
       lw    ra,12(sp)      /*restore ra*/
       addi  sp,sp,16       /*free*/
       jr    ra
```

A loop is a backward branch (i.e. "bne a5,zero,.L3").

When the optimization level is over 0, the unnecessary movements with the stack disappear.

while and do ... while loops are compiled the same way. For while loops, a first test handles the no iteration case. Then a do ... while loop is built.

4.2.4 Function Calls

Compile the code shown in Listing 4.7 with optimization level 1 and build the assembly file (the C source file is fib.c; the compilation command builds the fib.s file).

Listing 4.7 Compiling C function calls

```
#include <stdio.h>
unsigned int fibonacci(unsigned int n){
  unsigned int i, un, unm1=1, unm2=0;
  if (n==0) return unm2;
  if (n==1) return unm1;
  for (i=2; i<=n; i++){
    un   = unm1+unm2;
    unm2 = unm1;
    unm1 = un;
  }
  return(un);
}
void main(){
  printf("fibonacci(0)=%d\n",fibonacci(0));
  printf("fibonacci(1)=%d\n",fibonacci(1));
  printf("fibonacci(10)=%d\n",fibonacci(10));
  printf("fibonacci(11)=%d\n",fibonacci(11));
  printf("fibonacci(12)=%d\n",fibonacci(12));
}
```

The RISC-V assembly translation of the fibonacci function is shown in Listing 4.8.

Listing 4.8 Function calls in RISC-V assembly language: the fibonacci function

```
fibonacci:
      mv    a3,a0         /*copy n*/
      beq   a0,zero,.L1   /*if (n==0) goto .L1*/
      li    a5,1
      beq   a0,a5,.L1     /*if (n==1) goto .L1*/
      li    a2,0          /*unm2=0*/
      li    a4,1          /*unm1=1*/
      li    a5,2          /*i=2*/
      j     .L3           /*goto .L3*/
.L6:  mv    a4,a0         /*unm1=un*/
.L3:  add   a0,a4,a2      /*un=unm1+unm2*/
      addi  a5,a5,1       /*i++*/
      mv    a2,a4         /*unm2=unm1*/
      bgeu  a3,a5,.L6     /*if (n>=i) goto .L6*/
      ret                 /*return un*/
.L1:  ret                 /*return n*/
      .align  2
.LC0: .string "fibonacci(0)=%d\n"
      .align  2
.LC1: .string "fibonacci(1)=%d\n"
      .align  2
.LC2: .string "fibonacci(10)=%d\n"
      .align  2
.LC3: .string "fibonacci(11)=%d\n"
      .align  2
.LC4: .string "fibonacci(12)=%d\n"
      ...
```

The RISC-V assembly translation of the main function is shown in Listing 4.9.

Listing 4.9 Function calls in RISC-V assembly language: the main function

```
      ...
main: addi  sp,sp,-16
      sw    ra,12(sp)
      li    a0,0       /*n=0*/
      call  fibonacci  /*fibonacci(n)*/
      mv    a1,a0      /*a1=fibonacci(n)*/
      lui   a0,%hi(.LC0)
      addi  a0,a0,%lo(.LC0)
      call  printf  /*printf("fibonacci(0)=%d\n",fibonacci(n))*/
      li    a0,1       /*n=1*/
      call  fibonacci  /*fibonacci(n)*/
      mv    a1,a0      /*a1=fibonacci(n)*/
      lui   a0,%hi(.LC1)
      addi  a0,a0,%lo(.LC1)
      call  printf  /*printf("fibonacci(1)=%d\n",fibonacci(n))*/
      li    a0,10      /*n=10*/
      call  fibonacci  /*fibonacci(n)*/
      mv    a1,a0      /*a1=fibonacci(n)*/
      lui   a0,%hi(.LC2)
      addi  a0,a0,%lo(.LC2)
      call  printf  /*printf("fibonacci(10)=%d\n",fibonacci(n))*/
      li    a0,11      /*n=11*/
      call  fibonacci  /*fibonacci(n)*/
      mv    a1,a0      /*a1=fibonacci(n)*/
      lui   a0,%hi(.LC3)
      addi  a0,a0,%lo(.LC3)
      call  printf  /*printf("fibonacci(11)=%d\n",fibonacci(n))*/
```

```
        li    a0,12     /*n=12*/
        call  fibonacci /*fibonacci(n)*/
        mv    a1,a0     /*a1=fibonacci(n)*/
        lui   a0,%hi(.LC4)
        addi  a0,a0,%lo(.LC4)
        call  printf /*printf("fibonacci(12)=%d\n",fibonacci(n))*/
        lw    ra,12(sp)
        addi  sp,sp,16
        jr    ra
```

The calls apply the ABI constraints, i.e. transmit arguments through the a0-a7 registers, starting with a0 and return their result in a0.

There is much more to learn about assembly programming but this is enough to be able to build a processor.

References

1. https://riscv.org/specifications/isa-spec-pdf/
2. https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf
3. D. Patterson, A. Waterman: *The RISC-V Reader: An Open Architecture Atlas* (Strawberry Canyon, 2017)

5 Building a Fetching, Decoding, and Executing Processor

Abstract

This chapter prepares the building of your first RISC-V processor. First, a fetching machine is implemented. It is only able to fetch successive words from a code memory. Second, the fetching machine is upgraded to include a decoding mechanism. Third, the fetching and decoding machine is completed with an execution engine to run computation and control instructions, but not yet memory accessing ones.

5.1 General Programming Concepts for HLS

5.1.1 The Critical Path

If you are familiar with classic programming in C/C++, you will find surprising methods in HLS programming.

In classic programming, you improve your code either to save time (I mean *execution time*, i.e. running the program faster) or (but not excluding) to save space, i.e. use less memory words.

The piece of code in Listing 5.1 shows a loop representing the simulation of a processor. The loop iteration is composed of four functions representing successive phases in the processing of one instruction: fetch the instruction from the code memory, decode it, execute it, and set an *is_running* continuation condition.

It is a do ... while loop as the loop has to be run at least once and as it terminates depending on the processed instruction.

Listing 5.1 A basic processor simulation loop

```
do{
  fetch(pc, code_ram, &instruction);
  decode(instruction, &d_i);
  execute(pc, reg_file, data_ram, d_i, &pc);
```

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_5

```
    running_cond_update(instruction, reg_file, &is_running);
  } while (is_running);
```

In classic programming, the execution time of such a loop is the sum of the execution times of all the iterations.

For a given program to be simulated by the loop, the number of iterations cannot be changed. It is equal to the number of simulated instructions, i.e. the number of instructions of the program run. Hence, the only way to improve the execution time of the simulation is to speed up at least one iteration by improving the code of any of the four functions.

The simulation loop can be used in an HLS tool to define an IP. In this case, each iteration of the do ... while loop represents a *cycle* of the IP.

This is where the first very fundamental difference between a software simulator and an IP defined by an HLS program resides.

In an IP, all the iterations have the same IP cycle time, defined as the *critical path*, i.e. the longest delay to cross any path of successive gates from the cycle start (i.e. the iteration start) to the cycle end (i.e. the iteration end).

When looking at the iteration which successively calls the four functions, it seems that the delay is the same for all the iterations. However, the *execute* function is coded to be able to execute any instruction of the processor instruction set. An addition instruction does not use the same computing unit than a logical *and* for example. The delay of an adder is probably much longer than the delay of a simple AND gate (in this matter, hardware differs from software: in C, the "$a = b$ & c" expression is considered as time equivalent to the "$a = b + c$" expression).

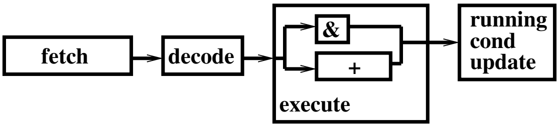
Figure 5.1 shows two different paths, one crossing an AND gate and the other crossing an adder. The *execute* unit has many other unshown paths to take care of all the other instructions in the ISA. For the two shown paths, the longest delay is the adder one. Hence, the critical path will be at least as long as the path crossing the adder.

For the IP, the run time of any instruction fits in the critical path, the logical *and* as well as the addition one.

Improving the IP requires to shorten the critical path, i.e. to improve the execution time of the longest iteration. Comparatively, improving the execution time of any iteration improves the software simulation time.

In the software simulator, reducing the execution time of one iteration benefits the overall execution time. In the IP, reducing the delay of all the paths excepting the critical one does not improve the IP cycle at all.

Fig. 5.1 Two paths with different delays



5.1.2 Computing More to Reduce the Critical Path

A second fundamental difference is illustrated by the piece of code in Listing 5.2 in which a computation is conditioned. In software simulation, conditioning the dosomething function execution helps to improve the overall computation time (each time the condition is false the dosomething computation time is saved). Hence, in software simulation, computing less improves the execution time.

Listing 5.2 Conditional computations

```
if (condition) dosomething();
```

In the IP, the "*if (condition)*" may lengthen the critical path, hence degrade the cycle. Removing it changes the semantic of course. But this change can have no impact on the IP behaviour. This is the case if the dosomething function has only a local impact on the current iteration but no global impact on the next iterations and on the computation after the loop exit.

In software simulation, the condition should not be removed because its elimination would degrade the performance. In the IP, the "*if (condition)*" removal should be considered. It implies computing more (i.e. uselessly computing "*dosomething*" when the "*condition*" is false). But in hardware, computing more can improve the cycle time. Including useless computations is a good choice if it helps to shorten the critical path.

5.1.3 Parallel Execution

A third difference is illustrated by the way loops are executed. In software simulation, there is no parallelism. Iterations are run one after the other. In an IP, a for loop with a static number of iterations is unrolled.

That means, an IP runs all the iterations of the loop in Listing 5.3 in parallel (dosomething(0) ... dosomething(15)) and the iteration counter i is eliminated (no "i++" update, no "$i < 16$" test).

Listing 5.3 for loop computation

```
for (i=0; i<16; i++)
  dosomething(i);
```

In an IP, parallelism also concerns the way functions are run, as illustrated in Listing 5.4. If functions f and g are independent (i.e. they could be permuted without changing the result, which is true if and only if f does not modify any variable used by g and conversely, g does not modify any variable used by f), they are implemented to be run in parallel in the IP but still sequentially called in the software simulation.

Listing 5.4 Independent functions

```
f(...);
g(...);
```

Adding an instruction to a computation increases the simulation time. On the FPGA though, such an added instruction may not change the critical path, because of the parallelism in the hardware.

For example, Listing 5.5 shows a fill function to fill the three fields of a structured variable given as an argument s.

Listing 5.5 Independent instructions

```
void fill(struct *s){
s->f1=...;
s->f2=...;
s->f3=...;
}
```

The execution time of the fill function simulation is the cumulated times of the three fields settings. The execution time on the FPGA is either independent of the fill function (if its run is not in the critical path) or related only to the longest of the three field value computations if these computations are independent with one another.

If you add a fourth field to the structure, the simulation time is increased but the IP critical path is probably not impacted (it is impacted only if the fill function run is in the critical path and if the new field computation is longer than the three other ones).

5.2 The Fundamental Execution Time Equation for a Processor

The execution time of a program applied to some data on a processor is given by the number of cycles of the run multiplied by the cycle duration. To highlight how the ISA and the processor implementation can impact the performance, this equation to compute the execution time can be further detailed by using three terms instead of two.

The execution time (in seconds) of a program P applied to a data D on a CPU C is the multiplication of three values: the number of machine instructions run (*nmi(P, D, C)*), the average number of processor cycles to run one instruction (*cpi(P, D, C)*, number of Cycles Per Instruction or CPI) and the duration (in seconds) of a cycle (*c(C)*), as formalized in Eq. 5.1.

$$time(P, D, C) = nmi(P, D, C) * cpi(P, D, C) * c(C) \tag{5.1}$$

To improve the overall performance of a CPU C, its cycle *c(C)* can be decreased. This improvement can be obtained by shortening the critical path either through a microarchitectural improvement (i.e. eliminating some gates on the path), or through a technological improvement (i.e. reducing the size of the transistors to decrease the delay to cross the critical path because of its reduced length). It can also be achieved by increasing the voltage to increase the processor component frequency through overclocking.

The performance is improved only if in the same time the number of cycles of the run is not increased (if we assume that the number of instructions run *nmi(P, D, C)* is unchanged, an increase of the number of cycles comes from an increase of the CPI).

On the FPGA, we cannot reduce the size of the transistors of course. It is possible to increase the frequency of the programmable part of the FPGA but I will not investigate this possibility in the book. Hence, the only way to improve the designs is to eliminate gates on the critical path through microarchitectural improvements.

Throughout the book, I will present successive implementations of a RISC-V processor. I will apply Eq. 5.1 to a set of program examples and compare the computed execution times to illustrate the improvements.

5.3 First Step: Build the Path to Update PC

Now that you know how to run an IP on a development board (Chap. 2), I will assume you are able to test all the codes given in the book. They have all been tested on my Pynq-Z1 board.

The code presented in the book is not the full code. Only the parts which deserve some explanations are presented in the listings throughout the book.

Each IP project is defined as a subfolder in the goossens-book-ip-projects/2022.1 folder. Each subfolder contains the full code of the corresponding IP project, which I recommend you to have a look at.

For example, the three IP projects presented in this chapter, i.e. fetching_ip, fetching_decoding_ip, and fde_ip, have their full code in the three folders with the same names.

Moreover, in each IP folder you will find prebuilt Vitis_HLS projects to help you simulate the proposed IPs without typing the code. You will also find predesigned Vivado projects including the Vitis_HLS prebuilt IPs. In these Vivado projects you will find pregenerated bitstreams.

Hence, to test the processor IPs defined in the book, you only have to set the helloworld drivers in the Vitis IDE workspaces from helloworld.c files included in the IP folders, as explained in Sect. 2.7.

5.3.1 The Fetching_ip Design

A processor is made of three complementary paths: the path to update the program counter (pc), the path to update the register file, and the path to update the data memory.

The fetching_ip design concentrates on the first path. Its duty is also to read the instruction addressed by pc from the code memory. In parallel, a new instruction is fetched (i.e. is read) and the new pc is computed.

A fetching IP is a component which reads instructions from a memory containing instruction words. The fetching process continues while a continuation condition keeps true. This continuation condition should be related to the fetched instructions themselves.

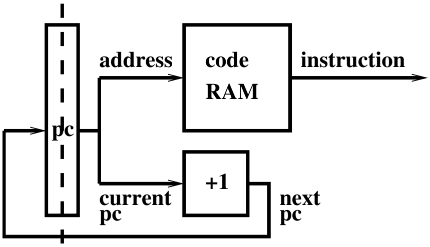


Fig. 5.2 The fetching IP design

In the fetching_ip design, I will define that condition as: "*the fetched instruction is not a RET*" (RET is the return from function pseudo-instruction).

The fetching IP moves along the memory. It starts from an initial address and moves forward (during each IP cycle, the address is incremented by the size of the fetched instruction, i.e. four bytes or one word as I restrict the recognized ISA to the RV32I set, excluding the RVC standard extension devoted to compressed 16-bit instructions).

Figure 5.2 shows the hardware I will implement in the fetching_ip design. The leftmost vertical rectangle represents a separation register, which is a clocked circuit. The dotted line represents the separation between the register input and the output.

At the beginning of each IP cycle, the present state at the left of the dotted line is copied to the right side. During the cycle, the left and the right sides are isolated from each other: what is computed after the "+1" square, labeled next pc, is driven to the input of the pc separation register but does not cross the dotted line before the next cycle starts.

The code RAM box is a memory circuit. The addressed word is copied to the instruction output.

The "+1" box is an adder circuit. It outputs the incremented value of its input.

5.3.2 The Fetching_ip Top Function

All the source files related to the fetching_ip can be found in the fetching_ip folder.

5.3.2.1 The Top Function Prototype

The code in Listing 5.6 is a part of the fetching_ip.cpp file showing the fetching_ip top function prototype.

The IP has a pinout described as the arguments of the top function.

The start_pc argument is an input which should contain the start address of the run (in this design, the initial value for pc at processor power up can be externally set).

The code_ram argument is a pointer (in C, an array is a constant pointer) used to address the external memory.

Listing 5.6 The fetching_ip top function prototype

```
void fetching_ip(
  unsigned int start_pc,
  unsigned int code_ram[CODE_RAM_SIZE]){
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=code_ram
#pragma HLS INTERFACE s_axilite port=return
  ...
```

The pinout of the top function is associated to a set of INTERFACE pragmas (one pragma for each argument).

HLS pragmas are informations for the synthesizer, as mentioned in Sect. 2.5.4. They are ignored by the C compiler when the code is compiled before running an IP simulation. The synthesizer uses the pragmas to build his translation to RTL.

The INTERFACE pragmas are used by the synthesizer to organize the way the IP is connected to its surrounding. They offer many protocols to exchange data from/to the IP. I will progressively introduce these different protocols.

The axilite protocol used in the fetching_ip is related to the AXI communication interface (I present the AXI interface in Chap. 11).

The s_axilite name designates a slave, i.e. the fetching_ip component is a slave device of the AXI interconnect. A slave receives requests (i.e. reads or writes) from masters and serves them.

An IP using an AXI interconnection interface can be connected to an AXI interconnect IP which links IPs together and allows them to communicate.

In the preceding chapter, I have connected the adder_ip to the Zynq7 Processing System IP through an AXI interconnect IP (in fact, the interconnection was done automatically by Vivado). Later in this chapter, I connect the fetching_ip the same way.

The top function arguments using the s_axilite interface should be typed with the C/C++ basic integer types (i.e. int, unsigned int, char, short and pointers).

You should not use the template types provided by Vitis through the ap_int.h header file (these types are presented in Sect. 5.3.2.4). The reason is that the AXI interface uses a standard 32-bit bus with byte enable, hence IP arguments should have a one, two or four bytes width.

To run the fetching_ip, its input arguments should be set, i.e. the start_pc variable with the starting point of the code to be run and the code_ram memory with the RISC-V code itself.

The Zynq IP sends these initial values to the fetching_ip through the AXI interconnect IP as shown in Fig. 5.3.

The Zynq IP writes to the AXI interconnect bridge. The write base address identifies the fetching_ip component. The write address offset identifies the fetching_ip top function argument, i.e. either start_pc or code_ram. Multiple writes are necessary to initialize the fetching_ip component.

For example, to initialize the start_pc argument as address 0, the Zynq IP writes word 0 to address *(fetching_ip + start_pc).

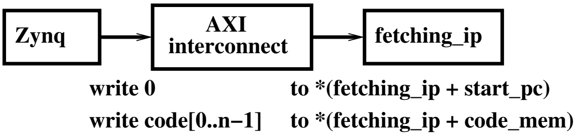


Fig. 5.3 Writing the fetching_ip input arguments through the AXI interconnect IP

5.3.2.2 The Top Function Do ... While Loop

The fetching_ip top function contains a do ... while loop, shown in Listing 5.7.

Listing 5.7 The fetching_ip top function main loop

```
  ...
  code_address_t pc;
  instruction_t  instruction;
  bit_t          is_running;
  pc = start_pc;
  do{
#pragma HLS PIPELINE off
    fetch(pc, code_ram, &instruction);
    execute(pc, &pc);
    running_cond_update(instruction, &is_running);
  } while (is_running);
}
```

Each iteration represents the IP cycle, not to be mixed with the FPGA cycle.

The FPGA cycle is set to 10 ns (100 MHz) in my designs (this is automatically done by the Vitis synthesizer).

The fetching IP cycle is the time needed to run one iteration of the main loop. It is computed by the synthesizer while it places the hardware gates and circuits to implement the code and compute their delays.

If the iteration critical path fits in the FPGA cycle, then the IP cycle is equal to the FPGA cycle.

In the fetching IP, even though it is still very modest, the iteration critical path will not fit in the FPGA cycle, mainly because the fetch function accesses a memory (the code_ram array, implemented as a memory, and holding the RISC-V code to fetch). Such an access takes longer than 10ns.

In the beginning of the do ... while loop, the HLS PIPELINE off pragma informs the synthesizer that it should fully separate two successive iterations (no overlapping).

The loop iterates a non static number of times as the number of iterations depends on the fetched instruction. The loop control was not apparent in Fig. 5.2. The synthesizer will build this control.

Any loop style is acceptable for the main IP loop: for, while and do ... while. However, other embedded loops should only be for ones with a static number of iterations (i.e. loops that you could statically unroll).

5.3.2.3 The Top Function Header File

The fetching_ip.h file (see Listing 5.8) contains the definitions of the constants (the code memory size and the RET instruction encoding) and types used in the code. It exports the prototype of the fetching_ip top function, which is used in the testbench file.

The code memory size is defined as 2^{16} instructions, i.e. 2^{18} bytes (256 KB) (if the RISC-V processor implementations are to be tested on a Basys3 development board, this size should be reduced to 64 KB because the XC7A35T FPGA it uses has only 200 KB of RAM: set LOG_CODE_RAM_SIZE to 14 in all the designs).

Listing 5.8 The fetching_ip.h file

```
#ifndef __FETCHING_IP
#define __FETCHING_IP
#include "ap_int.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
#define RET               0x8067
typedef unsigned int                instruction_t;
typedef ap_uint<LOG_CODE_RAM_SIZE> code_address_t;
typedef ap_uint<1>                  bit_t;
void fetching_ip(
  unsigned int start_pc,
  unsigned int code_ram[CODE_RAM_SIZE]);
#endif
```

5.3.2.4 The Ap_uint<size> and Ap_int<size> Template Types

The Vitis_HLS environment offers the ap_<size> and ap_int<size> template types (accessible through the ap_int.h header file). They are C++ parameterized types to define bit fields of any size (in bits; any width between 1 bit and 1024 bits). The ap_<size> type is unsigned and the ap_int<size> type is signed.

In the fetching_ip.h file, I define the bit_t type which is a single unsigned bit (ap_uint<1>).

All your variables should be typed with the exact needed bit size. One bit more than needed creates unnecessary wires and logic, i.e. wastes resources. One bit less than necessary leads to bugs.

For example, the code_address_t type is an unsigned bit field having the exact necessary width to address the code memory (i.e. LOG_CODE_RAM_SIZE bits to address a CODE_RAM_SIZE instructions memory) (LOG_CODE_RAM_SIZE is defined as log_2(CODE_RAM_SIZE)).

A RISC-V instruction has the 32-bit type instruction_t which is basically an unsigned int.

5.3.2.5 Global Variables

You may have noticed that the code defining the fetching_ip does not use any global variables.

As a general rule, I never use global declarations. All the variables in a function are either arguments or locals. Vitis_HLS does not forbid the usage of globals but I recommend to avoid them.

The reason is that once you globally declare something, you have a strong temptation to use it everywhere you need it. In HLS code there might appear some difficulty for the synthesizer. If there are multiple places in the code where a variable is written, the synthesizer has to handle false dependencies between the reads and the writes and might conservatively serialize accesses which may be in fact independent.

A good policy is to have a local only visibility on every variable, extend this visibility with argument transmission and, as much as possible, restrict writing accesses to a single point in the code.

Avoid for example to write something into argument v of function f and something else into argument w of function g with v and w sharing the same address. Even if the two updates are exclusive from each other, the synthesizer might not be able to see that the two writes may not happen in the same cycle. The result would be an unnecessary serialization of the two functions to perform one write after the other.

5.3.2.6 Parallelism

The fetching IP top function successively calls three functions: fetch, execute, and running_cond_update.

The fetch function reads an instruction, the execute function computes the next pc, and the running_cond_update function computes the is_running condition to continue or end the do ... while loop.

This succession of function calls leads to a sequential run according to the C semantic and it is how the IP will be simulated. In hardware however, the gates and circuits implementing the functions are linked together if they have some producer/-consumer dependencies between their arguments.

When two functions are independent, they are run in parallel. It is so for all the instructions implementing an IP.

While the synthesizer implements your code, it will point out the computation independencies, hence their parallelism. When you think of your code, you should always keep in mind the hardware reordering of the computations.

The Schedule Viewer of the Vitis_HLS tool will help you to visualize when an operation is done, how long it lasts and what it is depending on.

5.3.3 The Fetch Function

The fetch.cpp file is shown in Listing 5.9.

In the fetch function, the fetch itself is a read from the code_ram array.

This is one of the nice things when it comes to using HLS tools: to access memory, just access the C array representing the memory. As a general rule, each time you need to access to some hardware unit, just use a C variable or a C operator and let the synthesizer transform your C code into hardware.

For example, if you want to compute a product between integer values, just use the C multiplication operator on two integer variables (e.g. a*b). The synthesizer will miraculously build the necessary integer multiplier in the FPGA. If you want to multiply two floating-point numbers, just type the variables as float or double and the synthesizer will use a floating-point multiplier instead of an integer one.

Listing 5.9 The fetch.cpp file

```
#include "debug_fetching_ip.h"
#include "fetching_ip.h"
#ifndef __SYNTHESIS__
#ifdef DEBUG_FETCH
#include <stdio.h>
#endif
#endif
void fetch(
  code_address_t pc,
  instruction_t *code_ram,
  instruction_t *instruction){
#pragma HLS INLINE off
  *instruction = code_ram[pc];
#ifndef __SYNTHESIS__
#ifdef DEBUG_FETCH
  printf("%04d: %08x\n", (int)(pc<<2), *instruction);
#endif
#endif
}
```

5.3.3.1 The PC Register

Memories are addressed by byte. For example, you can address a memory to read the byte at address 1005, or 0x3ed.

However, the code memory only contains instructions, which are 32-bit words. Hence, the fetch function reads the code memory by words, i.e. four aligned bytes (e.g. fetch the instruction at the word aligned address 1004, which is composed of the four bytes 1004–1007, as illustrated in Fig. 5.4: the red addresses are word aligned, the green addresses are not).

The pc register addresses the code memory which is defined as a CODE_RAM_SIZE words memory, i.e. 4*CODE_RAM_SIZE bytes.

Hence, the pc register should have a width of LOG_CODE_RAM_SIZE+2 bits.

To fetch, the pc register should point on an instruction, i.e. on a four bytes aligned word (0, 4, 8, ...). Such a word aligned pointer is always ended by two cleared low order bits (e.g. address 0x3ec or 0b0011 1110 1100, as shown in Fig. 5.4).

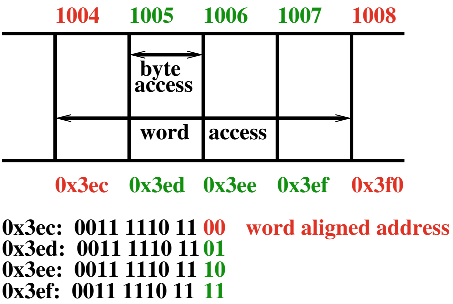


Fig. 5.4 Accessing a byte and accessing an aligned word

To minimize the number of flip-flops used to hold the pc register, the two cleared low-order bits are kept implicit. Thus, the pc register has a width of LOG_CODE_RAM_SIZE bits (it is a word pointer; the code_address_t type is defined as ap_uint<LOG_CODE_RAM_SIZE> in the fetching_ip.h file; see Listing 5.8).

5.3.3.2 The Debugging Flags

The debug_fetching_ip.h file is shown in Listing 5.10.

It is important to add some debugging tools in a Vitis_HLS project. They are run during the simulation, but are ignored during the synthesis. In general, the best debugging tools are built to survive to the debugging phase and to provide the user with useful informations on the run.

The first debugging tool outputs the trace of the fetched instructions.

The printf function in Listing 5.9 left shifts the pc register of two positions to reconstitute a byte pointer with LOG_CODE_RAM_SIZE+2 bits. Hence, the debug trace prints byte addresses, which is customary.

In the printf function, I have casted the type of pc to int (its base type is code_address_t, i.e. ap_uint<LOG_CODE_RAM_SIZE> or ap_uint<16>). You should always do so for ap_uint based types as printf would print arbitrary values for the missing bits. The cast to int forces 0 into the 16 upper bits of the 32-bit word produced.

Listing 5.10 The debug_fetching_ip.h file

```
#ifndef __DEBUG_FETCHING_IP
#define __DEBUG_FETCHING_IP
//comment the next line to turn off
//fetch debugging prints
#define DEBUG_FETCH
#endif
```

5.3.3.3 The __SYNTHESIS__ Constant

Debugging tools should be excluded from the synthesis as your IP is not supposed to print anything.

Vitis_HLS gives you the possibility to quickly turn on/off the debug prints by surrounding your prints with a #ifndef applied to the Vitis_HLS __SYNTHESIS__ constant. This constant is implicitly defined every time you start a synthesis and undefined otherwise (see an example of the usage of the __SYNTHESIS__ constant at the end of Listing 5.9).

5.3.3.4 The INLINE Pragma

In the fetch function, the INLINE pragma is an indication to the synthesizer to inline the function, i.e. to eliminate the transmission of the arguments before the call and of the results after the return. If however we do not want to inline, the INLINE off pragma is added at the start of the function, as shown on the fetch code in Listing 5.9.

When inlining is off, the function can be thought of a component inside the component in which it is called. The design gets more readable as a component (e.g. in the Schedule Viewer), abstracting the function. But the synthesizer uses resources to implement the arguments transmission, so usually the design uses more resources (e.g. LUTs and FFs) with INLINE off and might be slower (i.e. requires more FPGA cycles to fit the IP critical path).

5.3.4 The Execute Function

The execute function is defined in the execute.cpp file and shown in Listing 5.11. It only computes the next pc.

Listing 5.11 The execute function

```
void execute(
  code_address_t  pc,
  code_address_t *next_pc){
#pragma HLS INLINE off
  *next_pc = compute_next_pc(pc);
}
```

The compute_next_pc function is also defined in the execute.cpp. It is shown in Listing 5.12. It corresponds to the "+1" box in Fig. 5.2.

As pc has been defined as a word pointer (see Sect. 5.3.3.1), the next_pc computation adds 1 to pc. This saves resources (smaller adder, less interconnecting wires ...).

Listing 5.12 The compute_next_pc function

```
static code_address_t compute_next_pc(
  code_address_t pc){
#pragma HLS INLINE
  return (code_address_t)(pc + 1);
}
```

The next_pc variable always points to the next instruction in the code memory. The fetching IP reads successive addresses. In the fetching_decoding_ip design presented in Sect. 5.4, I will add the possibility to change the control flow through branches and jumps.

5.3.5 The IP Running Condition

At the end of the do ... while loop (see Listing 5.7), the running_cond_update function sets the loop continuation condition.

For a processor, this should correspond to the last instruction run, i.e. the return from the main function.

In the fetching_ip, I stop the run when the first RET instruction is met.

The running_cond_update function is defined in the fetching_ip.cpp file and shown in Listing 5.13.

Listing 5.13 The running_cond_update function

```
static void running_cond_update(
  instruction_t instruction,
  bit_t         *is_running){
#pragma HLS INLINE off
  *is_running = (instruction != RET);
}
```

5.3.6 The IP Simulation with the Testbench

Experimentation

To simulate the fetching_ip, in Vitis_HLS select **Open Project**, navigate to the fetching_ip/fetching_ip folder and click on **Open** (the folder to open is the second fetching_ip in the hierarchy; the opened folder contains a .apc folder). Then in the Vitis_HLS **Explorer** frame, right-click on **TestBench**, **Add Test Bench File**, and add the testbench_fetching_ip.cpp file in the fetching_ip folder. In the **Flow Navigator** frame, click on **Run C Simulation** and OK. The result of the run should print in the fetching_ip_csim.log tab.

The testbench_fetching_ip.cpp file (see Listing 5.14) contains the main function to simulate this IP. The main function calls the fetching_ip top function which runs the RISC-V code placed in the code_ram array.

The array is initialized with a #include directive to the preprocessor. The test_op_imm_0_text.hex hexadecimal file to be included should have been built as explained in Sect. 3.1.3, with the objcopy and the hexdump commands applied to the test_op_imm_0.elf file (this file is the result of the text based 0 compilation of test_op_imm.s).

Listing 5.14 The testbench_fetching_ip.cpp testbench file

```
#include <stdio.h>
#include "fetching_ip.h"
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_op_imm_0_text.hex"
};
int main(){
  fetching_ip(0, code_ram);
  printf("done\n");
  return 0;
}
```

The main function must have an int result (void is not permitted by Vitis_HLS).

In the fetching_ip project, the test_op_imm program is considered as an arbitrary set of RISC-V instructions ended by a RET instruction.

5.3.7 The Simulation Prints

The run prints according to the debugging flags defined.

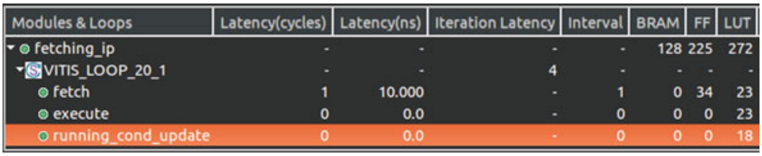
If the DEBUG_FETCH flag is defined in the debug_fetching_ip.h file, the simulation prints what is shown in Listing 5.15.

Listing 5.15 The simulation run output

```
0000: 00500593
0004: 00158613
0008: 00c67693
0012: fff68713
0016: 00576793
0020: 00c7c813
0024: 00d83893
0028: 00b83293
0032: 01c81313
0036: ff632393
0040: 7e633e13
0044: 01c35e93
0048: 41c35f13
0052: 00008067
done
```

5.3.8 The Fetching_ip Synthesis

When the simulation is successful, the synthesis can be done (**Run C Synthesis**). Figure 5.5 shows the synthesizer report. The BRAM column gives the number of BRAM blocks used in the FPGA (128 are used; 128 × 36 Kb or 4 KB blocks out of 140, to map the fetching_ip top function arguments among which the 256 KB code_ram). The FF column gives the number of Flip-Flops used by the implemented logic (225 out of 106,400). The LUT column gives the number of LUTs used (272 out of 53,200).



Modules & Loops	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	BRAM	FF	LUT
fetching_ip	-	-	-	-	128	225	272
VITIS_LOOP_20_1	-	-	4	-	-	-	-
fetch	1	10.000	-	1	0	34	23
execute	0	0.0	-	0	0	0	23
running_cond_update	0	0.0	-	0	0	0	18

Fig. 5.5 The fetching_ip synthesis report

Figure 5.6 shows the schedule. The VITIS_LOOP_20_1 (i.e. the do ... while loop; so named by the synthesizer because it starts at line 20 in the source code) has a latency of three FPGA cycles. Hence, the processor cycle is 30 ns (33 MHz).

The schedule shows that the functions fetch and execute are run in parallel (cycles 1 and 2 for fetch, cycle 2 for execute).

In the upper left part of the window, select the **Module Hierarchy** tab. Expand **VITIS_LOOP_20_1**. Click on **fetch** to display the fetch function schedule and click on the **code_ram_addr(getelementptr)** line.

In the fetch function schedule (Fig. 5.7), the code memory is loaded (code_ram_load(read)) during cycles 0 and 1. After one and a half FPGA cycle, the instruction is obtained (cycle 1).

If you click on the **Properties** tab in the bottom central frame, you get informations on the signal you are viewing, as shown in Fig. 5.8.

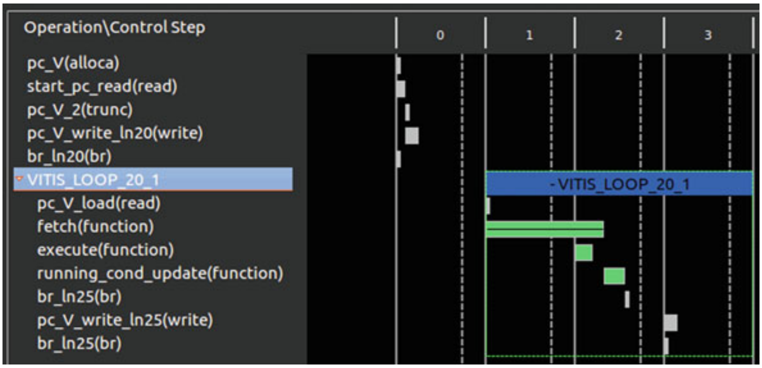


Fig. 5.6 The fetching_ip schedule

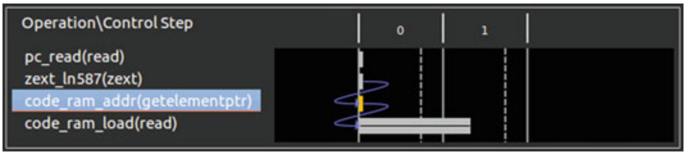
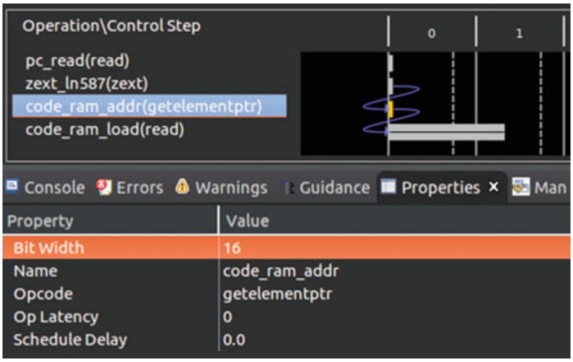


Fig. 5.7 The fetch function schedule



Property	Value
Bit Width	16
Name	code_ram_addr
Opcode	getelementptr
Op Latency	0
Schedule Delay	0.0

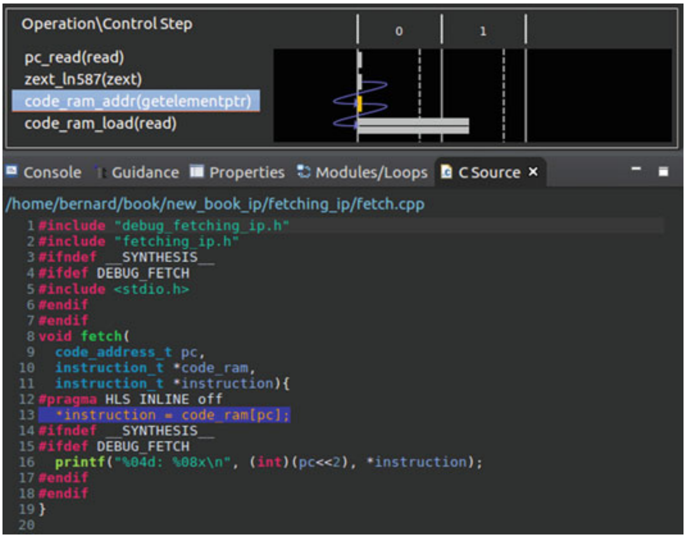
Fig. 5.8 The code_ram_addr variable: dependency, bit width

The code_ram_addr(getelemptr) bus is 16 bits wide (Bit Width field in the **Properties** tab).

Click on **code_ram_addr(getelementptr)** to select it. Then, you right-click on it and validate **Go to source**.

The **Code Source** tab opens and displays the code of the fetch function in the fetch.cpp file (see Fig. 5.9). Line 13 is highlighted which indicates that it is the source of code_ram_addr(getelementptr), i.e. "*instruction = code_ram[pc]".

In the execute function schedule (Fig. 5.10), the pc argument is read at cycle 0, i.e. the second cycle of the do ... while loop iteration (pc_read(read) in Fig. 5.10) and incremented in the same cycle (add_ln232(+)).



/home/bernard/book/new_book_ip/fetching_ip/fetch.cpp

```
 1 #include "debug_fetching_ip.h"
 2 #include "fetching_ip.h"
 3 #ifndef __SYNTHESIS__
 4 #ifdef DEBUG_FETCH
 5 #include <stdio.h>
 6 #endif
 7 #endif
 8 void fetch(
 9   code_address_t pc,
10   instruction_t *code_ram,
11   instruction_t *instruction){
12 #pragma HLS INLINE off
13   *instruction = code_ram[pc];
14 #ifndef __SYNTHESIS__
15 #ifdef DEBUG_FETCH
16   printf("%04d: %08x\n", (int)(pc<<2), *instruction);
17 #endif
18 #endif
19 }
20
```

Fig. 5.9 The code_ram_addr variable: the matching source code line

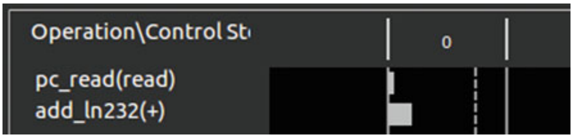


Fig. 5.10 The execute function schedule

The Schedule Viewer also indicates the dependencies between the different phases of the computation. In Fig. 5.8, there is an incoming purple arrow on the left of the code_ram_load(read) line, coming from the code_ram_addr(getelementptr) line (the memory read depends on the addressing pc).

5.3.9 The Z1_fetching_ip Vivado Project

In the Vivado tool, a block design can be built, as shown in Fig. 5.11.

You can use the prebuilt z1_fetching_ip.xpr file in the fetching_ip folder (in Vivado, open project, navigate to the fetching_ip folder and open the z1_fetching_ip.xpr file, then **Open Block Design**).

The report after the pre-generated bitstream is shown in Fig. 5.12 (**Reports** tab, scroll down to **Implementation/Place Design/Utilization - Place Design**; in the **Utilization - Place Design - impl_1** frame, scroll down to **1. Slice Logic**). The Pynq-Z1 design uses 1538 LUTs (2.89%) instead of 272.

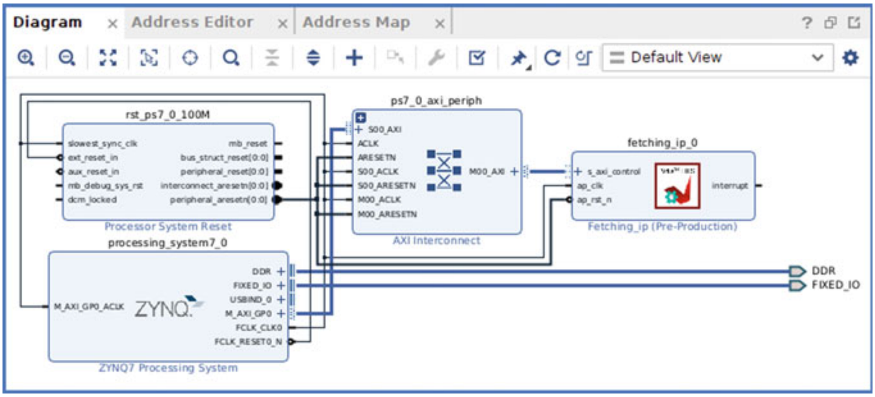


Fig. 5.11 The z1_fetching_ip block design

5.3.10 The Helloworld.c Program to Drive the Fetching_ip on the FPGA

Experimentation

To run the fetching_ip on the development board, plug in your board and switch it on.

Launch Vitis IDE (in a terminal with the /opt/Xilinx/Vitis/2022.1 folder as the current directory, run the "source settings64.sh" command, then run the "vitis" command).

Name the workspace as workspace_fetching_ip.

In a terminal, run "sudo putty", select **Serial**, set the **Serial line** as **/dev/ttyUSB1**, set the **Speed** as 115200 and click on **Open**.

In Vitis IDE, build an **Application Project**, **Create a new platform from hardware (XSA)**, browse to the goossens-book-ip-projects/2022.1/fetching_ip folder, select the design_1_wrapper.xsa file and click on **Open**, then on **Next**.

Name the **Application project** as z1_00, click on **Next** twice, then on **Finish**.

In a terminal with goossens-book-ip-projects/2022.1/fetching_ip as the current directory, run the update_helloworld.sh shell script.

In Vitis IDE, click on the Launch Target Connection button (refer back to Fig. 2.88). In the Target Connections Dialog Box, expand Hardware Server. Double-click on Local [default]. In the Target Connection Details pop-up window, click on OK. Close the Target Connections Dialog Box.

In Vitis IDE **Explorer** frame, expand z1_00_system/z1_00/src and double click on helloworld.c to open the file. You can replace the default helloworld program with the code in the goossens-book-ip-projects/2022.1/fetching_ip/helloworld.c file.

Right-click on z1_00_system, **Build Project**. Wait until the Build Finished message is printed in the **Console**.

Right-click again on z1_00_system and **Run As/Launch Hardware**. The result of the run (i.e. **done**) should be printed in the putty window.

The code in Listing 5.16 is the helloworld.c driver associated to the fetching IP (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script). It mimics the main function in the testbench_fetching_ip.cpp file (i.e. runs the IP and prints the done message at the end).

Listing 5.16 The helloworld.c file

```
#include <stdio.h>
#include "xfetching_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE      (1<<LOG_CODE_RAM_SIZE)
XFetching_ip_Config *cfg_ptr;
```

```
XFetching_ip          ip;
word_type code_ram[CODE_RAM_SIZE]={
#include "test_op_imm_0_text.hex"
};
int main(){
  cfg_ptr = XFetching_ip_LookupConfig(XPAR_XFETCHING_IP_0_DEVICE_ID
      );
  XFetching_ip_CfgInitialize(&ip, cfg_ptr);
  XFetching_ip_Set_start_pc(&ip, 0);
  XFetching_ip_Write_code_ram_Words(&ip, 0, code_ram, CODE_RAM_SIZE
      );
  XFetching_ip_Start(&ip);
  while (!XFetching_ip_IsDone(&ip));
  printf("done\n");
}
```

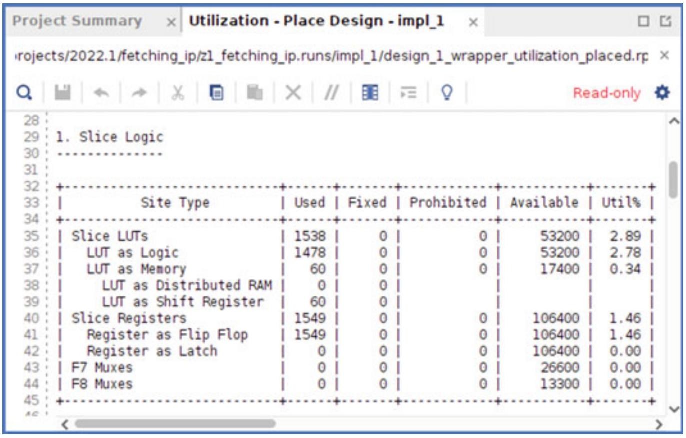
All the functions prefixed with XFetching_ip_ are defined in the xfetching_ip.h file which is built by the Vitis_HLS tool.

A copy of xfetching_ip.h can be found in the Vitis_HLS project when exploring solution1/impl/misc/drivers/fetching_ip_v1_0/src (it can also be viewed within the Vitis IDE environment by double clicking on the xfetching_ip.h header file name in the **Outline** frame on the right side of the Vitis IDE page).

The XFetching_ip_LookupConfig function builds a configuration structure associated to the IP given as the argument and defined as a constant in the xparameters.h file (XPAR_XFETCHING_IP_0_DEVICE_ID which is the fetching_ip component in the Vivado design). It returns a pointer to the created structure.

The XFetching_ip_CfgInitialize function initializes the configuration structure, i.e. creates the IP and returns a pointer to it.

The XIp_name_LookupConfig and XIp_name_CfgInitialize functions should be used to create and handle any ip_name IP imported from the added repositories.


Project Summary
Utilization - Place Design - impl_1
rojects/2022.1/fetching_ip/z1_fetching_ip.runs/impl_1/design_1_wrapper_utilization_placed.rp
Read-only

```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 1538 |     0 |          0 |     53200 |  2.89 |
|   LUT as Logic             | 1478 |     0 |          0 |     53200 |  2.78 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 1549 |     0 |          0 |    106400 |  1.46 |
|   Register as Flip Flop    | 1549 |     0 |          0 |    106400 |  1.46 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |    0 |     0 |          0 |     26600 |  0.00 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 5.12 The implementation utilization report

Once the fetching_ip has been created and can be addressed through the ip pointer returned by the XFetching_ip_CfgInitialize function, initial values should be sent.

The start_pc argument is initialized with a call to the XFetching_ip_Set_start_pc function. There is one XFetching_ip_Set_ function for each scalar input argument and one XFetching_ip_Get_ function per scalar output argument.

The code_ram array argument is initialized with a call to the XFetching_ip_Write_code_ram_Words function which sends a succession of words on the AXI bus in burst mode.

The fourth argument of the XFetching_ip_Write_code_ram_Words function defines the number of words sent (CODE_RAM_SIZE). The second argument is the start address of the write to the destination array (0).

Once the start pc and the code memory have been initialized, the fetching_ip can be started with a call to the XFetching_ip_Start function. The Zynq7 Processing System IP sends the start signal to the fetching_ip through the AXI interconnect.

The main function should then wait until the fetching_ip has finished its run, i.e. has exited the do ... while loop of the fetching_ip top function. This is the role of the while loop in the helloworld main function, controlled by the value returned by the XFetching_ip_IsDone function.

You may feel a bit disappointed after the run on the FPGA, as the only output is the done message. You know that your IP has done something but you do not see what has been done as the debug messages showing the fetched code are not printed.

The prints will get more convincing in the next designs. However, you should keep in mind that a processor does not output anything. Mainly, it computes in its internal registers which are not visible from the external world, and stores and loads from its memory. When the external world is given an access to the memory, the result of a computation can be observed by dumping the data memory. The computed program should store its results to memory rather than simply saving them to registers.

5.4 Second Step: Add a Bit of Decoding to Compute the Next PC

5.4.1 The RISC-V Instruction Encoding

Once an instruction has been fetched from memory, it must be decoded to prepare its execution.

The instruction is a 32-bit word (in RV32I, **32** refers to the data width, not to the instruction width: e.g. RV64I refers to 32-bit instructions operating on 64-bit data).

The instruction word is composed of fields as defined in Sect. 2.2 (Base Format) of the RISC-V specification [1] and presented in Sect. 4.1.3 of the present book.

Decoding the instruction means decomposing a single word into the set of its fields.

The components building the RISC-V instruction encoding are the major opcode, the minor opcode func3, the source registers rs1 and rs2, the destination register rd, an operation specifier func7, and an immediate value imm.

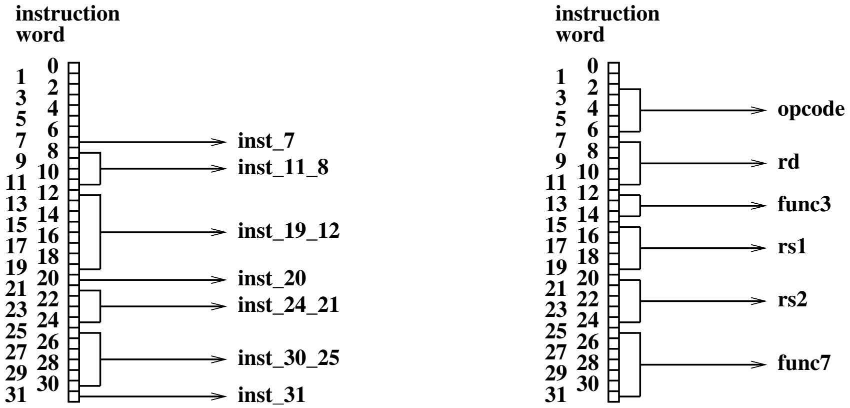


Fig. 5.13 Instruction decoding

According to its format, a RV32I instruction code is the concatenation of a subset of these components (e.g. imm, rs1, func3, rd, and opcode for an I-TYPE format and func7, rs2, rs1, func3, rd, and opcode for an R-TYPE format).

For example, "addi a0, a0, 1" (an I-TYPE format) is encoded as the hexadecimal 32-bit value 0x00150513.

This encoding includes and represents the immediate value 1 (imm), the left source register a0 (rs1), the minor opcode addi (func3), the destination register a0 (rd), and the major opcode OP_IMM (opcode).

The role of the decoding phase is to split the value 0x00150513 into these different components.

The "sub a0, a1, a2" instruction encoding (an R-TYPE format) is composed of the operation specifier sub (func7), the right source register a2 (rs2), the left source register a1 (rs1), the minor opcode add/sub (func3), the destination register a0 (rd), and the major opcode OP (opcode).

Figure 5.13 shows the hardware used to decode an RV32I instruction.

The partitioning uses no gates nor circuits. It is simply a set of wires pulled out of the instruction word.

Figure 5.13 shows two partitionings. I use the left one to form the imm immediate value and the right one to decompose the instruction into its main other components opcode, rd, func3, rs1, rs2, and func7.

The RISC-V ISA defines 32 different opcodes (encoded on the five bits of the major opcode field), each of which is associated to one of the six defined formats: R-TYPE (register type, format number 1), I-TYPE (immediate type, format number 2), S-TYPE (store type, format number 3), B-TYPE (branch type, format number 4), U-TYPE (upper type, format number 5) and J-TYPE (jump type, format number 6).

The format numbering is not part of the specification. It is defined in my implementation.

Table 5.1 Opcode and format association (opcodes with bits 4-2 between 000 and 011)

opcode[65][432]	000	001	010	011
00	LOAD	LOAD-FP	CUSTOM-0	MISC-MEM
	I-TYPE	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE
01	STORE	STORE-FP	CUSTOM-1	AMO
	S-TYPE	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE
10	MADD	MSUB	NMSUB	NMADD
	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE
11	BRANCH	JALR	RESERVED-1	JAL
	B-TYPE	I-TYPE	OTHER-TYPE	J-TYPE

Table 5.2 Opcode and format association (opcodes with bits 4-2 between 100 and 111)

opcode[65][432]	100	101	110	111
00	OP-IMM	AUIPC	OP-IMM-32	RV48-0
	I-TYPE	U-TYPE	OTHER-TYPE	OTHER-TYPE
01	OP	LUI	OP-32	RV64
	R-TYPE	U-TYPE	OTHER-TYPE	OTHER-TYPE
10	OP-FP	RESERVED-0	CUSTOM2-RV128	RV48-1
	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE
11	SYSTEM	RESERVED-2	CUSTOM3-RV128	RV80
	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE	OTHER-TYPE

I have added two more format numbers: UNDEFINED-TYPE (format number 0) and OTHER-TYPE (format number 7, used for all the instructions not part of the RV32I set).

Tables 5.1 and 5.2 show the association between opcodes and formats according to the RISC-V specification. It is not limited to the RV32I subset but to all the instructions of the full non privileged RISC-V ISA.

Table 5.1 shows the opcodes with the low order bits ranging from 0 to 3.

Table 5.2 shows the opcodes with the low order bits ranging from 4 to 7.

Figure 5.14 shows the circuits to decode the instruction formats. It is a set of four multiplexers in parallel and a serial fifth one. The four multiplexers select one among their eight 3-bit format number entries. The selected entry is the one addressed by the 3-bit selection code from the instruction word bits 2–4 (named opcl in Fig. 5.14).

For example, when the opcl 3-bit code is 0b100, the four multiplexers output their fourth entry (numbered from 0 to 7, from the top to the bottom of the multiplexer), i.e. the I-TYPE encoding for the top first multiplexer (0b010, format number 2), the R-TYPE encoding for the second multiplexer (0b001), the OTHER-TYPE encoding for the third multiplexer (0b111), and the OTHER-TYPE encoding again for the fourth bottom multiplexer.

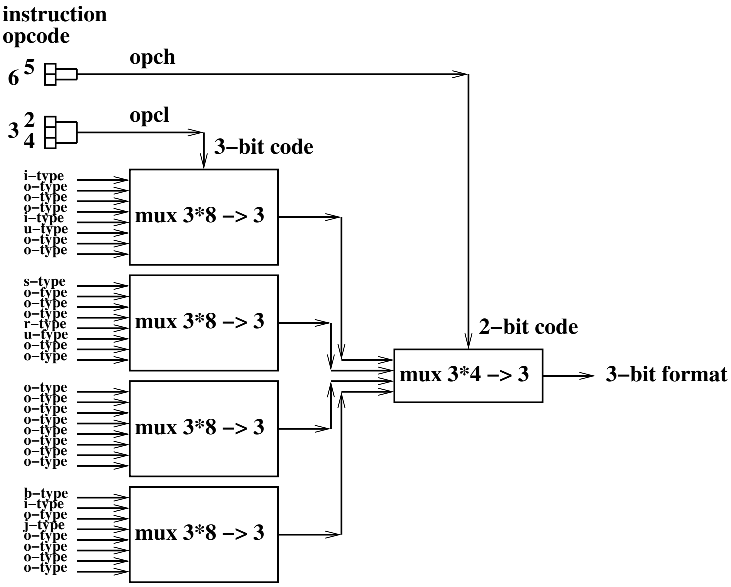


Fig. 5.14 Decoding the instruction format

The rightmost multiplexer selects one of the outputs of the four leftmost multiplexers, according to its 2-bit opch code (instruction word bits 5 and 6).

For example, when the 2-bit code is 0b00, the rightmost multiplexer outputs the I-TYPE encoding (0b010) coming from the uppermost left multiplexer. Hence, the format associated to opcode 0b00100 (OP-IMM) is I-TYPE.

The synthesizer simplifies the circuits. For example, the third left multiplexer has eight times the same OTHER-TYPE input. It can be eliminated and replaced by driving the OTHER-TYPE code directly to the right multiplexer second entry (numbered from 0 to 3, from the top to the bottom of the multiplexer).

5.4.2 The Fetching_decoding_ip

All the source files related to the fetching_decoding_ip can be found in the fetching_decoding_ip folder.

The fetching_decoding_ip function is defined in the fetching_decoding_ip.cpp file (see Listing 5.8).

There are a few differences compared to the preceding fetching_ip top function shown in Figs. 5.5 and 5.6.

I have added a third argument (nb_instruction) to the IP to provide the number of instructions fetched and decoded. It helps to check that the correct path has been followed.

Listing 5.17 The fetching_decoding_ip top function

```
void fetching_decoding_ip(
  unsigned int  start_pc,
  unsigned int  code_ram[CODE_RAM_SIZE],
  unsigned int *nb_instruction){
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=code_ram
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=return
  code_address_t        pc;
  instruction_t         instruction;
  bit_t                 is_running;
  unsigned int          nbi;
  decoded_instruction_t d_i;
  pc  = start_pc;
  nbi = 0;
  do{
#pragma HLS PIPELINE II=3
    fetch(pc, code_ram, &instruction);
    decode(instruction, &d_i);
    execute(pc, d_i, &pc);
    statistic_update(&nbi);
    running_cond_update(instruction, &is_running);
  } while (is_running);
  *nb_instruction = nbi;
}
```

The instruction count is computed in a local counter nbi which is copied to the nb_instruction output argument at the end of the run. The nbi counter is updated in the statistic_update function shown in Listing 5.18 and defined in the fetching_decoding_ip.cpp file.

Listing 5.18 The statistic_update function

```
static void statistic_update(
  unsigned int *nbi){
#pragma HLS INLINE off
  *nbi = *nbi + 1;
}
```

As every iteration fetches and decodes one instruction, the number of instructions fetched and decoded is also the number of IP cycles.

The nb_instruction argument has been associated to the s_axilite INTERFACE, like all the other arguments. But, as it is an output it is represented as a pointer.

I have used the HLS PIPELINE pragma to keep the loop iteration duration within three FPGA cycles.

The HLS PIPELINE II=3 sets an Initiation Interval (II) of 3. This interval is the number of cycles before the next iteration can start.

If a loop iteration has a duration of d cycles and the II is set as i (with $i <= d$), then an iteration starts every i cycles and each new iteration overlaps of d-i cycles with its predecessor.

The lower the II, the faster the design. The II value gives the throughput of the pipeline. With II=3, the pipeline outputs one instruction every three cycles. With II=1, the output rate is one instruction per cycle, i.e. three times more.

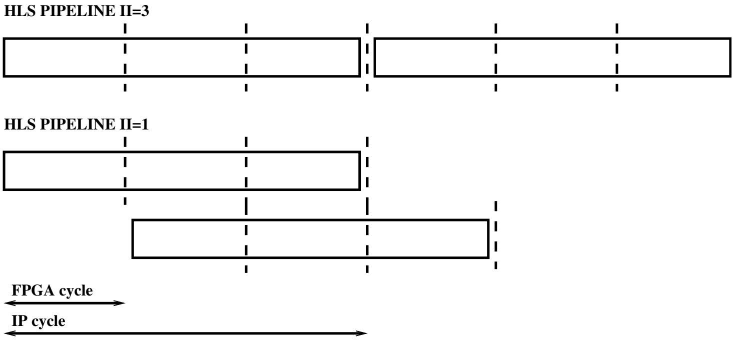


Fig. 5.15 Pipelining or not pipelining the IP

Figure 5.15 shows the difference between HLS PIPELINE II=3 and HLS PIPELINE II=1.

When II is set to *i*, it indicates that the next iteration should, if possible, start *i* FPGA cycles after the current one.

In my design, an II of 1 would lead to two overlapping FPGA cycles for two successive iterations. This is incompatible with the fact that the next pc is computed during the third cycle (it takes three FPGA cycles to fetch, decode, and select the incremented pc in the execute function), too late to be used by the next iteration if it is scheduled one FPGA cycle after its predecessor. Even an II of 2 would fail (in this case, the synthesizer detects an II violation).

The do ... while loop contains five function calls, with the adding of the calls to the decode function and to the statistic_update one.

5.4.3 The Fetching_decoding_ip.h File

5.4.3.1 The Opcode Definitions

The fetching_decoding_ip.h file contains the definition of the 32 opcodes (see Listing 5.19). These values are defined in the specification of the RISC-V architecture.

Listing 5.19 The RISC-V opcode definitions in the fetching_decoding_ip.h file

```
...
#define LOAD           0b00000
#define LOAD_FP        0b00001
#define CUSTOM_0       0b00010
#define MISC_MEM       0b00011
...
#define JAL            0b11011
#define SYSTEM         0b11100
#define RESERVED_2     0b11101
#define CUSTOM_3_RV128 0b11110
#define RV80           0b11111
...
```

5.4.3.2 The RISC-V Instruction Formats

The fetching_decoding_ip.h file also contains the definition of the six RISC-V instruction formats (the numbering is not part of the RISC-V specification), shown in Listing 5.20.

Listing 5.20 The RISC-V format definitions in the fetching_decoding_ip.h file

```
...
#define UNDEFINED_TYPE 0
#define R_TYPE         1
#define I_TYPE         2
#define S_TYPE         3
#define B_TYPE         4
#define U_TYPE         5
#define J_TYPE         6
#define OTHER_TYPE     7
...
```

5.4.3.3 The Fields in the RV32I Instruction Encoding

The fetching_decoding_ip.h file contains the following type definitions related to the RV32I instruction encoding (refer back to the formats presented in Sect. 4.1.3):

- type_t: the RISC-V instruction formats.
- i_immediate_t: the I-TYPE immediate values.
- s_immediate_t: the S-TYPE immediate values.
- b_immediate_t: the B-TYPE immediate values.
- u_immediate_t: the U-TYPE immediate values.
- j_immediate_t: the J-TYPE immediate values.
- opcode_t: the major opcodes.
- reg_num_t: the register numbers.
- func3_t: the minor opcodes.
- func7_t: the operation specifiers.

Even though the fetching_decoding_ip has no register file yet, the instruction encoding contains fields using register numbers (first source register rs1, second source register rs2, and destination register rd).

Listing 5.21 shows the type definitions in the fetching_decoding_ip.h file.

Listing 5.21 The type definitions in the fetching_decoding_ip.h file

```
...
typedef unsigned int                instruction_t;
typedef ap_uint<LOG_CODE_RAM_SIZE>  code_address_t;
typedef ap_uint<3>                  type_t;
typedef ap_int<20>                  immediate_t;
typedef ap_int<12>                  i_immediate_t;
typedef ap_int<12>                  s_immediate_t;
typedef ap_int<12>                  b_immediate_t;
typedef ap_int<20>                  u_immediate_t;
typedef ap_int<20>                  j_immediate_t;
typedef ap_uint<5>                  opcode_t;
typedef ap_uint<5>                  reg_num_t;
typedef ap_uint<3>                func3_t;
typedef ap_uint<7>                func7_t;
typedef ap_uint<1>                bit_t;
...
```

5.4.3.4 The Decoded_instruction_t Type

The decode_instruction_t type shown in Listing 5.22 is defined in the fetching_decoding_ip.h file.

The structure contains the main field partitioning of the instruction word: major opcode, destination register rd, minor opcode func3, left source register rs1, right source register rs2, operation specifier func7, instruction format type, and immediate value imm.

Listing 5.22 The decoded_instruction_t type definition in the fetching_decoding_ip.h file

```
typedef struct decoded_instruction_s{
  opcode_t    opcode;
  reg_num_t   rd;
  func3_t     func3;
  reg_num_t   rs1;
  reg_num_t   rs2;
  func7_t     func7;
  type_t      type;
  immediate_t imm;
} decoded_instruction_t;
```

5.4.3.5 The Decoded_immediate_t Type

The immediate value is to be decoded from different bit fields of the instruction, as shown on the left of Fig. 5.13. This partitioning shown in Listing 5.23 is defined as the decoded_immediate_t type in the fetching_decoding_ip.h file.

Listing 5.23 The decoded_immediate_t type definition in the fetching_decoding_ip.h file

```
typedef struct decoded_immediate_s{
  bit_t       inst_31;
  ap_uint<6>  inst_30_25;
  ap_uint<4>  inst_24_21;
  bit_t       inst_20;
  ap_uint<8>  inst_19_12;
  ap_uint<4>  inst_11_8;
  bit_t       inst_7;
} decoded_immediate_t;
```

5.4.4 The Fetch Function and the Running_cond_update Function

The fetch function and the running_cond_update function are unchanged from the fetching_ip.

As a general rule, the successive IPs you will build are incrementally designed. A function is changed only if it is necessary in the new implementation. Otherwise, what has been designed in an IP is kept in the successors.

5.4.5 The Decode Function

The decode function is defined in the decode.cpp file (see Listing 5.24).

Listing 5.24 The decode function in the decode.cpp file

```
void decode(
  instruction_t          instruction,
  decoded_instruction_t *d_i){
#pragma HLS INLINE off
  decode_instruction(instruction, d_i);
  decode_immediate  (instruction, d_i);
#ifndef __SYNTHESIS__
#ifdef DEBUG_DECODE
  print_decode(*d_i);
#endif
#endif
}
```

The decode function decodes an instruction in two decompositions, represented by functions decode_instruction and decode_immediate.

In debug mode, when the DEBUG_DECODE constant is defined in the debug_fetching_decoding_ip.h file, the decode function prints the decoded fields (call of the print_decode function).

5.4.5.1 The Decode_instruction Function

The decode_instruction function (see Listing 5.25) is defined in the decode.cpp file.

It fills the d_i structure, i.e. sets the fields of the decoded_instruction_t type. The d_i structure fields match the ones shown on the right part of Fig. 5.13.

Listing 5.25 The decode_instruction function

```
static void decode_instruction(
  instruction_t          instruction,
  decoded_instruction_t *d_i){
#pragma HLS INLINE
  d_i->opcode      = (instruction >>  2);
  d_i->rd          = (instruction >>  7);
  d_i->func3       = (instruction >> 12);
  d_i->rs1         = (instruction >> 15);
  d_i->rs2         = (instruction >> 20);
  d_i->func7       = (instruction >> 25);
  d_i->type        = type(d_i->opcode);
}
```

The type field (i.e. the instruction format) is set by the type function, according to the instruction opcode.

5.4.5.2 The Type Function

The type.cpp file contains the type function.

The type function decodes the instruction format as shown in Listing 5.26.

The opcode argument is divided into a 2-bit field opch (two most significant bits of the opcode) and a 3-bit field opcl (three least significant bits of the opcode).

The right shift applied to opcode to set opch in the "opch = opcode>>3" instruction is synthesized as a selection of the two upper bits of opcode.

The "opcl = opcode" instruction shrinks the opcode value to the three bits size of the opcl destination variable.

The switch-case branch on the opch value corresponds to the rightmost multiplexer in Fig. 5.14.

Listing 5.26 The type function in the type.cpp file

```
type_t type(opcode_t opcode){
#pragma HLS INLINE
  ap_uint<2> opch;
  ap_uint<3> opcl;
  opch = opcode>>3;
  opcl = opcode;
  switch(opch){
    case 0b00: return type_00(opcl);
    case 0b01: return type_01(opcl);
    case 0b10: return type_10(opcl);
    case 0b11: return type_11(opcl);
  }
  return UNDEFINED_TYPE;
}
```

Functions type_00 (LOAD, OP_IMM, and AUIPC), type_01 (STORE, OP, and LUI), type_10 (instructions out of the RV32I ISA) and type_11 (BRANCH, JALR, and JAL) are also defined in the type.cpp file.

They correspond to the four leftmost multiplexers in Fig. 5.14.

I only present the type_00 function (see Listing 5.27) as the other functions are based on the same model.

Listing 5.27 The type_00 function in the type.cpp file

```
static type_t type_00(ap_uint<3> opcl){
#pragma HLS INLINE
  switch(opcl){
    case 0b000: return I_TYPE;     //LOAD
    case 0b001: return OTHER_TYPE; //LOAD-FP
    case 0b010: return OTHER_TYPE; //CUSTOM-0
    case 0b011: return OTHER_TYPE; //MISC-MEM
    case 0b100: return I_TYPE;     //OP-IMM
    case 0b101: return U_TYPE;     //AUIPC
    case 0b110: return OTHER_TYPE; //OP-IMM-32
    case 0b111: return OTHER_TYPE; //RV48-0
  }
  return UNDEFINED_TYPE;
}
```

The type functions will be left unchanged in all the successive designs you will build.

5.4.5.3 The Decode_immediate Function

The decode_immediate function (see Listing 5.28) is defined in the decode.cpp file.

It fills the imm field of the d_i structure. The field receives the decoding of the immediate. The decoding is the one shown on the left part of Fig. 5.13.

Listing 5.28 The decode_immediate function

```
static void decode_immediate(
  instruction_t          instruction,
  decoded_instruction_t *d_i){
#pragma HLS INLINE
  decoded_immediate_t d_imm;
  d_imm.inst_31    = (instruction >> 31);
  d_imm.inst_30_25 = (instruction >> 25);
  d_imm.inst_24_21 = (instruction >> 21);
  d_imm.inst_20    = (instruction >> 20);
  d_imm.inst_19_12 = (instruction >> 12);
  d_imm.inst_11_8  = (instruction >>  8);
  d_imm.inst_7     = (instruction >>  7);
  switch(d_i->type){
    case UNDEFINED_TYPE: d_i->imm = 0; break;
    case R_TYPE:         d_i->imm = 0; break;
    case I_TYPE:         d_i->imm = i_immediate(d_imm); break;
    case S_TYPE:         d_i->imm = s_immediate(d_imm); break;
    case B_TYPE:         d_i->imm = b_immediate(d_imm); break;
    case U_TYPE:         d_i->imm = u_immediate(d_imm); break;
    case J_TYPE:         d_i->imm = j_immediate(d_imm); break;
    case OTHER_TYPE:     d_i->imm = 0; break;
  }
}
```

5.4.5.4 The I_immediate Function

The i_immediate, s_immediate, b_immediate, u_immediate, and j_immediate functions are defined in the immediate.cpp file.

They assemble the different fields to build the I-TYPE (LOAD, JALR, and OP_IMM instructions), S-TYPE (STORE instructions), B-TYPE (BRANCH instructions), U-TYPE (LUI and AUIPC instructions), and J-TYPE constants (JAL instructions) as presented in Sect. 4.1.3 of the present book and defined in Sect. 2.3 of the RISC-V specification [1].

R-TYPE, i.e. OP instructions have no encoded constant.

For example, Listing 5.29 shows the i_immediate function.

Listing 5.29 The i_immediate function

```
i_immediate_t i_immediate(decoded_immediate_t d_imm){
#pragma HLS INLINE
 return (((i_immediate_t)d_imm.inst_31   <<11) |
         ((i_immediate_t)d_imm.inst_30_25<< 5) |
         ((i_immediate_t)d_imm.inst_24_21<< 1) |
         ((i_immediate_t)d_imm.inst_20      ));
}
```

The other functions are built similarly.

The immediate decoding functions will be left unchanged in all the successive designs you will build.

5.4.6 The Instruction Execution (Computing Next PC)

The execute function is defined in the execute.cpp file. It is shown in Listing 5.30.

It computes the next pc according to the instruction format saved in the d_i structure after decoding.

Listing 5.30 The execute function

```
void execute(
  code_address_t          pc,
  decoded_instruction_t d_i,
  code_address_t         *next_pc){
#pragma HLS INLINE off
  *next_pc = compute_next_pc(pc, d_i);
}
```

The compute_next_pc function (see Listing 5.31) is also defined in the execute.cpp file. It is a slight extension of the version presented in Listing 5.12. It takes care of immediate jump instructions (JAL opcode, e.g. "jal foo" to call the foo function).

The JAL instructions belong to the J-TYPE. They contain an encoded constant which is a displacement from the instruction position. The processor adds this displacement to the current pc to set the next pc.

The displacement value is extracted from the instruction by the j_immediate function and decoded in the d_i.imm field.

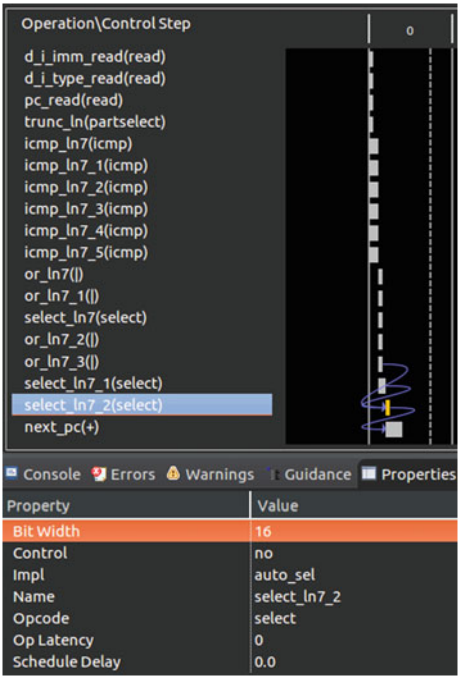
For all the other instructions (including BRANCH ones), the next pc is always set to point to the next instruction (i.e. pc + 1). Conditional branches and indirect jumps will be considered in the fde_ip design in Sect. 5.5.

The RISC-V specification states that the J-TYPE constant should be multiplied by 2, i.e. shifted of one position on the left, before being added to the current pc to form the jump target address.

But, as we shrink the two lowest bits to address the code_ram word memory (word pointer), the computed displacement is shifted of one position on the right ("next_pc = pc + (d_i.imm>>1)").

Listing 5.31 The compute_next_pc function

```
code_address_t compute_next_pc(
  code_address_t          pc,
  decoded_instruction_t d_i){
#pragma HLS INLINE
  code_address_t next_pc;
  switch(d_i.type){
    case R_TYPE:
      next_pc = pc + 1;
      break;
    ...
    case J_TYPE:
      next_pc = pc + (d_i.imm>>1);
      break;
    default:
      next_pc = pc + 1;
      break;
  }
  return next_pc;
}
```



```
d_i_imm_read(read)
d_i_type_read(read)
pc_read(read)
trunc_ln(partselect)
icmp_ln7(icmp)
icmp_ln7_1(icmp)
icmp_ln7_2(icmp)
icmp_ln7_3(icmp)
icmp_ln7_4(icmp)
icmp_ln7_5(icmp)
or_ln7(|)
or_ln7_1(|)
select_ln7(select)
or_ln7_2(|)
or_ln7_3(|)
select_ln7_1(select)
select_ln7_2(select)
next_pc(+)
```

Property	Value
Bit Width	16
Control	no
Impl	auto_sel
Name	select_ln7_2
Opcode	select
Op Latency	0
Schedule Delay	0.0

Fig. 5.16 Schedule of the execute function

The d_i.imm field has type immediate_t (see Listing 5.22), which is 20 bits wide.

However, the next_pc destination of the computation involving the d_i.imm value has type code_address_t, which is 16 bits wide (LOG_CODE_RAM_SIZE; see Listing 5.8).

The synthesizer shrinks the d_i.imm value to its 16 least significant bits and adds this 16-bit value to pc.

This can be checked after synthesis on the Schedule Viewer by navigating down to the execute graph. Then, you look at the **Properties** of the select_ln7_2(select) line (i.e. line 7 in the execute.cpp file, i.e. the "switch(d_i.type)" instruction in the compute_next_pc function).

The **Bit Width** property is 16 bits (see Fig. 5.16).

The next line (next_pc(+)) is the addition with pc. Again the **Bit Width** is 16.

5.4.7 The Fetching_decoding_ip Simulation with the Testbench

Experimentation

To simulate the fetching_decoding_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with fetching_decoding_ip.

The testbench_fetching_decoding_ip.cpp file (see Listing 5.32) adds the new nbi argument counting the number of fetched and decoded instructions. The test_op_imm_0_text.hex file is unchanged.

Listing 5.32 The testbench_fetching_decoding_ip.cpp file

```
#include <stdio.h>
#include "fetching_decoding_ip.h"
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_op_imm_0_text.hex"
};
int main() {
  unsigned int nbi;
  fetching_decoding_ip(0, code_ram, &nbi);
  printf("%d fetched and decoded instructions\n", nbi);
  return 0;
}
```

The IP runs the code in the test_op_imm_0_text.hex file.

The run should print what is shown in Listing 5.33 (only the first and last instruction decodings are shown) (the printing of the decoding is done in the decode function; refer back to Listing 5.24).

Listing 5.33 The fetching_decoding_ip simulation print

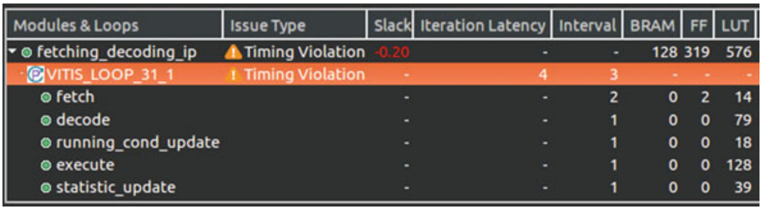
```
0000: 00500593
opcode:       4
rd:           b
func3:        0
rs1:          0
rs2:          5
func7:        0
I_TYPE
...
0052: 00008067
opcode:      19
rd:           0
func3:        0
rs1:          1
rs2:          0
func7:        0
I_TYPE
14 fetched and decoded instructions
```

5.4.8 The Fetching_decoding_ip Synthesis

Figure 5.17 shows the synthesis report.

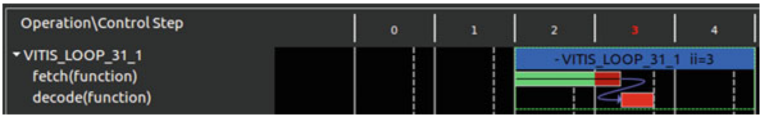
The synthesis report shows a Timing Violation warning. This is not critical as it concerns operations inside the processor cycle and does not impact the loop scheduling (it is not critical when the II interval is equal to the iteration latency, implying no overlapping for successive iterations).

The synthesizer could not fit the end of the fetch function and the start of the decode one within FPGA cycle 3 (see Fig. 5.18, obtained by right clicking on the synthesis report **Timing Violation** warning, then selecting **Go To Timing Violation**, which opens the Schedule Viewer).



Modules & Loops	Issue Type	Slack	Iteration Latency	Interval	BRAM	FF	LUT
fetching_decoding_ip	⚠ Timing Violation	-0.20	-	-	128	319	576
VITIS_LOOP_31_1	⚠ Timing Violation	-	4	3	-	-	-
fetch		-	-	2	0	2	14
decode		-	-	1	0	0	79
running_cond_update		-	-	1	0	0	18
execute		-	-	1	0	0	128
statistic_update		-	-	1	0	0	39

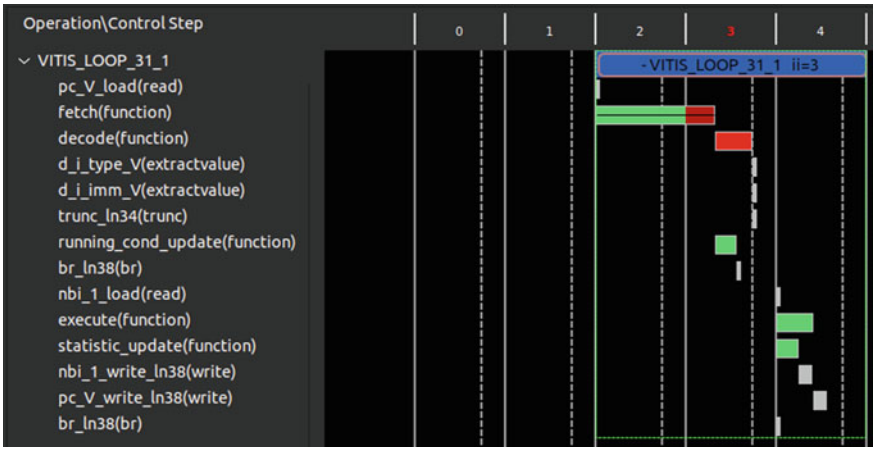
Fig. 5.17 The fetching_decoding_ip synthesis report



Operation\Control Step

VITIS_LOOP_31_1
fetch(function)
decode(function)

Fig. 5.18 Timing Violation



Operation\Control Step

VITIS_LOOP_31_1
pc_V_load(read)
fetch(function)
decode(function)
d_i_type_V(extractvalue)
d_i_imm_V(extractvalue)
trunc_ln34(trunc)
running_cond_update(function)
br_ln38(br)
nbi_1_load(read)
execute(function)
statistic_update(function)
nbi_1_write_ln38(write)
pc_V_write_ln38(write)
br_ln38(br)

Fig. 5.19 The fetching_decoding_ip schedule

As a general rule, do not pay too much attention to the Timing Violation warnings, except when they concern the end of the IP cycle, i.e. operations which do not fit in the last FPGA cycle of the main loop.

You can try to use the synthesis in Vivado. If the IP works fine on the development board, you can forget about the timing violation.

Figure 5.19 shows the fetching_decoding_ip schedule. The loop latency is three FPGA cycles (30ns, 33 Mhz), as expected from the HLS PIPELINE II=3 pragma (refer back to Listing 5.24).

5.4.9 The Z1_fetching_decoding_ip Vivado Project

The z1_fetching_decoding_ip Vivado project defines the block design shown in Fig. 5.20.

Figure 5.21 shows the implementation report. The IP uses 1318 LUTs (2.48%).

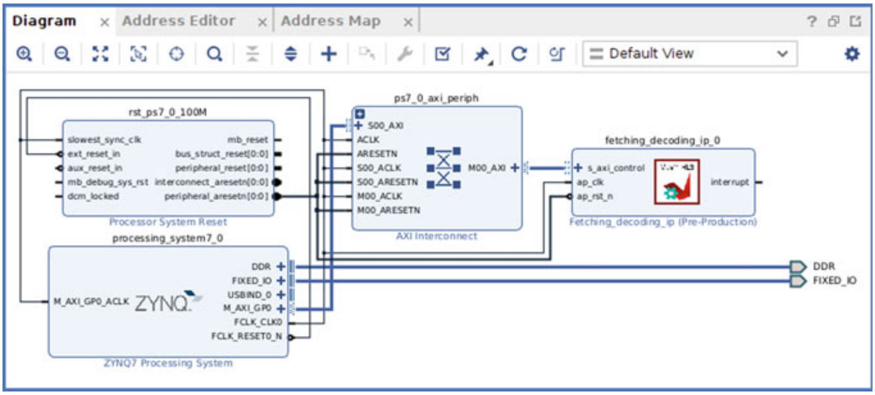
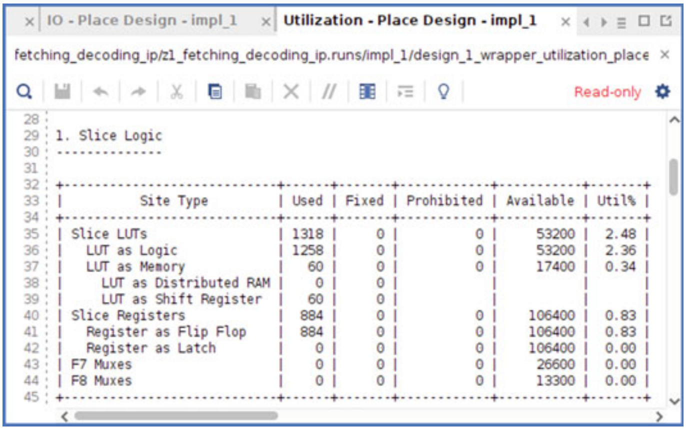


Fig. 5.20 The z1_fetching_decoding_ip block design in Vivado


IO - Place Design - impl_1
Utilization - Place Design - impl_1
fetching_decoding_ip/z1_fetching_decoding_ip.runs/impl_1/design_1_wrapper_utilization_place
Read-only

```

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|         Site Type          | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 1318 |     0 |          0 |     53200 |  2.48 |
|   LUT as Logic             | 1258 |     0 |          0 |     53200 |  2.36 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            |  884 |     0 |          0 |    106400 |  0.83 |
|   Register as Flip Flop    |  884 |     0 |          0 |    106400 |  0.83 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |    0 |     0 |          0 |     26600 |  0.00 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 5.21 The fetching_decoding_ip implementation report

5.4.10 The Helloworld.c Code to Drive the Fetching_decoding_ip

Experimentation

To run the fetching_decoding_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with fetching_decoding_ip.

The code in the helloworld.c file is given in Listing 5.34 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script).

Listing 5.34 The helloworld.c file in the fetching_decoding_ip folder

```c
#include <stdio.h>
#include "xfetching_decoding_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
XFetching_decoding_ip_Config *cfg_ptr;
XFetching_decoding_ip         ip;
word_type code_ram[CODE_RAM_SIZE]={
#include "test_op_imm_0_text.hex"
};
int main(){
  cfg_ptr = XFetching_decoding_ip_LookupConfig(
      XPAR_XFETCHING_DECODING_IP_0_DEVICE_ID);
  XFetching_decoding_ip_CfgInitialize(&ip, cfg_ptr);
  XFetching_decoding_ip_Set_start_pc(&ip, 0);
  XFetching_decoding_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XFetching_decoding_ip_Start(&ip);
  while (!XFetching_decoding_ip_IsDone(&ip));
  printf("%d fetched and decoded instructions\n",
    (int)XFetching_decoding_ip_Get_nb_instruction(&ip));
}
```

Listing 5.35 shows what the run of the RISC-V code in the test_op_imm.s file should print on the putty terminal.

Listing 5.35 The helloworld print when running the test_op_imm RISC-V code on the Pynq Z1 board

```
14 fetched and decoded instructions
```

5.5 Third Step: Filling the Execute Stage to Build the Register Path

All the source files related to the fde_ip design can be found in the fde_ip folder.

5.5.1 A Fetching, Decoding, and Executing IP: The Fde_ip Design

The Vitis_HLS fde_ip project (fetch, decode, and execute) adds a register file to the processor. Figure 5.22 shows the fde_ip component. Contrarily to the code_ram memory block, the reg_file entity belongs to the IP and is not externally visible.

A register file is a multi-ported memory. The fde_ip register file groups 32 registers. Each register memorizes a 32-bit value. The register file can be addressed in three simultaneous ways: reading from two sources and writing to one destination, with three different ports.

For example, instruction "add a0,a1,a2" reads registers a1 and a2 and writes their sum into register a0.

Listing 5.36 shows the prototype, local declarations, and initializations of the fde_ip top function defined in the fde_ip.cpp file.

The IP clears all the registers before it starts running the code. This is my choice: it is not part of the RISC-V specification.

The HLS ARRAY_PARTITION pragma is used to partition the reg_file variable.

The choosen partitioning indicates to the synthesizer that each element of the array (i.e. each register) should be considered as individually accessible. Consequently, the one dimension array will be mapped on FPGA flip-flops instead of a BRAM block (i.e. a memory).

When an array is implemented with a BRAM block, it has at most two access ports (i.e. you can access at most two entries of the array simultaneously). When it is implemented with flip-flops (i.e. one flip-flop per memorized bit), all the entries can be accessed simultaneously.

Small arrays should be implemented as flip-flops through an ARRAY_PARTITION pragma, at least to keep BRAM blocks for big arrays (e.g. the code and data memories).

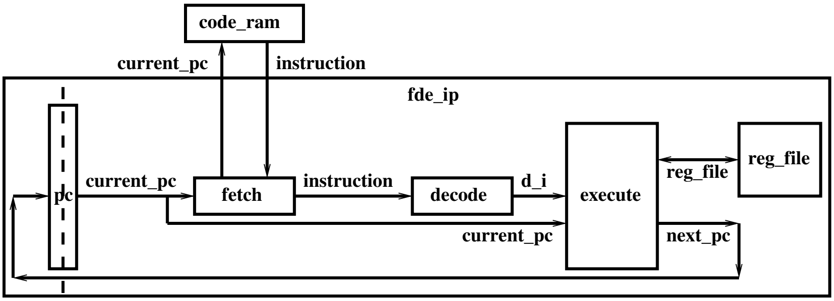


Fig. 5.22 The fde_ip component

Listing 5.36 The fde_ip function prototype, local variable declarations, and initializations

```
void fde_ip(
  unsigned int  start_pc,
  unsigned int  code_ram[CODE_RAM_SIZE],
  unsigned int *nb_instruction){
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=code_ram
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=return
  code_address_t        pc;
  int                   reg_file[NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file dim=1 complete
  instruction_t         instruction;
  bit_t                 is_running;
  unsigned int          nbi;
  decoded_instruction_t d_i;
  for (int i=0; i<NB_REGISTER; i++) reg_file[i] = 0;
  pc  = start_pc;
  nbi = 0;
  ...
```

The HLS PIPELINE II=6 pragma in the do .. while loop of the fde_ip top function (see Listing 5.37) bounds the IP cycle to six FPGA cycles (16.67 MHz). As the complexity increases, the computations to be done in one processor cycle need more time.

Listing 5.37 The fde_ip function do ... while loop

```
  ...
  do{
#pragma HLS PIPELINE II=6
    fetch(pc, code_ram, &instruction);
    decode(instruction, &d_i);
#ifndef __SYNTHESIS__
#ifdef DEBUG_DISASSEMBLE
    disassemble(pc, instruction, d_i);
#endif
#endif
    execute(pc, reg_file, d_i, &pc);
    statistic_update(&nbi);
    running_cond_update(instruction, pc, &is_running);
  } while (is_running);
  *nb_instruction = nbi;
#ifndef __SYNTHESIS__
#ifdef DEBUG_REG_FILE
  print_reg(reg_file);
#endif
#endif
}
```

5.5.2 Two Debugging Tools: Register File Dump and Code Disassembling

I have added two debugging features to the IP top function.

The print_reg function (the source code is in the print.cpp file; see Listing 5.38) can be useful to dump the content of the register file at any time during the run (in the fde_ip project, it is used to view the last state of the register file after the run).

The disassemble function (I do not show the code but it can be found in the disassemble.cpp file) is a tool to trace the assembly code while it is fetched and decoded.

These two functions will remain unchanged in all of the processor designs.

The debugging functions are not part of the synthesis. There are no restrictions to C/C++ programming and to library function usage as long as they are excluded from the synthesis (using the __SYNTHESIS__ constant). Local variables should have the C/C++ types rather than the Vitis ap_uint ones in these debugging functions.

Listing 5.38 The print_reg function

```
void print_reg(int *reg_file){
  unsigned int i;
  for (i=1; i<NB_REGISTER; i++){
    print_reg_name(i);
    printf(" ");
#ifdef SYMB_REG
    if (i!=26 && i!=27) printf(" ");
#else
    if (i<10) printf(" ");
#endif
    printf("= %16d (%8x)\n", reg_file[i],
               (unsigned int)reg_file[i]);
  }
}
```

In the print_reg function, the print_reg_name function (defined in the print.cpp file) prints the name of the registers. It can switch between the two namings with the SYMB_REG constant defined in the print.h file (see Listing 5.39).

If the constant is defined, the symbolic naming applies (registers zero, ra, sp, a0 ...). Otherwise, the ordinal numbering applies (registers x0, x1, x2 ...).

Listing 5.39 The print.h file with the SYMB_REG constant

```
#ifndef __PRINT
#define __PRINT
#ifndef __SYNTHESIS__
#include "fde_ip.h"
//register names are printed as x0, x1, x2 ...
//to print symbolic register names (zero, ra, sp ...)
//uncomment next line
#define SYMB_REG
void print_reg_name(reg_num_t r);
void print_op(func3_t func3, func7_t func7);
void print_op_imm(func3_t func3, func7_t func7);
void print_msize(func3_t func3);
void print_branch(func3_t func3);
void print_reg(int *reg_file);
#endif
#endif
```

The disassembling can be switched on or off by defining/not defining the DEBUG_DISASSEMBLE constant in the debug_fde_ip.h file shown in Listing 5.40.

Other constants can be defined/not defined to switch on or off other debugging prints: debugging print of the fetch function (define DEBUG_FETCH to print the fetch address and the fetched instruction word), debugging print of the execute function (define DEBUG_EMULATE to turn on emulation of the instruction, described in Sect. 5.5.5), debugging print of the register file content (define DEBUG_REG_FILE to print the last content of the register file).

The debugging functions are coded in a way to allow any combination of debugging features (e.g. debug fetch and emulation, without disassembly debug prints).

Listing 5.40 The debug_fde_ip.h file

```
#ifndef __DEBUG_FDE_IP
#define __DEBUG_FDE_IP
//comment the next line to turn off
//fetch debugging prints
#define DEBUG_FETCH
//comment the next line to turn off
//disassembling debugging prints
#define DEBUG_DISASSEMBLE
//comment the next line to turn off
//register file dump debugging prints
#define DEBUG_REG_FILE
//comment the next line to turn off
//emulation debugging prints
#define DEBUG_EMULATE
#endif
```

5.5.3 The IP Running Condition

The running_cond_update function is defined in the fde_ip.cpp file. It is shown in Listing 5.41.

The is_running loop exit condition has been updated. To finish the run, the IP must reach a RET instruction with a cleared return address (next pc should be null, which means the returning function is main).

Listing 5.41 The running_cond_update function

```
static void running_cond_update(
  instruction_t  instruction,
  code_address_t pc,
  bit_t          *is_running){
#pragma HLS INLINE off
  *is_running = (instruction != RET || pc != 0);
}
```

5.5.4 The Fde_ip.h File

The fde_ip.h file contains the constants associated to the register file size, as shown in Listing 5.42.

Listing 5.42 The fde_ip.h file (register file size related constants)

```
...
#define LOG_REG_FILE_SIZE  5
#define NB_REGISTER        (1<<LOG_REG_FILE_SIZE)
...
```

It also contains (see Listing 5.43) the constants defining the RISC-V comparison operators matching the different branch instructions (B-TYPE format, opcode BRANCH). These constants are part of the RISC-V specification (refer to [1], Chap. 24, p. 130, funct3 field).

Listing 5.43 The fde_ip.h file (comparison operator related constants)

```
...
#define BEQ            0
#define BNE            1
#define BLT            4
#define BGE            5
#define BLTU           6
#define BGEU           7
...
```

The arithmetic and logical operator related constants are defined (see Listing 5.44). They match the computation instructions between two register sources (R-TYPE format, opcode OP). These constants are also part of the RISC-V specification (refer to [1], Chap. 24, p. 130, funct3 field).

Listing 5.44 The fde_ip.h file (arithmetic and logical operator related constants for register-register instructions)

```
...
#define ADD            0
#define SUB            0
#define SLL            1
#define SLT            2
#define SLTU           3
#define XOR            4
#define SRL            5
#define SRA            5
#define OR             6
#define AND            7
...
```

The immediate version of the same operators related constants are defined (see Listing 5.45). They match the computation instructions between one register source and one constant (I-TYPE format, opcode OP-IMM). These constants are part of the RISC-V specification too (refer to [1], Chap. 24, p. 130, funct3 field).

Listing 5.45 The fde_ip.h file (arithmetic and logical operator related constants for register-constant instructions)

```
...
#define ADDI          0
#define SLLI          1
#define SLTI          2
#define SLTIU         3
#define XORI          4
#define SRLI          5
#define SRAI          5
#define ORI           6
#define ANDI          7
...
```

The fde_ip.h file also contains the definition of the register file related types reg_num_t and reg_num_p1_t (p1 stands for plus 1), as shown in Listing 5.46.

Listing 5.46 The fde_ip.h file (reg_num_t and reg_num_p1_t types)

```
...
typedef ap_uint<LOG_REG_FILE_SIZE+1> reg_num_p1_t;
typedef ap_uint<LOG_REG_FILE_SIZE>   reg_num_t;
...
```

Type names with the p1_t suffix indicate that the variables of the type use one more bit. Such a plus 1 type is necessary when a variable is used as a loop control.

For example, in "for (i=0; i<16; i++)", variable *i* should be five bits wide to be compared to constant 16, i.e. 0b10000 in binary, even though in the loop *i* ranges between 0 and 15, i.e. requires only four bits. Hence, I would write what is shown in Listing 5.47.

Listing 5.47 Plus 1 type example

```
typedef ap_uint<4> loop_counter_t;
typedef ap_uint<5> loop_counter_p1_t;
loop_counter_p1_t i1;
loop_counter_t    i;
for (i1=0; i1<16; i1++){
  i = i1;//"i1" is shrinked to 4 bits to fit in "i"
  ...//iteration body using "i"
}
```

The synthesizer produces 4-bit values each time variable *i* is used throughout the loop body and only for the loop control, generates a 5-bit value to increment and test variable *i1*.

Other constants and types in the fde_ip.h file are unchanged from the fetching_decoding_ip project.

5.5.5 The Decode Function and the Execute Function

As in the fetching_decoding_ip project, the main loop still contains the five function calls: fetch, decode, execute, statistic_update, and running_cond_update.

The fetch function is unchanged (fetch.cpp file).

The decode function (see Listing 5.48; decode.cpp file) is unchanged except that the debugging print of the decoding has been removed.

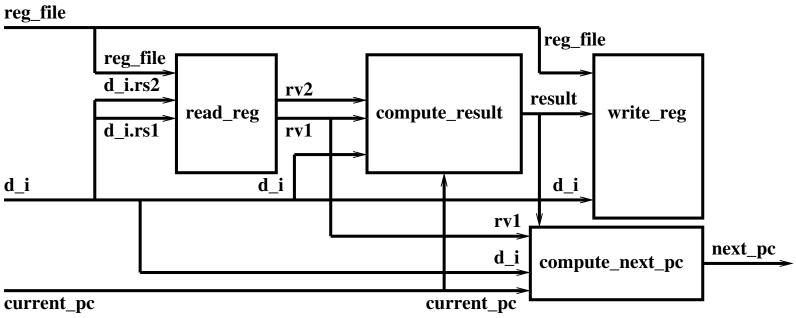


Fig. 5.23 The execute unit

Listing 5.48 The decode function

```
void decode(
  instruction_t           instruction,
  decoded_instruction_t *d_i){
#pragma HLS INLINE off
  decode_instruction(instruction, d_i);
  decode_immediate  (instruction, d_i);
}
```

The execute unit is shown in Fig. 5.23.

It is implemented in the execute function, defined in the execute.cpp file.

In the read_reg function (see Listing 5.50; defined in the execute.cpp file), the rv1 and rv2 values are read from the register file. They are passed to the compute_result function (see Listing 5.51; execute.cpp file). The computed result is written back to the register file by the write_reg function (see Listing 5.50; execute.cpp file).

In the same time, the next pc is computed from the current pc, from the rv1 value read in the read_reg function, and from the least significant bit of the result computed in the compute_result function (see Listing 5.54).

The rv1 value is used for the next pc computation if the decoded instruction is a JALR indirect jump (e.g. "jalr a0" is a call to a function at address a0; rv1 is the function address).

The result bit is used as a branching condition if the decoded instruction is a conditional branch (it is unused otherwise).

For example, in the "beq a0,a1,label" instruction, the least significant bit of result is set or cleared in the compute_result function (see Listing 5.51), depending on the a0==a1 condition. This bit is passed to the compute_next_pc function (fourth argument) and used in the B_TYPE case of the switch instruction as the cond condition to decide on the branch target.

The execute function code is shown in Listing 5.49.

Listing 5.49 The execute function in the execute.cpp file

```
void execute(
  code_address_t         pc,
  int                   *reg_file,
  decoded_instruction_t d_i,
  code_address_t        *next_pc){
#pragma HLS INLINE off
```

```
  int rv1, rv2, result;
  read_reg(reg_file, d_i.rs1, d_i.rs2, &rv1, &rv2);
  result = compute_result(rv1, rv2, d_i, pc);
  write_reg(reg_file, d_i, result);
  *next_pc = compute_next_pc(pc, rv1, d_i, (bit_t)result);
#ifndef __SYNTHESIS__
#ifdef DEBUG_EMULATE
  emulate(reg_file, d_i, *next_pc);
#endif
#endif
}
```

The emulate function is a debugging feature to print the updates to the register file (if the instruction writes to a destination register) and to the pc (if the instruction is a jump or a taken branch).

The emulate function is an equivalent of the spike simulator (the difference between an *emulator* and a *simulator* is not significant enough to deserve more explanation; you can just take the two terms as synonyms in our context).

I do not present the function. You can find its full code in the emulate.cpp file.

5.5.6 The Register File

Figure 5.24 shows the read and write accesses from and to the register file. The write_enable signal enables the writing access to the register file when the signal is set (i.e. when the destination register is not register zero, as specified in the RISC-V ISA, and when the instruction is not a conditional branch; notice that if the instruction is a JAL or a JALR, there is a destination: they write a link address to destination rd).

The code of the read_reg and write_reg functions is shown in Listing 5.50.

Listing 5.50 The read_reg and write_reg functions in the execute.cpp file

```
static void read_reg(
  int      *reg_file,
  reg_num_t rs1,
  reg_num_t rs2,
  int      *rv1,
  int      *rv2){
#pragma HLS INLINE
  *rv1 = reg_file[rs1];
  *rv2 = reg_file[rs2];
}
static void write_reg(
  int                  *reg_file,
  decoded_instruction_t d_i,
  int                   result){
#pragma HLS INLINE
  if (d_i.rd     != 0      &&
      d_i.opcode != BRANCH &&
      d_i.opcode != STORE)
    reg_file[d_i.rd] = result;
}
```

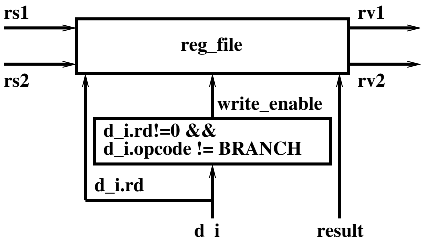


Fig. 5.24 The register file accesses

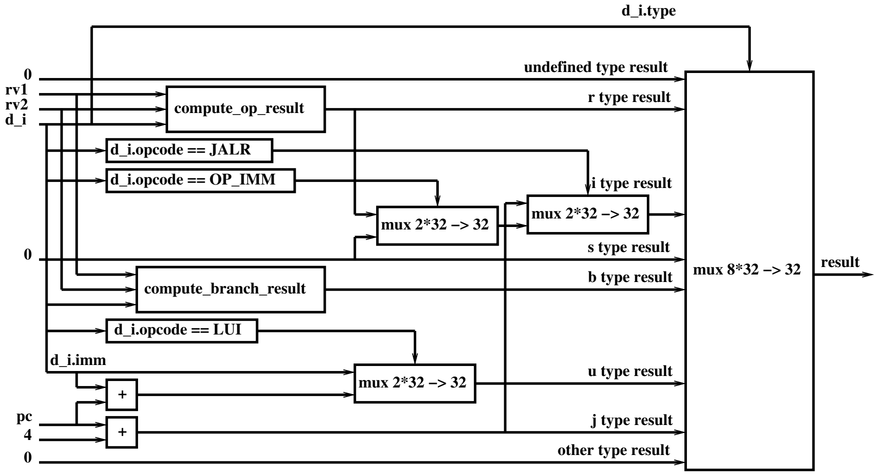


Fig. 5.25 Computing and selecting the instruction result

5.5.7 Computing

Figure 5.25 shows how the different results according to the instruction format are computed and how the final result is selected.

The compute_result function code is shown in Listing 5.51.

Figure 5.25 rightmost multiplexer is implemented as a switch on the d_i.type variable (in the figure, the multiplexer is named mux 8*32 -> 32, meaning that it selects one 32-bit word out of eight).

Listing 5.51 The compute_result function in the execute.cpp file

```
static int compute_result(
  int                   rv1,
  int                   rv2,
  decoded_instruction_t d_i,
  code_address_t        pc){
#pragma HLS INLINE
  int            imm12 = ((int)d_i.imm)<<12;
  code_address_t pc4   = pc<<2;
  code_address_t npc4  = pc4 + 4;
  int            result;
  switch(d_i.type){
    case R_TYPE:
      result = compute_op_result(rv1, rv2, d_i);
      break;
    case I_TYPE:
      if (d_i.opcode == JALR)
        result = npc4;
      else if (d_i.opcode == LOAD)
        result = 0;
      else if (d_i.opcode == OP_IMM)
        result = compute_op_result(rv1, (int)d_i.imm, d_i);
      else
        result = 0;//(d_i.opcode == SYSTEM)
      break;
    case S_TYPE:
      result = 0;
      break;
    case B_TYPE:
      result = (unsigned int)
               compute_branch_result(rv1, rv2, d_i);
      break;
    case U_TYPE:
      if (d_i.opcode == LUI)
        result = imm12;
      else//d_i.opcode == AUIPC
        result = pc4 + imm12;
      break;
    case J_TYPE:
      result = npc4;
      break;
    default:
      result = 0;
      break;
  }
  return result;
}
```

The main block in the compute_result unit is the compute_op_result, devoted to compute the result of the RISC-V OP and OP-IMM opcode operations (ADD/ADDI, SUB, SLL/SLLI, SLT/SLTI, SLTU/SLTIU, XOR/XORI, SRL/SRLI, SRA/SRAI, OR/ORI, and AND/ANDI). It is shown in Fig. 5.26.

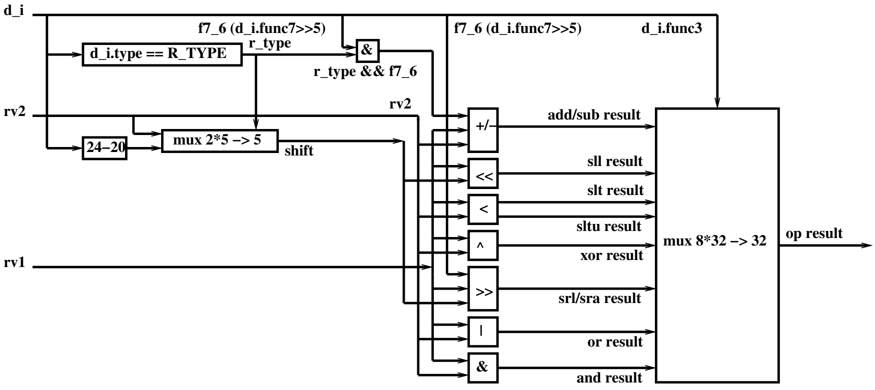


Fig. 5.26 Computing and selecting the OP operation result

It should be understood that the unit computes all the possible results in parallel and the func3 field is used to select the instruction result among them (remember the general programming concepts for HLS in Sect. 5.1: computing more can improve the critical path).

The code to implement the compute_op_result function is shown in Listing 5.52. It is defined in the execute.cpp file.

Listing 5.52 The compute_op_result function in the execute.cpp file

```
static int compute_op_result(
  int                   rv1,
  int                   rv2,
  decoded_instruction_t d_i){
#pragma HLS INLINE
  bit_t      f7_6   = d_i.func7>>5;
  bit_t      r_type = d_i.type == R_TYPE;
  ap_uint<5> shift;
  int        result;
  if (r_type)
    shift = rv2;
  else//I_TYPE
    shift = d_i.rs2;
  switch(d_i.func3){
    case ADD : if (r_type && f7_6)
                 result = rv1 - rv2;//SUB
               else
                 result = rv1 + rv2;
               break;
    case SLL : result = rv1 << shift;
               break;
    case SLT : result = rv1 < rv2;
               break;
    case SLTU: result = (unsigned int)rv1 < (unsigned int)rv2;
               break;
    case XOR : result = rv1 ^ rv2;
               break;
    case SRL : if (f7_6)
                 result = rv1 >> shift;//SRA
               else
                 result = (unsigned int)rv1 >> shift;
               break;
    case OR  : result = rv1 | rv2;
               break;
    case AND : result = rv1 & rv2;
               break;
  }
  return result;
}
```

The compute_branch_result function (see Listing 5.53) computes the branch condition (it returns a bit_t result). Signs are important as the synthesizer will not use the same order comparators (< and >=) for signed and unsigned integers.

Listing 5.53 The compute_branch_result function in the execute.cpp file

```
static bit_t compute_branch_result(
  int                   rv1,
  int                   rv2,
  decoded_instruction_t d_i){
#pragma HLS INLINE
  switch(d_i.func3){
    case BEQ : return (rv1 == rv2);
    case BNE : return (rv1 != rv2);
    case 2   :
    case 3   : return 0;
    case BLT : return (rv1 <  rv2);
    case BGE : return (rv1 >= rv2);
    case BLTU: return (unsigned int)rv1 <  (unsigned int)rv2;
    case BGEU: return (unsigned int)rv1 >= (unsigned int)rv2;
  }
  return 0;
}
```

The compute_next_pc function (see Listing 5.54) is completed to take care of BRANCH and indirect jump instructions (JR/JALR).

Listing 5.54 The compute_next_pc function in the execute.cpp file

```
static code_address_t compute_next_pc(
  code_address_t         pc,
  int                    rv1,
  decoded_instruction_t  d_i,
  bit_t                  cond){
#pragma HLS INLINE
  code_address_t next_pc;
  switch(d_i.type){
    case R_TYPE:
      next_pc = (code_address_t)(pc+1);
      break;
    case I_TYPE:
      next_pc = (d_i.opcode==JALR)?
                (code_address_t)
                (((rv1 + (int)d_i.imm)&0xfffffffe)>>2):
                (code_address_t)(pc+1);
      break;
    case S_TYPE:
      next_pc = (code_address_t)(pc+1);
      break;
    case B_TYPE:
      next_pc = (cond)?
                (code_address_t)(pc + (d_i.imm>>1)):
                (code_address_t)(pc + 1);
      break;
    case U_TYPE:
      next_pc = (code_address_t)(pc + 1);
      break;
    case J_TYPE:
      next_pc = (code_address_t)(pc + (d_i.imm>>1));
      break;
    default:
      next_pc = (code_address_t)(pc + 1);
      break;
  }
  return next_pc;
}
```

5.5.8 Simulating the Fde_ip With the Testbench

Experimentation

To simulate the fde_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with fde_ip.

You can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

I have added six new test programs: test_branch.s to test the BRANCH instructions, test_jal_jalr.s to test the JAL and JALR instructions, test_lui_auipc.s to test LUI and AUIPC instructions, test_op.s to test OP instructions, and test_sum.s to sum the 10 first natural numbers. These test programs will be used to test all the processor IPs (more test programs will be added in the next chapter for the data memory manipulations).

The testbench_fde_ip.cpp file shown in Listing 5.55 is applied to the test_op_imm_0_text.hex hexadecimal code file (obtained from the test_op_imm.s source).

To run another test code, you just need to replace the test_op_imm_0_text.hex name in the code_ram array declaration. To build any .hex file, use the build.sh script: "./build.sh test_branch" builds test_branch_0_text.hex (do not pay attention to the warning message).

Anyway, the fde_ip folder contains prebuilt hex files for all the proposed test codes.

Listing 5.55 The testbench_fde_ip.cpp file

```
#include <stdio.h>
#include "fde_ip.h"
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_sum_0_text.hex"
};
int main(){
  unsigned int nbi;
  fde_ip(0, code_ram, &nbi);
  printf("%d fetched, decoded and executed instructions\n",
          nbi);
  return 0;
}
```

The test_branch.s file contains a code to test the branch instructions (see Listing 5.56).

Listing 5.56 The test_branch.s file

```
       .globl  main
main:
       li      a0,-8       /*a0=-8*/
       li      a1,5        /*a1=5*/
       beq     a0,a1,.L1   /*if (a0==a1) goto .L1*/
       li      a2,1        /*a2=1*/
.L1:
       bne     a0,a1,.L2   /*if (a0!=a1) goto .L2*/
       li      a2,2        /*a2=2*/
.L2:
       blt     a0,a1,.L3   /*if (a0<a1) goto .L3*/
       li      a3,1        /*a3=1*/
.L3:
       bge     a0,a1,.L4   /*if (a0>=a1) goto .L4*/
       li      a3,2        /*a3=2*/
.L4:
       bltu    a0,a1,.L5   /*if (a0<a1) goto .L5  (unsigned)*/
       li      a4,1        /*a4=1*/
.L5:
       bgeu    a0,a1,.L6   /*if (a0>=a1) goto .L6 (unsigned)*/
       li      a4,2        /*a4=2*/
.L6:
       ret
```

Its run produces the output shown in Listing 5.57 (SYMB_REG defined in the print.h file; registers containing a null value are not shown).

Listing 5.57 The test_branch.s code output

```
0000: ff800513      li a0, -8
     a0   =               -8 (fffffff8)
0004: 00500593      li a1, 5
     a1   =                5 (        5)
0008: 00b50463      beq a0, a1, 16
     pc   =               12 (        c)
0012: 00100613      li a2, 1
     a2   =                1 (        1)
0016: 00b51463      bne a0, a1, 24
     pc   =               24 (       18)
0024: 00b54463      blt a0, a1, 32
     pc   =               32 (       20)
0032: 00b55463      bge a0, a1, 40
     pc   =               36 (       24)
0036: 00200693      li a3, 2
     a3   =                2 (        2)
0040: 00b56463      bltu a0, a1, 48
     pc   =               44 (       2c)
0044: 00100713      li a4, 1
     a4   =                1 (        1)
0048: 00b57463      bgeu a0, a1, 56
     pc   =               56 (       38)
0056: 00008067      ret
     pc   =                0 (        0)
...
a0   =               -8 (fffffff8)
a1   =                5 (       5)
a2   =                1 (       1)
a3   =                2 (       2)
a4   =                1 (       1)
...
12 fetched and decoded instructions
```

The test_jal_jalr.s file (see Listing 5.58) contains a code to test the jump and link instructions JAL and JALR.

Listing 5.58 The test_jal_jalr.s file

```
       .globl  main
main:
       mv      t0,ra     /*t0=ra (save return address)*/
here0:
       auipc   a0,0      /*a0=pc+0   (a0=4)*/
here1:
       auipc   a1,0      /*a1=pc+0   (a0=8)*/
       li      a2,0      /*a2=0*/
       li      a4,0      /*a4=0*/
       j       .L1       /*goto .L1*/
.L1:
       addi    a2,a2,1   /*a2++*/
       jal     f         /*f()            (call f)*/
       li      a3,3      /*a3=3*/
       jalr    52(a1)    /*(*(a1+52))()   (call f)*/
       jr      44(a0)    /*goto *(a0+44)  (goto there)*/
       addi    a4,a4,1   /*a4++*/
there:
       addi    a4,a4,1   /*a4++*/
       mv      ra,t0     /*ra=t0 (restore return address)*/
       ret
f:
       addi    a2,a2,1   /*a2++*/
       ret
```

It produces the output shown in Listing 5.59 (SYMB_REG defined; registers containing a null value are not shown).

Listing 5.59 The test_jal_jalr.s code output

```
0000: 00008293      addi t0, ra, 0
     t0  =                0 (       0)
0004: 00000517      auipc a0, 0
     a0  =                4 (       4)
0008: 00000597      auipc a1, 0
     a1  =                8 (       8)
0012: 00000613      li a2, 0
     a2  =                0 (       0)
0016: 00000713      li a4, 0
     a4  =                0 (       0)
0020: 0040006f      j 24
     pc  =               24 (      18)
0024: 00160613      addi a2, a2, 1
     a2  =                1 (       1)
0028: 020000ef      jal ra, 60
     pc  =               60 (      3c)
     ra  =               32 (      20)
0060: 00160613      addi a2, a2, 1
     a2  =                2 (       2)
0064: 00008067      ret
     pc  =               32 (      20)
0032: 00300693      li a3, 3
     a3  =                3 (       3)
0036: 034580e7      jalr 52(a1)
     pc  =               60 (      3c)
     ra  =               40 (      28)
0060: 00160613      addi a2, a2, 1
     a2  =                3 (       3)
0064: 00008067      ret
     pc  =               40 (      28)
0040: 02c50067      jr 44(a0)
     pc  =               48 (      30)
0048: 00170713      addi a4, a4, 1
     a4  =                1 (       1)
0052: 00028093      addi ra, t0, 0
     ra  =                0 (       0)
0056: 00008067      ret
```

```
     pc  =                0  (        0)
...
a0  =                4  (        4)
a1  =                8  (        8)
a2  =                3  (        3)
a3  =                3  (        3)
a4  =                1  (        1)
...
18 fetched and decoded instructions
```

The test_lui_auipc.s file (see Listing 5.60) contains a code to test the upper instructions LUI and AUIPC.

Listing 5.60 The test_lui_auipc.s file

```
       .globl  main
main:
       lui     a1,0x1     /*a1=(1<<12)        (4096)*/
       auipc   a2,0x1     /*a2=pc+(1<<12)  (pc+4096)*/
       sub     a2,a2,a1   /*a2-=a1*/
       addi    a2,a2,20   /*a2+=20*/
       jr      a2         /*goto a2  (.L1)*/
       li      a1,3       /*a1=3*/
.L1:
       li      a3,100     /*a3=100*/
       ret
```

It produces the output shown in Listing 5.61 (SYMB_REG defined; registers containing a null value are not shown).

Listing 5.61 The test_lui_auipc.s code output

```
0000: 000015b7      lui a1, 4096
     a1  =             4096  (     1000)
0004: 00001617      auipc a2, 4096
     a2  =             4100  (     1004)
0008: 40b60633      sub a2, a2, a1
     a2  =                4  (        4)
0012: 01460613      addi a2, a2, 20
     a2  =               24  (       18)
0016: 00060067      jr a2
     pc  =               24  (       18)
0024: 06400693      li a3, 100
     a3  =              100  (       64)
0028: 00008067      ret
     pc  =                0  (        0)
...
a1  =             4096  (     1000)
a2  =               24  (       18)
a3  =              100  (       64)
...
7 fetched and decoded instructions
```

The test_op.s file (see Listing 5.62) contains a code to test the OP instructions (register-register operations, i.e. with two register sources and no immediate value).

Listing 5.62 The test_op.s file

```
       .globl  main
main:
       li      a0,13       /*a0=13*/
       li      a4,12       /*a4=12*/
       li      a1,7        /*a1=7*/
```

```
        li      t0,28       /*t0=28*/
        li      t6,-10      /*t6=-10*/
        li      s2,2022     /*s2=2022*/
        add     a2,a1,zero  /*a2=a1*/
        and     a3,a2,a0    /*a3=a2&a0*/
        or      a5,a3,a4    /*a5=a3|a4*/
        xor     a6,a5,t0    /*a6=a5^t0*/
        sub     a6,a6,a1    /*a6-=a1*/
        sltu    a7,a6,a0    /*a7=a6<a0  (unsigned)*/
        sll     t1,a6,t0    /*t1=a6<<t0*/
        slt     t2,t1,t6    /*t2=t1<t6    (signed)*/
        sltu    t3,t1,s2    /*t3=t1<s2  (unsigned)*/
        srl     t4,t1,t0    /*t4=t1>>t0 (unsigned)*/
        sra     t5,t1,t0    /*t5=t1>>t0   (signed)*/
        ret
```

It produces the output shown in Listing 5.63 (SYMB_REG defined; registers containing a null value are not shown).

Listing 5.63 The test_op.s code output

```
0000: 00d00513      li a0, 13
     a0  =               13 (        d)
0004: 00c00713      li a4, 12
     a4  =               12 (        c)
0008: 00700593      li a1, 7
     a1  =                7 (        7)
0012: 01c00293      li t0, 28
     t0  =               28 (       1c)
0016: ff600f93      li t6, -10
     t6  =              -10 (fffffff6)
0020: 7e600913      li s2, 2022
     s2  =             2022 (      7e6)
0024: 00058633      add a2, a1, zero
     a2  =                7 (        7)
0028: 00a676b3      and a3, a2, a0
     a3  =                5 (        5)
0032: 00e6e7b3      or a5, a3, a4
     a5  =               13 (        d)
0036: 0057c833      xor a6, a5, t0
     a6  =               17 (       11)
0040: 40b80833      sub a6, a6, a1
     a6  =               10 (        a)
0044: 00a838b3      sltu a7, a6, a0
     a7  =                1 (        1)
0048: 00581333      sll t1, a6, t0
     t1  =      -1610612736 (a0000000)
0052: 01f323b3      slt t2, t1, t6
     t2  =                1 (        1)
0056: 01233e33      sltu t3, t1, s2
     t3  =                0 (        0)
0060: 00535eb3      srl t4, t1, t0
     t4  =               10 (        a)
0064: 40535f33      sra t5, t1, t0
     t5  =               -6 (fffffffa)
0068: 00008067      ret
     pc  =                0 (        0)
...
t0  =               28 (       1c)
t1  =      -1610612736 (a0000000)
t2  =                1 (        1)
...
a0  =               13 (        d)
```

```
a1  =                7 (       7)
a2  =                7 (       7)
a3  =                5 (       5)
a4  =               12 (       c)
a5  =               13 (       d)
a6  =               10 (       a)
a7  =                1 (       1)
s2  =             2022 (     7e6)
...
t4  =               10 (       a)
t5  =               -6 (fffffffa)
t6  =              -10 (fffffff6)
18 fetched and decoded instructions
```

The test_op_imm.s file has already been presented (refer back to Listing 3.13).

It produces the output shown in Listing 5.64 (SYMB_REG defined; registers containing a null value are not shown).

Listing 5.64 The test_op_imm.s code output

```
0000: 00500593      li a1, 5
     a1  =                5 (       5)
0004: 00158613      addi a2, a1, 1
     a2  =                6 (       6)
0008: 00c67693      andi a3, a2, 12
     a3  =                4 (       4)
0012: fff68713      addi a4, a3, -1
     a4  =                3 (       3)
0016: 00576793      ori a5, a4, 5
     a5  =                7 (       7)
0020: 00c7c813      xori a6, a5, 12
     a6  =               11 (       b)
0024: 00d83893      sltiu a7, a6, 13
     a7  =                1 (       1)
0028: 00b83293      sltiu t0, a6, 11
     t0  =                0 (       0)
0032: 01c81313      slli t1, a6, 28
     t1  =      -1342177280 (b0000000)
0036: ff632393      slti t2, t1, -10
     t2  =                1 (       1)
0040: 7e633e13      sltiu t3, t1, 2022
     t3  =                0 (       0)
0044: 01c35e93      srli t4, t1, 28
     t4  =               11 (       b)
0048: 41c35f13      srai t5, t1, 28
     t5  =               -5 (fffffffb)
0052: 00008067      ret
     pc  =                0 (       0)
...
t1  =      -1342177280 (b0000000)
t2  =                1 (       1)
...
a1  =                5 (       5)
a2  =                6 (       6)
a3  =                4 (       4)
a4  =                3 (       3)
a5  =                7 (       7)
a6  =               11 (       b)
a7  =                1 (       1)
...
t4  =               11 (       b)
t5  =               -5 (fffffffb)
```

```
...
14 fetched and decoded instructions
```

The test_sum.s file (see Listing 5.65) contains a code to sum the 10 first integers into register a0 (x10).

Listing 5.65 The test_sum.s file

```
       .globl  main
main:
       li      a0,0        /*a0=0*/
       li      a1,0        /*a1=0*/
       li      a2,10       /*a2=10*/
.L1:
       addi    a1,a1,1     /*a1++*/
       add     a0,a0,a1    /*a0+=a1*/
       bne     a1,a2,.L1   /*if (a1!=a2) goto .L1*/
       ret
```

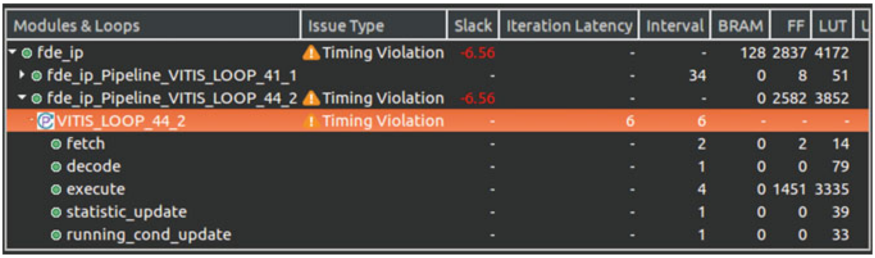
It produces the output shown in Listing 5.66 (SYM_REG defined; registers containing a null value are not shown; intermediate iterations are not shown).

Listing 5.66 The test_sum.s code output

```
0000: 00000513      li a0, 0
     a0  =                0 (       0)
0004: 00000593      li a1, 0
     a1  =                0 (       0)
0008: 00a00613      li a2, 10
     a2  =               10 (       a)
0012: 00158593      addi a1, a1, 1
     a1  =                1 (       1)
0016: 00b50533      add a0, a0, a1
     a0  =                1 (       1)
0020: fec59ce3      bne a1, a2, 12
     pc  =               12 (       c)
...
0012: 00158593      addi a1, a1, 1
     a1  =               10 (       a)
0016: 00b50533      add a0, a0, a1
     a0  =               55 (      37)
0020: fec59ce3      bne a1, a2, 12
     pc  =               24 (      18)
0024: 00008067      ret
     pc  =                0 (       0)
...
a0  =               55 (      37)
a1  =               10 (       a)
a2  =               10 (       a)
...
34 fetched and decoded instructions
```

5.5.9 The Fde_ip Synthesis

For the synthesis, all the functions are inlined (pragma HLS INLINE) except the highest level ones fetch, decode, execute, statistic_update, and running_cond_update. Figure 5.27 shows the synthesis report.



Modules & Loops	Issue Type	Slack	Iteration Latency	Interval	BRAM	FF	LUT	U
fde_ip	Timing Violation	-6.56	-	-	128	2837	4172	
fde_ip_Pipeline_VITIS_LOOP_41_1		-	-	34	0	8	51	
fde_ip_Pipeline_VITIS_LOOP_44_2	Timing Violation	-6.56	-	-	0	2582	3852	
VITIS_LOOP_44_2	Timing Violation	-	6	6	-	-	-	
fetch		-	-	2	0	2	14	
decode		-	-	1	0	0	79	
execute		-	-	4	0	1451	3335	
statistic_update		-	-	1	0	0	39	
running_cond_update		-	-	1	0	0	33	

Fig. 5.27 Synthesis report for the fde_ip

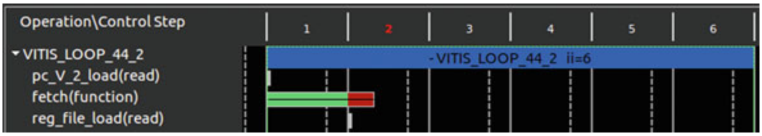


Fig. 5.28 The fde_ip schedule

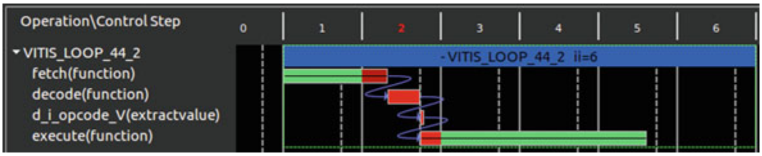


Fig. 5.29 Timing Violation

The Schedule Viewer shows that the IP cycle corresponds to six FPGA cycles (see Fig. 5.28) as requested by the HLS PIPELINE II=6 pragma.

The synthesis report shows a Timing Violation warning. This is not critical as it concerns operations inside the processor cycle and does not impact the six cycles scheduling. The synthesizer could not fit the end of the fetch function, the decode one, and the beginning of the execute function within loop cycle 2 (see Fig. 5.29).

5.5.10 The Z1_fde_ip Vivado Project

The z1_fde_ip Vivado project defines the design shown in Fig. 5.30.

The bitstream generation shows a resource utilization of 3313 LUTs, 6.23% of the available LUTs on the FPGA (see Fig. 5.31).

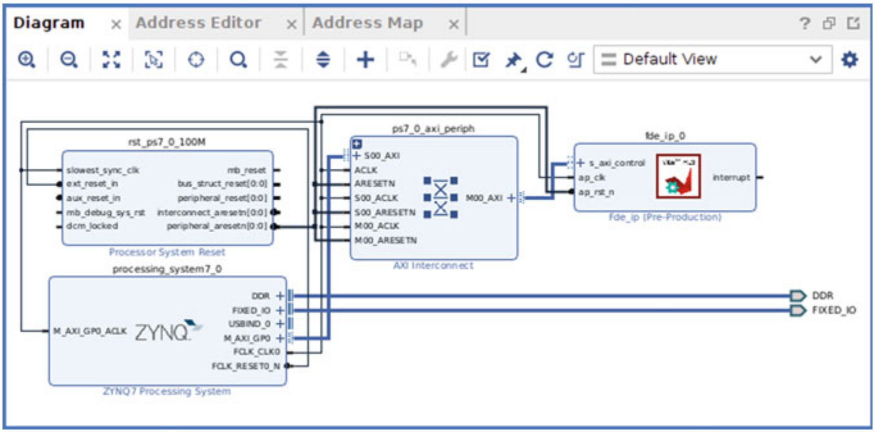
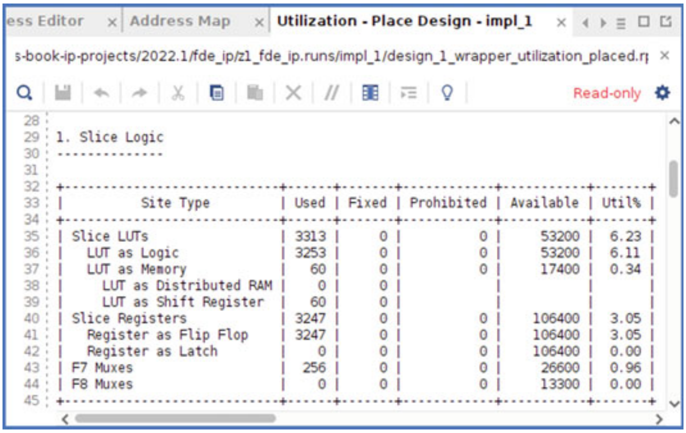


Fig. 5.30 The z1_fde_ip block design



```
s-book-ip-projects/2022.1/fde_ip/z1_fde_ip.runs/impl_1/design_1_wrapper_utilization_placed.rp
```

```

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 3313 |     0 |          0 |     53200 |  6.23 |
|   LUT as Logic             | 3253 |     0 |          0 |     53200 |  6.11 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 3247 |     0 |          0 |    106400 |  3.05 |
|   Register as Flip Flop    | 3247 |     0 |          0 |    106400 |  3.05 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  256 |     0 |          0 |     26600 |  0.96 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 5.31 Vivado utilization report for the fde_ip IP

5.5.11 The Helloworld.c Program to Drive the Fde_ip on the FPGA

Experimentation

To run the fde_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with fde_ip.

You can play with your IP, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

The code in the helloworld.c file is shown in Listing 5.67 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script).

Listing 5.67 The helloworld.c file

```c
#include <stdio.h>
#include "xfde_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
XFde_ip_Config *cfg_ptr;
XFde_ip         ip;
word_type code_ram[CODE_RAM_SIZE]={
#include "test_op_imm_0_text.hex"
};
int main(){
  cfg_ptr = XFde_ip_LookupConfig(XPAR_XFDE_IP_0_DEVICE_ID);
  XFde_ip_CfgInitialize(&ip, cfg_ptr);
  XFde_ip_Set_start_pc(&ip, 0);
  XFde_ip_Write_code_ram_Words(&ip, 0, code_ram, CODE_RAM_SIZE);
  XFde_ip_Start(&ip);
  while (!XFde_ip_IsDone(&ip));
  printf("%d fetched, decoded and executed instructions\n",
    (int)XFde_ip_Get_nb_instruction(&ip));
}
```

If you run the RISC-V code in the test_op_imm_0_text.hex file, your putty terminal should print what is shown in Listing 5.68.

Listing 5.68 The FPGA execution print on the putty terminal for the run of the test_op_imm_0_text.hex RISC-V code

```
14 fetched, decoded and executed instructions
```

Reference

1. https://riscv.org/specifications/isa-spec-pdf/

Building a RISC-V Processor

6

Abstract

This chapter makes you build your first RISC-V processor. The implemented microarchitecture proposed in this first version is not pipelined. The IP cycle encompasses the fetch, the decoding, and the execution of an instruction.

6.1 The Rv32i_npp_ip Top Function

The Vitis_HLS project is named rv32i_npp_ip. It will be your first implementation of the RV32I subset of the RISC-V ISA (npp stands for non pipelined).

All the source files related to the rv32i_npp_ip can be found in the rv32i_npp_ip folder.

6.1.1 The Rv32i_npp_ip Top Function Prototype, Local Declarations, and Initializations

The rv32i_npp_ip top function is defined in the rv32i_npp_ip.cpp file.

The rv32i_npp_ip top function prototype, local declarations, and initializations are shown in Listing 6.1.

A data_ram has been added to the arguments. It is the array where the code reads and writes data through LOAD and STORE instructions.

Listing 6.1 The rv32i_npp_ip top function prototype, local declarations, and initializations

```
void rv32i_npp_ip(
  unsigned int   start_pc,
  unsigned int   code_ram[CODE_RAM_SIZE],
  int            data_ram[DATA_RAM_SIZE],
  unsigned int  *nb_instruction){
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=code_ram
```

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_6

```
#pragma HLS INTERFACE s_axilite port=data_ram
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INLINE recursive
  code_address_t        pc;
  int                   reg_file[NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file dim=1 complete
  instruction_t         instruction;
  bit_t                 is_running;
  unsigned int          nbi;
  decoded_instruction_t d_i;
  for (int i=0; i<NB_REGISTER; i++) reg_file[i] = 0;
  pc  = start_pc;
  nbi = 0;
  ...
```

A legitimate question is: why two memory arrays (code_ram and data_ram)? Why do I separate the data from the code? In a processor, there is usually one memory, shared by the code and the data. When the compiler builds the executable file, it maps both the code and the data on the same memory space.

We need two separate spaces because we need two accesses in the same time. If the processor has a single memory array with a single port, the access for the code (i.e. the instruction fetch) and the data access (i.e. the execution of a LOAD or a STORE instruction) must be done at different moments, i.e. should start at two different FPGA cycles.

The rv32i_npp_ip processor is not pipelined. Hence, the processing of each instruction goes through multiple successive steps: fetch, decode, execution with memory access for loads/stores, and writeback. It is easy to separate fetch from execution in time and have the fetch access start at a different FPGA cycle than the load/store access. So, for the rv32i_npp_ip implementation, I could have used a single memory. But Chap. 8 will introduce a pipelined design in which in the same FPGA cycle the processor starts the fetch phase and the load/store one.

However, even though both accesses are simultaneous there is a way to avoid separating the data and the code memory on Xilinx FPGAs. Each BRAM (Block RAM) has two access ports, allowing to use one port for a fetch and the other for a memory access. But I will use the second port for another purpose in the second part of the book, when implementing multicore processors. The second port is used to provide a remote access (a core *i* accesses a memory word inside a core *j* data memory).

So, to avoid having different memory models across the various implementations, I decided to have separate code and data memories. This is not much different from a classic processor which has separate code and data first level caches.

This will impact the way the executable files are built but I will explain how to adapt to that later (see Sect. 7.3.1).

The DATA_RAM_SIZE constant in the rv32i_npp_ip.h file defines the data memory size as 2^{16} words (2^{18} bytes, 256 KB) (if the RISC-V processor implementations are to be tested on a Basys3 development board, this size should be reduced to 64 KB because the XC7A35T FPGA has only 200 KB of RAM: set LOG_DATA_RAM_SIZE to 14, i.e. 16K words, in all the designs).

The data memory is externally accessed through the s_axilite interface protocol. Hence, I will use the AXI connection to externally view the content of the memory after the run.

To optimize the iteration time, inlining has been systematically enabled with the pragma HLS INLINE recursive added after the HLS INTERFACE pragmas.

The *recursive* option for the INLINE pragma implies inlining for all the functions called by the top function.

6.1.2 The Do ... While Loop

The rv32i_npp_ip top function do ... while loop code is shown in Listing 6.2.

Listing 6.2 The rv32i_npp_ip top function do ... while loop

```
  ...
  do{
#pragma HLS PIPELINE II=7
    fetch(pc, code_ram, &instruction);
    decode(instruction, &d_i);
#ifndef __SYNTHESIS__
#ifdef DEBUG_DISASSEMBLE
    disassemble(pc, instruction, d_i);
#endif
#endif
    execute(pc, reg_file, data_ram, d_i, &pc);
    statistic_update(&nbi);
    running_cond_update(instruction, pc, &is_running);
  } while (is_running);
  *nb_instruction = nbi;
#ifndef __SYNTHESIS__
#ifdef DEBUG_REG_FILE
  print_reg(reg_file);
#endif
#endif
}
```

The processor cycle is set as seven FPGA cycles (pragma HLS PIPELINE II=7).

6.2 Decoding Update

The fetch function is defined in the fetch.cpp file. It is unchanged (see Sect. 5.3.3).

I have added some precomputed Boolean fields as the result of the decoding of the opcode and added them to the decoded_instruction_t type (e.g. "d_i.opcode == LOAD" sets bit_t variable is_load). It is less expensive to drive a single bit than to compare 5-bit values multiple times.

I have added the necessary decodings to replace all the opcode comparisons found in the execute function and its dependencies. There are eight new bit fields in the definition of the decoded_instruction_t type in the rv32i_npp_ip.h file. The new definition is shown in Listing 6.3.

Listing 6.3 The decoded_instruction_t type

```
label
typedef struct decoded_instruction_s{
  opcode_t    opcode;
  ...
  immediate_t imm;
  bit_t       is_load;
  bit_t       is_store;
  bit_t       is_branch;
  bit_t       is_jalr;
  bit_t       is_op_imm;
  bit_t       is_lui;
  bit_t       is_ret;
  bit_t       is_r_type;
} decoded_instruction_t;
```

The decode_instruction function is defined in the decode.cpp file. It is extended to fill the new fields, as shown in Listing 6.4.

Listing 6.4 The decoded_instruction function

```
label
static void decode_instruction(
  instruction_t          instruction,
  decoded_instruction_t *d_i){
  d_i->opcode     = (instruction >>  2);
  ...
  d_i->func7      = (instruction >> 25);
  d_i->is_load    = (d_i->opcode == LOAD);
  d_i->is_store   = (d_i->opcode == STORE);
  d_i->is_branch  = (d_i->opcode == BRANCH);
  d_i->is_jalr    = (d_i->opcode == JALR);
  d_i->is_ret     = (instruction == RET);
  d_i->is_lui     = (d_i->opcode == LUI);
  d_i->is_op_imm  = (d_i->opcode == OP_IMM);
  d_i->type       = type(d_i->opcode);
  d_i->is_r_type  = (d_i->type   == R_TYPE);
}
```

6.3 Data Memory Accesses: Alignment and Endianness

The instruction memory is an ordered set of instructions. In the RV32I ISA, an instruction is a 32-bit word. Hence, the memory is built as a set of 32-bit words. The fetch function reads aligned words as was presented in Sect. 5.3.4.

The data memory is an ordered set of *bytes*. Adjacent bytes can be grouped by two, to form *half words* (16 bits). They can also be grouped by four, to form *words* (32 bits). Half words can be made of two bytes *aligned* on a 16-bit word boundary, as shown on the left part of Fig. 6.1, or *unaligned* as shown on the right part of Fig. 6.1. A half word is aligned if its address is even, i.e. the least significant bit is 0.

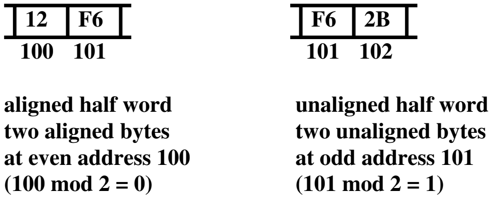


Fig. 6.1 Aligned and unaligned half words

Similarly, a word is aligned if the two least significant bits of its address are both 0, as shown in Fig. 6.2.

When a word is stored to memory, its bytes can be written in two opposite orders.

A *little endian* processor writes the bytes starting with the least significant one (i.e. the least significant byte is written to the byte with the lowest address).

A *big endian* processor writes the bytes starting with the most significant one (i.e. the most significant byte is written to the byte with the lowest address).

Figure 6.3 shows the difference between little and big endian stores.

A little endian processor loads the byte at the lowest address (i.e. byte 0xc3 in the left part of Fig. 6.3) and writes it to the least significant byte position in the destination register (0x12F62BC3 is loaded into the destination register).

A big endian processor loads and writes it (i.e. byte 0x12 in the right part of Fig. 6.3) to the highest significant byte position (0x12F62BC3 is loaded into the destination register).

The RISC-V specification does not precise the endianness of the implementation. From the C code used to define the load and store RISC-V operations (in Sect. 6.4), the Vitis HLS synthesizer builds a little endian byte ordering.

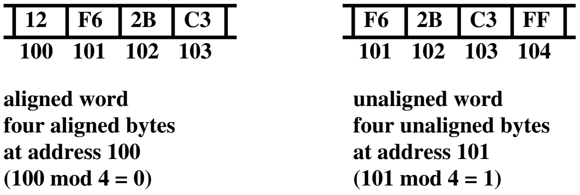


Fig. 6.2 Aligned and unaligned words

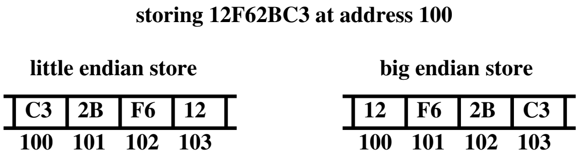


Fig. 6.3 Little and big endian

6.4 The Execute Function

6.4.1 The Computation of the Accessed Address

The execute function (see Listing 6.5; defined in the execute.cpp file) has the data memory pointer as a new argument (data_ram). It also includes the execution of the LOAD and STORE instructions (calls to the mem_load and mem_store functions).

Listing 6.5 The execute function

```
void execute(
  code_address_t        pc,
  int                  *reg_file,
  int                  *data_ram,
  decoded_instruction_t d_i,
  code_address_t       *next_pc){
  int              rv1, rv2, result;
  b_data_address_t address;
  read_reg(reg_file, d_i.rs1, d_i.rs2, &rv1, &rv2);
  result  = compute_result(rv1, rv2, d_i, pc);
  address = result;
  if (d_i.is_store)
    mem_store(data_ram, address, rv2, (ap_uint<2>)d_i.func3);
  if (d_i.is_load)
    result = mem_load(data_ram, address, d_i.func3);
  write_reg(reg_file, d_i, result);
  *next_pc = compute_next_pc(pc, rv1, d_i, (bit_t)result);
#ifndef __SYNTHESIS__
#ifdef DEBUG_EMULATE
  emulate(reg_file, d_i, *next_pc);
#endif
#endif
}
```

In case of a memory access instruction (e.g. "lw a0,4(a1)"), the compute_result function returns the access address which is the sum of the rs1 register value and an immediate offset (e.g. a1 + 4 for the "lw a0,4(a1)" instruction).

After the access address has been computed, the memory access itself can take place. The mem_store function or the mem_load function is called according to the d_i.is_load and d_i.is_store pre-computed bits.

The mem_load function returns the loaded value which is written to the destination register in the write_reg function.

The mem_store function writes the rv2 value to be stored to the data_ram array.

6.4.2 The Compute_result Function

The compute_result function (see Listing 6.6; defined in the execute.cpp file) is completed to take care of the S-TYPE instructions (STORE opcode) and the LOAD variant of the I-TYPE format.

In both cases, the computed result is the accessed address, obtained from the sum of rv1 and d_i.imm.

Listing 6.6 The compute_result function

```
static int compute_result(
  int                   rv1,
  int                   rv2,
  decoded_instruction_t d_i,
  code_address_t        pc){
  int            imm12 = ((int)d_i.imm)<<12;
  code_address_t pc4   = pc<<2;
  code_address_t npc4  = pc4 + 4;
  int            result;
  switch(d_i.type){
    ...
    case I_TYPE:
      if (d_i.is_jalr)
        result = npc4;
      else if (d_i.is_load)
        result = rv1 + (int)d_i.imm;
      else if (d_i.is_op_imm)
        result = compute_op_result(rv1, (int)d_i.imm, d_i);
      else
        result = 0;//(d_i.opcode == SYSTEM)
      break;
    case S_TYPE:
      result = rv1 + (int)d_i.imm;
      break;
    ...
  }
  return result;
}
```

The compute_next_pc and the compute_op_result functions (both defined in the execute.cpp file) are updated to use the new bit fields in the instruction decoding (d_i.is_jalr and d_i.is_r_type).

6.4.3 The Mem_store Function

The mem_store function (see Listing 6.7; defined in the execute.cpp file) is organized as a switch on the access width msize, i.e. the 3-bit func3 field.

The value to be stored is either the lower byte rv2_0 (SB or store byte), the lower half word rv2_01 (SH or store half word) or the full rv2 value (SW or store word).

The write address is either a byte pointer (char *), a half word pointer (short *), or a word pointer (int *).

The synthesizer sets the byte enables to restraint the writes into the addressed word to the concerned bytes (a single byte anywhere in the addressed word is enabled for SB, a pair of aligned adjacent bytes, i.e. either at the start or at the end of the addressed word are enabled for SH, or all the bytes of the addressed word are enabled for SW).

The RISC-V specification leaves the implementation free to decide what to do with misaligned accesses.

In my implementation the two least significant bits of the address are discarded (two bits right shift) before a SW or LW access, forcing the 4-byte boundary alignment. For SH, LH, and LHU, the least significant bit of the address is discarded (one bit right shift), forcing the 2-byte boundary alignment.

So, even though an access address is misaligned, the processor performs an aligned access (e.g. if the address is $4a + 3$, the accessed word is the aligned word at address $4a$, composed of bytes $4a$, $4a + 1$, $4a + 2$ and $4a + 3$).

Listing 6.7 The mem_store function

```
label
static void mem_store(
  int                *data_ram,
  b_data_address_t    address,
  int                 rv2,
  ap_uint<2>          msize){
  h_data_address_t    a1 = (address >> 1);
  w_data_address_t    a2 = (address >> 2);
  char                rv2_0;
  short               rv2_01;
  rv2_0  = rv2;
  rv2_01 = rv2;
  switch(msize){
    case SB:
      *((char*) (data_ram) + address) = rv2_0;
      break;
    case SH:
      *((short*)(data_ram) + a1)      = rv2_01;
      break;
    case SW:
      data_ram[a2]                    = rv2;
      break;
    case 3:
      break;
  }
}
```

In other words, the programmer should align his/her multibyte data. For example, a structure should be padded to avoid misalignment. Listing 6.8 shows an example of such a padded structure. Two padding fields of one byte each (*pad1* and *pad2*) are added to ensure the alignment of the *j* field in both *s1* and *s2* variables.

Listing 6.8 A structure with aligned fields

```
//it is assumed that the start of the s1 structure is word aligned
struct s_s{
  int   i;//word aligned
  short s;//half word aligned
  char  pad1;//byte aligned
  char  pad2;//byte aligned
  int   j;//word aligned, thanks to the pad1 and pad2 fields
} s1, s2;
```

The addresses for LOAD and STORE instructions are either word addresses with type w_data_address_t, or h_data_address_t with one more bit for half word addresses, or b_data_address_t with two more bits for byte addresses.

These types are defined in the rv32i_npp_ip.h file (see Listing 6.9).

Listing 6.9 The data memory type definitions in the rv32i_npp_ip.h file

```
...
typedef ap_uint<LOG_DATA_RAM_SIZE>   w_data_address_t;
typedef ap_uint<LOG_DATA_RAM_SIZE+1> h_data_address_t;
typedef ap_uint<LOG_DATA_RAM_SIZE+2> b_data_address_t;
...
```

6.4.4 The Mem_load Function

The mem_load function (see Listing 6.10; defined in the execute.cpp file) works differently from the store function.

The store function writes the value to the addressed bytes and only them are accessed, thanks to the byte write enable bits.

The implementation of the load function accesses a full aligned word, from which the bytes requested by the LW, LH, LHU, LB, or LBU instruction are extracted.

The load function accesses four bytes aligned on the addressed word boundary (the address argument is a byte address, from which a word address a2 is derived with a two bits right shift).

Listing 6.10 The mem_load function: load the addressed word

```
label
static int mem_load(
  int               *data_ram,
  b_data_address_t  address,
  func3_t           msize){
  ap_uint<2>        a01 =  address;
  bit_t             a1  = (address >> 1);
  w_data_address_t  a2  = (address >> 2);
  int               result;
  char              b, b0, b1, b2, b3;
  unsigned char     ub, ub0, ub1, ub2, ub3;
  short             h, h0, h1;
  unsigned short    uh, uh0, uh1;
  int               w, ib, ih;
  unsigned int      iub, iuh;
  w    = data_ram[a2];
  b0   = w;
  ub0  = b0;
  b1   = w>>8;
  ub1  = b1;
  h0   = ((ap_uint<16>)ub1<<8) | (ap_uint<16>)ub0;
  uh0  = h0;
  b2   = w>>16;
  ub2  = b2;
  b3   = w>>24;
  ub3  = b3;
  h1   = ((ap_uint<16>)ub3<<8) | (ap_uint<16>)ub2;
  uh1  = h1;
  ...
```

The loaded word is used to build the requested data according to the size (byte, half-word, or word) and to the address least significant bits (a01 for a byte access, a1 for a half word access).

For LB and LH, the built data is sign extended to the size of the destination register (i.e. left padding the word data with copies of the loaded value sign).

For LBU and LHU, the loaded value is 0-extended (i.e. left padding the word data with zeros).

Listing 6.11 The mem_load function: build the loaded value

```
label
  ...
  switch(a01){
    case 0b00: b = b0; break;
    case 0b01: b = b1; break;
    case 0b10: b = b2; break;
    case 0b11: b = b3; break;
  }
  ub  = b;
  ib  = (int)b;
  iub = (unsigned int)ub;
  h   = (a1)?h1:h0;
  uh  = h;
  ih  = (int)h;
  iuh = (unsigned int)uh;
  switch(msize){
    case LB:
      result = ib;  break;
    case LH:
      result = ih;  break;
    case LW:
      result = w;   break;
    case 3:
      result = 0;   break;
    case LBU:
      result = iub; break;
    case LHU:
      result = iuh; break;
    case 6:
    case 7:
      result = 0;   break;
  }
  return result;
}
```

6.4.5 The Write_reg Function

The write_reg function (see Listing 6.12; defined in the execute.cpp file) writes the result back into the destination register, except if the instruction is a STORE, a BRANCH, or if the destination is register zero.

Listing 6.12 The write_reg function

```
static void write_reg(
  int                  *reg_file,
  decoded_instruction_t d_i,
  int                   result){
  if (d_i.rd != 0    &&
     !d_i.is_branch &&
     !d_i.is_store)
    reg_file[d_i.rd] = result;
}
```

6.5 Simulating the Rv32i_npp_ip With the Testbench

Experimentation

To simulate the rv32i_npp_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with rv32i_npp_ip.

You can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

The main function in the testbench_rv32i_npp_ip.cpp file prints the content of the data memory at the end of the run (see Listing 6.13). To avoid a long dump, only the non null words are printed.

Listing 6.13 The testbench_rv32i_npp_ip.cpp file

```
#include <stdio.h>
#include "rv32i_npp_ip.h"
int          data_ram[DATA_RAM_SIZE];
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_mem_0_text.hex"
};
int main(){
  unsigned int nbi;
  int          w;
  rv32i_npp_ip(0, code_ram, data_ram, &nbi);
  printf("%d fetched and decoded instructions\n", nbi);
  printf("data memory dump (non null words)\n");
  for (int i=0; i<DATA_RAM_SIZE; i++){
    w = data_ram[i];
    if (w != 0)
      printf("m[%5x] = %16d (%8x)\n", 4*i, w,
             (unsigned int)w);
  }
  return 0;
}
```

The code_ram array initialization includes the RISC-V code to be run. This code is obtained from the .text section of the ELF file.

Two new test programs have been added: test_load_store.s (see Listing 6.14) to test the various size of loads and stores defined in the RV32I ISA and test_mem.s (see Listing 6.18) which sets an array of the 10 first natural numbers and sums them. Their hex translations are available in the rv32i_npp_ip folder.

Listing 6.14 The test_load_store.s file

```
main:
       li      t0,1      /*t0=1*/
       li      t1,2      /*t1=2*/
       li      t2,-3     /*t2=-3*/
       li      t3,-4     /*t3=-4*/
       li      a0,0      /*a0=0*/
       sw      t0,0(a0)  /*t[a0]=t0        (word access)*/
       addi    a0,a0,4   /*a0+=4*/
       sh      t1,0(a0)  /*t[a0]=t1   (half word access)*/
       sh      t0,2(a0)  /*t[a0+2]=t0 (half word access)*/
```

```
        addi    a0,a0,4    /*a0+=4*/
        sb      t3,0(a0)   /*t[a0  ]=t3      (byte access)*/
        sb      t2,1(a0)   /*t[a0+1]=t2      (byte access)*/
        sb      t1,2(a0)   /*t[a0+2]=t1      (byte access)*/
        sb      t0,3(a0)   /*t[a0+3]=t0      (byte access)*/
        lb      a1,0(a0)   /*a1=t[a0  ]      (byte access)*/
        lb      a2,1(a0)   /*a2=t[a0+1]      (byte access)*/
        lb      a3,2(a0)   /*a3=t[a0+2]      (byte access)*/
        lb      a4,3(a0)   /*a4=t[a0+3]      (byte access)*/
        lbu     a5,0(a0)   /*a5=t[a0]   (unsigned byte access)*/
        lbu     a6,1(a0)   /*a6=t[a0+1] (unsigned byte access)*/
        lbu     a7,2(a0)   /*a7=t[a0+2] (unsigned byte access)*/
        addi    a0,a0,-4   /*a0-=4*/
        lh      s0,2(a0)   /*s0=t[a0+2] (half word access)*/
        lh      s1,0(a0)   /*s1=t[a0]   (half word access)*/
        lhu     s2,4(a0)   /*s2=t[a0+4] (unsigned h.w. access)*/
        lhu     s3,6(a0)   /*s3=t[a0+6] (unsigned h.w. access)*/
        addi    a0,a0,-4   /*a0-=4*/
        lw      s4,8(a0)   /*s4=t[a0+8]      (word access)*/
        ret
```

The store instructions forming the first part of the run print what is shown in Listing 6.15.

Listing 6.15 The test_load_store.s print: store instructions

```
label
0000: 00100293      li t0, 1
     t0  =                1 (       1)
0004: 00200313      li t1, 2
     t1  =                2 (       2)
0008: ffd00393      li t2, -3
     t2  =               -3 (fffffffd)
0012: ffc00e13      li t3, -4
     t3  =               -4 (fffffffc)
0016: 00000513      li a0, 0
     a0  =                0 (       0)
0020: 00552023      sw t0, 0(a0)
     m[       0] =                1 (       1)
0024: 00450513      addi a0, a0, 4
     a0  =                4 (       4)
0028: 00651023      sh t1, 0(a0)
     m[       4] =                2 (       2)
0032: 00551123      sh t0, 2(a0)
     m[       6] =                1 (       1)
0036: 00450513      addi a0, a0, 4
     a0  =                8 (       8)
0040: 01c50023      sb t3, 0(a0)
     m[       8] =               -4 (fffffffc)
0044: 007500a3      sb t2, 1(a0)
     m[       9] =               -3 (fffffffd)
0048: 00650123      sb t1, 2(a0)
     m[       a] =                2 (       2)
0052: 005501a3      sb t0, 3(a0)
     m[       b] =                1 (       1)
```

The load instructions forming the last part of the run print what is shown in Listing 6.16.

Listing 6.16 The test_load_store.s print: load instructions

```
0056: 00050583      lb a1, 0(a0)
     a1  =               -4 (fffffffc)    (m[       8])
0060: 00150603      lb a2, 1(a0)
     a2  =               -3 (fffffffd)    (m[       9])
0064: 00250683      lb a3, 2(a0)
     a3  =                2 (       2)    (m[       a])
0068: 00350703      lb a4, 3(a0)
     a4  =                1 (       1)    (m[       b])
0072: 00054783      lbu a5, 0(a0)
     a5  =              252 (      fc)    (m[       8])
0076: 00154803      lbu a6, 1(a0)
     a6  =              253 (      fd)    (m[       9])
0080: 00254883      lbu a7, 2(a0)
     a7  =                2 (       2)    (m[       a])
0084: ffc50513      addi a0, a0, -4
     a0  =                4 (       4)
0088: 00251403      lh s0, 2(a0)
     s0  =                1 (       1)    (m[       6])
0092: 00051483      lh s1, 0(a0)
     s1  =                2 (       2)    (m[       4])
0096: 00455903      lhu s2, 4(a0)
     s2  =            65020 (    fdfc)    (m[       8])
0100: 00655983      lhu s3, 6(a0)
     s3  =              258 (     102)    (m[       a])
0104: ffc50513      addi a0, a0, -4
     a0  =                0 (       0)
0108: 00852a03      lw s4, 8(a0)
     s4  =         16973308 ( 102fdfc)    (m[       8])
0112: 00008067      ret
     pc  =                0 (       0)
```

After the run, the processor prints the register file and the testbench program prints the number of instructions run and the non null memory words, as shown in Listing 6.17.

Listing 6.17 The test_load_store.s print: register file and memory dump

```
label
ra  =                0 (       0)
...
tp  =                0 (       0)
t0  =                1 (       1)
t1  =                2 (       2)
t2  =               -3 (fffffffd)
s0  =                1 (       1)
s1  =                2 (       2)
a0  =                0 (       0)
a1  =               -4 (fffffffc)
a2  =               -3 (fffffffd)
a3  =                2 (       2)
a4  =                1 (       1)
a5  =              252 (      fc)
a6  =              253 (      fd)
a7  =                2 (       2)
s2  =            65020 (    fdfc)
s3  =              258 (     102)
s4  =         16973308 ( 102fdfc)
s5  =                0 (       0)
...
s11 =                0 (       0)
t3  =               -4 (fffffffc)
```

```
t4   =                0 (        0)
...
29 fetched and decoded instructions
data memory dump (non null words)
m[     0] =                1 (        1)
m[     4] =            65538 (    10002)
m[     8] =         16973308 (  102fdfc)
```

The second test program is test_mem.s, shown in Listing 6.18 (which computes the sum of the 10 elements of an array initialized with the 10 first natural numbers; in the comments, I give the C equivalent semantic of each RISC-V assembly instruction).

Listing 6.18 The test_mem.s file

label

```
       .globl  main
main:
       li      a0,0         /*a0=0*/
       li      a1,0         /*a1=0*/
       li      a2,0         /*a2=0*/
       addi    a3,a2,40     /*a3=40*/
.L1:
       addi    a1,a1,1      /*a1++*/
       sw      a1,0(a2)     /*t[a2]=a1*/
       addi    a2,a2,4      /*a2+=4*/
       bne     a2,a3,.L1    /*if (a2!=a3) goto .L1*/
       li      a1,0         /*a1=0*/
       li      a2,0         /*a2=0*/
.L2:
       lw      a4,0(a2)     /*a4=t[a2]*/
       add     a0,a0,a4     /*a0+=a4*/
       addi    a2,a2,4      /*a2+=4*/
       bne     a2,a3,.L2    /*if (a2!=a3) goto .L2*/
       sw      a0,4(a2)     /*t[a2+4]=a0*/
       ret
```

The first part of the run of the test_mem.s file prints what is shown in Listing 6.19.

Listing 6.19 The test_mem.s print: first iteration of the write array loop

label

```
0000: 00000513      li a0, 0
     a0   =                0 (        0)
0004: 00000593      li a1, 0
     a1   =                0 (        0)
0008: 00000613      li a2, 0
     a2   =                0 (        0)
0012: 02860693      addi a3, a2, 40
     a3   =               40 (       28)
0016: 00158593      addi a1, a1, 1
     a1   =                1 (        1)
0020: 00b62023      sw a1, 0(a2)
     m[       0] =                1 (        1)
0024: 00460613      addi a2, a2, 4
     a2   =                4 (        4)
0028: fed61ae3      bne a2, a3, 16
     pc   =               16 (       10)
0016: 00158593      addi a1, a1, 1
     a1   =                2 (        2)
0020: 00b62023      sw a1, 0(a2)
```

```
     m[       4] =                 2 (       2)
...
```

After the array has been initialized, it is read to accumulate its values in the a0 register. The second part of the run prints what is shown in Listing 6.20.

Listing 6.20 The test_mem.s print: out of the write array loop and in the read array loop

```
...
0024: 00460613      addi a2, a2, 4
     a2   =                40 (      28)
0028: fed61ae3      bne a2, a3, 16
     pc   =                32 (      20)
0032: 00000593      li a1, 0
     a1   =                 0 (       0)
0036: 00000613      li a2, 0
     a2   =                 0 (       0)
0040: 00062703      lw a4, 0(a2)
     a4   =                 1 (       1)    (m[       0])
0044: 00e50533      add a0, a0, a4
     a0   =                 1 (       1)
0048: 00460613      addi a2, a2, 4
     a2   =                 4 (       4)
0052: fed61ae3      bne a2, a3, 40
     pc   =                40 (      28)
0040: 00062703      lw a4, 0(a2)
     a4   =                 2 (       2)    (m[       4])
...
```

When all the elements of the array have been added, the final sum is saved to memory. The third part of the run prints what is shown in Listing 6.21.

Listing 6.21 The test_mem.s print: out of the read array loop and store the result

label

```
...
0044: 00e50533      add a0, a0, a4
     a0   =                55 (      37)
0048: 00460613      addi a2, a2, 4
     a2   =                40 (      28)
0052: fed61ae3      bne a2, a3, 40
     pc   =                56 (      38)
0056: 00a62223      sw a0, 4(a2)
     m[      2c] =                55 (      37)
0060: 00008067      ret
     pc   =                 0 (       0)
```

The processor prints the final state of its register file, with the sum in a0 as shown in Listing 6.22.

Listing 6.22 The test_mem.s print: the register file

label

```
...
a0  =               55 (      37)
a1  =                0 (       0)
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
```

The testbench program prints the number of executed instructions and the non null memory words, i.e. the array and the sum, as shown in Listing 6.23.

Listing 6.23 The test_mem.s print: the memory

```
label
88 fetched and decoded instructions
data memory dump (non null words)
m[    0] =                  1 (         1)
m[    4] =                  2 (         2)
m[    8] =                  3 (         3)
m[    c] =                  4 (         4)
m[   10] =                  5 (         5)
m[   14] =                  6 (         6)
m[   18] =                  7 (         7)
m[   1c] =                  8 (         8)
m[   20] =                  9 (         9)
m[   24] =                 10 (         a)
m[   2c] =                 55 (        37)
```

6.6 Synthesis of the Rv32i_npp_ip

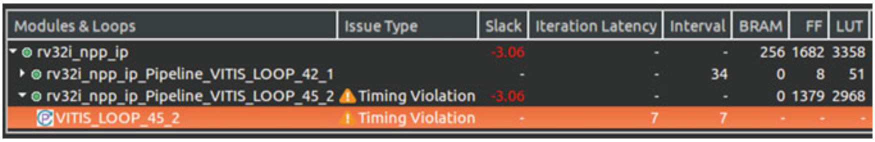
The synthesis report shows that the IP cycle corresponds to seven FPGA cycles (see Fig. 6.4).

The Schedule Viewer confirms the seven cycles (see Fig. 6.5).

6.7 The Z1_rv32i_npp_ip Vivado Project

The block design in the Vivado project is shown in Fig. 6.6.

The Vivado implementation for the Pynq-Z1 board uses 4091 LUTs, 7.69% of the available LUTs on the FPGA (see Fig. 6.7).



Modules & Loops	Issue Type	Slack	Iteration Latency	Interval	BRAM	FF	LUT
▾ ● rv32i_npp_ip		-3.06	-	-	256	1682	3358
▸ ● rv32i_npp_ip_Pipeline_VITIS_LOOP_42_1		-	-	34	0	8	51
▾ ● rv32i_npp_ip_Pipeline_VITIS_LOOP_45_2	⚠ Timing Violation	-3.06	-	-	0	1379	2968
VITIS_LOOP_45_2	! Timing Violation	-	7	7	-	-	-

Fig. 6.4 Synthesis report for the rv32i_npp_ip IP

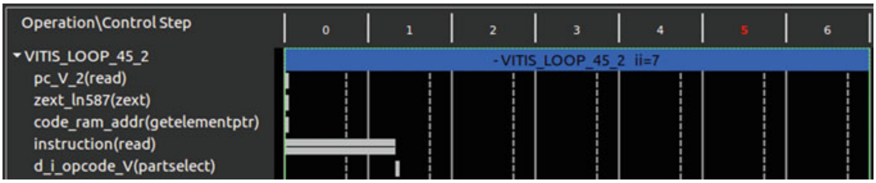


Fig. 6.5 The rv32i_npp_ip schedule

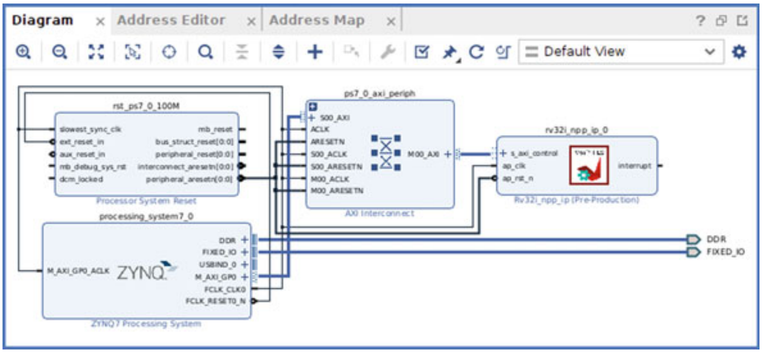
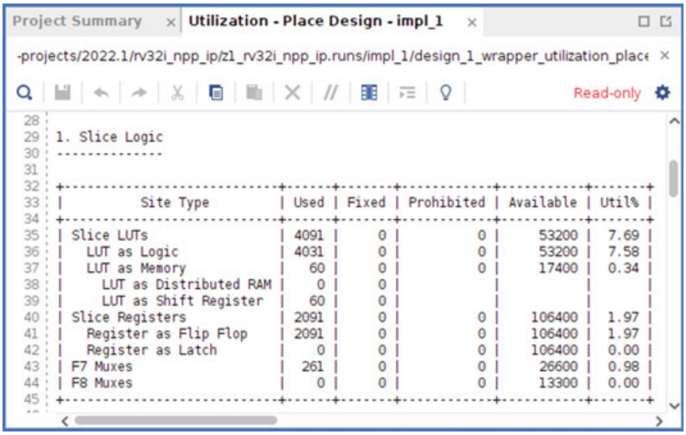


Fig. 6.6 The Vivado z1_rv32i_npp_ip block design


Project Summary
Utilization - Place Design - impl_1
-projects/2022.1/rv32i_npp_ip/z1_rv32i_npp_ip.runs/impl_1/design_1_wrapper_utilization_place
Read-only

```

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 4091 |     0 |          0 |     53200 |  7.69 |
|   LUT as Logic             | 4031 |     0 |          0 |     53200 |  7.58 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 2091 |     0 |          0 |    106400 |  1.97 |
|   Register as Flip Flop    | 2091 |     0 |          0 |    106400 |  1.97 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  261 |     0 |          0 |     26600 |  0.98 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 6.7 Vivado utilization report for the rv32i_npp_ip

6.8 The Helloworld.c Program to Drive the Rv32i_npp_ip on the FPGA

Experimentation

To run the rv32i_npp_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with rv32i_npp_ip.
You can play with your IP, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

The code in the helloworld.c file is shown in Listing 6.24 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script; to run another test program, update the #include line in the initialization of the code_ram array).

Listing 6.24 The helloworld.c program to run the test_mem.s code on the Pynq-Z1 board

```
#include <stdio.h>
#include "xrv32i_npp_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE 16
//size in words
#define DATA_RAM_SIZE     (1<<LOG_DATA_RAM_SIZE)
XRv32i_npp_ip_Config *cfg_ptr;
XRv32i_npp_ip         ip;
word_type code_ram[CODE_RAM_SIZE]={
#include "test_mem_0_text.hex"
};
int main(){
  word_type w;
  cfg_ptr = XRv32i_npp_ip_LookupConfig(
      XPAR_XRV32I_NPP_IP_0_DEVICE_ID);
  XRv32i_npp_ip_CfgInitialize(&ip, cfg_ptr);
  XRv32i_npp_ip_Set_start_pc(&ip, 0);
  XRv32i_npp_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XRv32i_npp_ip_Start(&ip);
  while (!XRv32i_npp_ip_IsDone(&ip));
  printf("%d fetched and decoded instructions\n",
    (int)XRv32i_npp_ip_Get_nb_instruction(&ip));
  printf("data memory dump (non null words)\n\r");
  for (int i=0; i<DATA_RAM_SIZE; i++){
    XRv32i_npp_ip_Read_data_ram_Words(&ip, i, &w, 1);
    if (w != 0)
      printf("m[%5x] = %16d (%8x)\n", 4*i, (int)w,
             (unsigned int)w);
  }
}
```

The run of the RISC-V code in the test_mem.s file prints the output shown in Listing 6.25 on the putty terminal.

Listing 6.25 The helloworld.c program output on the putty terminal

```
88 fetched and decoded instructions
data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
```

Testing Your RISC-V Processor

7

Abstract

This chapter lets you test your first RISC-V processor in three steps: test all the instructions in their most frequent usage (my six test programs), pass the official riscv-tests and test benchmark programs from the mibench suite and from the official riscv-tests.

7.1 Testing the Rv32i_npp_ip Processor with my Test Programs

I have already presented six RISC-V test programs: test_branch.s to test BRANCH instructions, test_jal_jalr.s to test JAL and JALR instructions, test_lui_auipc.s to test LUI and AUIPC instructions, test_load_store.s to test LOAD and STORE instructions, test_op.s to test OP instructions and test_op_imm.s to test OP_IMM instructions.

They are enough to make sure that the decoder recognizes all the instructions in the RV32I ISA and that the execution unit can run them.

However, there are many special situations which are not checked by these six programs. For example, is register zero preserved from any writeback? As a source, does it provide the value 0? Are the constants in every format properly decoded (there are a lot of cases to test because the decoded immediate value is composed of many fields from different bits in the instruction word, assembled in different ways according to the instruction format)?

The RISC-V organization provides a set of programs to test all the instructions more exhaustively than what I do in my own codes. However, you should first run my codes before running the official riscv-tests programs, for two reasons.

First, the riscv-tests codes are embedded and the embedding code is itself made of RISC-V instructions. So, if your processor is buggy, you might not be able to run the part of the code to launch the tests.

Second, because you will learn that debugging hardware is not as simple as debugging software. On the FPGA, you do not have a debugger at hand (you do, but

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_7

just for the helloworld driver code run on the ARM processor within the Zynq SoC, not for the RISC-V code run on the IP implemented on the FPGA). When your FPGA does not send anything to the putty terminal, the only way to debug your IP is to think about your code and evaluate every instruction, step by step, again and again. My simple RISC-V programs are less likely to bug because of the processor implementation than the complex riscv-tests ones.

Once your processor has successfully run my six test codes (not only on the simulator but also on the FPGA), you can try to pass the riscv-tests.

7.2 More Testing with the Official Riscv-Tests

All the source files related to the riscv-tests can be found in the riscv-tests folder.

7.2.1 Running the Riscv-Tests With Spike

To ensure that the processor implementation is respecting the RISC-V specification, you must pass the riscv-tests provided by the RISC-V organization.

I have adapted the riscv-tests to the Vitis_HLS environment. My version of riscv-tests is provided in the riscv-tests folder.

In this folder, the env and isa subfolders are original versions taken from https://github.com/riscv-software-src/riscv-tests.

The benchmarks, my_env, and my_isa subfolders are modified versions adapted to the Vitis_HLS environment.

To build the tests and run them with spike, apply make run in the riscv-tests/isa folder (see Listing 7.1). Each test run appends an error message to the standard error stream, redirected to a file. For the addi instruction test, the file is named rv32ui-p-addi.out32. If the test is passed, the file is empty.

Listing 7.1 Running the riscv-tests with spike

```
$ cd $HOME/goossens-book-ip-projects/2022.1/riscv-tests/isa
$ make run XLEN=32
spike --isa=rv32gc rv32ui-p-simple 2> rv32ui-p-simple.out32
spike --isa=rv32gc rv32ui-p-add 2> rv32ui-p-add.out32
spike --isa=rv32gc rv32ui-p-addi 2> rv32ui-p-addi.out32
...
$
```

The pk interpreter is not used. The simulated code (e.g. rv32ui-p-add) contains its own OS proxy and it is loaded by spike itself.

As the spike simulator has no known bug, all the files are empty after the run.

7.2.2 The Riscv-Tests Structure

Before I describe the port of the riscv-tests to the Vitis_HLS environment, I must present their structure in the original code.

7.2.2.1 The Test Files

The tests are the .S files in the riscv-tests/isa/rv64ui folder (from add.S to xori.S, each *x.S* file contains a piece of assembly code to test various situations involving the *x* instruction).

For example, the add.S file is partly shown in Listing 7.2.

Listing 7.2 The add.S file

```
...
#include "riscv_test.h"
#include "test_macros.h"

RVTEST_RV64U
RVTEST_CODE_BEGIN

  #---------------------------------------------------------------
  # Arithmetic tests
  #---------------------------------------------------------------

  TEST_RR_OP( 2,  add, 0x00000000, 0x00000000, 0x00000000 );
  TEST_RR_OP( 3,  add, 0x00000002, 0x00000001, 0x00000001 );
  TEST_RR_OP( 4,  add, 0x0000000a, 0x00000003, 0x00000007 );
...
```

7.2.2.2 The Macros

Each test file is composed of *macros* defined in the riscv-tests/isa/macros/scalar/test_macros.h file.

The RVTEST_RV64U and RVTEST_CODE_BEGIN macros have no arguments. The TEST_RR_OP macro has five arguments.

For example, the TEST_RR_OP macro is defined as shown in Listing 7.3.

Listing 7.3 The TEST_RR_OP macro definition in the test_macros.h file

```
#define TEST_RR_OP( testnum, inst, result, val1, val2 ) \
    TEST_CASE( testnum, x14, result, \
      li  x1, MASK_XLEN(val1); \
      li  x2, MASK_XLEN(val2); \
      inst x14, x1, x2; \
    )
```

A macro is defined in three parts: the #define keyword, the name of the macro with its possible arguments (i.e. TEST_RR_OP(testnum, inst, result, val1, val2)) and its definition (the remaining of the line, possibly extended with \ characters).

7.2.2.3 The Preprocessor Processing of the Macros

Macros are manipulated by the *preprocessor* of the compiler. The preprocessor replaces every instance of a macro by its definition, adapting the arguments with the actual values. For example, the two first occurrences of TEST_RR_OP in the add.S file are replaced as shown in Listing 7.4.

Listing 7.4 The TEST_RR_OP macro substitution in the add.S file

```
#include "riscv_test.h"
#include "test_macros.h"

RVTEST_RV64U
RVTEST_CODE_BEGIN

  #-------------------------------------------------
  # Arithmetic tests
  #-------------------------------------------------

  TEST_CASE( 2, x14, 0x00000000, \
    li  x1, MASK_XLEN(0x00000000); \
    li  x2, MASK_XLEN(0x00000000); \
    add x14, x1, x2; \
  );
  TEST_CASE( 3, x14, 0x00000002, \
    li  x1, MASK_XLEN(0x00000001); \
    li  x2, MASK_XLEN(0x00000001); \
    add x14, x1, x2; \
  )
```

TEST_CASE and MASK_XLEN are also macros (see Listing 7.5).

Listing 7.5 The MASK_XLEN and TEST_CASE macro definitions in the test_macros.h file

```
#define MASK_XLEN(x) ((x) \& ((1 << (__riscv_xlen - 1) << 1) - 1))
#define TEST_CASE( testnum, testreg, correctval, code... ) \
test_ ## testnum: \
    code; \
    li  x7, MASK_XLEN(correctval); \
    li  TESTNUM, testnum; \
    bne testreg, x7, fail;
```

The substitution process continues until all the macros have been replaced. You can check these substitutions by running the compiler only for the preprocessor job with the -E option (see Listing 7.6).

Listing 7.6 The preprocessor substitution job on the add.S file

```
$ cd $HOME/goossens-book-ip-projects/2022.1/riscv-tests/isa/rv64ui
$ riscv32-unknown-elf-gcc -I../../env/p -I../macros/scalar -E add.S
      > add_preprocessor.S
$ cat add_preprocessor.S
...
  #-------------------------------------------------
  # Arithmetic tests
  #-------------------------------------------------

  test_2: li x1, ((0x00000000) & ((1 << (32 - 1) << 1) - 1)); li
      x2, ((0x00000000) & ((1 << (32 - 1) << 1) - 1)); add x14, x1,
       x2;; li x7, ((0x00000000) & ((1 << (32 - 1) << 1) - 1)); li
      gp, 2; bne x14, x7, fail;;
  test_3: li x1, ((0x00000001) & ((1 << (32 - 1) << 1) - 1)); li
      x2, ((0x00000001) & ((1 << (32 - 1) << 1) - 1)); add x14, x1,
       x2;; li x7, ((0x00000002) & ((1 << (32 - 1) << 1) - 1)); li
      gp, 3; bne x14, x7, fail;;
...
```

7.2.2.4 The Test Files After the Preprocessing

It is not necessary to understand the full substitution process though. What is necessary is to understand what the test files are made of.

When the compiler starts compiling, the add.S file has been translated into a text containing RISC-V instructions, among which the add instruction appears in some tested situations (this is why it is essential to be sure that your processor is at least able to run the instructions surrounding the tested one).

For example, in the test_2 code block, registers x1 and x2 are cleared, then added into x14. Register x7 is initialized with the expected result (i.e. 0+0 should be 0). Register gp receives the test number (i.e. 2). Register x14 and x7 are compared. If they do not match, it means that the test has failed and the run branches to the fail label defined somewhere else (through another macro to be shown in 7.2.3).

If the test does not fail, the run continues to the next test (i.e. test_3).

In test_3, the code is identical except for the register initializations: x1 and x2 are set to 1, x7 is set to 2 and gp is set to 3.

There are 38 tests for the add instruction as the end of the add_preprocessor.S file shows in Listing 7.7.

Listing 7.7 The end of the add_preprocessor.S file

```
$ cat add_preprocessor.S
...
  test_38: li x1, ((16) & ((1 << (32 - 1) << 1) - 1)); li x2, ((30)
       & ((1 << (32 - 1) << 1) - 1)); add x0, x1, x2;; li x7, ((0)
      & ((1 << (32 - 1) << 1) - 1)); li gp, 38; bne x0, x7, fail;;

  bne x0, gp, pass; fail: fence; 1: beqz gp, 1b; sll gp, gp, 1; or
      gp, gp, 1; li a7, 93; addi a0, gp, 0; ecall; pass: fence; li
      gp, 1; li a7, 93; li a0, 0; ecall
...
```

A more readable version is shown in Listing 7.8.

Listing 7.8 A readable version of the end of the add_preprocessor.S file

```
test_38: li   x1, ((16) & ((1 << (32 - 1) << 1) - 1))
         li   x2, ((30) & ((1 << (32 - 1) << 1) - 1))
         add  x0, x1, x2
         li   x7, ((0) & ((1 << (32 - 1) << 1) - 1))
         li   gp, 38
         bne  x0, x7, fail
         bne  x0, gp, pass
fail:    fence
1:       beqz gp, 1b
         sll  gp, gp, 1
         or   gp, gp, 1
         li   a7, 93
         addi a0, gp, 0
```

```
          ecall
pass:     fence
          li    gp, 1
          li    a7, 93
          li    a0, 0
          ecall
```

The last test adds 16 to 30 into register x0. Register x0, alias zero, should never be written. Hence, register x0 is compared to a cleared x7. If they do not match, the run branches to fail. Otherwise, it branches to pass.

Whatever the result of the equality comparison between x0 and x7, the run calls the 93 system call (in a spike simulated RISC-V machine, the ecall instruction calls the system call indicated by the x7 register, i.e. 93).

The called system call belongs to spike. It receives arguments through the a0 to a5 registers (a0 is cleared if pass and set to 2*test number + 1 if fail; gp is set to 1 if pass and to the test number if fail; this is probably why the first test number is 2 instead of 1).

It is not very clear (because undocumented) how this system call operates. It certainly writes a fail message to the standard error stream. The spike command launched by make run redirects this standard error stream to an error file.

7.2.3 Adapting the Riscv-Tests Structure to the Vitis_HLS Environment

The test programs run on spike must be adapted to be run on the rv32i_npp_ip simulator and on the FPGA. For example, the ebreak instruction and the system calls included in spike are not present in the rv32i_npp_ip implementation.

I have adapted the riscv_test.h file included by each x.S test program to the Vitis_HLS environment (I mostly adapted the changes for Vitis_HLS from a proposition I found on Peter Gu's blog, thanks to him (https://www.ustcpetergu.com/MyBlog/experience/2021/07/09/about-riscv-testing.html), himself being inspired by the PicoRV32 implementation at https://github.com/YosysHQ/picorv32).

The new version is my_riscv_test.h shown in Listing 7.9. It is located in the riscv-tests/my_env/p folder.

For each tested instruction, the result of the test is available in the a0 register (null if passed) and saved in the result_zone memory (starting at byte address 0x2000 or word address 0x800).

The RVTEST_CODE_BEGIN macro only contains the TEST_FUNC_NAME label definition (see the "#define RVTEST_CODE_BEGIN" line in Listing 7.9).

TEST_FUNC_NAME is a macro to be defined as the name of the tested instruction (e.g. addi).

This macro is not defined in the code. It is to be defined directly in the compilation command through the -D option (the shell script including the compilation command and the -D option is presented in 7.2.6).

The RVTEST_FAIL and RVTEST_PASS macros are defined to save their a0 result into the result zone (see the "#define RVTEST_FAIL" and "#define RVTEST_PASS" lines in Listing 7.9).

The RVTEST_FAIL and RVTEST_PASS macros jump to TEST_FUNC_RET.

The TEST_FUNC_RET macro is like the TEST_FUNC_NAME one. It should be defined in the compilation command with the -D option.

The TEST_FUNC_RET macro is to be defined as the name of the tested instruction followed by the "_ret" suffix.

For example for the addi instruction the TEST_FUNC_RET macro is defined as "addi_ret".

Listing 7.9 The my_riscv_test.h file

```
// See LICENSE for license details.
#ifndef _ENV_PHYSICAL_SINGLE_CORE_H
#define _ENV_PHYSICAL_SINGLE_CORE_H
#define RVTEST_RV32U
#define RVTEST_RV64U
#define TESTNUM gp
#define INIT_XREG                                    \
        .text;                                       \
        li x1, 0;                                    \
...
        li x31, 0;
#define RVTEST_CODE_BEGIN                            \
        .text;                                       \
        .global  TEST_FUNC_NAME;                     \
        .global  TEST_FUNC_RET;                      \
TEST_FUNC_NAME:
#define RVTEST_CODE_END
#define RVTEST_PASS                                  \
        .equ     result_zone,0x2000;                 \
        .text;                                       \
        li       a0,0;                               \
        lui      t0,%hi(result_zone);                \
        addi     t0,t0,%lo(result_zone);             \
        lw       t1,0(t0);                           \
        sw       a0,0(t1);                           \
        addi     t1,t1,4;                            \
        sw       t1,0(t0);                           \
        jal      zero,TEST_FUNC_RET;
#define RVTEST_FAIL                                  \
        .text;                                       \
        mv       a0,TESTNUM;                         \
        lui      t0,%hi(result_zone);                \
        addi     t0,t0,%lo(result_zone);             \
        lw       t1,0(t0);                           \
        sw       a0,0(t1);                           \
        addi     t1,t1,4;                            \
        sw       t1,0(t0);                           \
        jal      zero,TEST_FUNC_RET;
#define RVTEST_DATA_BEGIN                            \
        .data;                                       \
        .align   4;
#define RVTEST_DATA_END                              \
        .data;                                       \
        .align   4;
#endif
```

Similarly, I have adapted the test_macros.h file. The my_test_macros.h file in riscv-tests/my_isa/macros/scalar is my adaptation.

7.2.4 Adding a _start.S Program to Glue All the Tests Together

The tests are all glued together in a single program named _start.S, shown in Listing 7.10. The _start.S file is in the riscv-tests/my_isa/my_rv32ui folder.

The _start.S file runs the tests of the 37 instructions consecutively (plus the simple.S test) and saves their result in the result_zone array.

The _start.S file defines the TEST(n) macro.

The TEST(n) macro jumps to label *n* ("jal zero, *n*"; e.g. TEST(addi) jumps to label addi defined in RVTEST_CODE_BEGIN, which starts the run of addi.S).

The macro also defines label *n*_ret (e.g. TEST(addi) defines label addi_ret which is the jump address after RVTEST_PASS or RVTEST_FAIL).

To summarize the construction let me detail the addi instruction test example.

TEST(addi) in _start.S jumps to label addi ("jal zero, addi" after the TEST macro expansion). The addi label is defined in the addi.S file by the expansion of the RVTEST_CODE_BEGIN macro.

The addi.S tests are run (the succession of macros in addi.S, starting with TEST_IMM_OP). The addi.S code ends with the TEST_PASSFAIL macro which is defined in the my_test_macros.h file and expanded as a branch to the fail or pass labels.

At the pass label, the RVTEST_PASS macro expands to the code given in the my_riscv_test.h file, which clears the current result_zone word (meaning the test is passed) and jumps to addi_ret.

The addi_ret label is at the end of the TEST(addi) macro expansion. It is followed by the TEST(add) macro which starts the add instruction test.

Listing 7.10 The _start.S file

```
#include "../../my_env/p/my_riscv_test.h"
        .text
        .globl   _start
        .equ     result_zone,0x2000
_start:
        lui      a0,%hi(result_zone)
        addi     a0,a0,%lo(result_zone)
        addi     a1,a0,4
        sw       a1,0(a0)
        INIT_XREG
#define TEST(n)                           \
        .global  n;                       \
        jal      zero,n;                  \
        .global  n ## _ret;               \
n ## _ret:
        TEST(addi  )
        TEST(add   )
...
        TEST(xori  )
        TEST(xor   )
        li       ra,0
        ret
```

7.2.5 The Testbench to Simulate the Tests in Vitis_HLS

A new testbench code is written (see Listing 7.11) to read the result_zone and print the result of each test (testbench_riscv_tests_rv32i_npp_ip.cpp file in the riscv-tests/my_isa/my_rv32ui folder).

The code run by the rv32i_npp_ip processor is defined as the test_0_text.hex file (to be built with a shell script presented in 7.2.6). This file contains the RISC-V instruction codes provided by the compiler from the compilation of the _start.S file.

Concerning the test_0_data.hex file, the LOAD instructions in the tests read from memory words which must be provided by the data hex file.

The data these LOAD instructions access are defined in the .S files (lw.S, lh.S, lhu.S, lb.S, and lbu.S).

They are incorporated to the test_0_data.hex file which serves to initialize the data_ram array (this file is built by the same shell script) (remember that the code and data memories are separate, hence two initialization files: test_0_data.hex to initialize the data RAM and test_0_code.hex to initialize the code RAM).

When the rv32i_npp_ip function is called, the code in the code_ram array is run, i.e. the _start.S file. The results of the tests are saved in the result_zone in the data_ram array (at word address 0x801 is the result of the first instruction tested, i.e. addi). When the memory word is null, the test is passed. Otherwise, the value is the identification number of the first failing test.

Listing 7.11 The testbench_riscv_tests_rv32i_npp_ip.cpp file

```
#include <stdio.h>
#include "../../../rv32i_npp_ip/rv32i_npp_ip.h"
unsigned int data_ram[DATA_RAM_SIZE]={
#include "test_0_data.hex"
};
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_0_text.hex"
};
char        *name[38] = {
  "addi  ", "add   ", "andi  ", "and   ", "auipc ",
  "beq   ", "bge   ", "bgeu  ", "blt   ", "bltu  ", "bne   ",
  "jalr  ", "jal   ",
  "lb    ", "lbu   ", "lh    ", "lhu   ", "lui   ", "lw    ",
  "ori   ", "or    ",
  "sb    ", "sh    ", "simple",
  "slli  ", "sll   ", "slti  ", "sltiu ", "slt   ", "sltu  ",
  "srai  ", "sra   ", "srli  ", "srl   ", "sub   ", "sw    ",
  "xori  ", "xor   "
};
int main(){
  unsigned int nbi;
  int          w;
  rv32i_npp_ip(0, (instruction_t*)code_ram, (int*)data_ram, &nbi);
  for (int i=0; i<38; i++){
    printf("%s:",name[i]);
    if (data_ram[0x801+i]==0)
      printf(" all tests passed\n");
    else
      printf(" test %d failed\n",data_ram[0x801+i]);
  }
  return 0;
}
```

7.2.6 Running the Riscv-Tests in the Vitis_HLS Environment

Experimentation

To simulate the riscv-tests on the rv32i_npp_ip, in a terminal with riscv-tests/my_isa/my_rv32ui as the current directory, run ./my_build_all.sh.

Proceed as explained in 5.3.6, replacing fetching_ip with rv32i_npp_ip and testbench_fetching_ip.cpp with riscv-tests/my_isa/my_rv32ui/ testbench_riscv_tests_rv32i_npp_ip.cpp.

The simulation result in the rv32i_npp_ip_csim.log tab should show that all the tests have been passed.

7.2.6.1 Building the Riscv-Tests Environment for Vitis_HLS

The riscv-tests environment for Vitis_HLS is built with the my_build_all.sh shell script in the riscv-tests/my_isa/my_rv32ui folder. Run the commands in Listing 7.12.

Listing 7.12 Build the test_0_text.hex and test_0_data.hex files

```
$ cd $HOME/goossens-book-ip-projects/2022.1/riscv-tests/my_isa/
    my_rv32ui
$ ./my_build_all.sh
$
```

The my_build_all.sh script (shown in Listing 7.13) compiles each .S file, defining the TEST_FUNC_NAME and TEST_FUNC_RET variables (for example "-DTEST_FUNC_NAME=addi" and "-DTEST_FUNC_RET=addi_ret" for the compilation of addi.S).

Then it compiles the _start.S file.

When all the source files have been compiled, the script links them to build the test.elf file. Usually, the linker builds a memory with the concatenation of the code and the data (the .text section followed by the .data section).

The rv32i_npp_ip processor has separate code and data memory banks. For this reason, the script orders the linker to base the code and the data at the same address 0 (-Ttext 0 and -Tdata 0). The linker complains when two sections are overlapping. To avoid this, the script uses the -no-check-sections option. This option is not for the gcc compiler but for the ld linker (-Wl,-no-check-sections).

Moreover, the _start label defined in _start.S should be the entry point of the run. The script uses the -nostartfiles to prevent the linker to add some OS related start code. Similarly, no library should be added (-nostdlib) to keep the code as short as possible in the code_ram array.

The test.elf file built by the linker is structured in sections. There may be many sections and among them, a .text section for the code, a .rodata section for read-only data, a .data section for the initialized global data and a .bss section for the uninitialized global data.

The ELF file is to be manipulated by a loader. The OS provides such a loader, which is used each time you run a program. The loader dispatches the sections in the processor memory, i.e. loads the code from the .text section and the data from the .data section.

As the rv32i_npp_ip processor has no OS, there is no loader. In this case, you must extract the .text and .data sections and place them in the code_ram and data_ram arrays.

From the test.elf file, the .text and the .data sections are extracted with the objcopy tool and saved in the test_0_text.bin and test_0_data.bin files.

Then, with the hexdump tool, the hexadecimal values are transformed in their ASCII representation (e.g. the one word hexadecimal value 0xdeadbeef is translated to the 10 characters '0', 'x', 'd', 'e', 'a', 'd', 'b', 'e', 'e', 'f'). This transformation builds a file which can be included in the initialization of the code_ram or data_ram array.

Listing 7.13 The my_build_all.sh script file

```
riscv32-unknown-elf-gcc -c -DTEST_FUNC_NAME=addi -DTEST_FUNC_RET=addi_ret addi.S -o addi.o
...
riscv32-unknown-elf-gcc -c -DTEST_FUNC_NAME=xor -DTEST_FUNC_RET=xor_ret xor.S -o xor.o
riscv32-unknown-elf-gcc -c _start.S -o _start.o
riscv32-unknown-elf-gcc -O3 -static -nostartfiles -nostdlib -o test.elf -Ttext 0 -Tdata 0 -Wl,-no-check-sections _start.o [a-z]*.o
riscv32-unknown-elf-objcopy -O binary --only-section=.text test.elf test_0_text.bin
riscv32-unknown-elf-objcopy -O binary --only-section=.data test.elf test_0_data.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' test_0_text.bin > test_0_text.hex
hexdump -v -e '"0x" /4 "%08x" ",\n"' test_0_data.bin > test_0_data.hex
```

7.2.6.2 The Riscv-Tests Prints

If the debugging constant definitions are uncommented in the debug_rv32i_npp_ip.h file, the run of the rv32i_npp_ip function called in the main function of the testbench_riscv_tests_rv32i_npp_ip.cpp file prints the disassembling and emulation of the RISC-V instructions run, as shown in Listings 7.14–7.20.

The INIT_XREG macro at the beginning of the _start.S code (refer back to Listings 7.9 and 7.10) clears all the registers. Its run produces the disassembling and emulation shown in Listing 7.14 (from code memory address 0016 to 0136).

Listing 7.14 The print of the rv32i_npp_ip function

```
0000: 00002537      lui a0, 8192
     a0   =             8192 (    2000)
0004: 00050513      addi a0, a0, 0
     a0   =             8192 (    2000)
0008: 00450593      addi a1, a0, 4
     a1   =             8196 (    2004)
0012: 00b52023      sw a1, 0(a0)
     m[    2000] =             8196 (    2004)
0016: 00000093      li ra, 0
     ra   =                0 (       0)
0020: 00000113      li sp, 0
     sp   =                0 (       0)
...
0132: 00000f13      li t5, 0
     t5   =                0 (       0)
0136: 00000f93      li t6, 0
     t6   =                0 (       0)
0140: 0a00006f      j 300
     pc   =              300 (     12c)
...
```

Then, the run prints the disassembling and emulation of the addi first test (from 0300 to 0316) as shown in Listing 7.15.

Listing 7.15 The print of the rv32i_npp_ip function

```
...
0300: 00000093      li ra, 0
     ra   =                0 (       0)
0304: 00008713      addi a4, ra, 0
     a4   =                0 (       0)
0308: 00000393      li t2, 0
     t2   =                0 (       0)
0312: 00200193      li gp, 2
     gp   =                2 (       2)
0316: 26771c63      bne a4, t2, 948
     pc   =              320 (     140)
...
```

Then (see Listing 7.16), it prints the disassembling and emulation of the other tests run for addi until test 25 (addi test number 25 from 0924 to 0940). If all the tests for addi have been successfull, the run continues at address 0980, i.e. RVTEST_PASS (0948 otherwise, i.e. RVTEST_FAIL). When the success has been reported in the result zone, the run continues at address 0144 (addi_ret).

Listing 7.16 The print of the rv32i_npp_ip function

```
...
0924: 02100093      li ra, 33
     ra   =               33 (      21)
0928: 03208013      addi zero, ra, 50

0932: 00000393      li t2, 0
     t2   =                0 (       0)
0936: 01900193      li gp, 25
     gp   =               25 (      19)
0940: 00701463      bne zero, t2, 948
     pc   =              944 (     3b0)
0944: 02301263      bne zero, gp, 980
     pc   =              980 (     3d4)
```

```
0980: 00000513      li a0, 0
     a0  =                0 (       0)
0984: 000022b7      lui t0, 8192
     t0  =             8192 (    2000)
0988: 00028293      addi t0, t0, 0
     t0  =             8192 (    2000)
0992: 0002a303      lw t1, 0(t0)
     t1  =             8196 (    2004)    (m[    2000])
0996: 00a32023      sw a0, 0(t1)
     m[    2004] =                0 (       0)
1000: 00430313      addi t1, t1, 4
     t1  =             8200 (    2008)
1004: 0062a023      sw t1, 0(t0)
     m[    2000] =             8200 (    2008)
1008: ca1ff06f      j 144
     pc  =              144 (      90)
...
```

All the instructions are successively tested until xor (test 2 from 30288 to 30320 in Listing 7.17).

Listing 7.17 The print of the rv32i_npp_ip function

```
...
30288: ff0100b7      lui ra, 65536
     ra  =        -16711680 (ff010000)
30292: f0008093      addi ra, ra, -256
     ra  =        -16711936 (ff00ff00)
30296: 0f0f1137      lui sp, -61440
     sp  =        252645376 ( f0f1000)
30300: f0f10113      addi sp, sp, -241
     sp  =        252645135 ( f0f0f0f)
30304: 0020c733      xor a4, ra, sp
     a4  =       -267390961 (f00ff00f)
30308: f00ff3b7      lui t2, -4096
     t2  =       -267390976 (f00ff000)
30312: 00f38393      addi t2, t2, 15
     t2  =       -267390961 (f00ff00f)
30316: 00200193      li gp, 2
     gp  =                2 (       2)
30320: 4a771063      bne a4, t2, 31504
     pc  =            30324 (    7674)
...
```

The last test for xor is test number 27 (from 31468 to 31496). The last test report is saved to the result zone (from 31536 to 31564; see Listing 7.18). The run ends with the ret to 0.

Listing 7.18 The print of the rv32i_npp_ip function

```
...
31468: 111110b7      lui ra, 69632
     ra  =        286330880 (11111000)
31472: 11108093      addi ra, ra, 273
     ra  =        286331153 (11111111)
31476: 22222137      lui sp, 139264
     sp  =        572661760 (22222000)
31480: 22210113      addi sp, sp, 546
     sp  =        572662306 (22222222)
31484: 0020c033      xor zero, ra, sp

31488: 00000393      li t2, 0
     t2  =                0 (       0)
```

```
31492: 01b00193      li gp, 27
     gp  =               27 (      1b)
31496: 00701463      bne zero, t2, 31504
     pc  =            31500 (    7b0c)
31500: 02301263      bne zero, gp, 31536
     pc  =            31536 (    7b30)
31536: 00000513      li a0, 0
     a0  =                0 (       0)
31540: 000022b7      lui t0, 8192
     t0  =             8192 (    2000)
31544: 00028293      addi t0, t0, 0
     t0  =             8192 (    2000)
31548: 0002a303      lw t1, 0(t0)
     t1  =             8344 (    2098)    (m[    2000])
31552: 00a32023      sw a0, 0(t1)
     m[    2098] =                0 (       0)
31556: 00430313      addi t1, t1, 4
     t1  =             8348 (    209c)
31560: 0062a023      sw t1, 0(t0)
     m[    2000] =             8348 (    209c)
31564: dd8f806f      j 292
     pc  =              292 (     124)
0292: 00000093      li ra, 0
     ra  =                0 (       0)
0296: 00008067      ret
     pc  =                0 (       0)
...
```

The registers are printed (prints from the rv32i_npp_ip function, after the do ... while loop exit in Listing 7.19).

Listing 7.19 The print of the rv32i_npp_ip function

```
...
ra  =                0 (       0)
sp  =        572662306 (22222222)
gp  =               27 (      1b)
tp  =                2 (       2)
t0  =             8192 (    2000)
t1  =             8348 (    209c)
t2  =                0 (       0)
...
a1  =               96 (      60)
a2  =                0 (       0)
a3  =             9476 (    2504)
a4  =        267390960 ( ff00ff0)
a5  =                0 (       0)
...
```

Eventually, the testbench main function prints the result of the tests as shown in Listing 7.20.

Listing 7.20 The print of the main function in the testbench_riscv_tests_rv32i_npp_ip.cpp file

```
...
addi  : all tests passed
add   : all tests passed
andi  : all tests passed
and   : all tests passed
...
sub   : all tests passed
sw    : all tests passed
xori  : all tests passed
```

```
xor   : all tests passed
```

If a test failed, the message would be for example "addi : test 8 failed".

7.2.7 Running the Tests on the FPGA

Experimentation

To run the riscv-tests on the development board, proceed as explained in 5.3.10, replacing fetching_ip with rv32i_npp_ip.
The helloworld.c driver is the riscv-tests/my_isa/my_rv32ui/helloworld_rv32i_npp_ip.c file.
As for the simulation, the run on the board should print in the putty window that all the tests passed.

The helloworld_rv32i_npp_ip.c file to drive the FPGA is located in the riscv-tests/my_isa/my_rv32ui folder.

It is shown in Listing 7.21 (do not forget to adapt the path to the hex files to your environment with the update_helloworld.sh shell script, in the same folder).

Listing 7.21 The helloworld_rv32i_npp_ip.c file to run the tests on the FPGA

```
#include <stdio.h>
#include "xrv32i_npp_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE 16
//size in words
#define DATA_RAM_SIZE     (1<<LOG_DATA_RAM_SIZE)
XRv32i_npp_ip_Config *cfg_ptr;
XRv32i_npp_ip         ip;
word_type data_ram[DATA_RAM_SIZE]={
#include "test_0_data.hex"
};
word_type code_ram[CODE_RAM_SIZE]={
#include "test_0_text.hex"
};
char     *name[38] = {
  "addi  ", "add   ", "andi  ", "and   ", "auipc ",
  "beq   ", "bge   ", "bgeu  ", "blt   ", "bltu  ", "bne   ",
  "jalr  ", "jal   ",
  "lb    ", "lbu   ", "lh    ", "lhu   ", "lui   ", "lw    ",
  "ori   ", "or    ",
  "sb    ", "sh    ", "simple",
  "slli  ", "sll   ", "slti  ", "sltiu ", "slt   ", "sltu  ",
  "srai  ", "sra   ", "srli  ", "srl   ", "sub   ", "sw    ",
  "xori  ", "xor   "
};
int main(){
  word_type d;
  cfg_ptr = XRv32i_npp_ip_LookupConfig(
      XPAR_XRV32I_NPP_IP_0_DEVICE_ID);
  XRv32i_npp_ip_CfgInitialize(&ip, cfg_ptr);
  XRv32i_npp_ip_Set_start_pc(&ip, 0);
  XRv32i_npp_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XRv32i_npp_ip_Write_data_ram_Words(&ip, 0, data_ram,
      DATA_RAM_SIZE);
  XRv32i_npp_ip_Start(&ip);
  while (!XRv32i_npp_ip_IsDone(&ip));
  for (int i=0; i<38; i++){
    printf("%s:",name[i]);
    XRv32i_npp_ip_Read_data_ram_Words(&ip, 0x801+i, &d, 1);
    if (d == 0)
      printf(" all tests passed\n");
    else
      printf(" test %d failed\n",(int)d);
  }
}
```

7.3 Running a Benchmark Suite on the Rv32i_npp_ip Processor

All the source files related to the mibench benchmark suite can be found in the mibench/my_mibench folder.

To further test your processor, you should run real programs and compare their results with the ones computed with spike.

A *benchmark suite* is such a set of programs which can be used to compare processors. In your case, the benchmark suite has two goals: testing your processor and comparing different designs (as you will implement other versions of the RISC-V processor in the next chapters).

I have selected the mibench suite. It is made of applications devoted to embedded computing, which is more the target of a home made processor than high performance computing applications.

There are more recent suites but the main problem with a processor on an FPGA is the size of the programmable part of the SoC. The Zynq XC7Z020 offers 140 blocks of BRAM, each block representing 4KB of data. Hence, the code and the data of the RISC-V application to be run on a Pynq-Z1/Pynq-Z2 development board should not be bigger than 560KB.

In the mibench suite there are some oversized codes which I had to remove.

Another constraint comes from the absence of an OS. All the benchmarks are composed of three parts: input, computation, output. The I/O parts use the I/O functions of the standard library, like scanf and printf. It is rather easy to substitute these I/O functions with memory access operations. The scanf inputs are replaced by a set of data initializing the processor data_ram array. The printf outputs are replaced by a set of produced data saved to the data_ram array.

However, some applications use other OS related functions. The most frequently used is malloc. The malloc function allocates memory handled by the OS, not by the compiler. To use malloc on a bare-metal platform, the user needs to mimic the

OS, implementing the management of a memory space out of the one addressed by the compiled code (the *heap* memory).

I decided to simplify the benchmark suite, and to keep only the applications which were easy to port to the no-OS environment.

However, normally a benchmark suite should not be modified. It would not be fair to compare the performance of your RISC-V processor running the modified suite to the performance of an existing processor based on the run of the original suite.

But, when you use a benchmark for your own comparisons (e.g. compare the non pipelined rv32i_npp_ip processor to the rv32i_pp_ip pipelined implementation presented in the next chapter), you can organize your benchmarking as you wish, and as long as the adaptations of the benchmark are the same for every single implementation of the processor. The modified benchmark should also be representative of the particularities of your designs.

Representative means that for example, if you implement caches you need to incorporate to your benchmark suite some programs with different memory access patterns to measure the impact of cache misses on the performance. If on the contrary the memory has a unique access time, the benchmark suite can be simplified.

7.3.1 The Basicmath_small Benchmark From the Mibench Suite

Experimentation

To simulate the mibench benchmarks, for example the basicmath_small benchmark, in a terminal with current directory mibench/my_mibench/my_automotive/basicmath, run the build.sh shell script.

In Vitis_HLS, open the rv32i_npp_ip project and set the testbench file as testbench_basicmath_rv32i_npp_ip.cpp file in the my_mibench/my_automotive/basicmath folder.

In the debug_rv32i_npp_ip.h file, comment all the degugging constant definitions (you open the file from the rv32i_npp_ip.cpp one, **Outline** frame).

Then, **Run C Simulation**.

You can work the same way with the other benchmarks proposed in the mibench/my_mibench and riscv-tests/benchmarks folders.

I have adapted the mibench suite to the Vitis_HLS environment.

In the mibench folder, the my_mibench subfolder contains the adapted version of the mibench suite. The mibench subfolder contains the original mibench files.

7.3.1.1 Make the Benchmarks

You can make all the benchmarks just by running make in each individual folder. For example, to make basicmath_small, run make as shown in Listing 7.22.

Listing 7.22 Making basicmath_small

```
$ cd $HOME/goossens-book-ip-projects/2022.1/mibench/my_mibench/
    my_automotive/basicmath
$ make
...
$
```

In the original mibench suite, for each benchmark, two data sets are provided, a small one and a large one. I have discarded the large data sets as either the text or the data is too big to fit in the data_ram or the code_ram array (the main problem is the size, not the time of the large run). Moreover, for some benchmarks (e.g. basicmath_small), I have reduced the computation to make it runnable on the FPGA (i.e. keep beyond the limit of 140 BRAM blocks available on the Zynq XC7Z020 FPGA).

The basicmath_small code is big because it contains floating-point computations and the compiler targets an ISA with no floating-point instructions (RV32I ISA). Hence, the floating-point operations are computed with library functions using integer instructions. These functions of the mathematical library are linked with the basicmath_small program.

7.3.1.2 Running the Benchmarks with Spike

To run basicmath_small with spike, just type the commands shown in Listing 7.23.

Listing 7.23 Running basicmath_small with spike

```
$ spike /opt/riscv/riscv32-unknown-elf/bin/pk
    ./basicmath_small > small.out
$ head small.out
bbl loader
********* CUBIC FUNCTIONS ***********
Solutions: 2.000000 6.000000 2.500000
Solutions: 2.500000
Solutions: 1.635838
Solutions: 13.811084
Solutions: -9.460302 0.152671 -0.692369
Solutions: -9.448407 0.260617 -0.812210
Solutions: -9.436449 0.348553 -0.912103
Solutions: -9.424429 0.424429 -1.000000
$
```

The code in the basicmath_small.c file uses the printf function to display the results. The spike simulator includes a display driver to print on your computer screen. But the rv32i_npp_ip processor does not.

So, if you compile calls to printf, spike runs the compiled code, which prints. But the same code run on the rv32i_npp_ip processor does not produce any output.

To have an output despite the lack of a driver, the rv32i_npp_ip processor saves its results in the data_ram array. The testbench program or the FPGA driver reads the data_ram array and reconstitutes the output.

7.3.1.3 Adapting the Basicmath_small Benchmark to the Vitis_HLS Environment

In the basicmath_small example, the basicmath_small.c file is replaced by the basicmath_small_no_print.c one (both files are in the my_mibench/my_automotive/basicmath folder).

As its name says, the new file does not contain any print. The two files are nearly identical though. Both compute the same results which are saved in the same memory locations. The only difference is that the printing version reads the memory at the end of the run and prints according to the original basicmath_small.c program.

The code in the basicmath_small.c file is to be run with spike. This run is useful as it provides the reference output.

The non printing basicmath_small_no_print.c version is to be run on the rv32i_npp_ip processor, with a testbench code to produce an output which should be compared to the reference one.

To build the code to be run on the rv32i_npp_ip processor, as for the riscv-tests program, you must build 0-based binary files (with a linker file linker.lds) which you translate to hex files. You can do this with the build.sh script in the my_mibench/my_automotive/basicmath folder (see Listing 7.24).

Listing 7.24 Building the basicmath_small_no_print_0_text.hex and basicmath_small_no_print_0_data.hex files

```
$ cd $HOME/goossens-book-ip-projects/2022.1/mibench/my_mibench/
    my_automotive/basicmath
$ ./build.sh
$
```

The shell script (see Listing 7.25) compiles, builds the two binary files with objcopy and translates to hexadecimal with hexdump.

Listing 7.25 The build.sh file

```
$ cat build.sh
riscv32-unknown-elf-gcc -static -O3 -nostartfiles -o
    basicmath_small_no_print.elf -T linker.lds -Wl,-no-check-
    sections basicmath_small_no_print.c rad2deg.c cubic.c isqrt.c -
    lm
riscv32-unknown-elf-objcopy -O binary --only-section=.text
    basicmath_small_no_print.elf basicmath_small_no_print_0_text.
    bin
riscv32-unknown-elf-objcopy -O binary --only-section=.data
    basicmath_small_no_print.elf basicmath_small_no_print_0_data.
    bin
hexdump -v -e '"0x" /4 "%08x" ",\n"'
    basicmath_small_no_print_0_text.bin >
    basicmath_small_no_print_0_text.hex
hexdump -v -e '"0x" /4 "%08x" ",\n"'
    basicmath_small_no_print_0_data.bin >
    basicmath_small_no_print_0_data.hex
$
```

7.3.1.4 The Linker.lds Script File

To build the basicmath_small_no_print.elf file with all the data gathered in the same .data section, you need a linker script file. The reason is that the compiler produces some read only data, placed in a .rodata section. You have to combine the .rodata and the .data sections into the data_ram array. With no linker script file, the linker puts the read only data in the text section.

A linker script file serves to assemble the sections from the compilation of the different source files (the input sections) into a set of output sections.

The linker.lds file to build the basicmath_small_no_print.elf file is shown in Listing 7.26. It is located in the my_mibench/my_automotive/basicmath folder.

Listing 7.26 The linker.lds file

```
ENTRY(main)
SECTIONS
{
    . = 0;
    .text :
    {
        *(.text.main);
        *(.text*);
    }
    . = 0;
    .data :
    {
        *(.*data*);
    }
    .bss :
    {
        *(.*bss*);
    }
}
```

The linker.lds file describes three *output* sections: the text section named .text, the initialized data section named .data and the uninitialized data section named .bss (they are the sections named out of the curly braces).

These output sections are built from the concatenation of *input* sections taken from an input ELF file (the input sections are named in the curly braces).

The .text output section is composed of all the input sections named .text.main followed by all the input sections prefixed by .text. The .text output section starts at address 0 in the code RAM (". = 0" sets the current address as 0).

The .data output section is also starting at address 0 in the data RAM. It is built from the concatenation of all the data sections in the input files, including the read only data (the read only data sections usually use the word rodata in their names).

The .bss output section is starting after the .data section in the data RAM. It is built from the concatenation of all the bss sections in the input files.

For example, the basicmath_small_no_print.elf is built from the compilation of four files: basicmath_small_no_print.c, rad2deg.c, cubic.c, and isqrt.c. Each compilation builds a .text section. Thus, the .text output section in the basicmath_small_no_print.elf file is the concatenation of the four input .text sections from the intermediate files produced by the compilation of the four ".c" files.

7.3.1.5 Placing the Main Function at Address 0 in the Code Memory

The basicmath_small_no_print.c file is the only one to have a .text.main section. It is defined with an __attribute__ directive associated to the main function (see Listing 7.27).

As the linker script file places the .text.main section in the first place of the .text output section, this is a way to ensure that the main function is placed at the beginning of the code memory, basically at address 0, so the processor starts running the main function.

Listing 7.27 The __attribute__((section)) directive and the result global array to save the computed results

```
#include <string.h>
#include "snipmath.h"
int result[3500];
void main() __attribute__((section(".text.main")));
void main(){
  double  a1 = 1.0, b1 = -10.5, c1 = 32.0, d1 = -30.0;
  double  a2 = 1.0, b2 = -4.5,  c2 = 17.0, d2 = -30.0;
  double  a3 = 1.0, b3 = -3.5,  c3 = 22.0, d3 = -31.0;
  double  a4 = 1.0, b4 = -13.7, c4 = 1.0,  d4 = -35.0;
  double  x[3], d, r;
  double  X;
  int     solutions;
  int     i, i3, i4, gi = 0;
  unsigned long l = 0x3fed0169L, u;
  struct  int_sqrt q;
  long    n = 0;
...
```

7.3.1.6 Saving the Results of the Basicmath_Small Run to Memory

The main function saves its results in the result[3500] global array (see Listing 7.28). The global index gi moves along the result array. Results are just added one after the other. For example, the number of solutions of the first cubic equation is followed by the solutions. The former is an integer and the latter are double precision floating-point numbers.

Listing 7.28 The gi global index

```
...
  /* solve some cubic functions */
  /* should get 3 solutions: 2, 6 & 2.5   */
  SolveCubic(a1, b1, c1, d1, &solutions, x);
  memcpy(&result[gi],&solutions,sizeof(int));
  gi++;
  for(i=0;i<solutions;i++){
    memcpy(&result[gi],&x[i],sizeof(double));
    gi+=2;
  }
...
```

The place of the result global array in memory (0xb18) can be obtained with the objdump tool (see Listing 7.29), after the basicmath_small_no_print.elf file has been built by the build.sh script (the script also builds the basicmath_small_no_ print_0_text.hex file for the code and the basicmath_small_no_print_0_data.hex file for the data):

Listing 7.29 The result symbol address in the .data section (start of the .bss)basicstyle

```
$ ./build.sh
$ riscv32-unknown-elf-objdump -t
    basicmath_small_no_print.elf | grep "result"
00000b18 g     O .bss   000036b0 result
$
```

7.3.1.7 Reconstitution of the Basicmath_Small Output

The testbench file has to reconstitute the output from the content of the result array in the data_ram. The testbench code is shown in Listing 7.30.

The testbench_basicmath_rv32i_npp_ip.cpp file is located in the my_mibench/my_automotive/basicmath folder.

Listing 7.30 The testbench_basicmath_rv32i_npp_ip.cpp file

```
#include <stdio.h>
#include "../../../../rv32i_npp_ip/rv32i_npp_ip.h"
#define PI      3.14159265358979323846
#define RESULT 0xb18
unsigned int data_ram[DATA_RAM_SIZE]={
#include "basicmath_small_no_print_0_data.hex"
};
unsigned int code_ram[CODE_RAM_SIZE]={
#include "basicmath_small_no_print_0_text.hex"
};
int main(){
  unsigned int  nbi, i, gi, solutions;
  double        d;
  double        X;
  int           i3, i4;
  unsigned long l = 0x3fed0169L, u;
  rv32i_npp_ip(0, (instruction_t*)code_ram, (int*)data_ram, &nbi);
  gi = RESULT/4;
  printf("********* CUBIC FUNCTIONS ***********\n");
  printf("Solutions:");
  memcpy(&solutions, &data_ram[gi], sizeof(int));
  gi++;
  for(i=0;i<solutions;i++){
    memcpy(&d,&data_ram[gi],sizeof(double));
    gi+=2;
    printf(" %f",d);
  }
  printf("\n");
  ...
  printf("********* ANGLE CONVERSION ***********\n");
  for (i=0, X = 0.0; X <= 360.0; i++, X += 1.0){
    memcpy(&d,&data_ram[gi],sizeof(double));
    gi+=2;
    printf("%3.0f degrees = %.12f radians\n", X, d);
  }
  puts("");
  for (i=0, X = 0.0; X <= (2 * PI + 1e-6); i++, X += (PI / 180)){
    memcpy(&d,&data_ram[gi],sizeof(double));
    gi+=2;
    printf("%.12f radians = %3.0f degrees\n", X, d);
  }
  printf("%d fetched, decoded and run instructions\n", nbi);
  return 0;
}
```

7.3.1.8 The Testbench Prints

The run prints what is shown in Listing 7.31 (30 million instructions run in about three minutes).

Listing 7.31 The main function prints

```
********* CUBIC FUNCTIONS ***********
Solutions: 2.000000 6.000000 2.500000
Solutions: 2.500000
Solutions: 1.635838
Solutions: 13.811084
Solutions: -9.460302 0.152671 -0.692369
Solutions: -9.448407 0.260617 -0.812210
Solutions: -9.436449 0.348553 -0.912103
Solutions: -9.424429 0.424429 -1.000000
...
6.178465552060 radians = 354 degrees
6.195918844580 radians = 355 degrees
6.213372137100 radians = 356 degrees
6.230825429620 radians = 357 degrees
6.248278722140 radians = 358 degrees
6.265732014660 radians = 359 degrees
6.283185307180 radians = 360 degrees
30897739 fetched, decoded and run instructions
```

7.3.2 Running the Basicmath_Small Benchmark on the FPGA

Experimentation

To run the basicmath_small benchmark on the development board, proceed as explained in 5.3.10, replacing fetching_ip with rv32i_npp_ip.

The helloworld.c driver is the mibench/my_mibench/my_automotive/basicmath/helloworld_rv32i_npp_ip.c file.

The printed results should be identical to the simulation ones and to the spike run ones.

The helloworld_rv32i_npp_ip.c file to drive the FPGA is shown in Listing 7.32 (do not forget to adapt the path to the hex files to your environment with the update_helloworld.sh shell script).

The helloworld_rv32i_npp_ip.c file is located in the my_mibench/my_automotive/basicmath folder.

Listing 7.32 The helloworld_rv32i_npp_ip.c file to run basicmath_small_no_print on the FPGA

```
#include <stdio.h>
#include "xrv32i_npp_ip.h"
#include "xparameters.h"
#define PI 3.14159265358979323846
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE 16
//size in words
#define DATA_RAM_SIZE     (1<<LOG_DATA_RAM_SIZE)
XRv32i_npp_ip_Config *cfg_ptr;
XRv32i_npp_ip          ip;
word_type data_ram[DATA_RAM_SIZE]={
#include "basicmath_small_no_print_0_data.hex"
};
word_type code_ram[CODE_RAM_SIZE]={
#include "basicmath_small_no_print_0_text.hex"
};
int main(){
  unsigned int  i, gi, solutions;
  double        d;
  double        X;
  int           i3, i4;
  unsigned long l = 0x3fed0169L, u;
  cfg_ptr = XRv32i_npp_ip_LookupConfig(
      XPAR_XRV32I_NPP_IP_0_DEVICE_ID);
  XRv32i_npp_ip_CfgInitialize(&ip, cfg_ptr);
  XRv32i_npp_ip_Set_start_pc(&ip, 0);
  XRv32i_npp_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XRv32i_npp_ip_Write_data_ram_Words(&ip, 0, data_ram,
      DATA_RAM_SIZE);
  XRv32i_npp_ip_Start(&ip);
  while (!XRv32i_npp_ip_IsDone(&ip));
  gi = 0xb18/4;
  printf("********* CUBIC FUNCTIONS ***********\n");
  printf("Solutions:");
  XRv32i_npp_ip_Read_data_ram_Words(&ip, gi, (word_type*)&solutions
     , 1);
  gi++;
  for(i=0;i<solutions;i++){
    XRv32i_npp_ip_Read_data_ram_Words(&ip, gi, (word_type*)&d, 2);
    gi+=2;
    printf(" %f",d);
  }
  printf("\n");
  ...
  for (i=0, X = 0.0; X <= (2 * PI + 1e-6); i++, X += (PI / 180)){
    XRv32i_npp_ip_Read_data_ram_Words(&ip, gi, (word_type*)&d, 2);
    gi+=2;
    printf("%.12f radians = %3.0f degrees\n", X, d);
  }
  printf("%d fetched, decoded and run instructions\n",
         (int)XRv32i_npp_ip_Get_nb_instruction(&ip));
  return 0;
}
```

The prints of the helloworld (run on the FPGA from Vitis IDE) and the testbench (simulated in Vitis HLS) are identical.

7.3.3 The Other Benchmarks of the Mibench Suite

To run the other benchmarks of the adapted mibench suite, you proceed the same way.

For a benchmark named bench.c in the mibench/my_mibench/my_dir folder, first make (the Makefile builds the bench executable file) (as bench.c is compiled with the riscv32-unknown-elf-gcc compiler with the standard linking, it is to be run only by spike, not by the rv32i_npp_ip processor).

Then, run spike on the bench executable. This produces the reference output file.

The second step is to build the bench_no_print_0_text.hex and bench_no_print_0_data.hex files which are used by the testbench program (you can find prebuilt hex files in each benchmark folder). You run build.sh which compiles the bench_no_print.c version of bench.c with no printing. The data and text sections of the ELF files are extracted by objcopy and the hex files are built with hexdump.

The third step is to run the rv32i_npp_ip processor. You add the testbench_bench_rv32i_npp_ip.cpp file found in the mibench/my_mibench/my_dir folder and you start the Vitis_HLS simulation.

After the simulation, you can compare its output to the reference one. They should be identical.

Then, you have to run the helloworld_rv32i_npp_ip.c on the FPGA (z1_rv32i_npp_ip Vivado project).

Before building the Vitis IDE project from the helloworld_rv32i_npp_ip.c program, make sure to adapt the path to the hex files to your own environment. For this purpose, run the update_helloworld.sh shell script.

Notice that some mibench benchmarks are not suited to the Basys3 (XC7A35T) or any board based on the XC7Z010 FPGA because of the number of BRAM blocks available. On the XC7Z020 chip, there are 140 blocks, i.e. 560KB of memory. On the XC7Z010 chip, there are only 60 blocks, i.e. 240KB and on the XC7A35T there are only 50 blocks, i.e. 200KB.

I have added seven benchmarks which are included in the riscv-tests (in the riscv-tests/benchmarks folder). Their testing procedure is identical to the mibench one (each benchmark folder includes a build.sh shell script, an update_helloworld.sh script and a Makefile to build the executable for spike).

7.3.4 The Mibench and Riscv-Tests Benchmarks Execution Times on the Rv32i_npp_ip Implementation

Table 7.1 shows the execution time (in seconds) on the FPGA implementation of the rv32i_npp_ip processor of the benchmarks as computed with equation 5.1 (*nmi* * *cpi* * *c*, where $c = 70$ ns). The CPI value is 1 (each instruction is fully processed in a single loop iteration, i.e. in a single processor cycle).

These execution times are the run times on the FPGA (not considering the reconstitution of the output). The run times on the Vitis_HLS simulator are significantly higher (for example, the mm benchmark is run in 11 s on the FPGA and 13 min on the simulator).

Table 7.1 Number of Machine Instructions run (*nmi*) and execution times of the mibench and riscv-tests suites on the rv32i_npp_ip processor

Benchmark	NMI	Time (s)
Basicmath	30,897,739	2.162841730
Bitcount	32,653,239	2.285726730
qsort	6,683,571	0.467849970
Stringsearch	549,163	0.038441410
Rawcaudio	633,158	0.044321060
Rawdaudio	468,299	0.032780930
crc32	300,014	0.021000980
fft	31,365,408	2.195578560
fft_inv	31,920,319	2.234422330
Median	27,892	0.001952440
mm	157,561,374	11.029296180
Multiply	417,897	0.029252790
qsort	271,673	0.019017110
spmv	1,246,152	0.087230640
Towers	403,808	0.028266560
vvadd	16,010	0.001120700

7.4 Proposed Exercises: The RISC-V M and F Instruction Extensions

7.4.1 Adapt the Rv32i_npp_ip Design to the RISC-V M Extension

This section will make you modify the rv32i_npp_ip processor to add the M RISC-V extension.

I do not provide the resulting IP. You must design it yourself until it works on your development board. I will simply list the different steps you will have to go through to achieve the exercise.

First, you should check the RISC-V specification document to learn about the M extension (Chap. 7 of [1]). In the same document, you will find the coding definition of these new instructions in Chap. 24 (page 131: RV32M standard extension).

Second, you should open a new Vitis_HLS project named rv32im_npp_ip with a top function having the same name. You can copy all the code from the rv32i_npp_ip folder to the rv32im_npp_ip one as a starting point.

Third, you should update the rv32im_npp_ip.h file to add the new constants to define the M extension (opcode, func3 and func7 fields in the decoding).

Fourth, you should update the execute function of the rv32im_npp_ip code. The fetch and the decode functions are not impacted (fetching a multiplication is not different from fetching an addition, decoding, i.e. decomposing into pieces, is not impacted either as the format of multiplication and division instructions is R-TYPE).

In the execute function, you will find the compute_result function which computes a result according to the format of the instruction. The R-TYPE computation calls the more specialized function compute_op_result which computes arithmetic operations according to the func3 and func7 fields of the instruction encoding.

You must update the switch in the function to take care of the different new encodings proposed in the M extension. For example, to multiply rv1 by rv2, just write rv1 * rv2 in C and the synthesizer will do the rest, implementing a hardware multiplier. For each RISC-V operation, you only need to find the matching C operator (e.g. remember that the % operator in C is the remainder of the division).

Once your compute_op_result function has been adapted, you can simulate your IP. You will need to write a RISC-V assembly code involving each of the newly implemented instructions.

It is probably faster to directly write in RISC-V assembly language than compile from C. Use the test_op.h code as a model.

If you want to produce assembly code from a C source, use the march option of the compiler to get multiplication/division RISC-V instructions: "riscv32-unknown-elf-gcc source.c -march=rv32im -mabi=ilp32 -o executable".

You keep the same testbench program and you make it run with your multiplication and division test code.

When your IP is correctly simulated, you can start the synthesis. You will have to adapt the processor cycle duration to the multiplication and division duration (the processor cycle should be long enough to compute a division). This means changing the HLS PIPELINE II value.

It is totally inefficient to align all the instructions execution time to the division one. However, this inefficiency comes from the non pipelined organization. In Chap. 9, I present the multicycle pipeline organization which is suited to different execution time operators.

Once the synthesis is done, you can export it.

In Vivado, you can build your new design involving your new rv32im_npp_ip. Then, you create the HDL wrapper, you generate the bitstream and you export the hardware, including the bitstream.

You switch your board on and you start a putty terminal.

In Vitis IDE, you create your application project with the exported bitstream, you generate an initial helloworld.c file, you replace it by a driver producing the same results than your testbench program.

You open a connection with the board.

You program the FPGA with the bitstream and you run the driver on the development board.

The whole process should be fulfilled in a time ranging from a single day to a full week.

7.4.2 Adapt the Rv32i_npp_ip Design to the RISC-V F Extension

A more ambitious project is to add the F extension (floating-point, simple precision). This one should take between a week and a month to be fulfilled.

To produce F and D extension instructions from a C source, use option march=rv 32if or march=rv32id (e.g. "riscv32-unknown-elf-gcc source.c -march=rv32if -mabi=ilp32 -o executable").

To handle the two register files (integer registers and floating-point registers), it is useful to define a *union* type mixing two decodings of a unique 32-bit value (see Listing 7.33). On one side, the value is to be interpreted as an integer. On the other side, it is to be interpreted as a single precision floating-point number.

Listing 7.33 A union type mixing two interpretations of a 32-bit value

```
typedef union int_spfp_u{
  int   i;
  float f;
} int_spfp_t
```

If *u* is declared as int_spfp_t, *u.i* is the integer interpretation of value *u* and *u.f* is its single precision floating-point interpretation, as illustrated in Listing 7.34.

Listing 7.34 Using a union variable

```
#include <stdio.h>
typedef union int_spfp_u{
  int   i;
  float f;
} int_spfp_t;
int_spfp_t u;
void main(){
  u.f = 1.0;
  printf("u.f=%f, u.i=%d, u=%8x\n",u.f,u.i,u.i);
}
```

The code in Listing 7.34 prints what is shown in Listing 7.35.

Listing 7.35 Using a union variable

```
u.f=1.000000, u.i=1065353216, u=3f800000
```

In the simulation phase, such a code works fine. But it assumes that the *u.i* and *u.f* fields of the union share the same memory location. This is not the case for the synthesized version of the union.

In Vitis_HLS, I recommend to use unions with care. They are safe if the utilization of the union is confined in a single function (e.g. set the union at the beginning of the function and use its fields in the function body).

This is what the Vitis documentation says about unions:

“Unlike C/C++ compilation, synthesis does not guarantee using the same memory (in the case of synthesis, registers) for all fields in the union. Vitis HLS perform the optimization that provides the most optimal hardware.”

7.5 Debugging Hints

If you are used to debugging your programs with fancy debuggers, you may find debugging hardware rather spartan. However, there are a few tricks to make it possible to debug an IP without going down to VHDL and chronograms.

7.5.1 Synthesis Is Not Simulation: You Can Disactivate Some Parts of the Simulation

The main hint comes from the remark that synthesis is not simulation. During synthesis, the code is not run. Hence, the code to be synthesized does not need to be runnable. You can easily disactivate some parts of your code.

As in all debugging work, the main technique is to isolate the faulty part (with the faulty code, there is an error, without there is no error). In the same manner, when your constraints cannot be satisfied by the synthesizer, try to reduce your code.

You eliminate the error by disactivating functions or by commenting some lines. Do not be shy in doing so: remember that the synthesis does not run the code. However, beware that the synthesizer will eliminate unused parts of the code and the resulting synthesis might become empty.

When the constraints are satisfied, start adding back the discarded parts of the code, step by step, until the problem arises again.

When the origin of the problem is detected, you can work on the code to reorganize it in order to obtain a synthesizable version.

7.5.2 Infinite Simulation: Replace the "Do ... While" Loop By a "For" Loop

A second hint concerns the simulation of the processors. The do ... while loop may sometimes run infinitely.

A way around this is to turn it into a for loop, running enough iterations to end the RISC-V test program (all the test programs I have given terminate in a few tens of cycles).

7.5.3 Frozen IP on the FPGA: Check Ap_int and Ap_uint Variables

A third hint concerns frozen IPs on the FPGA. Sometimes, your synthesis seems fine but when you run the bitstream on the board, nothing prints. This is because your IP did not reach the end of the do ... while loop. Hence, no IsDone signal is sent from the IP to the Zynq and your helloworld.c driver stays stuck waiting for ever in the "while (!X...IsDone(&ip));" loop.

This might come from a missing bit in an ap_uint or ap_int variable.

For example, the code shown in Listing 7.36 illustrates a badly typed loop counter.

Listing 7.36 Bad loop

```
typedef ap_uint<4> loop_counter_t;
loop_counter_t i;
for (i=0; i<16; i++){
  ...//iteration body using "i"
}
```

The simulation will be correct because the compiler will replace the loop_counter_t type by a char type (the smallest C defined type large enough to fit the loop_counter_t type). But the synthesizer will produce an RTL with exactly four bits for *i*. The *i*++ operation is computed on four bits, i.e. modulo 16, and *i* never reaches value 16. Hence, on the FPGA, the loop never ends.

Please refer back to the end of 5.5.4 to find the correct way to declare *i*.

7.5.4 Frozen IP on the FPGA: Check Computations Inside #Ifndef __SYNTHESIS__

An IP run may also block because some necessary computation is within #ifndef __SYNTHESIS__ and #endif. In this case, the simulation may be correct and the FPGA run incorrect.

7.5.5 Frozen IP on the FPGA: Replace the "While (!IsDone(...));" Loop By a "For" Loop

When an IP is not exiting, you can try to replace the while loop waiting for the Done signal by an empty for loop in the helloworld driver (i.e. replace "while (!X..._ip_IsDone(...));" by "for (int i=0; i<1000000; i++);").

The empty for loop does not wait for the IP to finish. The remaining of the driver should print the state of the memory to give some indications on how far the run went until the IP got stuck.

7.5.6 Frozen IP on the FPGA: Reduce the RISC-V Code Run

Another hint is related to the fact that your IP is a processor running a RISC-V code. As the IP is built to run any RISC-V code, you can eliminate some parts of the RISC-V code directly in the hex file included by the helloworld driver (e.g. comment the hexadecimal codes you want to remove).

For example, you can limitate the RISC-V code to its last RET instruction and see if you IP runs properly. By reducing the RISC-V code, you can isolate (i.e. eliminate) the faulty instructions and know which part of your HLS code should be corrected.

7.5.7 Non Deterministic Behaviour on the FPGA: Check Initializations

If multiple runs of the same RISC-V code produce different results on the FPGA, it probably means that some variable in the IP implementation code has not been initialized. There are no implicit initializations on the FPGA as there are in C. Hence, the simulation might be correct and the run on the FPGA be faulty.

7.5.8 Debugging Prints Along the Run on the FPGA

Even though there is no debugger for the IP itself (Vitis IDE has a debugger but it concerns only the helloworld driver, not the IP of course), you can make your processor IP give you some information about the progression of the run.

Instead of printing a message, which the RISC-V code cannot do, you can add some STORE instructions to the RISC-V code (of course, your IP must at least be able to run STORE instructions properly). In your driver program running in Vitis IDE, you can check the stores, e.g. read the respective memory addresses, like you would check successive prints.

Reference

1. https://riscv.org/specifications/isa-spec-pdf/

Building a Pipelined RISC-V Processor

8

Abstract

This chapter will make you build your second RISC-V processor. The implemented microarchitecture proposed in this second version is pipelined. Within a single processor cycle, the updated processor fetches and decodes instruction *i*, executes instruction *i-1*, accesses memory for instruction *i-2* and writes a result back for instruction *i-3*.

8.1 First Step: Control a Pipeline

All the source files related to the simple_pipeline_ip can be found in the simple_pipeline_ip folder.

8.1.1 The Difference Between a Non-pipelined and a Pipelined Microarchitecture

Figure 8.1 shows the difference between a non-pipelined microarchitecture (as the rv32i_npp_ip built in Chap. 6) and a pipelined one. The main difference is in the way the iteration of the do ... while loop is considered.

In the non-pipelined design, one iteration of the main loop (rectangular red box) contains all four of the main steps of an instruction processing: fetch, decode, execute (including memory access), and writeback. This was done in seven FPGA cycles on the rv32i_npp_ip design. One loop iteration matches the full processing of one instruction (blue box) from fetch to writeback. In the non-pipelined design, the blue box (which surrounds the instruction execution) and the red box (which surrounds the iteration) are identical.

In a pipelined design with multiple *pipeline stages* (two stages in Fig. 8.1), one iteration (vertical red box) contains the execute and writback ending steps of an

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_8

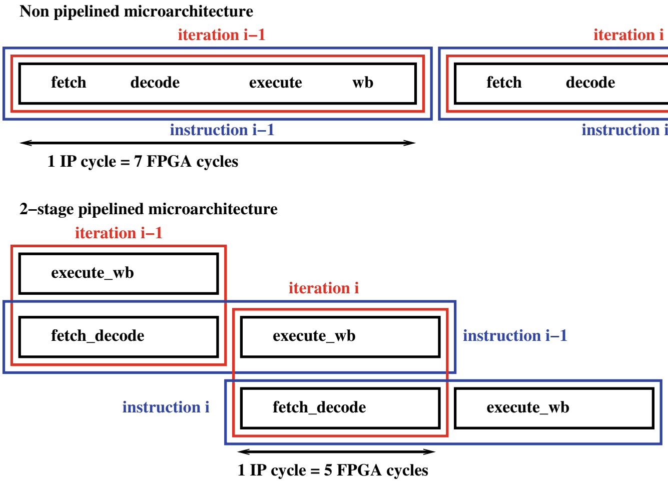


Fig. 8.1 Non-pipelined versus pipelined implementations

instruction processing (iteration *i* contains the execute_wb stage of instruction *i-1* pipelined processing) and the fetch and decode starting steps of the next instruction processing (iteration *i* contains the fetch_decode stage of instruction *i* pipelined processing).

The instruction processing (horizontal blue box) is distributed on two successive iterations. In the pipelined design, the red box extends vertically (to surround the different pipeline stages in the same cycle) and the blue box extends horizontally (to surround the instruction processing spanning successive cycles).

If the fetch_decode stage of instruction *i* is made independent from the execute_wb stage of instruction *i-1*, the two stages in iteration *i* can be run in parallel, resulting in a reduced iteration latency of five FPGA cycles.

In the non-pipelined design, the input of the execute function is the output of the decode function (what is decoded is executed in the same iteration). In the pipelined design, the output of the fetch_decode stage at iteration *i-1* (fetching and decoding of instruction *i-1*) is the input of the execute_wb stage at iteration *i* (execution and writeback of instruction *i-1*).

To implement this, I swap the functions in the main loop as shown in the code in Listing 8.1, which is the code presented in Listing 6.2, but with a permutation of all the function calls.

In Listing 6.2 code, there were Read After Write (RAW) dependencies on variables instruction, d_i, and on the reg_file array (there is a RAW dependency on the variable *v* if *v* is written and after that, *v* is read; the read depends on the write).

In Listing 8.1 code, these dependencies have been removed as the reads have been placed before the writes, like for example the read of reg_file in running_cond_update which precedes the write to reg_file in execute.

The dependence on pc is not removed by the permutation because the execute function writes to pc (the next pc) before the fetch function reads it (to fetch). However, the pc read by the fetch function should be the pc manipulated by the execute function *before* the next pc computation.

I duplicated pc with pc_i which is pc at iteration *i* and pc_ip1 which is pc at iteration *i+1*.

The execute function reads pc_i to compute pc_ip1. The fetch function reads the same pc_i and does not depend on execute. In such a way, fetch and execute can be run independently because they both work on their own copy of pc_i.

Listing 8.1 A pipelined processor

```
do{
  pc_i = pc_ip1;
  //read reg_file
  running_cond_update(instruction, reg_file, &is_running);
  //write reg_file and read d_i
  execute(pc_i, reg_file, data_ram, d_i, &pc_ip1);
  //write d_i and read instruction
  decode(instruction, &d_i);
  //write instruction and read pc
  fetch(pc_i, code_ram, &instruction);
} while (is_running);
```

With these two techniques (permutation and duplication), the loop contains independent functions which the synthesizer schedules in parallel.

8.1.2 The Inter-stage Connection Structures

I have added structured variables encapsulating what each of the two stages computes for the next iteration.

- f_to_f and f_to_e are variables written by the fetch_decode stage at the end of the iteration. At iteration *i*, the fetch_decode stage saves respectively what is to be used by the fetch_decode stage and by the execute_wb stage at iteration *i+1*. f_to_f stands for a communication link between the fetch_decode stage and itself. f_to_e stands for a communication link between the fetch_decode stage and the execute_wb stage.
- e_to_f and e_to_e are similar variables written by the execute_wb stage.

I have also added matching variables following the naming scheme y_from_x and x_to_y for stages *x* and *y*, e.g. e_from_f and f_to_e. Variable y_from_x serves as a duplication of variable x_to_y. Variable x_to_y is copied into y_from_x at the beginning of the iteration (like pc_ip1 was copied into pci).

I added the other _from_ variables to make all the function calls fully permutable. It turns out that when the calls are ordered with fetch_decode before execute_wb, the Vivado implementation uses less resources than when they are in the reverse order. I have no explanation for this.

The simple_pipeline_ip.h file (see Listing 8.2) contains the type definitions for the _from_ and _to_ variable declarations:

Listing 8.2 The inter-stage transmission types definitions in the simple_pipeline_ip.h file

```
typedef struct from_f_to_f_s{
  code_address_t next_pc;
} from_f_to_f_t;
typedef struct from_f_to_e_s{
  code_address_t        pc;
  decoded_instruction_t d_i;
#ifndef __SYNTHESIS__
#ifdef DEBUG_DISASSEMBLE
  instruction_t         instruction;
#endif
#endif
} from_f_to_e_t;
typedef struct from_e_to_f_s{
  code_address_t target_pc;
  bit_t          set_pc;
} from_e_to_f_t;
typedef struct from_e_to_e_s{
  bit_t cancel;
} from_e_to_e_t;
```

The fetch_decode stage sends its computed next pc to itself (for non branching/jumping instructions) with f_to_f.next_pc. It sends the fetching pc and the decoded instruction d_i to the next iteration execute_wb stage with f_to_e.pc and respectively f_to_e.d_i.

The execute_wb stage sends its computed target_pc and the set_pc bit to the next iteration of the fetch_decode stage with e_to_f.target_pc and respectively e_to_f.set_pc. The set_pc bit indicates if the executed instruction is a branching/jumping one.

The execute_wb stage sends a cancel bit to itself with e_to_e.cancel. This cancel bit is set if the executed instruction is a branching/jumping one (hence, set_pc sent to the fetch_decode stage and cancel sent to execute_wb stage have the same value). The cancel bit indicates that the next instruction execution should be cancelled (this is explained in Sect. 8.1.4).

The f_to_f, f_to_e, e_to_f, e_to_e, and the matching _from_ variables are declared in the simple_pipeline_ip function. They are initialized to initiate the pipeline as if the previous instruction executed is a branching/jumping one, with start_pc as the target (e_to_f.set_pc is set). The execute_wb stage should not do any work during the first iteration (e_to_e.cancel is set)

8.1.3 The IP Top Function

The simple_pipeline_ip function in the simple_pipeline_ip.cpp file has the same prototype as the rv32i_npp_ip one (see Listing 8.3).

Listing 8.3 The simple_pipeline_ip function: prototype, local declarations and initializations

```
void simple_pipeline_ip(
  unsigned int  start_pc,
  unsigned int  code_ram[CODE_RAM_SIZE],
  int           data_ram[DATA_RAM_SIZE],
  unsigned int *nb_instruction){
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=code_ram
#pragma HLS INTERFACE s_axilite port=data_ram
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INLINE recursive
  int            reg_file[NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file dim=1 complete
  from_f_to_f_t  f_to_f, f_from_f;
  from_f_to_e_t  f_to_e, e_from_f;
  from_e_to_f_t  e_to_f, f_from_e;
  from_e_to_e_t  e_to_e, e_from_e;
  bit_t          is_running;
  unsigned int   nbi;
  for (int i=0; i<NB_REGISTER; i++) reg_file[i] = 0;
  e_to_f.target_pc = start_pc;
  e_to_f.set_pc    = 1;
  e_to_e.cancel    = 1;
  nbi              = 0;
  ...
```

The main loop (see Listing 8.4) contains the current fetch (instruction *i*, fetch_decode function) and the execution of the instruction fetched and decoded in the previous iteration (instruction *i-1*, execute_wb function).

The Initiation Interval is set to 5 (II = 5). This will lead to a processor cycle set at five FPGA cycles (20Mhz).

Listing 8.4 The simple_pipeline_ip function: main loop

```
  ...
  do{
#pragma HLS PIPELINE II=5
    f_from_f = f_to_f; e_from_f = f_to_e;
    f_from_e = e_to_f; e_from_e = e_to_e;
    fetch_decode(f_from_f, f_from_e, code_ram, &f_to_f, &f_to_e);
    execute_wb(e_from_f, e_from_e, reg_file, data_ram, &e_to_f, &
        e_to_e);
    statistic_update(e_from_e.cancel, &nbi);
    running_cond_update(e_from_e.cancel, e_from_f.d_i.is_ret,
        e_to_f.target_pc, &is_running);
  } while (is_running);
  ...
}
```

The simple_pipeline_ip function is the description of a multi component design in which the two pipeline stages represent two components, as shown in Fig. 8.2 (fetch_decode red box and links and execute_wb blue box and links).

8.1.4 Control Flow Instructions Handling in the Pipeline

When the decoded instruction is a control flow one (i.e. either a jump or a taken conditional branch), the next fetch address (the computed control flow target address) is given by the execute_wb function of the next iteration.

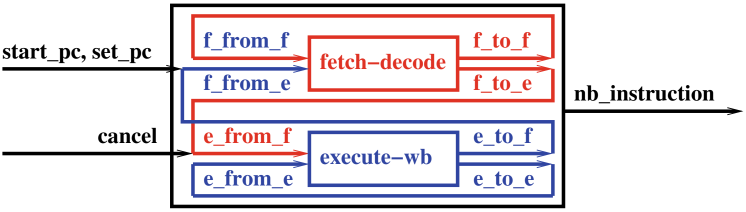


Fig. 8.2 Two components in an IP

Hence, the next iteration fetch_decode function fetches at a wrong address (pc + 1 instead of the computed target address).

To avoid running wrong instructions, we can cancel an execution by blocking its updates (at least, destination register writes and memory stores).

This is a general pattern: an instruction *i* has an impact on a later one *i+k* (e.g. *i* cancels *i+1*). A value computed by *i* is *forwarded* to *i+k* (e.g. a cancel bit is forwarded).

In the control flow instruction case, there are three forwarded values: the set_pc bit, the computed target address and the cancel bit. The set_pc bit is set by the execute_wb function if the instruction is a JAL, a JALR, or a taken branch. For any other instruction, the set_pc bit is cleared. The cancel bit is a copy of the set_pc bit.

The set_pc bit is used by the fetch_decode function called in the next iteration. It is transmitted by the e_to_f and f_from_e variables and used to decide which pc is valid for the next fetch: the next_pc computed in the fetch_decode stage (set_pc bit is 0) or the target_pc forwarded by the execute_wb stage (set_pc bit is 1).

Figure 8.3 illustrates this double transmission (in magenta) between the red rectangle (iteration *i*) and the green adjacent one (next iteration *i+1*).

The cancellation bit is forwarded by the execute_wb stage to itself (see the blue arrow in Fig. 8.4). When the cancel bit is set, it cancels the execution in the next iteration.

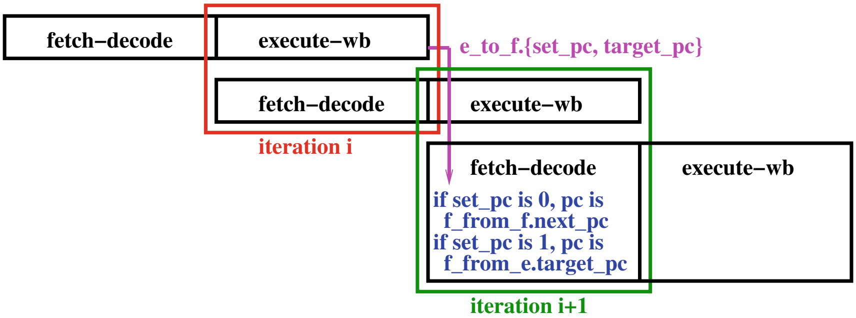


Fig. 8.3 Pipeline transmission (set_pc bit, target address)

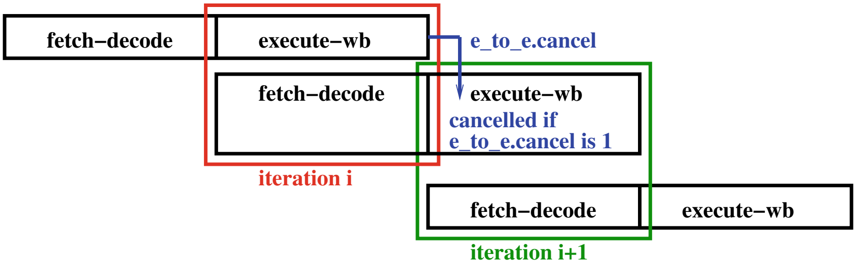


Fig. 8.4 Cancellation

8.1.5 The Fetch_decode Pipeline Stage

The fetch_decode function code (in the fetch_decode.cpp file) is shown in Listing 8.5.

The fetch pc is either the next pc computed by the fetch_decode function or the target pc computed by the execute_wb function at the preceding iteration. The selection bit is set_pc computed by the execute_wb function at the preceding iteration. The fetch_decode function sends pc to the execute_wb function through f_to_e->pc.

Listing 8.5 The fetch_decode function

```
void fetch_decode(
  from_f_to_f_t   f_from_f,
  from_e_to_f_t   f_from_e,
  unsigned int   *code_ram,
  from_f_to_f_t  *f_to_f,
  from_f_to_e_t  *f_to_e){
  code_address_t pc;
  instruction_t   instruction;
  pc = (f_from_e.set_pc)   ?
         f_from_e.target_pc : f_from_f.next_pc;
  fetch(pc, code_ram, &(f_to_f->next_pc), &instruction);
  decode(instruction, &(f_to_e->d_i));
  f_to_e->pc = pc;
#ifndef __SYNTHESIS__
#ifdef DEBUG_DISASSEMBLE
  f_to_e->instruction = instruction;
#endif
#endif
}
```

The fetch function code (in the fetch.cpp file) is shown in Listing 8.6.

Listing 8.6 The fetch function

```
void fetch(
  code_address_t   pc,
  instruction_t   *code_ram,
  code_address_t  *next_pc,
  instruction_t   *instruction){
  *next_pc = (code_address_t)(pc + 1);
  *instruction = code_ram[pc];
}
```

The decode function (in the decode.cpp file) is unchanged except for the adding of an is_jal bit (new field in the decoded_instruction_t type and set in the decode_instruction function). The is_jal bit is used in the execute function.

8.1.6 The Execute_wb Pipeline Stage

The execute_wb function (see Listings 8.7 to 8.10) is located in the execute_wb.cpp file.

It receives e_from_f and e_from_e and sends e_to_f and e_to_e.

Listing 8.7 The execute_wb function prototype and local declarations

```
void execute_wb(
  from_f_to_e_t   e_from_f,
  from_e_to_e_t   e_from_e,
  int            *reg_file,
  int            *data_ram,
  from_e_to_f_t  *e_to_f,
  from_e_to_e_t  *e_to_e){
  int    rv1, rv2, rs, op_result, result;
  bit_t bcond, taken_branch;
  ...
```

The execute_wb stage stays idle when the e_from_e.cancel bit is set. In the same iteration, the fetch_decode function fetches the target_pc instruction (see iteration *i* in Fig. 8.5).

When the e_from_e.cancel bit is set, the execute_wb function does not compute anything (see Listing 8.8). It only clears the e_to_f.set_pc and the e_to_e.cancel bits.

Listing 8.8 The execute_wb function computation: cancellation

```
  ...
  if (e_from_e.cancel){
    e_to_f->set_pc = 0;
    e_to_e->cancel = 0;
  }
  ...
```

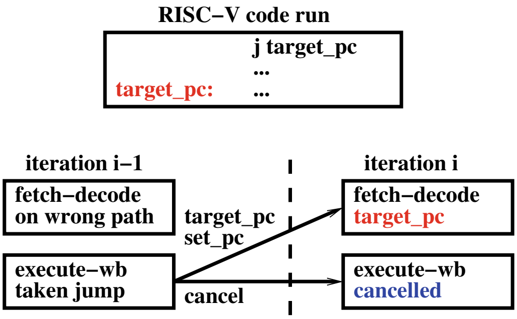


Fig. 8.5 The pipeline when the cancel bit is set

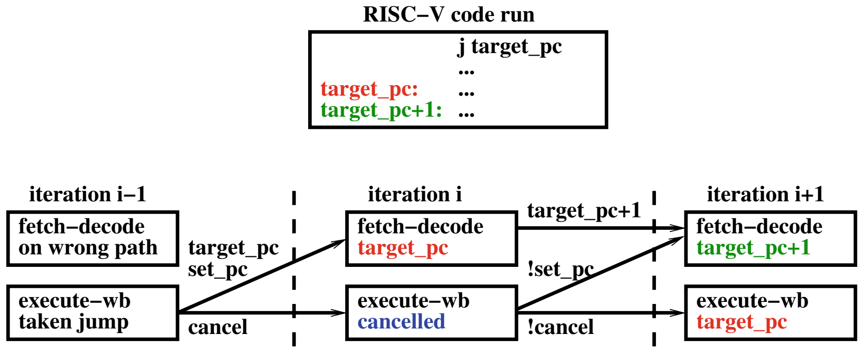


Fig. 8.6 The pipeline after the cancel bit is cleared

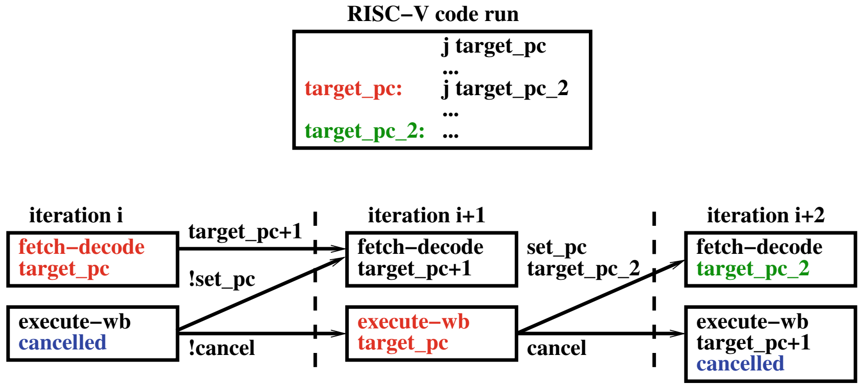


Fig. 8.7 Two back-to-back taken jumps

Figure 8.6 shows what happens on the *i+1* iteration after cancellation, due to cleared set_pc and cancel bits.

Figure 8.7 shows the pipeline if the target instruction is also a taken jump.

When the e_from_e.cancel bit is not set, the execute_wb stage executes the instruction fetched in the preceding iteration and computes the value of the e_to_f.set_pc and the e_to_e.cancel bits.

Listing 8.9 The execute_wb function computation: no cancellation

```
...
else{
  read_reg(reg_file, e_from_f.d_i.rs1, e_from_f.d_i.rs2, &rv1, &
      rv2);
  bcond        = compute_branch_result(rv1, rv2, e_from_f.d_i.
      func3);
  taken_branch = e_from_f.d_i.is_branch && bcond;
  rs           =(e_from_f.d_i.is_r_type)?
                 rv2:(int)e_from_f.d_i.imm;
  op_result    = compute_op_result(e_from_f.d_i, rv1, rs);
```

```
    result         = compute_result(e_from_f.pc, e_from_f.d_i, rv1,
        op_result);
    if (e_from_f.d_i.is_store)
      mem_store(data_ram, (b_data_address_t)result, rv2, e_from_f.
          d_i.func3);
    else if (e_from_f.d_i.is_load)
      result = mem_load(data_ram, (b_data_address_t)result,
          e_from_f.d_i.func3);
    write_reg(e_from_f.d_i, reg_file, result);
    e_to_f->target_pc =
      compute_next_pc(e_from_f.pc, e_from_f.d_i, bcond, rv1);
    e_to_f->set_pc    = e_from_f.d_i.is_jalr ||
                       (e_from_f.d_i.is_jal) ||
                        taken_branch;
    e_to_e->cancel    = e_to_f->set_pc;
    ...
```

The debugging prints are all placed in the execute_wb function. They are organized in a way to produce the same outputs than in the rv32i_npp_ip design.

Listing 8.10 The execute_wb function computation: debugging prints

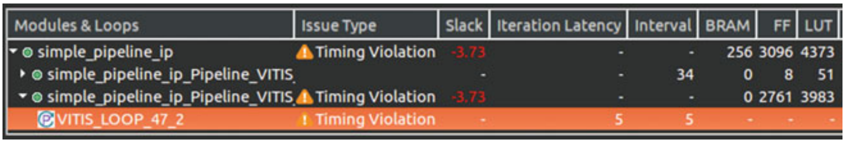
```
    ...
#ifndef __SYNTHESIS__
#ifdef DEBUG_FETCH
    printf("%04d: %08x      ",
     (int)(e_from_f.pc<<2), e_from_f.instruction);
#ifndef DEBUG_DISASSEMBLE
    printf("\n");
#endif
#endif
#ifdef DEBUG_DISASSEMBLE
    disassemble(e_from_f.pc, e_from_f.instruction,
                e_from_f.d_i);
#endif
#ifdef DEBUG_EMULATE
    emulate(reg_file, e_from_f.d_i, e_to_f->target_pc);
#endif
#endif
  }
}
```

The computation, memory access, register read, and write functions are mostly unchanged (they are all located in the execute.cpp file).

8.1.7 The Simulation and Synthesis of the IP

Experimentation

To simulate the simple_pipeline_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with simple_pipeline_ip.
You can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.



Modules & Loops	Issue Type	Slack	Iteration Latency	Interval	BRAM	FF	LUT
simple_pipeline_ip	⚠ Timing Violation	-3.73	-	-	256	3096	4373
simple_pipeline_ip_Pipeline_VITIS		-	-	34	0	8	51
simple_pipeline_ip_Pipeline_VITIS	⚠ Timing Violation	-3.73	-	-	0	2761	3983
VITIS_LOOP_47_2	! Timing Violation	-	5	5	-	-	-

Fig. 8.8 The simple_pipeline_ip synthesis report

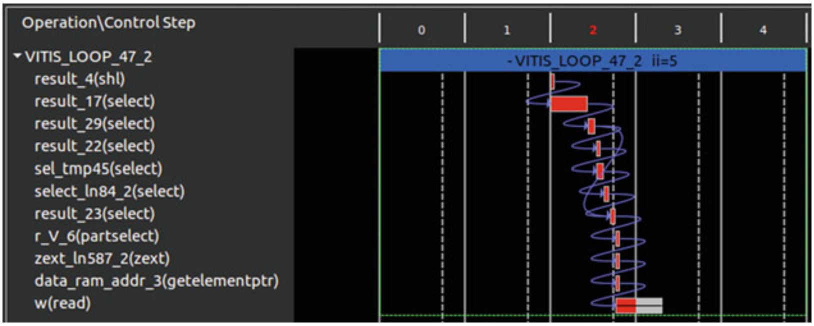


Fig. 8.9 Timing violation

All the functions are inlined (pragma HLS INLINE recursive) to optimize the size and speed.

The testbench code (in the testbench_simple_pipeline_ip.cpp file) is unchanged and the eight test files remain the same with an identical output.

Figure 8.8 shows the synthesis report. The iteration latency takes five FPGA cycles (Fig. 8.9).

There is a timing violation in cycle 2 of the iteration scheduling. It can be ignored.

8.1.8 The Vivado Project Using the IP

The Vivado project block design is shown in Fig. 8.10.

Figure 8.11 shows the Vivado implementation cost of the simple_pipeline_ip design (4720 LUTs, 8.87%).

8.1.9 The Execution of the Vivado Project on the Development Board

Experimentation

To run the simple_pipeline_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with simple_pipeline_ip.
You can play with your IP, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

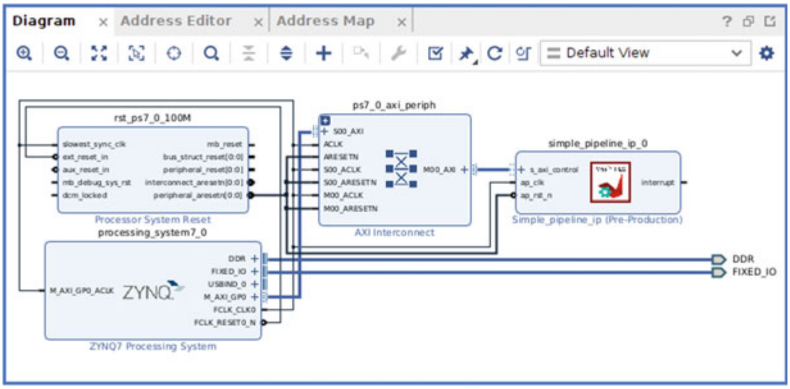
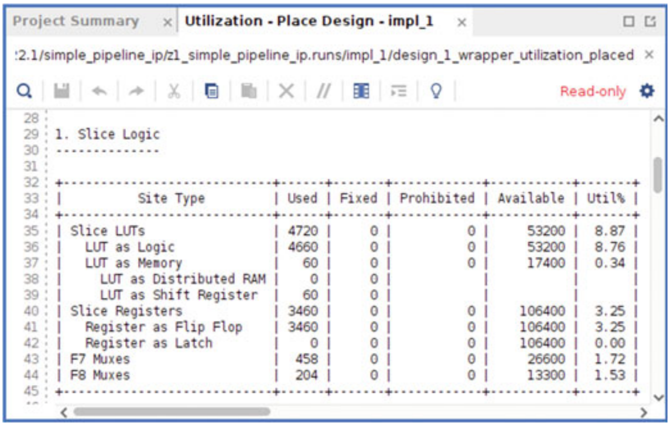


Fig. 8.10 The simple_pipeline_ip block design in Vivado



:2.1/simple_pipeline_ip/z1_simple_pipeline_ip.runs/impl_1/design_1_wrapper_utilization_placed

```

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 4720 |     0 |          0 |     53200 |  8.87 |
|   LUT as Logic             | 4660 |     0 |          0 |     53200 |  8.76 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 3460 |     0 |          0 |    106400 |  3.25 |
|   Register as Flip Flop    | 3460 |     0 |          0 |    106400 |  3.25 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  458 |     0 |          0 |     26600 |  1.72 |
| F8 Muxes                   |  204 |     0 |          0 |     13300 |  1.53 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 8.11 The simple_pipeline_ip Vivado implementation report

The helloworld.c file is shown in Listing 8.11 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script). To run another test program with the simple_pipeline_ip, you just need to update the "#include test_mem_0_text.hex" line.

Listing 8.11 The helloworld.c file to run test_mem_0_text.hex

```c
#include <stdio.h>
#include "xsimple_pipeline_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE      (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE 16
//size in words
#define DATA_RAM_SIZE      (1<<LOG_DATA_RAM_SIZE)
XSimple_pipeline_ip_Config *cfg_ptr;
XSimple_pipeline_ip         ip;
```

```
word_type code_ram[CODE_RAM_SIZE] = {
#include "test_mem_0_text.hex"
};
int main(){
  word_type w;
  cfg_ptr = XSimple_pipeline_ip_LookupConfig(
      XPAR_XSIMPLE_PIPELINE_IP_0_DEVICE_ID);
  XSimple_pipeline_ip_CfgInitialize(&ip, cfg_ptr);
  XSimple_pipeline_ip_Set_start_pc(&ip, 0);
  XSimple_pipeline_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XSimple_pipeline_ip_Start(&ip);
  while (!XSimple_pipeline_ip_IsDone(&ip));
  printf("%d fetched and decoded instructions\n",
    (int)XSimple_pipeline_ip_Get_nb_instruction(&ip));
  printf("data memory dump (non null words)\n");
  for (int i=0; i<DATA_RAM_SIZE; i++){
    XSimple_pipeline_ip_Read_data_ram_Words(&ip, i, &w, 1);
    if (w != 0)
      printf("m[%5x] = %16d (%8x)\n", 4*i, (int)w,
            (unsigned int)w);
  }
}
```

The run on the FPGA prints on the putty terminal what is shown in Listing 8.12.

Listing 8.12 The helloworld output

```
88 fetched and decoded instructions
data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
```

8.1.10 Further Testing of the Simple_pipeline_ip

It is wise to pass the riscv-tests. Some of the tests are specific to pipelines. They check that dependencies are respected.

To pass the riscv-tests on the Vitis_HLS simulator, you just need to use the testbench_riscv_tests_simple_pipeline_ip.cpp program in the riscv-tests/my_isa/my_rv32ui folder as the testbench.

To pass the riscv-tests on the FPGA, you must use the helloworld_simple_pipeline_ip.c in the riscv-tests/my_isa/my_rv32ui folder.

Normally, since you already ran the update_helloworld.sh shell script for the rv32i_npp_ip processor, you should have adapted the helloworld_simple_pipeline_ip.c paths to your environment. However if you did not, you must run the update_helloworld.sh shell script.

8.1.11 Comparison of the Non-pipelined Design with the Pipelined One

To compare the simple_pipeline_ip performance with the rv32i_npp_ip one, you must run the benchmark suite presented in the preceding chapter (mibench and riscv-tests benchmarks).

You find testbench_basicmath_simple_pipeline_ip.cpp and helloworld_simple_pipeline_ip.c files in the mibench/my_mibench/my_automotive/basicmath folder to run basicmath_small on the Vitis_HLS simulator and on the FPGA. There, you also find other testbench and helloworld programs for all the other benchmarks.

Make sure to build the hex files with the build.sh shell script if you did not run the benchmarks for the rv32i_npp_ip processor already. Also make sure to adapt the paths in the helloworld file with the update_helloworld.sh script.

Table 8.1 shows the execution time of the benchmarks as computed with Eq. 5.1 ($nmi * cpi * c$, where $c = 50$ns).

The CPI value is 1 for non control flow instructions and 2 for control flow ones (1.19 on the average for the 16 benchmarks).

The pipelined implementation runs faster (from 8% to 26%) than the baseline rv32i_npp_ip even though the CPI has increased (1.19 vs. 1.00), thanks to the division by 1.4 of the cycle duration (50 ns instead of 70 ns).

To further improve pipelining, the instruction execution can be divided into more pipeline stages.

Table 8.1 Execution time of the benchmarks on the 2-stage pipelined simple_pipeline_ip processor

suite	benchmark	Cycles	cpi	Time (s)	Baseline time (s)	Improvement (%)
mibench	basicmath	37,611,825	1.22	1.880591250	2.162841730	13
mibench	bitcount	37,909,997	1.16	1.895499850	2.285726730	17
mibench	qsort	8,080,719	1.21	0.404035950	0.467849970	14
mibench	stringsearch	634,391	1.16	0.031719550	0.038441410	17
mibench	rawcaudio	747,169	1.18	0.037358450	0.044321060	16
mibench	rawdaudio	562,799	1.20	0.028139950	0.032780930	14
mibench	crc32	330,014	1.10	0.016500700	0.021000980	21
mibench	fft	38,221,438	1.22	1.911071900	2.195578560	13
mibench	fft_inv	38,896,511	1.22	1.944825550	2.234422330	13
riscv-tests	median	35,469	1.27	0.001773450	0.001952440	9
riscv-tests	mm	193,397,241	1.23	9.669862050	11.029296180	12
riscv-tests	multiply	540,022	1.29	0.027001100	0.029252790	8
riscv-tests	qsort	322,070	1.19	0.016103500	0.019017110	15
riscv-tests	spmv	1,497,330	1.20	0.074866500	0.087230640	14
riscv-tests	towers	418,189	1.04	0.020909450	0.028266560	26
riscv-tests	vvadd	18,010	1.12	0.000900500	0.001120700	20

8.2 Second Step: Slice a Pipeline into Stages

All the source files related to the rv32i_pp_ip can be found in the rv32i_pp_ip folder.

8.2.1 A 4-Stage Pipeline

As I have already pointed out, the processing of a single instruction is done in several steps: fetch, decode, execute (which computes a memory address if the instruction is a memory access), memory access, and writeback.

The 2-stage pipeline organization can be further refined to divide the instruction processing in four phases: fetch and decode, execute, access to memory, register writeback (the fetch and decode steps are kept in the same pipeline stage as in the simple_pipeline_ip design; in Chap. 9, I will design a pipeline with separate fetch and decode stages).

If the instruction is not a memory access, nothing is done during the memory access stage, just propagating the already computed result up to the writeback stage.

Figure 8.12 shows a 4-stage pipeline: fetch and decode (f+d), execute, memory access (mem), and writeback (wb). The green horizontal rectangle is the 4-step processing of a single instruction. The red vertical rectangle is what should be inside the main loop of the top function.

8.2.2 The Inter-stage Connections

This pipeline is implemented in the rv32i_pp_ip project (pipelined IP to run the RV32I ISA).

New inter-stage links are added to connect the four stages. They correspond to the connections shown in Fig. 8.13.

The transmission types are defined in the rv32i_pp_ip.h file (see Listings 8.13 and 8.14).

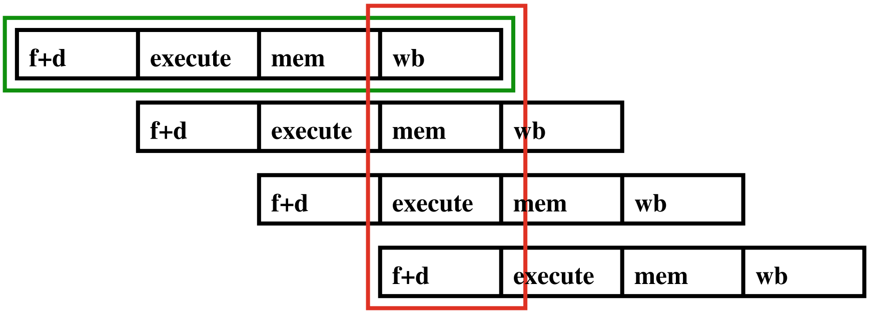


Fig. 8.12 A 4-stage pipeline

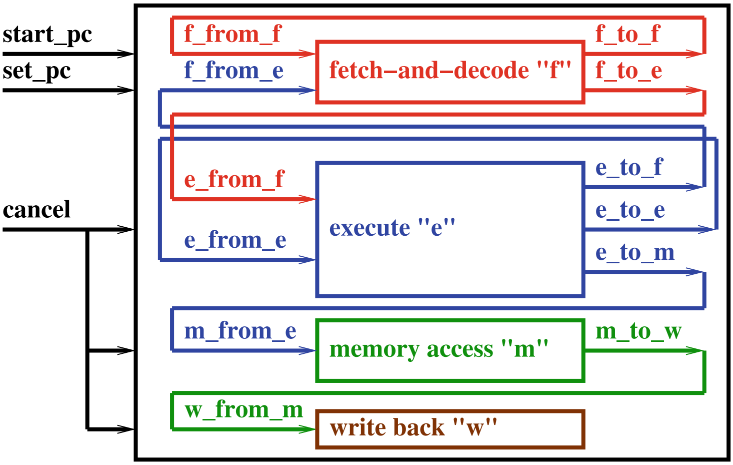


Fig. 8.13 Connections between the four stages of the pipeline

The transmissions from the f stage are unchanged.

The transmissions from the e stage to the f stage and to the e stage are also unchanged.

From e to m (see Listing 8.13), the cancel bit is the one received by the e stage and propagated to the m stage (when the e stage receives a cancel bit set from itself, it means that it will cancel its computation in the current iteration; the m stage should cancel its computation on the next iteration, hence the propagated copy of the cancel bit from e to m).

Listing 8.13 The from_e_to_m_t type

```
typedef struct from_e_to_m_s{
  bit_t                  cancel;
  int                    result;
  int                    rv2;
  decoded_instruction_t d_i;
#ifndef __SYNTHESIS__
#ifdef DEBUG_DISASSEMBLE
  code_address_t         pc;
  instruction_t          instruction;
#endif
#ifdef DEBUG_EMULATE
  code_address_t         next_pc;
#endif
#endif
} from_e_to_m_t;
```

The transmission from the m stage to the w stage (see Listing 8.14) sends the computed result (either the result received from the e stage or the one loaded in the m stage) and propagates the destination rd and the decoded bits is_ret and has_no_dest. It propagates the cancel bit received from the e stage.

Listing 8.14 The from_m_to_w_ type

```
typedef struct from_m_to_w_s{
  bit_t                 cancel;
  int                   result;
  reg_num_t             rd;
  bit_t                 is_ret;
  bit_t                 has_no_dest;
#ifndef __SYNTHESIS__
#ifdef DEBUG_DISASSEMBLE
  instruction_t         instruction;
  decoded_instruction_t d_i;
  code_address_t        pc;
#endif
#ifdef DEBUG_EMULATE
#ifndef DEBUG_DISASSEMBLE
  decoded_instruction_t d_i;
#endif
  code_address_t        next_pc;
#endif
#endif
} from_m_to_w_t;
```

8.2.3 The Decode Part of the Fetch_decode Stage

The decode and decode_immediate functions are unchanged (decode.cpp file).

The decoded_instruction_t type (rv32i_pp_ip.h file) has been extended to add a new has_no_dest bit. It is used to block the writeback to the register file. It is set when an instruction has no destination in the register file (i.e. BRANCH, STORE, or register zero). The decode_instruction function fills the new field.

8.2.4 The IP Top Function

The rv32i_pp_ip top function (rv32i_pp_ip.cpp file) is shown in Listings 8.15 and 8.16.

The rv32i_pp_ip adds a new nb_cycle argument. The cancellings imply that the pipeline does not output one instruction per cycle, hence the number of instructions run is not equal to the number of cycles of the run.

From the nb_cycle number of cycles and the nb_instruction number of instructions, I compute the IPC (nb_instruction/nb_cycle) in the testbench result (the IPC is the inverse of the CPI).

The cancel and set_pc bits are set to initialize the pipeline: all the stages are cancelled except the fetch and decode one (e_to_e.cancel set to cancel stage e, e_to_m.cancel set to cancel stage m and m_to_w.cancel set to cancel stage w).

Listing 8.15 The rv32i_pp_ip function prototype, local declarations, and initializations

```
void rv32i_pp_ip(
  unsigned int   start_pc,
  unsigned int   code_ram[CODE_RAM_SIZE],
  int            data_ram[DATA_RAM_SIZE],
  unsigned int  *nb_instruction,
  unsigned int *nb_cycle){
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=code_ram
#pragma HLS INTERFACE s_axilite port=data_ram
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=nb_cycle
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INLINE recursive
  int             reg_file[NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file dim=1 complete
  from_f_to_f_t   f_to_f, f_from_f;
  from_f_to_e_t   f_to_e, e_from_f;
  from_e_to_f_t   e_to_f, f_from_e;
  from_e_to_e_t   e_to_e, e_from_e;
  from_e_to_m_t   e_to_m, m_from_e;
  from_m_to_w_t   m_to_w, w_from_m;
  bit_t           is_running;
  unsigned int    nbi;
  unsigned int    nbc;
  for (reg_num_p1_t i=0; i<NB_REGISTER; i++) reg_file[i] = 0;
  e_to_f.target_pc = start_pc;
  e_to_f.set_pc    = 1;
  e_to_e.cancel    = 1;
  e_to_m.cancel    = 1;
  m_to_w.cancel    = 1;
  nbi              = 0;
  nbc              = 0;
  ...
```

The rv32i_pp_ip main loop (see Listing 8.16) reflects the 4-stage pipeline organization with six parallel calls: fetch_decode to fetch and decode instruction *i+3*, execute to execute instruction *i+2*, mem_access to access memory for instruction *i+1*, wb to writeback the result of instruction *i*, statistic_update to compute the number of instructions and the number of cycles of the run, and running_cond_update to update the running condition.

When the loop starts, the pipeline is empty. All of the stages except the fetch_decode one receive a cancel input bit set by the copy from the _to_ variables to the _from_ ones.

The HLS PIPELINE sets the Initiation Interval (II) to 3. Hence, the processor cycle is three FPGA cycles (30 ns, 33 Mhz).

Listing 8.16 The rv32i_pp_ip do ... while loop and return

```
  ...
  do{
#pragma HLS PIPELINE II=3
    f_from_f = f_to_f; f_from_e = e_to_f; e_from_f = f_to_e;
    e_from_e = e_to_e; m_from_e = e_to_m; w_from_m = m_to_w;
    fetch_decode(f_from_f, f_from_e, code_ram, &f_to_f,  &f_to_e);
    execute(f_to_e, e_from_f, e_from_e.cancel,
            m_from_e.cancel, m_from_e.d_i.has_no_dest,
            m_from_e.d_i.rd, m_from_e.result,
            w_from_m.cancel, w_from_m.has_no_dest,
            w_from_m.rd, w_from_m.result, reg_file,
           &e_to_f,  &e_to_e,  &e_to_m);
    mem_access(m_from_e, data_ram, &m_to_w);
    wb(w_from_m, reg_file);
    statistic_update(w_from_m.cancel, &nbi, &nbc);
    running_cond_update(w_from_m.cancel, w_from_m.is_ret, w_from_m.
        result, &is_running);
  } while (is_running);
  *nb_cycle       = nbc;
  *nb_instruction = nbi;
#ifndef __SYNTHESIS__
#ifdef DEBUG_REG_FILE
  print_reg(reg_file);
#endif
#endif
}
```

The fetch_decode.cpp and fetch.cpp files are unchanged.

8.2.5 The Bypass Mechanism in the Execute Stage

In the execute stage, the sources are read from the register file. However, two prior instructions are still in progress in the pipeline. They have computed their result but they still have not written it back to the register file. If one of the executed instruction sources is a register to be updated by one of the two preceding instructions, its result should *bypass* the register source read as illustrated in Fig. 8.14.

The blue result is the one computed in the m stage and the red result is the one to be written back in the w stage.

The mux 3->1 boxes are 3-to-1 multiplexers. Each multiplexer outputs its upper input if the m stage is not cancelled, if the instruction has a destination, and if the destination is the same as the e stage source (rs1 for the upper multiplexer, rs2 for the lower one). If not, the multiplexer outputs its middle input if the w stage is not cancelled, if the instruction has a destination, and if the destination is the same as the e stage source. If not, the multiplexer outputs its lower input (i.e. the value read from the register file).

Hence, the register sources are taken in priority order from the m stage value, from the w stage value, or from the register file read value.

In any pipeline, the priority for the bypass mechanism orders the pipeline stages from the pipeline stage next to the stage reading the register file (in this 4-stage pipeline, the execute stage reads the register file) up to the writeback stage.

Unfortunately, the bypass mechanism degrades the critical path as it inserts a 3-to-1 multiplexer between the register file read and the ALU.

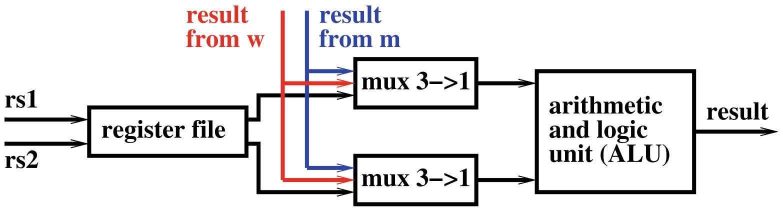


Fig. 8.14 Bypassing with in-flight sources

Moreover, the ALU input depends on the values coming from the m and w stages. The value transmitted by the w stage is not in the critical path because it is set at the cycle start and not modified by the writeback (the register file access time is longer than the value transmission from the w stage).

But for the m stage, the value to transmit is loaded from memory if the instruction is a load. The memory access time is much longer than the register file access time. So, for a load, the m stage value transmission is in the critical path.

To limit the increase of the critical path to a 3-to-1 multiplexer, what I transfer from the m stage to the multiplexers upper input is not the value computed in the m stage but only the one received by the m stage and propagated from the e stage (hence, bypassing applies to the e stage ALU computed values, not to the m stage memory loads; for LOAD instructions, there is a special processing explained in Sect. 8.2.7).

The execute function calls the get_source function, which calls the bypass function.

The bypass function in the execute.cpp file (see Listing 8.17) selects which source is the most up-to-date between the mem_result and the wb_result. The mem_result value sent by the m stage has priority over the wb_result sent by the w stage (more precisely, the bypass function returns the mem_result value if the m stage is bypassing, otherwise it returns the wb_result value).

Listing 8.17 The bypass function

```
static int bypass(
  bit_t     m_bp,
  int       mem_result,
  int       wb_result){
  if (m_bp) return mem_result;
  else      return wb_result;
}
```

The get_source function in the execute.cpp file (see Listing 8.18) computes bypass conditions for both rs1 and rs2 sources (bypass_rs1 and bypass_rs2).

Each condition is set when the bypass applies either for the m or w stage.

For example, the bypass applies for the rs1 source of the instruction *i* processed in stage m if the following four conditions are all true: stage m is not cancelled, the instruction *i* has a destination, the rs1 register source of instruction *i* is not register zero, and the instruction *j* processed in stage m has register rs1 as its destination. In this case, the Boolean variable m_bp_1 is set.

The Boolean variable w_bp_1 is set if the same four conditions are true, replacing the m stage with the w stage.

The Boolean variables m_bp_2 and w_bp_2 are set in the same conditions, replacing the rs1 register with the the rs2 register.

When a bypass condition is set (bypass_rs1 or bypass_rs2), the bypass function is called to choose between the mem_result and the wb_result values. Otherwise, the source is set to the value read from the register file and received as an argument (r1 for rs1 and r2 for rs2).

Listing 8.18 The get_source function

```
static void get_source(
  int           r1,
  int           r2,
  from_f_to_e_t e_from_f,
  bit_t         m_cancel,
  bit_t         m_has_no_dest,
  reg_num_t     m_rd,
  int           m_result,
  bit_t         w_cancel,
  bit_t         w_has_no_dest,
  reg_num_t     w_rd,
  int           w_result,
  int          *rv1,
  int          *rv2){
  bit_t     m_bp_1, m_bp_2, w_bp_1, w_bp_2;
  bit_t     bypass_rs1, bypass_rs2;
  reg_num_t rs1, rs2;
  rs1 = e_from_f.d_i.rs1;
  rs2 = e_from_f.d_i.rs2;
  m_bp_1 = (!m_cancel && !m_has_no_dest &&
            rs1!=0    && rs1==m_rd);
  w_bp_1 = (!w_cancel && !w_has_no_dest &&
            rs1!=0    && rs1==w_rd);
  bypass_rs1 = m_bp_1 || w_bp_1;
  m_bp_2 = (!m_cancel && !m_has_no_dest &&
            rs2!=0    && rs2==m_rd);
  w_bp_2 = (!w_cancel && !w_has_no_dest &&
            rs2!=0    && rs2==w_rd);
  bypass_rs2 = m_bp_2 || w_bp_2;
  *rv1 = (bypass_rs1)?bypass(m_bp_1, m_result, w_result):r1;
  *rv2 = (bypass_rs2)?bypass(m_bp_2, m_result, w_result):r2;
}
```

8.2.6 The Execute Stage

8.2.6.1 The Execute Function

The execute function is shown in Listings 8.19, 8.20, 8.23, and 8.24 (execute.cpp file).

The function reads the register file (see Listing 8.19) to get the rs1 and rs2 values (read_reg) and calls get_source to update these values if bypassing applies.

Listing 8.19 The execute function to get the sources

```
void execute(
  from_f_to_e_t  f_to_e,
  from_f_to_e_t  e_from_f,
  bit_t          e_cancel,
  bit_t          m_cancel,
  bit_t          m_has_no_dest,
  reg_num_t      m_rd,
  int            m_result,
  bit_t          w_cancel,
  bit_t          w_has_no_dest,
  reg_num_t      w_rd,
  int            w_result,
  int           *reg_file,
  from_e_to_f_t *e_to_f,
  from_e_to_e_t *e_to_e,
  from_e_to_m_t *e_to_m){
  int         r1, r2, rv1, rv2, rs;
  int         c_op_result, c_result, result;
  reg_num_t rs1, rs2;
  bit_t       bcond, taken_branch, load_delay, is_rs1_reg, is_rs2_reg
      ;
  opcode_t  opcode;
  rs1 = e_from_f.d_i.rs1;
  rs2 = e_from_f.d_i.rs2;
  r1  = read_reg(reg_file, rs1);
  r2  = read_reg(reg_file, rs2);
  get_source(r1, r2,
             e_from_f,
             m_cancel,
             m_has_no_dest,
             m_rd,
             m_result,
             w_cancel,
             w_has_no_dest,
             w_rd,
             w_result,
            &rv1, &rv2);
  ...
```

Once the sources are known, the execute function computes (see Listing 8.20) the branch condition (unchanged compute_branch_result function), the operation result (unchanged compute_op_result function), the instruction typed result (compute_result function) and the next pc (unchanged compute_next_pc function).

Listing 8.20 The execute function to compute

```
...
bcond = compute_branch_result(rv1, rv2, e_from_f.d_i.func3);
taken_branch = e_from_f.d_i.is_branch && bcond;
rs = (e_from_f.d_i.is_r_type)?
     rv2:(int)e_from_f.d_i.imm;
c_op_result = compute_op_result(e_from_f.d_i, rv1, rs);
c_result    = compute_result(e_from_f.pc, e_from_f.d_i, rv1);
result      = (e_from_f.d_i.is_r_type ||
               e_from_f.d_i.is_op_imm)?
               c_op_result:c_result;
e_to_f->target_pc =
         compute_next_pc(e_from_f.pc, e_from_f.d_i, rv1,
                         bcond);
...
```

8.2.6.2 The Compute_result Function

The computation functions have been grouped in a new file (compute.cpp).

The compute_result function has been modified to become independent from the compute_op_result function (see Listing 8.21). With this changing, the two functions are run in parallel.

Listing 8.21 The compute_result function

```
int compute_result(
  code_address_t        pc,
  decoded_instruction_t d_i,
  int                   rv1){
  int            imm12 = ((int)d_i.imm)<<12;
  code_address_t pc4   = pc<<2;
  code_address_t npc4  = pc4 + 4;
  int            result;
  switch(d_i.type){
    case R_TYPE:
      //computed by compute_op_result()
      result = 0;
      break;
    case I_TYPE:
      if (d_i.is_jalr)
        result = (unsigned int)npc4;
      else if (d_i.is_load)
        result = rv1 + (int)d_i.imm;
      else if (d_i.is_op_imm)
        //computed by compute_op_result()
        result = 0;
      else
        result = 0;//(opcode == SYSTEM)
      break;
    case S_TYPE:
      result = rv1 + (int)d_i.imm;
      break;
    case B_TYPE:
      //computed by compute_branch_result()
      result = 0;
      break;
    case U_TYPE:
      if (d_i.is_lui)
        result = imm12;
      else//d_i.opcode == AUIPC
        result = pc4 + imm12;
      break;
    case J_TYPE:
      result = (unsigned int)npc4;
      break;
    default:
      result = 0;
      break;
  }
  return result;
}
```

8.2.6.3 The Load_delay Bit

The load_delay bit is set in the execute function (see Listing 8.23) if the executed instruction is a non cancelled LOAD and the next instruction in the fetch and decode stage uses the loaded value as a source (e.g. f_to_e.d_i.rs1 is the destination register of the load e_from_f.d_i.rd).

For example, if the code to be run contains the instructions shown in Listing 8.22, the load_delay bit is set when the lw instruction is in the execute stage.

Listing 8.22 A loaded value used by a back-to-back instruction

```
          ...
          lw     a1,0(a2)
          addi   a1,a1,1
          ...
```

If the load_delay bit is set, the current fetch must be cancelled as explained in Sect. 8.2.7. The target_pc is set as the current fetch pc (i.e. the instruction next to the load should be refetched).

Listing 8.23 The execute function to set the load_delay bit

```
  ...
  opcode      = f_to_e.d_i.opcode;
  is_rs1_reg  = ((opcode != JAL)     && (opcode != LUI)    &&
                 (opcode != AUIPC)   && (f_to_e.d_i.rs1 != 0));
  is_rs2_reg  = ((opcode != OP_IMM)  && (opcode != LOAD)   &&
                 (opcode != JAL)     && (opcode != JALR)   &&
                 (opcode != LUI)     && (opcode != AUIPC)  &&
                 (f_to_e.d_i.rs2 != 0));
  load_delay  = !e_cancel   && e_from_f.d_i.is_load &&
               ((is_rs1_reg && (e_from_f.d_i.rd == f_to_e.d_i.rs1))
                    ||
                (is_rs2_reg && (e_from_f.d_i.rd == f_to_e.d_i.rs2)));
  if (load_delay) e_to_f->target_pc = e_from_f.pc + 1;
  ...
```

The set_pc bit is set (see Listing 8.24) if the instruction is a non cancelled jump, a taken branch, or if the load_delay bit is set.

The execute stage sends the set_pc bit to the f stage.

The execute stage sends a cancel bit to itself (the cancel bit is a copy of the set_pc one).

The cancel bit sent to itself is also sent to the m stage.

The execute stage sends its result to the m stage (the transmitted result is the target_pc if the instruction is a RET, i.e. the return address).

The execute stage sends rv2 (the stored value if the instruction is a STORE; it is the value after a potential bypass) and d_i (which contains rd to be propagated to the w stage, func3 which is the memory access size, and other decoded bits used by the m and w stages).

Other values (the instruction, its pc, and the computed target_pc) are sent for debugging purpose (they are not included in synthesis mode).

Listing 8.24 The execute function to set the transmitted values

```
  ...
  e_to_f->set_pc =(e_from_f.d_i.is_jalr ||
                   e_from_f.d_i.is_jal  ||
                   taken_branch         ||
                   load_delay)          &&
                  !e_cancel;
  e_to_e->cancel     = e_to_f->set_pc;
  e_to_m->cancel     = e_cancel;
  e_to_m->result     =(e_from_f.d_i.is_ret)?
                   (int)e_to_f->target_pc:result;
  e_to_m->rv2        = rv2;
  e_to_m->d_i        = e_from_f.d_i;
#ifndef __SYNTHESIS__
```

```
#ifdef DEBUG_DISASSEMBLE
  e_to_m->pc           = e_from_f.pc;
  e_to_m->instruction = e_from_f.instruction;
#endif
#ifdef DEBUG_EMULATE
  e_to_m->next_pc      = e_to_f->target_pc;
#endif
#endif
}
```

8.2.7 Memory Load Hazards

If a memory load is followed by a computation involving the loaded value as in the piece of code in Listing 8.22, the bypass technique is not sufficient as, when the memory access stage ends, the next instruction execute stage ends too, as depicted on the left part of Fig. 8.15 (the value returned by the mem stage should simultaneously be the input source of the instruction processed by the execute stage).

The easiest way to solve this problem is to let the programmer (or the compiler) avoid it by inserting any independent instruction between the load and the instruction using the loaded value.

Instead of implementing any bypass unit, I could have applied the same policy for dependent computing instructions. However, the general bypass situation is more frequent than the load dependency one and the compiler would have to insert two independent instructions instead of just one.

More generally, if there are *s* stages from register read to register writeback, including these two, the compiler must insert *s-1* independent instructions between the writing one and the next reading one to avoid the bypass mechanism for RAW dependencies.

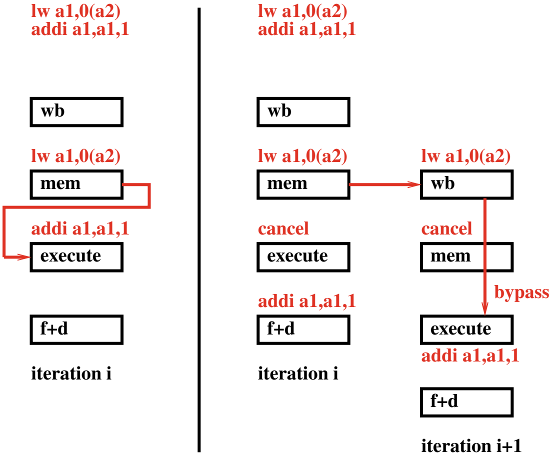


Fig. 8.15 Separating a load instruction from an instruction using the loaded value

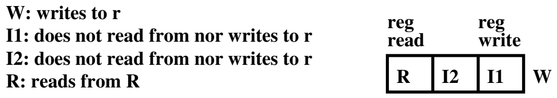

W: writes to r
I1: does not read from nor writes to r
I2: does not read from nor writes to r
R: reads from R
reg read
reg write
R
I2
I1
W

Fig. 8.16 No bypass is needed with the insertion of two independent instructions

Figure 8.16 shows an example where $s = 3$. There are four instructions: W, I1, I2 and R. The W instruction has register *r* as its destination. The R instruction has *r* as its source. No bypass is necessary if W and R are separated by at least two instructions (I1 and I2) not accessing to register R.

However, this lazy solution is usually inapplicable because of the compiler. The Gnu RISC-V cross compiler does not provide any option to handle delays between instructions (there is an option only for branch delays).

It is still possible to manipulate the assembly code produced by the compiler and insert NOP instructions where they are needed (a NOP is a neutral No-OPeration instruction, leaving the processor state unchanged).

This manipulation has to be done on the assembly code, before the production of the hexadecimal code, to keep correct code references (the computed displacements between label definitions and usages).

However, such a post-processing cannot be applied to the linked libraries, as they are already compiled.

For loads, another solution is to extend cancellation to load dependent computations.

On the left part of Fig. 8.15, the "addi a1,a1,1" instruction increments the value loaded by the preceding "lw a1,0(a2)" instruction. However, when the addi instruction is executed, the lw instruction is still loading. I could forward the value out of the load to the execute stage but this would serialize the mem stage after the execute stage. Such a serialization would ruin the benefits of pipelining.

On the right part of Fig. 8.15, the first fetch of the "addi a1,a1,1" instruction is cancelled, introducing in the pipeline an equivalent of a NOP instruction. The same addi instruction is refetched in iteration *i* (the set_pc bit is set if a use after load dependency is detected; the target_pc is set to refetch the use after load instruction: see the last line in Listing 8.23). In iteration *i+1*, the value in the wb stage (which is the one out of the iteration *i* mem stage) bypasses the register file value in the execute stage.

8.2.8 The Memory Access Stage

The mem stage (mem.cpp file) is composed of the mem_load, the mem_store, and the mem_access functions. The mem_load and the mem_store functions are unchanged.

The mem_access function in the mem.cpp file (see Listing 8.25) is slightly modified to add the case of cancellations (m_from_e.cancel). If the instruction in the mem stage is not a load, its result is not modified. Otherwise, it receives the loaded value. The mem_access function fills the m_to_w fields to either transmit the result of the load or propagate what was computed in the execute function.

Listing 8.25 The mem_access function

```
void mem_access(
  from_e_to_m_t   m_from_e,
  int            *data_ram,
  from_m_to_w_t  *m_to_w){
  b_data_address_t address;
  address          = m_from_e.result;
  m_to_w->cancel   = m_from_e.cancel;
  if (!m_from_e.cancel){
    m_to_w->result       =
     (m_from_e.d_i.is_load)                                 ?
        mem_load(data_ram, address, m_from_e.d_i.func3):
        m_from_e.result;
    if (m_from_e.d_i.is_store)
      mem_store(data_ram, address, m_from_e.rv2,
               (ap_uint<2>)m_from_e.d_i.func3);
  }
  m_to_w->rd            = m_from_e.d_i.rd;
  m_to_w->is_ret        = m_from_e.d_i.is_ret;
  m_to_w->has_no_dest   = m_from_e.d_i.has_no_dest;
#ifndef __SYNTHESIS__
#ifdef DEBUG_DISASSEMBLE
  m_to_w->pc            = m_from_e.pc;
  m_to_w->instruction   = m_from_e.instruction;
  m_to_w->d_i           = m_from_e.d_i;
#endif
#ifdef DEBUG_EMULATE
#ifndef DEBUG_DISASSEMBLE
  m_to_w->d_i           = m_from_e.d_i;
#endif
  m_to_w->next_pc       = m_from_e.next_pc;
#endif
#endif
}
```

8.2.9 The Writeback Stage

The wb function (see Listing 8.26) is in the wb.cpp file.

The wb stage is where the debugging facilities have been gathered. The debugging messages are all placed in the same single stage. It gives the same output as the rv32i_npp_ip, with no troublesome interleaving and no useless details like cancelled instructions.

Listing 8.26 The wb function

```
void wb(
  from_m_to_w_t   w_from_m,
  int            *reg_file){
  if (!w_from_m.cancel){
    if (!w_from_m.has_no_dest)
      reg_file[w_from_m.rd] = w_from_m.result;
#ifndef __SYNTHESIS__
#ifdef DEBUG_FETCH
    printf("%04d: %08x      ",
          (int)(w_from_m.pc<<2), w_from_m.instruction);
#ifndef DEBUG_DISASSEMBLE
    printf("\n");
#endif
#endif
#ifdef DEBUG_DISASSEMBLE
    disassemble(w_from_m.pc, w_from_m.instruction,
                w_from_m.d_i);
#endif
#ifdef DEBUG_EMULATE
    emulate(reg_file, w_from_m.d_i, w_from_m.next_pc);
#endif
#endif
  }
}
```

8.2.10 The Testbench Function

Experimentation

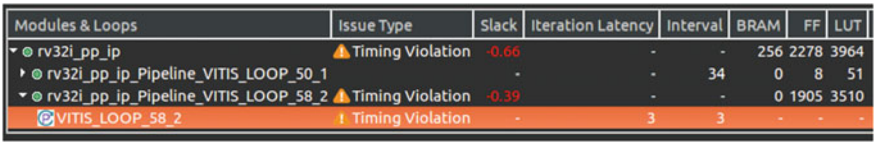
To simulate the rv32i_pp_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with rv32i_pp_ip.

You can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

The main function in the testbench_test_mem_rv32i_pp_ip.cpp file adds the new nbc argument to compute the number of cycles (see Listing 8.27).

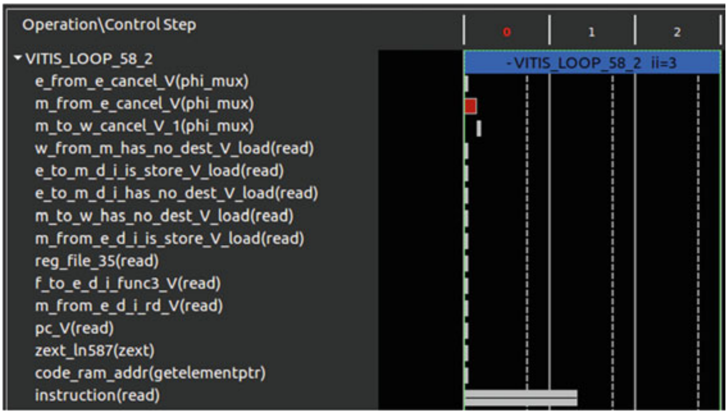
Listing 8.27 The testbench_test_mem_rv32i_pp_ip.cpp file

```
#include <stdio.h>
#include "rv32i_pp_ip.h"
int          data_ram[DATA_RAM_SIZE];
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_mem_0_text.hex"
};
int main() {
  unsigned int  nbi;
  unsigned int  nbc;
  int           w;
  rv32i_pp_ip(0, code_ram, data_ram, &nbi, &nbc);
  printf("%d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", nbi, nbc, ((float)nbi)/nbc);
  printf("data memory dump (non null words)\n");
  for (int i=0; i<DATA_RAM_SIZE; i++){
    w = data_ram[i];
    if (w != 0)
      printf("m[%5x] = %16d (%8x)\n", 4*i, w, (unsigned int)w);
  }
  return 0;
}
```



Modules & Loops	Issue Type	Slack	Iteration Latency	Interval	BRAM	FF	LUT
rv32i_pp_ip	⚠ Timing Violation	-0.66	-	-	256	2278	3964
rv32i_pp_ip_Pipeline_VITIS_LOOP_50_1		-	-	34	0	8	51
rv32i_pp_ip_Pipeline_VITIS_LOOP_58_2	⚠ Timing Violation	-0.39	-	-	0	1905	3510
VITIS_LOOP_58_2	⚠ Timing Violation	-	3	3	-	-	-

Fig. 8.17 The rv32i_pp_ip synthesis report



Operation\Control Step
VITIS_LOOP_58_2
e_from_e_cancel_V(phi_mux)
m_from_e_cancel_V(phi_mux)
m_to_w_cancel_V_1(phi_mux)
w_from_m_has_no_dest_V_load(read)
e_to_m_d_i_is_store_V_load(read)
e_to_m_d_i_has_no_dest_V_load(read)
m_to_w_has_no_dest_V_load(read)
m_from_e_d_i_is_store_V_load(read)
reg_file_35(read)
f_to_e_d_i_func3_V(read)
m_from_e_d_i_rd_V(read)
pc_V(read)
zext_ln587(zext)
code_ram_addr(getelementptr)
instruction(read)

Fig. 8.18 The rv32i_pp_ip main loop schedule

8.2.11 The IP Synthesis

Figure 8.17 shows the synthesis report.

Figure 8.18 shows the rv32i_pp_ip schedule of the main loop. The iteration latency is three FPGA cycles. The instruction fetch operation (instruction(read) on the bottom of the figure) spans cycles 0 and 1.

8.2.12 The Vivado Project

Figure 8.19 shows the Vivado project block design.

Figure 8.20 shows the Vivado implementation cost of the rv32i_pp_ip design (4,334 LUTs, 8.15%).

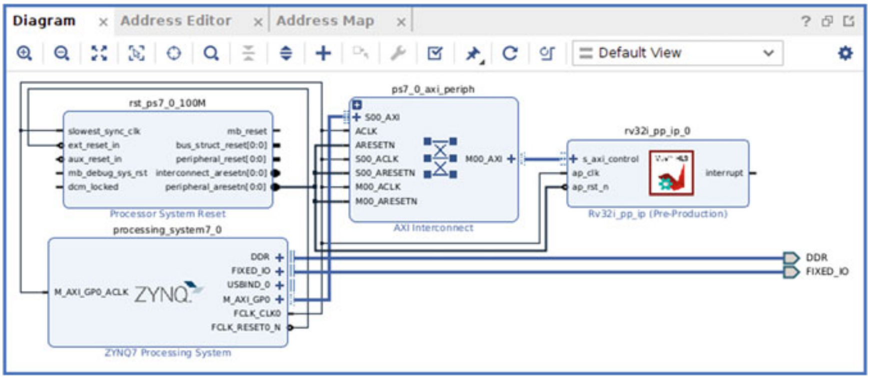
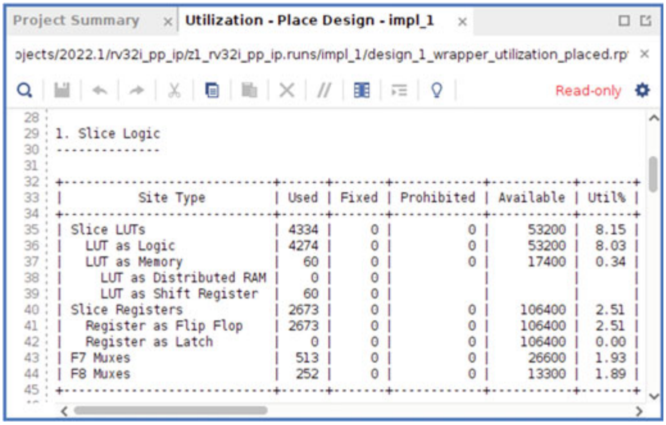


Fig. 8.19 The rv32i_pp_ip block design in Vivado


Project Summary
Utilization - Place Design - impl_1
ojects/2022.1/rv32i_pp_ip/z1_rv32i_pp_ip.runs/impl_1/design_1_wrapper_utilization_placed.rp
Read-only

```

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 4334 |     0 |          0 |     53200 |  8.15 |
|   LUT as Logic             | 4274 |     0 |          0 |     53200 |  8.03 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 2673 |     0 |          0 |    106400 |  2.51 |
|   Register as Flip Flop    | 2673 |     0 |          0 |    106400 |  2.51 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  513 |     0 |          0 |     26600 |  1.93 |
| F8 Muxes                   |  252 |     0 |          0 |     13300 |  1.89 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 8.20 The rv32i_pp_ip Vivado implementation report

8.2.13 The Execution of the Vivado Project on the Development Board

Experimentation

To run the rv32i_pp_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with rv32i_pp_ip.
You can play with your IP, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

The code in the helloworld.c file is shown in Listing 8.28 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script). The code_ram array initialization concerns the test_mem program.

Listing 8.28 The helloworld.c file

```
#include <stdio.h>
#include "xrv32i_pp_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE 16
//size in words
#define DATA_RAM_SIZE     (1<<LOG_DATA_RAM_SIZE)
XRv32i_pp_ip_Config *cfg_ptr;
XRv32i_pp_ip         ip;
word_type code_ram[CODE_RAM_SIZE] = {
#include "test_mem_0_text.hex"
};
int main(){
  unsigned int nbi, nbc;
  word_type    w;
  cfg_ptr = XRv32i_pp_ip_LookupConfig(XPAR_XRV32I_PP_IP_0_DEVICE_ID
      );
  XRv32i_pp_ip_CfgInitialize(&ip, cfg_ptr);
  XRv32i_pp_ip_Set_start_pc(&ip, 0);
  XRv32i_pp_ip_Write_code_ram_Words(&ip, 0, code_ram, CODE_RAM_SIZE
      );
  XRv32i_pp_ip_Start(&ip);
  while (!XRv32i_pp_ip_IsDone(&ip));
  nbi = XRv32i_pp_ip_Get_nb_instruction(&ip);
  nbc = XRv32i_pp_ip_Get_nb_cycle(&ip);
  printf("%d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", nbi, nbc, ((float)nbi)/nbc);
  printf("data memory dump (non null words)\n");
  for (int i=0; i<DATA_RAM_SIZE; i++){
    XRv32i_pp_ip_Read_data_ram_Words(&ip, i, &w, 1);
    if (w != 0)
      printf("m[%4x] = %16d (%8x)\n", 4*i, (int)w,
             (unsigned int)w);
  }
}
```

If the RISC-V code in the test_mem.h file is run, it produces the output shown in Listing 8.29.

Listing 8.29 The helloworld output

```
88 fetched and decoded instructions in 119 cycles (ipc = 0.74)
data memory dump (non null words)
m[   0] =                1 (       1)
m[   4] =                2 (       2)
m[   8] =                3 (       3)
m[   c] =                4 (       4)
m[  10] =                5 (       5)
m[  14] =                6 (       6)
m[  18] =                7 (       7)
m[  1c] =                8 (       8)
m[  20] =                9 (       9)
m[  24] =               10 (       a)
m[  2c] =               55 (      37)
```

Table 8.2 Execution time of the benchmarks on the 4-stage pipelined rv32i_pp_ip processor

suite	benchmark	Cycles	cpi	Time (s)	2-stage time (s)	Improvement (%)
mibench	basicmath	37,643,157	1.22	1.129294710	1.880591250	40
mibench	bitcount	37,909,999	1.16	1.137299970	1.895499850	40
mibench	qsort	8,086,191	1.21	0.242585730	0.404035950	40
mibench	stringsearch	635,830	1.16	0.019074900	0.031719550	40
mibench	rawcaudio	758,171	1.20	0.022745130	0.037358450	39
mibench	rawdaudio	562,801	1.20	0.016884030	0.028139950	40
mibench	crc32	360,016	1.20	0.010800480	0.016500700	35
mibench	fft	38,233,594	1.22	1.147007820	1.911071900	40
mibench	fft_inv	38,908,668	1.22	1.167260040	1.944825550	40
riscv-tests	median	35,471	1.27	0.001064130	0.001773450	40
riscv-tests	mm	193,405,475	1.23	5.802164250	9.669862050	40
riscv-tests	multiply	540,024	1.29	0.016200720	0.027001100	40
riscv-tests	qsort	330,630	1.22	0.009918900	0.016103500	38
riscv-tests	spmv	1,497,940	1.20	0.044938200	0.074866500	40
riscv-tests	towers	426,385	1.06	0.012791550	0.020909450	39
riscv-tests	vvadd	18,012	1.13	0.000540360	0.000900500	40

8.2.14 Further Testing of the Rv32i_pp_ip

To pass the riscv-tests on the Vitis_HLS simulator, you just need to use the testbench_riscv_tests_rv32i_pp_ip.cpp program in the riscv-tests/my_isa/my_rv32ui folder as the testbench.

To pass the riscv-tests on the FPGA, you must use the helloworld_rv32i_pp_ip.c in the riscv-tests/my_isa/my_rv32ui folder. Normally, since you already ran the update_helloworld.sh shell script for the other processors, the helloworld_rv32i_pp_ip.c file should have paths adapted to your environment. However if you did not, you must run the update_helloworld.sh shell script.

8.3 The Comparison of the 2-Stage Pipeline with the 4-Stage One

To run a benchmark from the mibench suite, say my_dir/bench, you set the testbench as the testbench_bench_rv32i_pp_ip.cpp file found in the mibench/my_mibench/my_dir/bench folder. For example, to run basicmath, you set the testbench as testbench_basicmath_rv32i_pp_ip.cpp in mibench/my_mibench/my_automotive/basicmath.

To run one of the official riscv-tests benchmarks, say bench, you set the testbench as the testbench_bench_rv32i_pp_ip.cpp file found in the riscv-tests/benchmarks/bench folder. For example, to run median, you set the testbench as testbench_median_rv32i_pp_ip.cpp in riscv-tests/benchmarks/median.

To run the same benchmarks on the FPGA, select helloworld_rv32i_pp_ip.c to run in Vitis IDE.

Table 8.2 shows the execution time of the benchmarks as computed with Eq. 5.1 ($nmi * cpi * c$, where c = 30ns).

Even though the pipeline is deeper, the average CPI is nearly unchanged (1.20 vs. 1.19) and the reduction of the cycle time from 50ns to 30ns ensures a 40% reduction of the execution time.

Compared to rv32i_npp_ip, the improvement is more than 50% and the size of the IP is less than 6% bigger (4,334 LUTs instead of 4,091). Pipelining is a must have in modern processor designs as a pipelined processor is faster at almost no cost in die area.

Building a RISC-V Processor with a Multicycle Pipeline

9

Abstract

This chapter will make you build your third RISC-V processor. The implemented microarchitecture proposed in this third version takes care of dependencies by blocking an instruction in the pipeline until the instructions it depends on are all out of the pipeline. For this purpose, a new issue stage is added. Moreover, the pipeline stages are organized to allow an instruction to stay multiple cycles in the same stage. The instruction processing is divided into six steps in order to further reduce the processor cycle to two FPGA cycles (i.e. 50Mhz): fetch, decode, issue, execute, memory access, and writeback. This multicycle pipeline microarchitecture is useful when the operators have different latencies, like multicycle arithmetic or memory accesses.

9.1 The Difference Between a Pipeline and a Multicycle Pipeline

All the source files related to the multicycle_pipeline_ip can be found in the multicycle_pipeline_ip folder.

9.1.1 The Wait Signal to Freeze Stages

The pipelined organization implemented in the preceding chapter has an important drawback. The instructions flow along the pipeline and an instruction may not stay for more than one cycle in a pipeline stage.

A consequence is that all the computations have to take the same execution delay (i.e. all computing their result in one cycle in the execute stage). This is not suited to an implementation of the RISC-V F or D floating-point extensions. Even the M extension which adds an integer division is a problem.

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_9

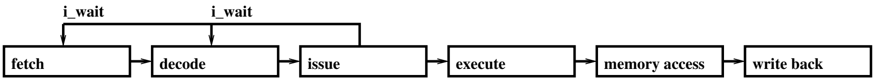


Fig. 9.1 The i_wait signal sent by the issue stage when a source is not ready

Another consequence is that an instruction cannot wait for its sources before entering the execution stage: they must all be ready when the instruction starts execution. I had to add a bypass mechanism to the pipeline to forward computed but not yet written values. This bypass hardware impacts the critical path. With no bypass, the execute stage can fit in the two FPGA cycle limit. With bypassing, the execute stage requires three FPGA cycles.

In the multicycle pipeline organization, when an instruction must stay in the same stage for multiple cycles, it sends a wait signal to the preceding stages as shown in Fig. 9.1.

Each stage receives some input (e.g. the stage *y* receives the structure x_to_y from the stage *x*), processes it and sends an output (e.g. the structure y_to_z is sent to the stage *z*). However, if the stage has a wait input signal, it stays frozen, which means that it does not change its output at all (it does not receive nor process its input and it keeps its output unchanged).

A stage waits until the wait signal it inputs is cleared (e.g. the issue stage is able to issue and sends a null wait signal to the fetch and decode stages).

When a waiting stage receives a cleared wait signal, it resumes and starts processing its input.

9.1.2 The Valid Input and Output Bits

The inputs and the outputs of a pipeline stage use the same structures as the ones described in the preceding chapter (see Sect. 8.1.2).

However, to allow waits from multicycle stages, each stage input and output structure contains a valid bit (see Fig. 9.2). While a multicycle stage is in progress, its output is invalid. The valid bit is set once the final result is ready and the wait condition is cleared.

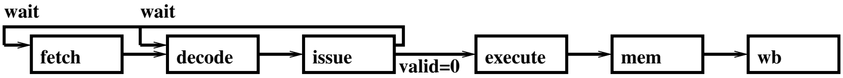


Fig. 9.2 A waiting issue operation invalidates its output until the wait is cleared

9.1.3 Fetching and Computing the Next PC

To keep the processor cycle within the limit of two FPGA cycles, the fetch stage cannot additionally decode the fetched instruction. The read operation from the code memory, the filling of the decoded instruction structure and the computation of the immediate value according to the instruction format are too much work for two FPGA cycles, i.e. 20ns. This is one of the two reasons why the rv32i_pp_ip processor cycle had to be 30ns (the other reason is the impact of the bypass on the execute stage delay).

Although the decoding can be moved to a dedicated decode stage, the computation of the next pc cannot. If there is no valid pc output from the fetch stage to itself, the next cycle is not able to fetch.

A sequential next pc can be computed in the fetch stage, as it can be done in parallel with the fetch. But if the fetched instruction is a taken jump or branch, the sequential pc is not correct.

Instead of cancelling an incorrect path as I did in the preceding chapter, the multicycle pipeline does some decoding in the fetch stage to block the fetch operation each time the next pc keeps unknown (i.e. when the fetched instruction is a BRANCH, a JAL or a JALR).

In case of a JAL, the next pc is known in the decode stage. In case of a BRANCH or a JALR, the next pc remains unknown until the target has been computed in the execute stage. The fetch stage keeps idle until it receives a valid input next pc.

If neither of the three cases occurs, BRANCH, JAL, or JALR, the fetch stage sends a valid next pc to itself. This next pc is pc + 1.

Figure 9.3 shows how the fetch stage input is set according to the incoming valid bits of the structures produced by itself, by the decode stage, or by the execute stage.

The fetch stage remains idle until a valid bit is present on either one of the three inputs. Each time a BRANCH or a JALR instruction is fetched, the fetch stage stays idle for three cycles (the time to move the control instruction to the execute stage where the target pc is computed and sent back to the fetch stage with the valid bit set). Each time a JAL is fetched, the fetch stage stays idle for one cycle.

Control instructions represent between 10% and 20% of the instructions fetched during a run, and most of them are branches. Hence, the impact on the performance is rather high and the microarchitecture built in the next chapter gives a way to fill the idle cycles with useful work.

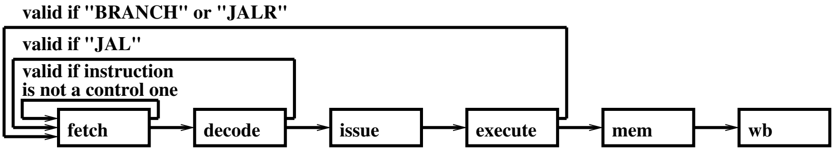

valid if "BRANCH" or "JALR"
valid if "JAL"
valid if instruction is not a control one
fetch
decode
issue
execute
mem
wb

Fig. 9.3 Setting the fetch stage pc

9.1.4 The Safe Structure of a Multicycle Stage

When a multicycle stage s takes more than one cycle to produce its output from its stable input, it raises a wait signal and sends it to its predecessors. It also clears the output valid bit sent to its successor.

In the same cycle, the prior stage computes, and at the end of the cycle sends a valid output.

On the following cycle, stage s receives the valid input but it should not use it while it is still computing on the preceding input.

Such multicycle stages have an internal safe structure (see Fig. 9.4). This structure serves to store the input. Instead of computing directly on the input, the stage computes on the saved data.

When there is a valid input and the stage is no longer in a waiting situation, the new input is saved in the stage safe.

When the stage finishes its processing, it empties its safe.

During a wait, a valid input is not saved. It keeps in stand-by until the wait is over. As the emitting stage is frozen, the input is preserved and stays stable until the stage resumes.

9.1.5 Multiple Multicycle Stages

If the pipeline contains multiple multicycle stages (e.g. a multicycle execute stage implementing the M or the F extensions of the RISC-V ISA and a multicycle memory access stage to access a hierarchized memory), each multicycle stage has a safe and emits a wait signal to its predecessors as shown in Fig. 9.5.

When a stage s' raises a wait signal while a sooner stage s has already signalled its own wait (e.g. e_wait signal is raised while i_wait signal is set), the new wait freezes the pipeline including stage s. However, stage s can continue its processing from its stable input in the safe.

If stage s finishes its processing before stage s', its output valid bit is set but the stage keeps frozen until stage s' clears its wait signal. Stages before s remain frozen too.

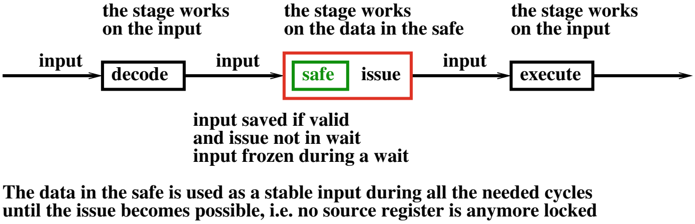
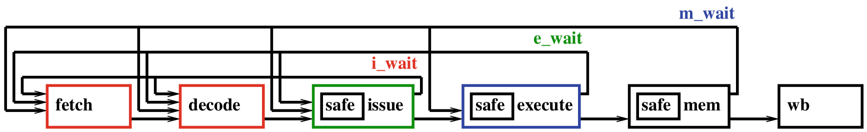

the stage works
on the input
the stage works
on the data in the safe
the stage works
on the input
input
decode
input
safe
issue
input
execute
input saved if valid
and issue not in wait
input frozen during a wait
The data in the safe is used as a stable input during all the needed cycles
until the issue becomes possible, i.e. no source register is anymore locked

Fig. 9.4 A safe in a multicycle stage



The red stages are frozen while the i_wait signal is set
The red and green stages are frozen while the e_wait signal is set
The red, green, and blue stages are frozen while the m_wait signal is set

If the i_wait signal is cleared while either the e_wait or the m_wait signal is set, the red stages keep frozen
If the i_wait signal is cleared while both the e_wait and the m_wait signals are clear, the red stages are resumed

Fig. 9.5 Multiple wait signals

If stage s and s' finish their processing in the same cycle, their outputs are simultaneously valid and the whole pipeline gets active again.

If stage s finishes its processing after s', the pipeline between s and s' resumes when the s' wait bit is cleared, but the pipeline until s remains frozen while the s wait bit keeps set. Hence, the stage after s receives an invalid input until s ends its processing. As the stage next to s has an invalid input, it does not process anything and it sends an invalid output.

9.2 The IP Top Function

The multicycle_pipeline_ip function is located in the multicycle_pipeline_ip.cpp file.

The prototype of the multicycle IP top function is the same as the rv32i_pp_ip one (see the prototype part in the code presented in Sect. 8.1.3).

The local declarations are shown in Listing 9.1. They include the stage interconnections, the i_wait signal, and the i_safe structure used by the issue stage.

The is_reg_computed array has been added. It is used to lock registers while they are computed in the pipeline. There is a 1-bit entry per register in the register file.

When an instruction is issued, its destination register r is locked, i.e. the is_reg_computed[r] bit is set. When the instruction enters the writeback stage, the is_reg_computed[r] bit is cleared.

Listing 9.1 The multicycle_pipeline_ip function local declarations

```
  ...
  int           reg_file       [NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file        dim=1 complete
  bit_t         is_reg_computed[NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=is_reg_computed dim=1 complete
  from_f_to_f_t f_to_f;
  from_f_to_d_t f_to_d;
  from_d_to_f_t d_to_f;
  from_d_to_i_t d_to_i;
  bit_t         i_wait;
  i_safe_t      i_safe;
  from_i_to_e_t i_to_e;
  from_e_to_f_t e_to_f;
  from_e_to_m_t e_to_m;
  from_m_to_w_t m_to_w;
  bit_t         is_running;
  counter_t     nbi;
  counter_t     nbc;
  ...
```

The initialization phase (see Listing 9.2) starts with a call to the init_reg_file function which initializes the register file (all registers are cleared) and the register locking bits in the is_reg_computed array (all registers are marked as unlocked).

Then, the connection links are all marked as invalid (is_valid bit is cleared) except for the f_to_f one. The f_to_f.next_pc receives the start_pc address. The issue stage is not in wait (i_wait cleared) and its safe is empty.

Listing 9.2 The multicycle_pipeline_ip function initializations

```
  ...
  init_reg_file (reg_file, is_reg_computed);
  f_to_f.is_valid  = 1;
  f_to_f.next_pc   = start_pc;
  f_to_d.is_valid  = 0;
  d_to_f.is_valid  = 0;
  d_to_i.is_valid  = 0;
  i_to_e.is_valid  = 0;
  e_to_f.is_valid  = 0;
  e_to_m.is_valid  = 0;
  m_to_w.is_valid  = 0;
  i_wait           = 0;
  i_safe.is_full   = 0;
  nbi              = 0;
  nbc              = 0;
  ...
```

The multicycle_pipeline_ip.cpp file contains the main do ... while loop shown in Listing 9.3.

The function calls are placed in a reverse order to avoid RAW dependencies (i.e. from writeback to fetch).

Listing 9.3 The do ... while loop in the multicycle_pipeline_ip function

```
  ...
  do{
#pragma HLS PIPELINE II=2
#pragma HLS LATENCY max=1
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("=============================================\n");
    printf("cycle %d\n", (int)nbc);
#endif
#endif
    statistic_update(i_to_e, &nbi, &nbc);
    running_cond_update(m_to_w, &is_running);
    write_back(m_to_w, reg_file, is_reg_computed);
    mem_access(e_to_m, data_ram, &m_to_w);
    execute(i_to_e,
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
            reg_file,
#endif
#endif
           &e_to_f, &e_to_m);
    issue(d_to_i, reg_file, is_reg_computed, &i_safe, &i_to_e, &
        i_wait);
    decode(f_to_d, i_wait, &d_to_f, &d_to_i);
    fetch(f_to_f, d_to_f, e_to_f, i_wait, code_ram, &f_to_f, &
        f_to_d);
  } while (is_running);
  ...
```

The "HLS LATENCY max=1" pragma instructs the synthesizer to try to use at most one separation register to implement the succession of operations, i.e. constraints the iteration timing to at most two FPGA cycles.

As a general rule, you should define "HLS LATENCY max=*n*" if you want your implementation to have a latency of at most *n+1* cycles.

If the "HLS LATENCY max=1" pragma is disactivated (e.g. the line is turned into a comment), the synthesizer expands the iteration processing on three FPGA cycles instead of two (however, it keeps the IP cycle as two FPGA cycles; hence there is a one cycle overlapping for two successive iterations). With an active "HLS LATENCY max=1" pragma, the iteration latency and the IP cycle coincide (II=2).

The fetch, decode, issue, execute, mem_access, and write_back functions implement the six pipeline stages.

They are organized and placed to be independent from each other and to be run in parallel.

There are some RAW dependencies though. This is the case for the retroaction link between the decode stage and the fetch stage (d_to_f), which is written by the decode function and read by the fetch function. It is the same situation for e_to_f (the execute function writes to the e_to_f structure and the fetch function reads from it) and also for i_wait (written by issue and read by decode and fetch).

These RAW dependencies do serialize some part of the computations. But it turns out that the synthesizer can fit everything in the do ... while loop into the two FPGA cycles limit, probably because these dependencies are not in the critical path.

The statistic_update and the running_cond_update functions play the same role as in the rv32i_pp_ip design.

The DEBUG_PIPELINE definition can be used to switch between a cycle by cycle dump of the processor work (at each cycle, the fetch stage prints which instruction it fetches, the decode stage prints what it decodes, and so on for each of the six pipeline stages) or the already implemented execution trace (i.e. the same output as previously implemented).

The constant is to be defined in the debug_multicycle_pipeline_ip.h file to set the cycle by cycle dump.

Listings 9.14 to 9.19 show what the print looks like when the DEBUG_PIPELINE constant is defined.

The end part of the top function after the do...while loop is unchanged from the rv32i_pp_ip one.

9.3 The Pipeline Stages

Each stage is built on the same pattern. A stage only works when a valid input is present and no wait signal is received. The computation is done in a stage_job function. The stage produces its output in a set_output_to_ function (one function per output recipient).

While the stage is waiting, the output is unchanged. When there is no input, the output is cleared.

9.3.1 The Fetch Stage

The code of the fetch function (in the fetch.cpp file) is shown in Listing 9.4.

The fetch function returns (i.e. the fetch stage stays frozen) if the i_wait condition is set (i.e. the issue stage is stopped by a locked register).

The fetch function sets the has_input bit if any pc emitting stage has sent a valid output at the end of the preceding cycle (the pc emitting stages are the fetch stage, the decode stage, or the execute stage).

When a control instruction is fetched, the fetch stage does not output any valid next_pc value to itself and the fetch is suspended. However, the fetch stage outputs the fetched instruction to the decode stage.

If the control instruction is a JAL, the decode stage in the next cycle sends the target_pc value back to the fetch stage, which resumes.

If the control instruction is a BRANCH or a JALR, the decode stage in the next cycle is not able to send a valid target_pc value. Hence, the fetch stage remains suspended. When the BRANCH or JALR target is computed, the execute stage sends the target_pc value and the fetch stage resumes.

A consequence is that at most one of the three valid bits from the stages emitting a pc can be set.

If there is no input (has_input is clear), the stage stays idle. It clears its output valid bits.

If there is an input, the stage does its job in the stage_job function (fetch and partial decoding).

The stage sets the outputs to itself (next sequential pc) and to the decode stage (current pc and fetched instruction).

The stage sets the output valid bits (the output to itself is valid only if the instruction is not a control one).

Listing 9.4 The fetch function

```
void fetch(
  from_f_to_f_t  f_from_f,
  from_d_to_f_t  f_from_d,
  from_e_to_f_t  f_from_e,
  bit_t          i_wait,
  instruction_t *code_ram,
  from_f_to_f_t *f_to_f,
  from_f_to_d_t *f_to_d){
  bit_t             has_input;
  instruction_t     instruction;
  decoded_control_t d_ctrl;
  bit_t             is_ctrl;
  code_address_t    pc;
  if (!i_wait){
    has_input = f_from_f.is_valid || f_from_d.is_valid || f_from_e.
        is_valid;
    if (has_input){
      if (f_from_f.is_valid)
        pc = f_from_f.next_pc;
      else if (f_from_d.is_valid)
        pc = f_from_d.target_pc;
      else if (f_from_e.is_valid)
        pc = f_from_e.target_pc;
      stage_job(pc, code_ram, &instruction, &d_ctrl);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
      printf("fetched  ");
      printf("%04d: %08x      \n",
        (int)(pc<<2), instruction);
#endif
#endif
      set_output_to_f(pc, f_to_f);
      set_output_to_d(pc, instruction, d_ctrl, f_to_d);
    }
    is_ctrl = d_ctrl.is_branch || d_ctrl.is_jalr ||
              d_ctrl.is_jal;
    f_to_f->is_valid = has_input && !is_ctrl;
    f_to_d->is_valid = has_input;
  }
}
```

The stage_job function (see Listing 9.5) fetches an instruction from the code_ram addressed by the input pc.

It does not decode the fetched instruction. However, it checks if the instruction is a control one (the decode_control function).

The stage_job function and the decode_control function are in the fetch.cpp file.

Listing 9.5 The decode_control and stage_job functions

```
static void decode_control(
  instruction_t      instruction,
  decoded_control_t *d_ctrl){
  opcode_t opcode;
  opcode = (instruction >> 2);
  d_ctrl->is_branch = (opcode == BRANCH);
  d_ctrl->is_jalr   = (opcode == JALR);
  d_ctrl->is_jal    = (opcode == JAL);
}
static void stage_job(
  code_address_t     pc,
  unsigned int      *code_ram,
  instruction_t     *instruction,
  decoded_control_t *d_ctrl){
  *instruction = code_ram[pc];
  decode_control(*instruction, d_ctrl);
}
```

The two set_output functions (see Listing 9.6; fetch.cpp file) fill the fields of the structures for the next decode stage and for the fetch stage itself. The output to the decode stage includes the bits computed by the decode_control function to avoid recomputing them.

Listing 9.6 The set_output_to_f and set_output_to_d functions

```
static void set_output_to_f(
  code_address_t pc,
  from_f_to_f_t *f_to_f){
```

```
  f_to_f->next_pc = pc + 1;
}
static void set_output_to_d(
  code_address_t    pc,
  instruction_t     instruction,
  decoded_control_t d_ctrl,
  from_f_to_d_t    *f_to_d){
  f_to_d->pc          = pc;
  f_to_d->instruction = instruction;
  f_to_d->is_branch   = d_ctrl.is_branch;
  f_to_d->is_jalr     = d_ctrl.is_jalr;
  f_to_d->is_jal      = d_ctrl.is_jal;
}
```

9.3.2 The Decode Stage

The code to implement the decode stage has the same organization as the one implementing the fetch stage.

Two bits have been added to the decoded_instruction_t type defined in the multicycle_pipeline_ip.cpp file (see Listing 9.7): is_reg_rs1 and is_reg_rs2. The is_reg_rs1 bit (respectively is_reg_rs2) is set if the decoded rs1 field (respectively rs2) represents a register source.

Listing 9.7 The decode_instruction_t type

```
typedef struct decoded_instruction_s{
  opcode_t     opcode;
  ...
  immediate_t imm;
  bit_t        is_rs1_reg;
  bit_t        is_rs2_reg;
  ...
  bit_t        is_r_type;
} decoded_instruction_t;
```

The decode_instruction function defined in the decode.cpp file (see Listings 9.8 and 9.9) is updated to take the already decoded bits concerning control instructions into account (is_branch, is_jalr, and is_jal).

The computation has also been reorganized to minimize the redundancy in the expressions, inserting many local bits (e.g. is_lui or is_not_auipc).

Listing 9.8 The decode_instruction function: compute local bits

```
static void decode_instruction(
  instruction_t          instruction,
  bit_t                  is_branch,
  bit_t                  is_jalr,
  bit_t                  is_jal,
  decoded_instruction_t *d_i){
  opcode_t opcode;
  bit_t    is_lui;
  bit_t    is_load;
  bit_t    is_store;
  bit_t    is_op_imm;
  bit_t    is_not_auipc;
  bit_t    is_not_jal;
  opcode        = (instruction >> 2);
  is_lui        = (opcode == LUI);
  is_load       = (opcode == LOAD);
  is_store      = (opcode == STORE);
  is_op_imm     = (opcode == OP_IMM);
  is_not_auipc  = (opcode != AUIPC);
  is_not_jal    = !is_jal;
...
```

The decode_instruction function is also updated to fill the two new is_rs1_reg and is_rs2_reg fields.

Listing 9.9 The decode_instruction function: fill the d_i fields

```
  ...
  d_i->opcode      =  opcode;
  ...
  d_i->is_rs1_reg  = (is_not_jal    && !is_lui       &&
                      is_not_auipc  && (d_i->rs1 !=  0));
  d_i->is_rs2_reg  = (!is_op_imm    && !is_load      &&
                      is_not_jal    && !is_jalr      &&
                      !is_lui       && is_not_auipc  &&
                      (d_i->rs2 !=  0));
  ...
  d_i->is_r_type   = (d_i->type    == R_TYPE);
}
```

The decoding of the immediate (decode.cpp file) is unchanged (see 5.4.5).

The decode function in the decode.cpp file (see Listing 9.10) returns if the i_wait condition is set (i.e. the decode stage stays frozen).

If there is no valid input, the output valid bits are cleared.

If there is a valid input from the fetch stage, the received instruction is decoded in the stage_job function. The outputs to the fetch and issue stages are filled in the two set_output functions. The output valid bits are set (the output to the fetch stage is set if the decoded instruction is a JAL).

Listing 9.10 The decode function

```
void decode(
  from_f_to_d_t  d_from_f,
  bit_t          i_wait,
  from_d_to_f_t *d_to_f,
  from_d_to_i_t *d_to_i){
  decoded_instruction_t d_i;
  code_address_t        target_pc;
  if (!i_wait){
    if (d_from_f.is_valid){
      stage_job(d_from_f.pc, d_from_f.instruction,
                d_from_f.is_branch, d_from_f.is_jalr,
                d_from_f.is_jal, &d_i, &target_pc);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
      printf("decoded  %04d: ", (int)(d_from_f.pc<<2));
      disassemble(d_from_f.pc, d_from_f.instruction, d_i);
#endif
#endif
      set_output_to_f(target_pc, d_to_f);
      set_output_to_i(d_from_f.pc, d_i,
#ifndef __SYNTHESIS__
                      d_from_f.instruction, target_pc,
#endif
                      d_to_i);
    }
    d_to_f->is_valid = d_from_f.is_valid && d_i.is_jal;
    d_to_i->is_valid = d_from_f.is_valid;
  }
}
```

The stage_job function in the decode.cpp file (see Listing 9.11) decodes the instruction, decodes the immediate, and computes the JAL target_pc.

Listing 9.11 The decode stage_job function

```
static void stage_job(
  code_address_t         pc,
  instruction_t          instruction,
  bit_t                  is_branch,
  bit_t                  is_jalr,
  bit_t                  is_jal,
  decoded_instruction_t *d_i,
  code_address_t        *target_pc){
  decode_instruction(instruction, is_branch, is_jalr, is_jal,
                     d_i);
  decode_immediate  (instruction, d_i);
  if (d_i->is_jal)
    *target_pc = pc+(code_address_t)(d_i->imm >> 1);
}
```

The set_output_to_f and set_output_to_i functions in the decode.cpp file fill the output structures d_to_f and d_to_i (see Listing 9.12).

Listing 9.12 The set_output_to_f and set_output_to_i functions

```
static void set_output_to_f(
  code_address_t target_pc,
  from_d_to_f_t *d_to_f){
  d_to_f->target_pc = target_pc;
}
static void set_output_to_i(
  code_address_t        pc,
  decoded_instruction_t d_i,
#ifndef __SYNTHESIS__
  instruction_t         instruction,
  code_address_t        target_pc,
#endif
  from_d_to_i_t        *d_to_i){
  d_to_i->pc          = pc;
  d_to_i->d_i         = d_i;
#ifndef __SYNTHESIS__
  d_to_i->instruction = instruction;
  d_to_i->target_pc   = target_pc;
#endif
}
```

9.3.3 The Issue Stage

The job of the issue stage is to read the register sources from the register file and send the values to the execute stage. The stage issues the instruction to the next stage only when no register source of the current instruction is being computed at the same time in the following stages of the pipeline (a source register *r* is being computed if is_reg_computed[*r*] is set; the set bit locks the register).

9.3.3.1 Instruction Scheduling in the Issue Stage

In a strict scheduling of the instructions, an instruction cannot be issued if its destination is locked. This might seem surprising. The RISC-V piece of code shown in Listing 9.13 illustrates why it can lead to incorrect execution to issue an instruction with a locked destination.

Listing 9.13 A RISC-V piece of code to illustrate that the issue should be blocked if the destination is locked

```
0000        li      a0, 18
0004        li      a0, 19
0008        addi    a0, a0, 1
0012        ret
```

The schedule shown in Listings 9.14 to 9.16 (printed from the run of the RISC-V code in the multicycle_pipeline_ip with the DEBUG_PIPELINE constant set) shows how the three instructions move in the pipeline when the destination is not checked before issue.

At cycle 2 (see Listing 9.14), register a0 is locked by instruction (0000) (the destination register is locked when the instruction is issued).

Listing 9.14 The schedule when the destination is not checked before issue: cycles 0 to 2

```
=============================================
cycle 0
fetched   0000: 01200513
=============================================
cycle 1
decoded   0000: li a0, 18
fetched   0004: 01300513
=============================================
cycle 2
issued    0000
decoded   0004: li a0, 19
fetched   0008: 00150513
```

Register a0 is the destination of instruction (0004). At cycle 3 (see Listing 9.15), even though register a0 is locked since cycle 2, as the lock on the destination register is not checked, it does not block the issue of instruction (0004) (a side effect is that register a0 is locked again).

At cycle 5, register a0 is unlocked by instruction (0000) (the destination register is unlocked when the instruction is written back).

At cycle 5, instruction (0008) is issued before a0 has been updated by instruction (0004) which is still in the mem stage. Hence, instruction (0008) reads 18 in a0, as it is being written by instruction (0000) (the read in the issue stage follows the write in the writeback stage as the issue function is called after the write_back one in the do ... while loop).

Listing 9.15 The schedule when the destination is not checked before issue: cycles 3 to 5

```
============================================
cycle 3
execute  0000
issued   0004
decoded  0008: addi a0, a0, 1
fetched  0012: 00008067
============================================
cycle 4
mem      0000
execute  0004
============================================
cycle 5
wb       0000
      a0  =               18 (      12)
mem      0004
issued   0008
decoded  0012: ret
```

At cycle 8 (see Listing 9.16), instruction (0008) writes value 19 into register a0 (i.e. 18 + 1) and this is the final printed register value.

Listing 9.16 The schedule when the destination is not checked before issue: cycles 6 to 9

```
============================================
cycle 6
wb       0004
      a0  =               19 (      13)
execute  0008
issued   0012
============================================
cycle 7
mem      0008
execute  0012
      pc  =                0 (       0)
============================================
cycle 8
wb       0008
      a0  =               19 (      13)
mem      0012
============================================
cycle 9
wb       0012
============================================
```

9.3.3.2 Instruction Scheduling in the Issue Stage: Strict Scheduling

This scheduling is to be compared to the one applied when the destination is checked, shown in Listings 9.17 to 9.19. The first three cycles are unchanged.

At cycle 3 (see Listing 9.17), instruction (0004) is not issued because its destination a0 is locked.

Instruction (0004) is issued at cycle 5, after register a0 is unlocked by the writeback of instruction (0000) in the same cycle (is_reg_computed[a0] is cleared in the writeback stage before it is read in the issue stage).

Listing 9.17 The schedule when the destination is checked before issue: cycles 3 to 5

```
=============================================
cycle 3
execute  0000
=============================================
cycle 4
mem      0000
=============================================
cycle 5
wb       0000
      a0  =               18 (        12)
issued   0004
decoded  0008: addi a0, a0, 1
fetched  0012: 00008067
```

Instruction (0008) is issued at cycle 8 (see Listing 9.18) after instruction (0004) writeback in the same cycle. Hence, instruction (0008) has read the value 19 in register a0.

Instruction (0008) computes value 20 in the execute stage at cycle 9.

Listing 9.18 The schedule when the destination is checked before issue: cycles 6 to 9

```
=============================================
cycle 6
execute  0004
=============================================
cycle 7
mem      0004
=============================================
cycle 8
wb       0004
      a0  =               19 (        13)
issued   0008
decoded  0012: ret
=============================================
cycle 9
execute  0008
issued   0012
```

Instruction (0008) writes value 20 to register a0 at cycle 11 (see Listing 9.19).

Listing 9.19 The schedule when the destination is checked before issue: cycles 10 to 12

```
=============================================
cycle 10
mem      0008
execute  0012
      pc  =                0 (         0)
=============================================
cycle 11
wb       0008
      a0  =               20 (        14)
mem      0012
=============================================
cycle 12
wb       0012
=============================================
```

9.3.3.3 Instruction Scheduling in the Issue Stage: Relaxed Scheduling

However, checking the destination register lock is only necessary when the code contains two back-to-back register initializations like in Listing 9.13, which should never occur in an optimized code (in an optimized code, the useless first initialization should be removed).

For example, the check is unnecessary for a succession of a load instruction followed by a computation on the loaded register (e.g. “lw a0,0(a1)” followed by “addi a0,a0,1”). In this case, the second instruction is not issued until the load has ended because of the RAW dependency on register *a0*: the lw instruction writes to *a0* which the addi instruction reads.

To save LUTs/FFs resources, the multicycle_pipeline_ip design applies a relaxed instruction scheduling policy in which the destination register locking is not checked.

9.3.3.4 The Issue Function

The code of the issue function (issue.cpp file) is shown in Listing 9.20.

If the issue stage is not waiting (i.e. if i_wait is not set) and if the safe is empty, the input is saved in the safe (save_input_from_d function). The safe is full if the input is valid.

If the safe is full (either because it has just been filled or because the issue stage is waiting), the is_reg_computed lock of each source of the instruction in the safe is checked: bits is_locked_1 (for register source rs1) and is_locked_2 (for register source rs2) are computed.

The stage issues and outputs to the execute stage only if the sources are not locked.

From the two lock check bits, a new i_wait value is set.

If the issue is possible (i.e. the new value for i_wait is 0), the sources are read in the register file by the stage_job function. The read values are copied in the output structure to be sent to the execute stage (the set_output_to_e function). The destination register rd (if any) is locked (is_reg_computed[i_safe->d_i.rd] bit set).

The output to the execute stage is valid only when there is an instruction in the safe and the stage is not waiting. Then, the safe is emptied (i.e. the is_full bit is cleared).

Listing 9.20 The issue function

```
void issue(
  from_d_to_i_t  i_from_d,
  int           *reg_file,
  bit_t         *is_reg_computed,
  i_safe_t      *i_safe,
  from_i_to_e_t *i_to_e,
  bit_t         *i_wait){
  bit_t is_locked_1;
  bit_t is_locked_2;
  int   rv1;
  int   rv2;
  if (!(*i_wait) && !i_safe->is_full){
    save_input_from_d(i_from_d, i_safe);
    i_safe->is_full = i_from_d.is_valid;
  }
  if (i_safe->is_full){
    is_locked_1 =
      i_safe->d_i.is_rs1_reg &&
      is_reg_computed[i_safe->d_i.rs1];
    is_locked_2 =
      i_safe->d_i.is_rs2_reg &&
      is_reg_computed[i_safe->d_i.rs2];
    *i_wait = is_locked_1 || is_locked_2;
    if (!(*i_wait)){
      stage_job(i_safe->d_i, reg_file, &rv1, &rv2);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
      printf("issued    ");
      printf("%04d\n", (int)(i_safe->pc<<2));
#endif
#endif
      set_output_to_e(i_safe->pc, i_safe->d_i, rv1, rv2,
#ifndef __SYNTHESIS__
                      i_safe->instruction,
                      i_safe->target_pc,
#endif
                      i_to_e);
      if (!i_safe->d_i.has_no_dest)
        is_reg_computed[i_safe->d_i.rd] = 1;
    }
  }
  i_to_e->is_valid = i_safe->is_full && !(*i_wait);
  i_safe->is_full  = (*i_wait);
}
```

However, if you want to apply strict scheduling, the issue function should be updated by adding a third check bit is_locked_d related to the instruction destination i_safe->d_i.rd, as shown in Listing 9.21.

Listing 9.21 The issue function

```
void issue(
  from_d_to_i_t  i_from_d,
  int           *reg_file,
  bit_t         *is_reg_computed,
  i_safe_t      *i_safe,
  from_i_to_e_t *i_to_e,
  bit_t         *i_wait){
  bit_t is_locked_1;
  bit_t is_locked_2;
  bit_t is_locked_d;
  ...
  if (i_safe->is_full){
    is_locked_1 =
      i_safe->d_i.is_rs1_reg &&
      is_reg_computed[i_safe->d_i.rs1];
    is_locked_2 =
      i_safe->d_i.is_rs2_reg &&
      is_reg_computed[i_safe->d_i.rs2];
    is_locked_d =
     !i_safe->d_i.has_no_dest &&
      is_reg_computed[i_safe->d_i.rd];
    *i_wait = is_locked_1 || is_locked_2 || is_locked_d;
  ...
}
```

9.3.3.5 The Stage_job Function

The stage_job function in the issue.cpp file (see Listing 9.22) reads the sources from the register file.

Listing 9.22 The issue stage_job function

```
static void stage_job(
  decoded_instruction_t d_i,
  int                  *reg_file,
  int                  *rv1,
  int                  *rv2){
  *rv1 = reg_file[d_i.rs1];
  *rv2 = reg_file[d_i.rs2];
}
```

9.3.3.6 The Set_output_to_e Function

The set_output_to_e function in the issue.cpp file (see Listing 9.23) fills the i_to_e structure with the source values rv1 and rv2, the decoding d_i, and pc (the pc is needed in the execute stage to compute the target of a BRANCH or a JALR instruction).

Listing 9.23 The set_output_to_e function

```
static void set_output_to_e(
  code_address_t        pc,
  decoded_instruction_t d_i,
  int                   rv1,
  int                   rv2,
#ifndef __SYNTHESIS__
  instruction_t         instruction,
  code_address_t        target_pc,
#endif
  from_i_to_e_t        *i_to_e){
  i_to_e->pc          = pc;
  i_to_e->d_i         = d_i;
  i_to_e->rv1         = rv1;
  i_to_e->rv2         = rv2;
#ifndef __SYNTHESIS__
  i_to_e->instruction = instruction;
  i_to_e->target_pc   = target_pc;
#endif
}
```

9.3.4 The Execute Stage

The execute stage is not concerned by the i_wait signal (like all the stages after the issue one).

9.3.4.1 The Execute Function

The code of the execute function (in the execute.cpp file) is shown in Listing 9.24.

If a valid input is received from the issue stage, the execute stage computes the BRANCH condition, the ALU result, the memory access address, and the BRANCH and JALR next_pc (the compute function groups all the computations).

Then, it sets the target_pc to be sent to the fetch stage (stage_job function). After that, it fills the output structure fields in the two set_output functions. Eventually, the execute stage sets the valid bits for the outputs.

If no input is valid, the outputs to the fetch stage and to the memory access stage are invalid too. Otherwise, the output to the memory access stage is valid. The output to the fetch stage is valid if the executed instruction is a BRANCH or a JALR.

If the instruction is a RET (a RET is a JALR with register RA as the source and register zero as the destination) and if register RA is null, it is the return from the main function, hence the last instruction run. In this case, the output to the fetch stage is invalid and the return address is not sent.

Listing 9.24 The execute function

```
void execute(
  from_i_to_e_t  e_from_i,
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
  int             *reg_file,
#endif
#endif
  from_e_to_f_t  *e_to_f,
  from_e_to_m_t  *e_to_m){
  bit_t           bcond;
  int             result1;
  int             result2;
  code_address_t  target_pc;
  code_address_t  next_pc;
  if (e_from_i.is_valid){
    compute  (e_from_i.pc, e_from_i.d_i, e_from_i.rv1,
              e_from_i.rv2, &bcond, &result1, &result2,
             &next_pc);
    stage_job(e_from_i.pc, e_from_i.d_i, bcond, next_pc,
             &target_pc);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("execute  ");
    printf("%04d\n", (int)(e_from_i.pc<<2));
    if (e_from_i.d_i.is_branch || e_from_i.d_i.is_jalr)
      emulate(reg_file, e_from_i.d_i, next_pc);
#endif
#endif
    set_output_to_f(target_pc, e_to_f);
    set_output_to_m(e_from_i.d_i, result1, result2, next_pc,
                    e_from_i.rv2, target_pc,
#ifndef __SYNTHESIS__
                    e_from_i.pc, e_from_i.instruction,
#endif
                    e_to_m);
  }
  //block fetch after last RET
  //(i.e. RET with 0 return address)
  e_to_f->is_valid   =
    e_from_i.is_valid      &&
   (e_from_i.d_i.is_branch ||
   (e_from_i.d_i.is_jalr   &&
  (!e_from_i.d_i.is_ret    || (next_pc != 0))));
  e_to_m->is_valid = e_from_i.is_valid;
}
```

9.3.4.2 The Compute Function

The compute function in the execute.cpp file (see Listing 9.25) computes in parallel the branch condition bcond (compute_branch_result), the result1 ALU result (compute_op_result), the result2 result (the result returned by the compute_result function, i.e. a memory access address, a LUI immediate, or an AUIPC sum), and the next_pc for BRANCH and JALR (compute_next_pc).

These computation functions are unchanged (see 8.2.6 for the compute_result function and 5.5.7 for the other computation functions). They are located in the compute.cpp file.

Listing 9.25 The compute function

```
static void compute(
  code_address_t         pc,
  decoded_instruction_t d_i,
  int                    rv1,
  int                    rv2,
  bit_t                 *bcond,
  int                   *result1,
  int                   *result2,
  code_address_t        *next_pc){
  *bcond   = compute_branch_result(rv1, rv2, d_i.func3);
  *result1 = compute_op_result(rv1, rv2, d_i);
  *result2 = compute_result(rv1, pc, d_i);
  *next_pc = compute_next_pc(pc, d_i, rv1);
}
```

9.3.4.3 The Stage_job Function

The stage_job function in the execute.cpp file (see Listing 9.26) sets the BRANCH and JALR instructions target_pc from the computed next_pc and bcond branch condition.

For a BRANCH, the target is next_pc if bcond is set. Otherwise, the target is the sequential pc (i.e. pc+1 for an untaken branch).

For a JALR instruction, the target is next_pc.

Listing 9.26 The stage_job function

```
static void stage_job(
  code_address_t         pc,
  decoded_instruction_t d_i,
  bit_t                  bcond,
  code_address_t         next_pc,
  code_address_t        *target_pc){
  *target_pc =
   (bcond || d_i.is_jalr)?next_pc:(code_address_t)(pc + 1);
}
```

9.3.4.4 The Set_output Functions

The set_output functions in the execute.cpp file set the execute stage output structures.

The set_output_to_f function is shown in Listing 9.27.

Listing 9.27 The set_output_to_f function

```
static void set_output_to_f(
  code_address_t target_pc,
  from_e_to_f_t *e_to_f){
  e_to_f->target_pc = target_pc;
}
```

The set_output_to_m function is shown in Listing 9.28.

The value field can be set by different sources, depending on the major opcode. For a RET instruction, the value is the target_pc, i.e. the return address. For a JAL and a JALR instruction (except RET ones), the value is result2, i.e. the link address computed in the compute_result function. The value is also result2 if the instruction has an upper immediate (LUI and AUIPC).

The value field is set with rv2 if the instruction is a STORE (i.e. the value to be stored).

For all the other instructions (i.e. OP or OP_IMM opcodes), the value is result1 computed in the compute_op_result function. LOAD instructions also set value to result1 but this is irrelevant as loads overwrite the value field with the loaded value.

The destination of the output is the memory access stage which is not only the stage where the memory accesses are done. It is also a transit stage for all the instructions between the execute stage and the writeback stage. For this reason, the execute stage sends everything needed either by the memory access stage or by the writeback stage (this is why the value field holds a computation result for any opcode).

Listing 9.28 The set_output_to_m function

```
static void set_output_to_m(
  decoded_instruction_t d_i,
  int                   result1,
  int                   result2,
  code_address_t        next_pc,
  int                   rv2,
  code_address_t        target_pc,
#ifndef __SYNTHESIS__
  code_address_t        pc,
  instruction_t         instruction,
#endif
  from_e_to_m_t        *e_to_m){
  e_to_m->rd            = d_i.rd;
  e_to_m->has_no_dest   = d_i.has_no_dest;
  e_to_m->is_load       = d_i.is_load;
  e_to_m->is_store      = d_i.is_store;
  e_to_m->func3         = d_i.func3;
  //e_to_m->is_ret is used by running_cond_update
  e_to_m->is_ret        = d_i.is_ret;
  e_to_m->address       = result2;
  e_to_m->value         =
    (d_i.is_ret)?
      (int)target_pc:
    (d_i.is_jal || d_i.is_jalr || (d_i.type == U_TYPE))?
      result2:
    (d_i.is_store)?
      rv2:
      result1;
  e_to_m->target_pc     =
    (d_i.is_jal || d_i.is_branch)?
      target_pc:
```

```
      next_pc;
#ifndef __SYNTHESIS__
  e_to_m->pc              = pc;
  e_to_m->instruction     = instruction;
  e_to_m->d_i             = d_i;
#endif
}
```

9.3.4.5 The Running_cond_update Function

The running_cond_update function in the multicycle_pipeline_ip.cpp file (see Listing 9.29) monitors the connection between the memory access stage and the writeback stage. The running_cond_update function is tracking a RET to address 0 (the output from the memory access stage to the writeback stage is valid, the is_ret bit is set, and the value is null). When such a condition is met, the is_running condition is updated to end the main loop in the top function.

This is why the is_ret bit field is present in the e_to_m structure and propagated to the m_to_w structure even though neither the memory access stage nor the writeback stage use it.

Listing 9.29 The running_cond_update function

```
static void running_cond_update(
  from_m_to_w_t w_from_m,
  bit_t         *is_running){
  *is_running =
    !w_from_m.is_valid ||
    !w_from_m.is_ret   ||
     w_from_m.value != 0;
}
```

9.3.5 The Memory Access Stage

The mem_access function in the mem_access.cpp file is shown in Listing 9.30.

If the input from the execute stage is valid, the memory access does its stage job.

The stage_job function loads or stores.

The set_output_to_w function fills the m_to_w output structure fields.

Eventually, the output valid bit is set if the input from the execute stage is valid, cleared otherwise.

Listing 9.30 The mem_access function

```
void mem_access(
  from_e_to_m_t  m_from_e,
  int           *data_ram,
  from_m_to_w_t *m_to_w){
  int value;
  if (m_from_e.is_valid){
    value = m_from_e.value;
    stage_job(m_from_e.is_load, m_from_e.is_store,
              m_from_e.address, m_from_e.func3, data_ram,
             &value);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("mem       ");
    printf("%04d\n", (int)(m_from_e.pc<<2));
#endif
#endif
    set_output_to_w(m_from_e.rd, m_from_e.has_no_dest,
                    m_from_e.is_ret, value,
#ifndef __SYNTHESIS__
                    m_from_e.pc,  m_from_e.instruction,
                    m_from_e.d_i, m_from_e.target_pc,
#endif
                    m_to_w);
  }
  m_to_w->is_valid = m_from_e.is_valid;
}
```

The stage_job function in the mem_access.cpp file (see Listing 9.29) either loads from or stores to memory. If the instruction is neither a load nor a store, the function just returns.

The mem_load and mem_store functions in the mem.cpp file are unchanged.

Listing 9.31 The stage_job function

```
static void stage_job(
  bit_t            is_load,
  bit_t            is_store,
  b_data_address_t address,
  func3_t          func3,
  int             *data_ram,
  int             *value){
  if (is_load)
    *value = mem_load(data_ram, address, func3);
  else if (is_store)
    mem_store(data_ram, address, *value, (ap_uint<2>)func3);
}
```

The set_output_to_w function in the mem_access.cpp file (see Listing 9.32) fills the m_to_w structure. It transmits the parts of the instruction decoding which are relevant for the writeback, i.e. the destination register rd and the indication that the instruction does not have a destination has_no_dest.

It also outputs the is_ret bit indicating if the instruction is a RET. As I have already mentioned, this bit is used by the top function main loop control, not by the writeback stage.

The set_output_to_w function sends its value argument to the writeback stage, which is the value loaded from the memory if the instruction is a LOAD or else the value computed by the execute stage.

Listing 9.32 The set_output_to_w function

```
static void set_output_to_w(
  reg_num_t             rd,
  bit_t                 has_no_dest,
  bit_t                 is_ret,
  int                   value,
#ifndef __SYNTHESIS__
  code_address_t        pc,
  instruction_t         instruction,
  decoded_instruction_t d_i,
  code_address_t        target_pc,
#endif
  from_m_to_w_t         *m_to_w){
  m_to_w->rd           = rd;
  m_to_w->has_no_dest  = has_no_dest;
  m_to_w->is_ret       = is_ret;
  m_to_w->value        = value;
#ifndef __SYNTHESIS__
  m_to_w->pc           = pc;
  m_to_w->instruction  = instruction;
  m_to_w->d_i          = d_i;
  m_to_w->target_pc    = target_pc;
#endif
}
```

9.3.6 The Writeback Stage

The write_back function in the wb.cpp file is shown in Listing 9.33.

If the writeback stage input is valid, the stage_job function writes the value into the destination register rd if the instruction has a destination.

Then, the write_back function unlocks the written register rd by clearing the is_reg_computed[w_from_m.rd] bit.

If the writeback stage input is not valid, the write_back function just returns.

Listing 9.33 The write_back function

```
void write_back(
  from_m_to_w_t w_from_m,
  int            *reg_file,
  bit_t          *is_reg_computed){
  if (w_from_m.is_valid){
    stage_job(w_from_m.has_no_dest, w_from_m.rd,
              w_from_m.value, reg_file);
    if (!w_from_m.has_no_dest)
      is_reg_computed[w_from_m.rd] = 0;
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("wb       ");
    printf("%04d\n", (int)(w_from_m.pc<<2));
    if (!w_from_m.d_i.is_branch && !w_from_m.d_i.is_jalr)
      emulate(reg_file, w_from_m.d_i, w_from_m.target_pc);
#else
#ifdef DEBUG_FETCH
    printf("%04d: %08x       ",
      (int)(w_from_m.pc<<2), w_from_m.instruction);
#ifndef DEBUG_DISASSEMBLE
    printf("\n");
#endif
#endif
#ifdef DEBUG_DISASSEMBLE
    disassemble(w_from_m.pc, w_from_m.instruction,
                w_from_m.d_i);
#endif
#ifdef DEBUG_EMULATE
    emulate(reg_file, w_from_m.d_i, w_from_m.target_pc);
#endif
#endif
#endif
  }
}
```

The writeback stage_job function in the wb.cpp file (see Listing 9.34) writes the value into the destination register if the instruction has a destination.

Listing 9.34 The writeback stage_job function

```
static void stage_job(
  bit_t      has_no_dest,
  reg_num_t rd,
  int        value,
  int        *reg_file){
  if (!has_no_dest) reg_file[rd] = value;
}
```

9.4 Simulating, Synthesizing, and Running the IP

9.4.1 Simulating and Synthesizing the IP

Experimentation

To simulate the multicycle_pipeline_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with multicycle_pipeline_ip.

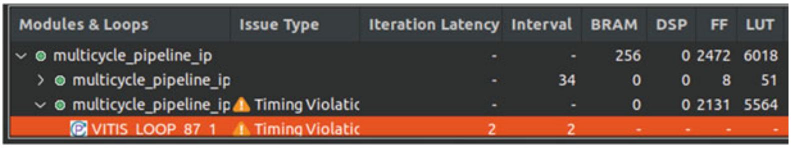
You can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

The testbench code in the testbench_multicycle_pipeline_ip.cpp file is unchanged. The simulation results are also unchanged (except for the number of cycles of the runs).

When the DEBUG_PIPELINE constant is defined (by uncommenting its definition line in the debug_multicycle_pipeline_ip.h file), the simulation result is presented as a cycle by cycle execution, with each stage printing its job (the print occurs in the cycle at which the outputs of the stage are valid). Refer back to the Listings 9.14 to 9.19 to have a preview of what it looks like.

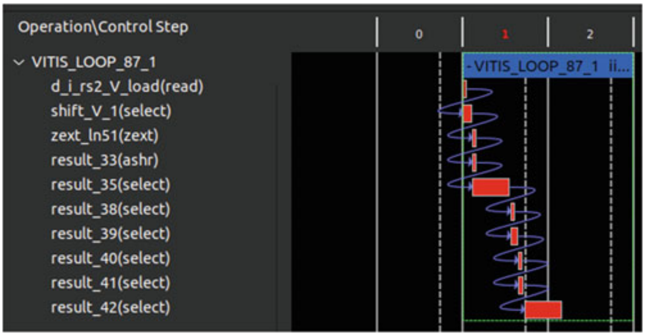
Figure 9.6 shows the synthesis report.

Figure 9.7 shows the main loop schedule with a timing violation. The schedule fits in two cycles (20 ns, 50 Mhz). The timing violation can be ignored.



Modules & Loops	Issue Type	Iteration Latency	Interval	BRAM	DSP	FF	LUT
⌄ ● multicycle_pipeline_ip		-	-	256	0	2472	6018
› ● multicycle_pipeline_ip		-	34	0	0	8	51
⌄ ● multicycle_pipeline_ip	⚠ Timing Violatic	-	-	0	0	2131	5564
VITIS_LOOP_87_1	⚠ Timing Violatic	2	2	-	-	-	-

Fig. 9.6 Multicycle pipeline IP synthesis report



VITIS_LOOP_87_1
d_i_rs2_V_load(read)
shift_V_1(select)
zext_ln51(zext)
result_33(ashr)
result_35(select)
result_38(select)
result_39(select)
result_40(select)
result_41(select)
result_42(select)

Fig. 9.7 Main loop schedule

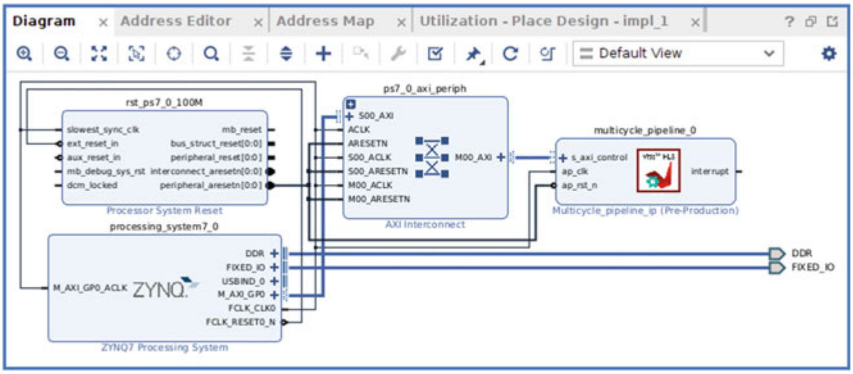


Fig. 9.8 Vivado multicycle pipeline IP block design

9.4.2 The Vivado Project and the Implementation Report

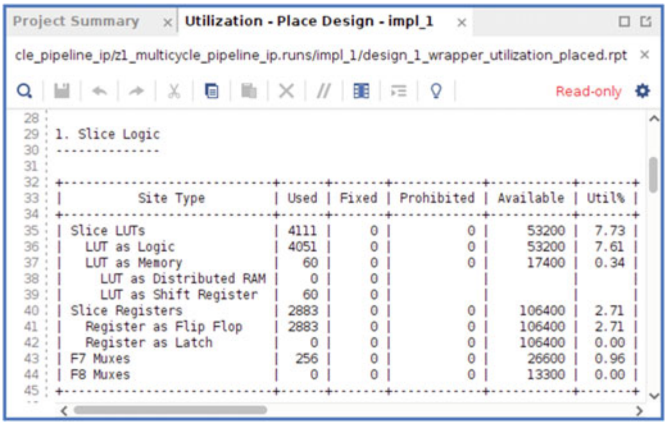
Figure 9.8 shows the Vivado block design.

Figure 9.9 shows the Vivado implementation report (4,111 LUTs, 7.73%).

9.4.3 Running the IP on the Development Board

Experimentation

To run the multicycle_pipeline_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with multicycle_pipeline_ip.
You can play with your IP, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.


Project Summary
Utilization - Place Design - impl_1
cle_pipeline_ip/z1_multicycle_pipeline_ip.runs/impl_1/design_1_wrapper_utilization_placed.rpt
Read-only

```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 4111 |     0 |          0 |     53200 |  7.73 |
|   LUT as Logic             | 4051 |     0 |          0 |     53200 |  7.61 |
|   LUT as Memory            |   60 |     0 |          0 |     17400 |  0.34 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 2883 |     0 |          0 |    106400 |  2.71 |
|   Register as Flip Flop    | 2883 |     0 |          0 |    106400 |  2.71 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  256 |     0 |          0 |     26600 |  0.96 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 9.9 Vivado multicycle pipeline IP implementation report

The code in the helloworld.c file is shown in Listing 9.35 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script).

Listing 9.35 The helloworld.c file to run the test_mem.s RISC-V program

```c
#include <stdio.h>
#include "xmulticycle_pipeline_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE      (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE 16
//size in words
#define DATA_RAM_SIZE      (1<<LOG_DATA_RAM_SIZE)
XMulticycle_pipeline_ip_Config *cfg_ptr;
XMulticycle_pipeline_ip          ip;
word_type code_ram[CODE_RAM_SIZE] = {
#include "test_mem_0_text.hex"
};
int main(){
  unsigned int nbi, nbc;
  word_type    w;
  cfg_ptr = XMulticycle_pipeline_ip_LookupConfig(
      XPAR_XMULTICYCLE_PIPELINE_IP_0_DEVICE_ID);
  XMulticycle_pipeline_ip_CfgInitialize(&ip, cfg_ptr);
  XMulticycle_pipeline_ip_Set_start_pc(&ip, 0);
  XMulticycle_pipeline_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XMulticycle_pipeline_ip_Start(&ip);
  while (!XMulticycle_pipeline_ip_IsDone(&ip));
  nbi = XMulticycle_pipeline_ip_Get_nb_instruction(&ip);
  nbc = XMulticycle_pipeline_ip_Get_nb_cycle(&ip);
  printf("%d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", nbi, nbc, ((float)nbi)/nbc);
  printf("data memory dump (non null words)\n");
  for (int i=0; i<DATA_RAM_SIZE; i++){
    XMulticycle_pipeline_ip_Read_data_ram_Words
      (&ip, i, &w, 1);
    if (w != 0)
      printf("m[%4x] = %16d (%8x)\n", 4*i, (int)w,
             (unsigned int)w);
  }
}
```

If the RISC-V code in the test_mem.h file is run, it produces the output shown in Listing 9.36 (217 cycles versus 119 cycles on the rv32i_pp_ip; the locking mechanism is less efficient than the bypass version from Chap. 8; the IPC is low, which means that the pipeline is far from filled):

Listing 9.36 The helloworld output

```
88 fetched and decoded instructions in 217 cycles (ipc = 0.41)
data memory dump (non null words)
m[   0] =                1 (       1)
m[   4] =                2 (       2)
m[   8] =                3 (       3)
m[   c] =                4 (       4)
m[  10] =                5 (       5)
m[  14] =                6 (       6)
m[  18] =                7 (       7)
m[  1c] =                8 (       8)
m[  20] =                9 (       9)
m[  24] =               10 (       a)
m[  2c] =               55 (      37)
```

9.4.4 Further Testing of the Multicycle_pipeline_ip

To pass the riscv-tests on the Vitis_HLS simulator, you just need to use the testbench_riscv_tests_multicycle_pipeline_ip.cpp program in the riscv-tests/my_isa/my_rv32ui folder as the testbench.

To pass the riscv-tests on the FPGA, you must use the helloworld_multicycle_pipeline_ip.c in the riscv-tests/my_isa/my_rv32ui folder. Normally, since you already ran the update_helloworld.sh shell script for the other processors, the helloworld_multicycle_pipeline_ip.c file should have paths adapted to your environment. However if you did not, you must run ./update_helloworld.sh.

9.5 Comparing the Multicycle Pipeline to the 4-Stage Pipeline

To run a benchmark from the mibench suite, say my_dir/bench, you set the testbench as the testbench_bench_multicycle_pipeline_ip.cpp file found in the mibench/my_mibench/my_dir/bench folder. For example, to run basicmath, you set the testbench as testbench_basicmath_multicycle_pipeline_ip.cpp in the mibench/my_mibench/my_automotive/basicmath folder.

To run one of the official riscv-tests benchmarks, say bench, you set the testbench as the testbench_bench_multicycle_pipeline_ip.cpp file found in the riscv-tests/benchmarks/bench folder. For example, to run median, you set the testbench as testbench_median_multicycle_pipeline_ip.cpp in the riscv-tests/benchmarks/median folder.

To run the same benchmarks on the FPGA, select helloworld_multicycle_pipeline_ip.c to run in Vitis IDE.

Table 9.1 shows the execution time of the benchmarks as computed with Eq. 5.1 ($nmi * cpi * c$, where $c = 20$ns). Even though the cycle duration has been reduced (30–20 ns), the CPI has been multiplied by 1.60 (1.92 vs. 1.20; mainly because of the register lock/unlock system which is less efficient than bypassing and because of the new issue stage which lengthens the pipeline, i.e. increases the latencies), resulting in a degradation of the performance compared to the 4-stage pipeline implemented in the rv32i_pp_ip design.

The higher the CPI, the higher the degradation of the performance (from +20% improvement with a 1.26 CPI for towers to a −30% degradation with a 2.26 CPI for stringsearch).

Notice that the run of the mm benchmark takes about one hour on the simulator (and a few seconds on the FPGA). You will save a lot of time if you run the benchmarks on the FPGA rather than on the Vitis_HLS simulator.

The multicycle pipeline should be implemented only if the ISA is to be extended to some multicycle instructions, e.g. F, D, or M extensions (or if the data memory should be hierarchized with caches, implying a multicycle execution of the memory accesses).

Table 9.1 Execution time of the benchmarks on the 6-stage pipelined multicycle_pipeline_ip processor

suite	benchmark	Cycles	cpi	Time (s)	4-stage time (s)	Improvement (%)
mibench	basicmath	62,723,992	2.03	1.254479840	1.129294710	−11
mibench	bitcount	57,962,065	1.78	1.159241300	1.137299970	−2
mibench	qsort	12,845,805	1.92	0.256916100	0.242585730	−6
mibench	stringsearch	1,240,390	2.26	0.024807800	0.019074900	−30
mibench	rawcaudio	1,363,673	2.15	0.027273460	0.022745130	−20
mibench	rawdaudio	942,834	2.01	0.018856680	0.016884030	−12
mibench	crc32	660,028	2.20	0.013200560	0.010800480	−22
mibench	fft	64,979,537	2.07	1.299590740	1.147007820	−13
mibench	fft_inv	66,054,232	2.07	1.321084640	1.167260040	−13
riscv-tests	median	53,141	1.91	0.001062820	0.001064130	0
riscv-tests	mm	328,860,252	2.09	6.577205040	5.802164250	−13
riscv-tests	multiply	745,904	1.78	0.014918080	0.016200720	8
riscv-tests	qsort	491,648	1.81	0.009832960	0.009918900	1
riscv-tests	spmv	2,426,687	1.95	0.048533740	0.044938200	−8
riscv-tests	towers	510,511	1.26	0.010210220	0.012791550	20
riscv-tests	vvadd	24,016	1.50	0.000480320	0.000540360	11

A way to improve the multicycle pipeline is to fill it more efficiently. The compiler can help by rearranging the instructions to minimize the number of waiting cycles (but you have to modify the compiler).

A major improvement would be to avoid the unused cycles due to the next pc computation latency. This can be achieved with a branch predictor, as shown in Fig. 9.10.

The branch predictor predicts the next pc from the current pc (the current pc is itself most of the time the result of a previous prediction) and a set of caches of the targets of the past control flow instructions (if the current pc is in one of the caches, it means it is the address of a control flow instruction and the cache gives a target address, which is the prediction; otherwise, the prediction is pc + 1).

The predicted next pc is forwarded all along the pipeline up to the writeback stage, where it is compared to the computed next pc. If they match, the run continues (the correction order bit is cleared, letting pc receive the lower input of the mux multiplexer shown on the left side of Fig. 9.10). Otherwise (the correction order bit is set and pc receives the upper input of the multiplexer), the computed next pc is sent back to the fetch stage to correct the instruction path.

The instructions in the pipeline are all cancelled. Whatever the correctness of the prediction, the caches of the targets in the branch predictor are updated.

Because the branch predictor is able to produce a prediction in a single processor cycle, the fetch stage receives a new predicted pc every cycle, even when a control instruction is fetched. Only when the prediction is wrong, the already fetched instructions on the wrong path are discarded and the corresponding cycles are lost. The best predictor [1] reaches a rate of 8 branch Mispredictions Per Kilo Instructions (MPKI), i.e. on the average, one miss every 125 instructions.

Another way to improve the performance of the design and decrease the CPI is to provide other instruction sources through multithreading, which will be your next design.

9.6 Proposed Exercise: Reduce II to 1

With the multicycle pipeline, the processor cycle has increased up to 50 Mhz. It is possible to double the speed though, with an initiation interval II = 1.

The exercise is to implement such an II = 1 design. Notice that the next pc computation in the fetch stage cannot benefit of any decoding at all because the fetch latency is two cycles (BRAM block latency; however, the BRAM access throughput is one access per cycle, i.e. you can start a new access every cycle).

As the fetch duration is two cycles, while the processor is fetching the instruction, a new fetch should start from a predicted address.

The simplest prediction is to systematically set next pc as pc+1. The MPKI for this static predictor is the rate of control instructions (in average, there are 15% of JAL, JALR, or taken branches), i.e. MPKI = 150 (one miss every six or seven instructions).

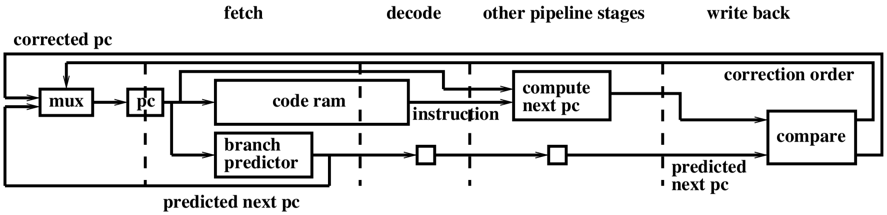


Fig. 9.10 A pipeline with branch prediction

A misprediction should be corrected as soon as possible, i.e. in the decode stage if the instruction is a JAL or in the execute stage if the instruction is a JALR or a taken BRANCH.

Once the II = 1 implementation is working, you can test it on the benchmarks, compute the average CPI and compare the performance of your design to the multicycle pipeline design presented in this chapter and to the 4-stage pipeline design presented in Chap. 8.

Then, you can improve the CPI by replacing your prediction mechanism with a dynamic branch predictor (e.g. Two-Level Adaptive Training Branch Prediction [2], Combining Branch Predictors (gshare) [3], A case for (partially) TAgged GEometric history length branch prediction (TAGE) [1]).

References

1. A. Seznec, P. Michaud, A case for (partially) TAgged GEometric history length branch prediction. J. Instruction Level Parallelism (2006)
2. T.-Y. Yeh, Y.N. Patt, Two-level adaptive training branch prediction, in *MICRO 24: Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61 (1991)
3. S. McFarling, *Combining Branch Predictors*. Digital Western Research Laboratory Technical Note 36 (1993)

Building a RISC-V Processor with a Multiple Hart Pipeline

10

Abstract

This chapter will make you build your fourth RISC-V processor. The implemented microarchitecture proposed in this fourth version improves the CPI by filling the pipeline with multiple instruction flows. At the OS level, a flow of control is a *thread*. The processor can be designed to host multiple threads and run them simultaneously (Simultaneous MultiThreading or SMT, as named by Tullsen in [1]). Such thread dedicated slots in the processor are called *harts* (for HARdware Threads). The multihart design presented in this chapter can host up to eight harts. The pipeline has six stages. The processor cycle is two FPGA cycles (i.e. 50 Mhz).

10.1 Handling Multiple Threads Simultaneously with a Multiple Hart Processor

In order to fill the pipeline and decrease the CPI as close to 1 as possible, there are multiple techniques. The lost cycles are due to waiting conditions, i.e. dependencies between instructions. One way to recover these cycles is to eliminate the dependencies with prediction. The control flow dependencies can be eliminated through branch prediction, as was presented in 9.5. The data dependencies can be eliminated through value prediction [2,3]. However, predictors are complex pieces of hardware and I will not give any implementation in this book.

Another possibility is to use independent instructions to fill empty pipeline slots. While an instruction is waiting for a prior result, the pipeline continues to fetch and process following instructions and make them bypass the stopped one. This is called out-of-order, or OoO computation [4]. OoO implementation is tricky and requires costly additional logic like a register renaming unit [5]. The reward is really only worth the complexity for speculative superscalar designs [6] in which the pipeline stages may process multiple instructions simultaneously and the addresses for the fetch unit are provided through a branch predictor. Again, I will avoid implementing

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_10

an OoO design. I will stick to a scalar design (i.e. which processes instructions one by one), with an optimal CPI of 1.

A third option to fill the pipeline is to interleave instructions from multiple flows (i.e. by running multiple threads), which is called *multithreading*.

It should be pointed out though that multithreading is a way to make the pipeline compute more but not a way to accelerate the run of a thread of computation. The pipeline is shared by the threads and each thread runs at the speed corresponding to its share (e.g. half the processor speed if there are two threads, each filling half of the pipeline slots).

To handle multiple threads, a processor must be provided with multiple pc and register files (in a multithreaded OoO design implementing register renaming like SMT, the register file is shared). The pipeline stages and the computing units are shared.

The inter-stage connections must specify which thread is emitting its result to the next stage (e.g. d_to_i.hart to send something from the decode stage to the issue stage).

Running an instruction of a thread means reading sources from and writing results to the dedicated register file. If the instruction is a control flow one, it updates the pc of the respective thread.

The multihart_ip hardware is designed as shown in Fig. 10.1 (the figure depicts a 4-hart design). The figure shows the six stages of the pipeline, from left to right in the upper half and continued from right to left in the lower half.

In the figure, the green rectangles represent hart slots.

Each pipeline stage has four slots (e.g. green rectangles named i0 to i3 for the issue stage). Each slot may host one instruction. In the fetch stage, the slots are name pc0 to pc3. Each can host the code memory address to the instruction of one running thread.

Each stage processes a single instruction per cycle. Hence, the hart to be processed is selected among the four slots in each pipeline stage. This selection is done in the same time in all the stages. In the same cycle, the stages may select different harts: for example, hart 0 for the fetch stage, hart 3 for the decode stage, hart 2 for the issue stage and so on.

The selection is done in two successive steps.

The first selection step is represented as a magenta vertical line. It selects one of the hosted threads in the four hart slots. The second step is represented as a red vertical line. It either selects the chosen thread from the magenta first step if there is any, or the incoming instruction from the previous stage output.

Each selection vertical line represents a multiplexer to choose one of its inputs.

A thread can be choosen if the instruction held in the stage or at the stage input is ready, i.e. fulfills some stage related condition. For example, an instruction in the issue stage is ready if its register sources are not locked.

The selection process follows a fixed priority order. The first step has priority over the second one. In the first step, harts are increasingly ordered (i.e. hart 0 has the highest priority and hart 3 has the lowest one). In the fetch stage, the pc incoming from the decode stage has priority over the one incoming from the execute stage.

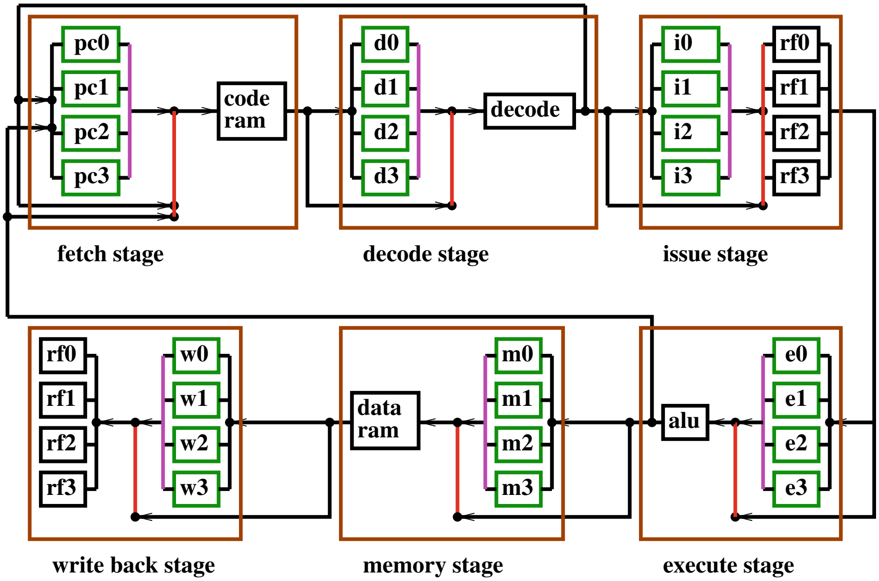


Fig. 10.1 A pipeline for multiple harts

Hence in the figure the upmost input on any magenta line has the highest priority and the lowest input on any red line has the lowest priority.

The hart selection adds some delay in the critical path. I reorganized some of the computations to fit every path into the two FPGA cycles limit.

10.2 A Multiple Hart Memory Model

In an OS based processor, the memory model is imposed by the OS. The memory is physically an array of bytes but the OS manages the space by paging it and allocating and freeing pages on demand of the processes or threads. As a result, a thread cannot directly access the physical memory. Its memory is made of non contiguous pages, assembled through tables of page references.

I will not detail further the OS organization of the memory.

In a no-OS or bare-metal processor, we are more free to organize the memory as we wish. Up to this point in the book, I have considered a processor running a single program. In this case, the program has full access to the processor memory, which is an array of bytes.

However, I have separated (see Chap. 6) the code and the data memories (which has some drawbacks like prohibiting JIT (just-in-time) compilation, e.g. building bit-banging instructions directly: the building program is unable to transfer the built instructions from the data memory to the code memory; however, I separated the two memories for reasons related to the number of access ports on the memory banks).

As I introduce multithreading, I need to reconsider how the memory is managed.

For the code memory, I can keep the same organization. There is a single array of instructions, fully accessible to all of the threads through the fetch stage (remember though that in a cycle only one thread is selected to fetch).

In each cycle, the fetch stage reads a single instruction from a single thread. The threads can be placed anywhere in the code memory. Multiple threads can even share their code.

For the data memory, the single byte array model should be reconsidered.

First, contrarily to the code memory, the data memory is writable. It is necessary to protect the memory space of one thread from the writings of the other threads. However, we must keep flexible by, on one side, prohibiting concurrent accesses from multiple threads, and on another side, by allowing to share memory spaces between threads.

Secondly, each thread may use a stack space, which should be fully private.

So, I adapt the data memory as follows: it is now an array of sub-memories, one per hart. This partitioning ensures protection. A hart accesses its own partition. A partition is divided in two parts: the static data part at one end and the stack at the other end. Both parts converge to one another. The programmer should take care that the stack never overlaps the static data part (as there is no OS to ensure this condition).

However, to allow memory sharing, a hart may access the other harts memory partitions. This should be done with care (the *programmer* should be careful). Of course, the stack parts of the other harts memory partitions should not be accessed, only their static data parts.

By allowing memory sharing, we allow parallelism. A computation may be divided into threads and the computed data may be partitioned and distributed into the hart memories.

Of course, as the processor runs a single hart per cycle, the threads are not really run in parallel. They are interleaved. However, as I will show in this chapter, the efficiency of the pipeline is improved by a multihart organization and so, a multithreaded computation is run slightly faster than its sequential version.

The processor is designed to take care of the partitioned memory model. A load or store instruction computes an access address to the data memory which is viewed by the running hart as relative to its own partition. This is illustrated in Fig. 10.2.

In the upper part of the figure, each hart accesses the first memory word of its own partition. For example, hart 1 loads from address 0, which is turned into an access to the red square, i.e. memory location 0 in hart 1 partition. Hart 2 also loads from address 0, but this address relative to its own data memory partition is turned into an access to the green square.

In the lower part of the figure, hart 1 successively accesses to words in different partitions of the memory. The HART_DATA_RAM_SIZE constant is the size (in words) of the hart memory partition.

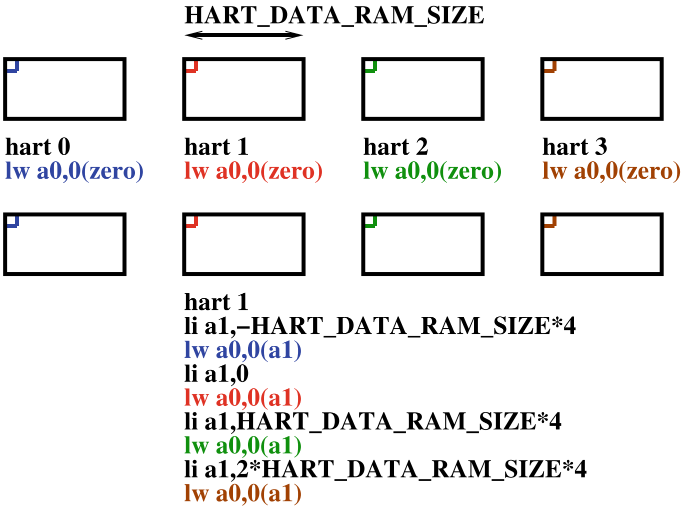


Fig. 10.2 How a hart addresses the partitioned memory (upper part: local accesses, lower part: accesses to the whole memory)

The first load instruction (in blue) accesses the first word of the first partition, at the relative word address -HART_DATA_RAM_SIZE, i.e. the absolute word address 0.

The second load (in red) accesses the first word of the second partition, at the relative word address 0, i.e. the absolute word address HART_DATA_RAM_SIZE.

The third load (in green) accesses the first word of the third partition, at the relative word address HART_DATA_RAM_SIZE, i.e. the absolute word address 2*HART_DATA_RAM_SIZE.

The fourth load (in brown) accesses the first word of the last partition, at the relative word address 2*HART_DATA_RAM_SIZE, i.e. the absolute word address 3*HART_DATA_RAM_SIZE.

In this four banks memory, the hart 0 relative word addresses range from 0 to 4*HART_DATA_RAM_SIZE-1.

For hart 1, the relative word addresses range from -HART_DATA_RAM_SIZE to 3*HART_DATA_RAM_SIZE-1.

For hart 2, the relative word addresses range from -2*HART_DATA_RAM_SIZE to 2*HART_DATA_RAM_SIZE-1.

For hart 3, the relative word addresses range from -3*HART_DATA_RAM_SIZE to HART_DATA_RAM_SIZE-1.

For any hart, a negative relative address is an access to a partition of a preceding hart (hart 0 relative addresses are never negative) and a positive relative address is an access either to the local partition (if the relative word address is less than HART_DATA_RAM_SIZE) or to a partition of a succeeding hart.

The main advantages of the hart-partitioned memory model on bare-metal is that protection and sharing are directly handled by the hardware (with a careful programmer: he/she should ensure that a thread which writes to a data and another thread which reads from this data are properly synchronized, with the reading instruction execution following the writing one; there is no hardware in the multihart design to ensure such a synchronization; in Sect. 10.4.2, I present a parallel program with multiple threads properly self-synchronized).

10.3 The Multihart Pipeline

All the source files related to the multihart_ip can be found in the multihart_ip folder.

The pipeline has the same six stages as the multicycle one presented in the preceding chapter.

The fetch stage has been slightly modified (see Fig. 10.3). As the pipeline is intended to run multiple threads, it is not mandatory to provide a new pc every cycle. If a new pc is available every other cycle, a 2-hart pipeline can alternate threads, the first one using the even cycles and the second one using the odd cycles. A drawback is that if only one thread is running, half of the cycles are lost.

The fetch stage does not decode instructions at all. It cannot distinguish control flow ones. It does not compute any next pc. The next pc computation is done in the decode stage (if the instruction is neither a BRANCH nor a JALR). This design gives better performance for runs with at least two threads than the multicycle pipeline organization where the fetch stage produces a next pc for itself. Moreover, this simplification saves LUTs.

In the figure, the fetch stage forwards the fetch pc to the decode stage, which computes the next pc. This next pc is sent to the fetch stage where it is received two cycles after the initial fetch. Hence, the pipeline can be filled only when at least two threads are running (red thread in hart 0 and green thread in hart 1).

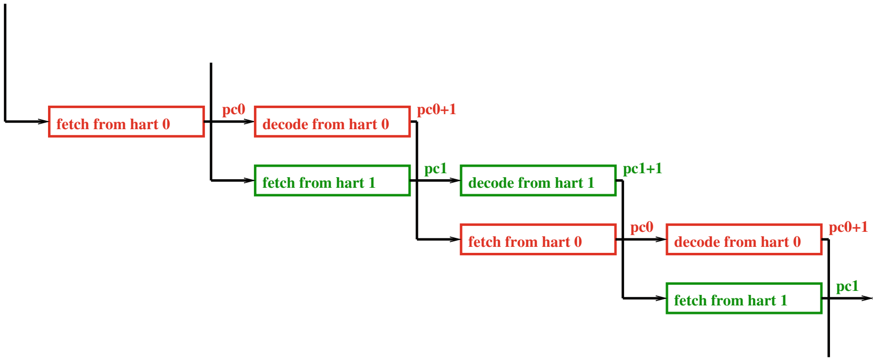


Fig. 10.3 The fetch stage does not decode

10.3.1 The Number of Harts

The number of harts is defined in the multihart_ip.h file (see Listing 10.1). To change it, you need to change the definition of the LOG_NB_HART constant (1 for two harts, 2 for four harts and 3 for eight harts). The number of harts cannot be 1 (LOG_NB_HART should not be null) and cannot be greater than eight.

The code memory is organized as a single bank, shared by all the harts. The codes they run are all mixed in the same memory.

The data memory is partitioned. Each hart has a private partition of HART_DATA_RAM_SIZE words. The total data memory size is DATA_RAM_SIZE, i.e. 2^{16} words (256KB), whatever the number of harts. Hence, when the number of harts increases, the hart memory partition size decreases. For example, with two harts, each hart has a 2^{15} words partition (128 KB). With four harts, the size of the partitions is 2^{14} words (64 KB). With eight harts, the size is 2^{13} words (32 KB).

Listing 10.1 The multihart_ip.h file (partial)

```
#ifndef __MULTIHART_IP
#define __MULTIHART_IP
#include "ap_int.h"
#include "debug_multihart_ip.h"
#define LOG_NB_HART             1
#define NB_HART                (1<<LOG_NB_HART)
#define LOG_CODE_RAM_SIZE      16
#define CODE_RAM_SIZE          (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE      16
#define DATA_RAM_SIZE          (1<<LOG_DATA_RAM_SIZE)
#define LOG_HART_DATA_RAM_SIZE (LOG_DATA_RAM_SIZE-LOG_NB_HART)
#define HART_DATA_RAM_SIZE     (1<<LOG_HART_DATA_RAM_SIZE)
#define LOG_REG_FILE_SIZE       5
#define NB_REGISTER            (1<<LOG_REG_FILE_SIZE)
...
```

10.3.2 The Multihart Stage States

An instruction stays in a pipeline stage until it is selected. Each pipeline stage has an internal array to hold its waiting instructions.

The array is named from the pipeline stage initial letter and the state suffix (e.g. e_state for the execute stage array).

The array has one entry per hart (hence, a pipeline stage may not hold more than one waiting instruction per running thread).

A state array entry is a structure gathering all the input and output fields of the pipeline stage.

For example, the f_state array entry has two fields: fetch_pc and instruction. The fetch_pc field is used to hold the incoming pc sent by the decode stage on the d_to_f link or by the execute stage on the e_to_f link. The instruction field is used to hold the fetched instruction to be sent to the decode stage on the f_to_d link.

Figure 10.4 shows the state arrays and the inter-stage links (red arrows).

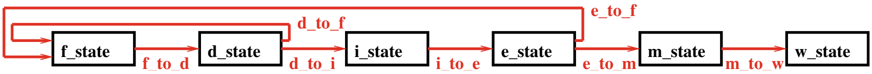


Fig. 10.4 The six pipeline stages, their state arrays, and their inter-stage links

The types of the state arrays and the types of the inter-stage links for the six pipeline stages are shown in Listings 10.2 to 10.7, which are part of the multihart_ip.h file.

10.3.2.1 The F_state_t and From_f_to_d_t Types

For the fetch stage, the f_state_t and from_f_to_d_t type definitions are shown in Listing 10.2.

Listing 10.2 The f_state_t and from_f_to_d_t type definitions

```
typedef struct f_state_s{
  code_address_t fetch_pc;
  instruction_t  instruction;
} f_state_t;
typedef struct from_f_to_d_s{
  bit_t          is_valid;
  hart_num_t     hart;
  code_address_t fetch_pc;
  instruction_t  instruction;
} from_f_to_d_t;
```

10.3.2.2 The D_state_t, From_d_to_f_t, and From_d_to_i_t Types

For the decode stage, the d_state_t, from_d_to_f_t, and from_d_to_i_t types are shown in Listing 10.3.

Listing 10.3 The d_state_t and from_d_to_x_t type definitions

```
typedef struct d_state_s{
  code_address_t         fetch_pc;
  instruction_t          instruction;
  decoded_instruction_t d_i;
  code_address_t         relative_pc;
} d_state_t;
typedef struct from_d_to_f_s{
  bit_t          is_valid;
  hart_num_t     hart;
  code_address_t relative_pc;
} from_d_to_f_t;
typedef struct from_d_to_i_s{
  bit_t                  is_valid;
  hart_num_t             hart;
  code_address_t         fetch_pc;
  decoded_instruction_t d_i;
  code_address_t         relative_pc;
#ifndef __SYNTHESIS__
  instruction_t          instruction;
#endif
} from_d_to_i_t;
```

10.3.2.3 The I_state_t and From_i_to_e_t Types

For the issue stage, the i_state_t and from_i_to_e_t types are shown in Listing 10.4.

Listing 10.4 The i_state_t and from_i_to_e_t type definitions

```
typedef struct i_state_s{
  code_address_t        fetch_pc;
  decoded_instruction_t d_i;
  int                   rv1;
  int                   rv2;
  code_address_t        relative_pc;
  bit_t                 wait_12;
#ifndef __SYNTHESIS__
  instruction_t         instruction;
#endif
} i_state_t;
typedef struct from_i_to_e_s{
  bit_t                 is_valid;
  hart_num_t            hart;
  code_address_t        fetch_pc;
  decoded_instruction_t d_i;
  int                   rv1;
  int                   rv2;
  code_address_t        relative_pc;
#ifndef __SYNTHESIS__
  instruction_t         instruction;
#endif
} from_i_to_e_t;
```

10.3.2.4 The E_state_t, From_e_to_f_t, and From_e_to_m_t Types

For the execute stage, the e_state_t, from_e_to_f_t, and from_e_to_m_t types are shown in Listing 10.5.

Listing 10.5 The e_state_t and from_e_to_x_t type definitions

```
typedef struct e_state_s{
  int                   rv1;
  int                   rv2;
  code_address_t        fetch_pc;
  decoded_instruction_t d_i;
  code_address_t        relative_pc;
  code_address_t        target_pc;
  bit_t                 is_target;
#ifndef __SYNTHESIS__
  instruction_t         instruction;
#endif
} e_state_t;
typedef struct from_e_to_f_s{
  bit_t          is_valid;
  hart_num_t     hart;
  code_address_t target_pc;
} from_e_to_f_t;
typedef struct from_e_to_m_s{
  bit_t                 is_valid;
  hart_num_t            hart;
  reg_num_t             rd;
  bit_t                 has_no_dest;
  bit_t                 is_load;
  bit_t                 is_store;
  func3_t               func3;
  bit_t                 is_ret;
  b_data_address_t      address;
  int                   value;
#ifndef __SYNTHESIS__
  code_address_t        fetch_pc;
  instruction_t         instruction;
  decoded_instruction_t d_i;
  code_address_t        target_pc;
#endif
} from_e_to_m_t;
```

10.3.2.5 The M_state_t and From_m_to_w_t Types

For the memory access stage, the m_state_t and from_m_to_w_t types are shown in Listing 10.6.

Listing 10.6 The m_state_t and from_m_to_w_t type definitions

```
typedef struct m_state_s{
  reg_num_t             rd;
  bit_t                 has_no_dest;
  bit_t                 is_load;
  bit_t                 is_store;
  func3_t               func3;
  bit_t                 is_ret;
  b_data_address_t      address;
  int                   value;
  hart_num_t            accessed_h;
  bit_t                 is_local;
#ifndef __SYNTHESIS__
  code_address_t        fetch_pc;
  instruction_t         instruction;
  decoded_instruction_t d_i;
  code_address_t        target_pc;
#endif
} m_state_t;
typedef struct from_m_to_w_s{
  bit_t                 is_valid;
  hart_num_t            hart;
  reg_num_t             rd;
  bit_t                 has_no_dest;
  bit_t                 is_ret;
  int                   value;
#ifndef __SYNTHESIS__
  code_address_t        fetch_pc;
  instruction_t         instruction;
  decoded_instruction_t d_i;
  code_address_t        target_pc;
#endif
} from_m_to_w_t;
```

10.3.2.6 The W_state_t Type

For the writeback stage, the w_state_t type is shown in Listing 10.7.

Listing 10.7 The w_state_t type definition

```
typedef struct w_state_s{
  int                     value;
  reg_num_t               rd;
  bit_t                   has_no_dest;
  bit_t                   is_ret;
#ifndef __SYNTHESIS__
  code_address_t          fetch_pc;
  instruction_t           instruction;
  decoded_instruction_t d_i;
  code_address_t          target_pc;
#endif
} w_state_t;
```

10.3.3 The Occupation Arrays

The instructions in the multihart pipeline move unevenly along the stages. An instruction may stay in the issue stage because its source registers are locked. It may stay in any stage because a higher priority hart is selected.

In each stage, the processing starts by a hart selection to choose which instruction is to be processed. To be selected, a hart must have its state array entry filled with an instruction.

Moreover, selecting a hart *h* is possible only if the next stage is able to host the output of the processing, i.e. if the hart *h* state array entry in the next stage is empty.

This selection procedure is not optimal though, because a full entry in stage *s+1* blocks a selection in stage *s* even if this entry is to be processed in the same cycle. However, an optimal selection algorithm would rely on a serialization of the stage selections, each stage selection depending on what hart is selected in the next stage. Anyway, the non-optimal selection algorithm used in the design is efficient, as will be shown by the performance measures at the end of the chapter.

So, the selection algorithm requires to be able to keep track of the occupation of the different state array entries. Stage *s* should be aware of which state array entries are empty in stage *s+1*.

For this purpose, for each stage an array of occupation bits is transmitted to its predecessor (see Fig. 10.5, in which each green arrow represents an array of NB_HART bits).

For example, the d_state_is_full array links the decode stage to the fetch stage. When hart *h* array entry is occupied in the decode stage, d_state_is_full[*h*] is set. In this case, hart *h* cannot be selected for fetch in the fetch stage.

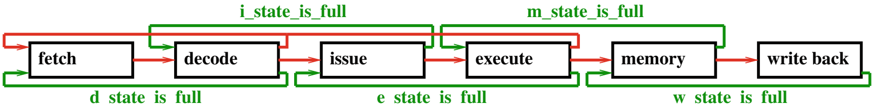


Fig. 10.5 Inter-stage retro links (in green) to show which state array entries are occupied

10.3.4 The Multihart_ip Top Function

10.3.4.1 The Multihart_ip Top Function Prototype

The multihart_ip top function in the multihart_ip.cpp file has the same prototype as the multicycle_pipeline_ip function except for its first two arguments (see the prototype in Listing 10.8).

The first running_hart_set argument is the set of running harts presented as a bit map. For an 8-hart IP, the value 0b10101010 for the running_hart_set argument starts the run of the four odd number harts 1, 3, 5, and 7.

The second start_pc argument is an array containing the set of starting pc addresses for the harts to be run.

The code_ram memory is common to all of the harts and should contain all the codes to be run. If the same address is given as a start pc for all the running threads, they run the same code (this is how I will test the IP).

As explained in the memory model description in Sect. 10.2, the data_ram memory is partitioned, with one partition of HART_DATA_RAM_SIZE words per hart. Each hart has access to the full memory.

Listing 10.8 The multihart_ip function prototype

```
void multihart_ip(
  unsigned int  running_hart_set,
  unsigned int  start_pc[NB_HART],
  unsigned int  code_ram[CODE_RAM_SIZE],
  int           data_ram[NB_HART][HART_DATA_RAM_SIZE],
  unsigned int *nb_instruction,
  unsigned int *nb_cycle){
#pragma HLS INTERFACE s_axilite port=running_hart_set
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=code_ram
#pragma HLS INTERFACE s_axilite port=data_ram
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=nb_cycle
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INLINE recursive
  ...
```

10.3.4.2 The Register File Declaration

Most of the local declarations are turned into arrays of harts. For example, there is one register file per hart (see Listing 10.9).

In the ARRAY_PARTITION pragma, the dim option indicates the dimension on which the partitioning should apply. Up to now, we only had single dimension arrays (i.e. vectors) to partition, so dim was systematically 1. In the multihart design, the arrays are matrices. The partitioning should apply to all the dimensions (i.e. each element should have its individual access port), hence the dim option is set as dim=0 complete (e.g. NB_HART*NB_REGISTER ports for reg_file).

Listing 10.9 The multihart_ip function local declarations (1)

```
  ...
  int    reg_file                [NB_HART][NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file       dim=0 complete
  bit_t  is_reg_computed         [NB_HART][NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=is_reg_computed dim=0 complete
  ...
```

10.3.4.3 The State Array and Inter-stage Link Declarations

The state array declarations are shown in Listing 10.10.

For a pipeline stage *x*, the x_state array variable has one entry per hart (to host one instruction).

The x_state_is_full array variable is a vector of bits (one bit per hart) for pipeline stage *x*. It is transmitted from one stage to its predecessor to indicate which hart entries are full in the state array.

Inter-stage links are also shown in Listing 10.10. They are not vectorized. Only one instruction can be selected, hence processed, and transmitted to the next stage. The link structure contains the emitting hart number (e.g. f_to_d.hart) and a valid bit to indicate if the link contains a valid output (refer back to Listings 10.2 to 10.6).

To avoid most of the RAW dependencies, each x_to_y variable is duplicated with a matching y_from_x variable. As was already explained in 8.1.2, at the beginning of the cycle the _to_ variables are all copied into the matching _from_ variables (see Listing 10.17: the new_cycle function does the copy job). Each stage function reads from the _from_ variables and writes to the _to_ variables.

Listing 10.10 The multihart_ip function local declarations (2)

```
  ...
  from_d_to_f_t f_from_d;
  from_e_to_f_t f_from_e;
  bit_t         f_state_is_full[NB_HART];
#pragma HLS ARRAY_PARTITION variable=f_state_is_full dim=1 complete
  f_state_t     f_state         [NB_HART];
#pragma HLS ARRAY_PARTITION variable=f_state         dim=1 complete
  from_f_to_d_t f_to_d;
  from_f_to_d_t d_from_f;
  bit_t         d_state_is_full[NB_HART];
#pragma HLS ARRAY_PARTITION variable=d_state_is_full dim=1 complete
  d_state_t     d_state         [NB_HART];
#pragma HLS ARRAY_PARTITION variable=d_state         dim=1 complete
  from_d_to_f_t d_to_f;
  from_d_to_i_t d_to_i;
  from_d_to_i_t i_from_d;
  bit_t         i_state_is_full[NB_HART];
#pragma HLS ARRAY_PARTITION variable=i_state_is_full dim=1 complete
  i_state_t     i_state         [NB_HART];
#pragma HLS ARRAY_PARTITION variable=i_state         dim=1 complete
```

```
  from_i_to_e_t i_to_e;
  from_i_to_e_t e_from_i;
  bit_t         e_state_is_full[NB_HART];
#pragma HLS ARRAY_PARTITION variable=e_state_is_full dim=1 complete
  e_state_t     e_state        [NB_HART];
#pragma HLS ARRAY_PARTITION variable=e_state         dim=1 complete
  from_e_to_f_t e_to_f;
  from_e_to_m_t e_to_m;
  from_e_to_m_t m_from_e;
  bit_t         m_state_is_full[NB_HART];
#pragma HLS ARRAY_PARTITION variable=m_state_is_full dim=1 complete
  m_state_t     m_state        [NB_HART];
#pragma HLS ARRAY_PARTITION variable=m_state         dim=1 complete
  from_m_to_w_t m_to_w;
  from_m_to_w_t w_from_m;
  bit_t         w_state_is_full[NB_HART];
#pragma HLS ARRAY_PARTITION variable=w_state_is_full dim=1 complete
  w_state_t     w_state        [NB_HART];
#pragma HLS ARRAY_PARTITION variable=w_state         dim=1 complete
  ...
```

10.3.4.4 The Has_exited Array Declaration

The has_exited array (see Listing 10.11) is used to detect the end of the run. At the start of the run, the set of running harts have their has_exited array entry cleared (the other harts have their entry set; see Listing 10.14).

When a thread reaches its ending RET instruction (i.e. a RET with a null return address), its has_exited array entry is set. When all the entries are set, the is_running variable is cleared (see Listing 10.40), which ends the do ... while loop in the multihart_ip top function.

Listing 10.11 The multihart_ip function local declarations (3)

```
  ...
  bit_t         has_exited     [NB_HART];
#pragma HLS ARRAY_PARTITION variable=has_exited      dim=1 complete
  bit_t         is_running;
  ...
```

10.3.4.5 The Lock Related Variable Declarations

The is_lock, is_unlock, i_hart, w_hart, i_destination, and w_destination variables (see Listing 10.12) are used to transmit the locking and unlocking arguments from the issue and writeback stages to the lock_unlock_update function.

The lock_unlock_update function is added to centralize the register locking (in the issue stage) and unlocking (in the writeback stage). This is to facilitate the job of the synthesizer.

Listing 10.12 The multihart_ip function local declarations (4)

```
  ...
  bit_t         is_lock;
  bit_t         is_unlock;
  reg_num_t     i_destination;
  reg_num_t     w_destination;
  hart_num_t    i_hart;
  hart_num_t    w_hart;
  counter_t     nbi;
  counter_t     nbc;
#ifndef __SYNTHESIS__
#ifdef DEBUG_REG_FILE
  hart_num_p1_t h1;
  hart_num_t    h;
#endif
#endif
  ...
```

10.3.4.6 The Centralization of the Lock/Unlock Operations

Figure 10.6 shows the lock_unlock_update function.

In the bottom left of the figure, the instruction from hart *ih* is issued in the issue stage. Its destination register *id* is locked by setting is_reg_computed[*ih*][*id*] in the lock_unlock_update function in the upper right of the figure.

Similarly, in the bottom right of the figure, the instruction from hart *wh* is written by the writeback stage. Its destination register *wd* is unlocked by clearing is_reg_computed[*wh*][*wd*] in the lock_unlock_update function in the upper right of the figure.

However, the lock_unlock_update function should not be confused with a pipeline stage. What is done in the function is run after the issue and writeback stages, in the same cycle. It ends both stages.

The lock_unlock_update function is presented in 10.3.12.

10.3.4.7 Initializations in the Top Function

Listing 10.13 shows the initializations of the top function before the main do ... while loop.

The has_exited array is initialized to identify the running harts (init_exit function).

For each pipeline stage *x*, its *x*_state_is_full array is initialized (init_*x*_state function).

The register file and the is_reg_computed array of register locks are initialized (init_file function).

The inter-stage links valid bits are cleared (no stage output is valid).

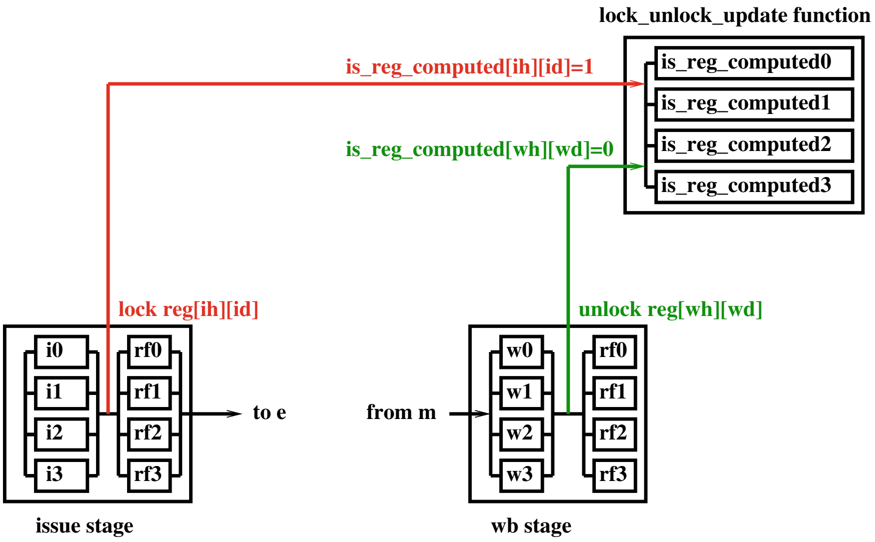


Fig. 10.6 The lock/unlock mechanism in the multihart pipeline

Listing 10.13 The initializations in the multihart_ip function

```
...
init_exit    (running_hart_set, has_exited);
init_f_state(running_hart_set, start_pc,
             f_state, f_state_is_full);
init_d_state(d_state_is_full);
init_i_state(i_state_is_full);
init_e_state(e_state_is_full);
init_m_state(m_state_is_full);
init_w_state(w_state_is_full);
init_file    (reg_file, is_reg_computed);
f_to_d.is_valid = 0;
d_to_f.is_valid = 0;
d_to_i.is_valid = 0;
i_to_e.is_valid = 0;
e_to_f.is_valid = 0;
e_to_m.is_valid = 0;
m_to_w.is_valid = 0;
nbi             = 0;
nbc             = 0;
...
```

10.3.4.8 The Init_exit Function

The init_exit function is shown in Listing 10.14 (it is located in the multihart_ip.cpp file).

Non running harts are marked as exited.

Listing 10.14 The init_exit function

```
static void init_exit(
  hart_set_t running_hart_set,
  bit_t     *has_exited){
  hart_num_p1_t h1;
  hart_num_t    h;
  bit_t         h_running;
  for (h1=0; h1<NB_HART; h1++){
#pragma HLS UNROLL
    h             = h1;
    h_running     = (running_hart_set>>h);
    has_exited[h] = !h_running;
  }
}
```

10.3.4.9 The Init_f_state Function

The init_f_state function (in fetch.cpp; see Listing 10.15) initializes the running harts according to the running hart set (running_hart_set argument of the IP prototype).

Running harts have a full starting fetch state (their f_state_is_full bit is set). They also have a start pc (their f_state entry is initialized with the start_pc transmitted to the IP).

Listing 10.15 The init_f_state function

```
void init_f_state(
  hart_set_t    running_hart_set,
  unsigned int *start_pc,
  f_state_t    *f_state,
  bit_t        *f_state_is_full){
  hart_num_p1_t h1;
  hart_num_t    h;
  bit_t         h_running;
  for (h1=0; h1<NB_HART; h1++){
#pragma HLS UNROLL
    h                            = h1;
    h_running                    = (running_hart_set>>h);
    f_state_is_full[h]           = h_running;
    f_state        [h].fetch_pc = start_pc[h];
  }
}
```

10.3.4.10 The Init_file Function

The init_file function (located in the multihart_ip.cpp file) is shown in Listing 10.16. It initializes reg_file and the is_reg_computed array of register locks.

All the registers are unlocked (is_reg_computed[h][r] is cleared).

The registers are cleared except registers a1 and sp.

Register a1 (x11) is initialized with its hart identifying number. This number can be useful for RISC-V programmers. For example, the register x11 of hart 2 receives the initial value 2.

The stack pointer register sp points on the next word after the hart data memory partition. As it is moving backward, the first word written to the stack is placed at the end of the hart partition. As a result, each hart has its own local stack.

Listing 10.16 The init_file function

```
//a1/x11 is set to the hart number
static void init_file(
  int    reg_file       [][NB_REGISTER],
  bit_t is_reg_computed[][NB_REGISTER]){
  hart_num_p1_t h1;
  hart_num_t    h;
  reg_num_p1_t  r1;
  reg_num_t     r;
  for (h1=0; h1<NB_HART; h1++){
#pragma HLS UNROLL
    h = h1;
    for (r1=0; r1<NB_REGISTER; r1++){
#pragma HLS UNROLL
      r = r1;
      is_reg_computed[h][r] = 0;
      if (r==11)
        reg_file     [h][r] = h;
      else if (r==SP)
        reg_file     [h][r] = (1<<(LOG_HART_DATA_RAM_SIZE+2));
      else
        reg_file     [h][r] = 0;
    }
  }
}
```

10.3.4.11 The Do ... While Loop

The main do ... while loop is shown in Listing 10.17. It starts with the new_cycle function which copies the _to_ variables into the _from_ ones.

The order of the pipeline stages goes from fetch to writeback to avoid a serialization from RAW dependencies on the accesses to the x_state_is_full arrays.

For example, the fetch function reads d_state_is_full, which is written by the decode function. With the call to fetch placed before the call to decode, there is no RAW dependency as the read precedes the write.

Listing 10.17 The do ... while loop in the multihart_ip function

```
  ...
  do {
#pragma HLS PIPELINE II=2
#pragma HLS LATENCY max=1
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("============================================\n");
    printf("cycle %d\n", (unsigned int)nbc);
#endif
#endif
    new_cycle(f_to_d, d_to_f, d_to_i, i_to_e, e_to_f, e_to_m,
              m_to_w, &f_from_d, &f_from_e, &d_from_f,
             &i_from_d, &e_from_i, &m_from_e, &w_from_m);
    statistic_update(e_from_i, &nbi, &nbc);
    running_cond_update(has_exited, &is_running);
    fetch(f_from_d, f_from_e, d_state_is_full,
          code_ram, f_state, &f_to_d, f_state_is_full);
    decode(d_from_f, i_state_is_full, d_state, &d_to_f,
          &d_to_i, d_state_is_full);
    issue(i_from_d, e_state_is_full, reg_file,
          is_reg_computed, i_state, &i_to_e, i_state_is_full,
         &is_lock, &i_hart, &i_destination);
    execute(e_from_i, m_state_is_full,
#ifndef __SYNTHESIS__
            reg_file,
#endif
            e_state, &e_to_f, &e_to_m, e_state_is_full);
    mem_access(m_from_e, w_state_is_full, data_ram, m_state,
              &m_to_w, m_state_is_full);
    write_back(w_from_m, reg_file, w_state, w_state_is_full,
              &is_unlock, &w_hart, &w_destination,
               has_exited);
    lock_unlock_update(is_lock, i_hart, i_destination,
                       is_unlock, w_hart, w_destination,
                       is_reg_computed);
  } while (is_running);
  ...
```

When the number of harts is four or eight, the main loop iteration duration is three FPGA cycles (to check this, you must synthesize and have a look at the Schedule Viewer; the synthesis time is rather long for eight harts: half an hour on my laptop).

However, the II value keeps to 2 and the multihart IP cycle stays at two FPGA cycles. This implies an overlapping of one cycle between two successive iterations, as shown in Fig. 10.7.

The overlapping does not cause any problem if everything used in the next iteration is set by the current iteration before the end of its second FPGA cycle. The synthesizer ensures that this condition is met (otherwise, it would raise an II violation).

10.3.5 The New_cycle Function to Copy the _to_ Structures Into the _from_ Ones

The new_cycle function in the new_cycle.cpp file copies the _to_ variables into the _from_ ones (see Listing 10.18).

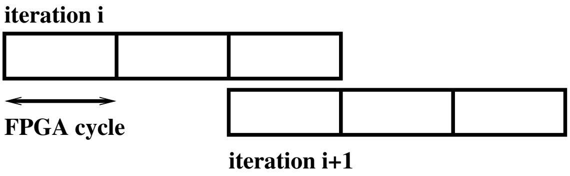


Fig. 10.7 Overlapping of two successive iterations with II=2 and iteration latency=3 FPGA cycles

Listing 10.18 The new_cycle function

```
void new_cycle(
  from_f_to_d_t  f_to_d,
  from_d_to_f_t  d_to_f,
  from_d_to_i_t  d_to_i,
  from_i_to_e_t  i_to_e,
  from_e_to_f_t  e_to_f,
  from_e_to_m_t  e_to_m,
  from_m_to_w_t  m_to_w,
  from_d_to_f_t *f_from_d,
  from_e_to_f_t *f_from_e,
  from_f_to_d_t *d_from_f,
  from_d_to_i_t *i_from_d,
  from_i_to_e_t *e_from_i,
  from_e_to_m_t *m_from_e,
  from_m_to_w_t *w_from_m){
  *f_from_d = d_to_f;
  *f_from_e = e_to_f;
  *d_from_f = f_to_d;
  *i_from_d = d_to_i;
  *e_from_i = i_to_e;
  *m_from_e = e_to_m;
  *w_from_m = m_to_w;
}
```

10.3.6 The Multihart Fetch Stage

10.3.6.1 The Fetch Function

The fetch function (in the fetch.cpp file) is shown in Listing 10.19.

The fetch stage starts by selecting the fetching hart for the current cycle (call to the select_hart function).

The select_hart function sets the Boolean is_selected which indicates if at least one hart is ready.

The function chooses the highest priority ready hart and the selected hart index is saved in the selected_hart variable (hart 0 has the highest priority). This is common to all of the pipeline stages, even though the selection procedure is specific to each stage.

In parallel, the save_input_from_d and save_input_from_e functions are run to save the input instructions in the f_state array if the matching inputs are valid (f_from_d.is_valid and f_from_e.is_valid). The f_state_is_full array entries are set (f_state_is_full[f_from_d.hart] and f_state_is_full[f_from_e.hart] are both set if the matching inputs are valid).

Notice that the stage may receive two inputs in the same cycle. However, they may not belong to the same hart: if an input is received for hart *h* from the execute stage, it means that hart *h* has decoded a BRANCH or JALR instruction a few cycles ago and since then, it is stopped. The decode stage may not at the same time send anything to the fetch stage for hart *h*, but it may send something for a different hart.

If is_selected is set (i.e. the select_hart function has found a selectable hart in the f_state array) the is_selected and selected_hart values are copied in the is_fetching and fetching_hart variables.

If no selectable hart is found in the state array, is_fetching may still be set if there is a valid input. Then, the fetching_hart is set as the inputting hart (f_from_d.hart if f_from_d.is_valid is set, f_from_e.hart otherwise; the input from the decode stage has priority over the input of the execute stage).

An is_fetching bit set indicates that a fetch is done in the current cycle. The fetching_hart value is the hart number of the fetching hart. The matching entry in the f_state_is_full array is cleared.

The fetching hart fetches in the stage_job function. The fetched instruction is transmitted to the decode stage (set_output_to_d function and f_to_d->is_valid).

Listing 10.19 The fetch function

```
void fetch(
  from_d_to_f_t   f_from_d,
  from_e_to_f_t   f_from_e,
  bit_t          *d_state_is_full,
  instruction_t *code_ram,
  f_state_t      *f_state,
  from_f_to_d_t *f_to_d,
  bit_t          *f_state_is_full){
  bit_t       is_selected;
  hart_num_t selected_hart;
  bit_t       is_fetching;
  hart_num_t fetching_hart;
  select_hart(f_state_is_full, d_state_is_full,
             &is_selected,  &selected_hart);
  if (f_from_d.is_valid){
    f_state_is_full[f_from_d.hart] = 1;
    save_input_from_d(f_from_d, f_state);
  }
  if (f_from_e.is_valid){
    f_state_is_full[f_from_e.hart] = 1;
    save_input_from_e(f_from_e, f_state);
  }
  is_fetching    =
    is_selected                                           ||
   (f_from_d.is_valid && !d_state_is_full[f_from_d.hart]) ||
   (f_from_e.is_valid && !d_state_is_full[f_from_e.hart]);
  fetching_hart =
   (is_selected)?selected_hart:
   (f_from_d.is_valid && !d_state_is_full[f_from_d.hart])?
    f_from_d.hart:f_from_e.hart;
  if (is_fetching){
    f_state_is_full[fetching_hart] = 0;
    stage_job(fetching_hart, f_state, code_ram);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("hart %d: fetched  ", (int)fetching_hart);
    printf("%04d: %08x      \n",
      (int)(f_state[fetching_hart].fetch_pc<<2),
            f_state[fetching_hart].instruction);
#endif
#endif
    set_output_to_d(fetching_hart, f_state, f_to_d);
  }
  f_to_d->is_valid = is_fetching;
}
```

10.3.6.2 Hart Selection in the Fetch Stage

The select_hart function in the fetch.cpp file (see Listings 10.20 and 10.21) is built as a set of successive #if ... #endif sections. According to the value of the NB_HART constant, a binary tree of height LOG_NB_HART computes the is_selected Boolean value indicating if a ready hart has been found to fetch.

In a first step (see Listing 10.20), a set of NB_HART conditions *c[h]* are computed in parallel. The *c[h]* condition is set if f_state_is_full[*h*] is set, i.e. the fetch stage state contains a valid pc for hart *h*, and if d_state_is_full[*h*] is clear, i.e. d_state[*h*] is empty (i.e. the fetched instruction can be moved to the decode stage).

Listing 10.20 The fetch stage select_hart function: c conditions

```
void select_hart(
  bit_t      *f_state_is_full,
  bit_t      *d_state_is_full,
  bit_t      *is_selected,
  hart_num_t *selected_hart){
  bit_t c[NB_HART];
  c[0] = (f_state_is_full[0] && !d_state_is_full[0]);
#if (NB_HART>1)
  c[1] = (f_state_is_full[1] && !d_state_is_full[1]);
#endif
#if (NB_HART>2)
  c[2] = (f_state_is_full[2] && !d_state_is_full[2]);
  c[3] = (f_state_is_full[3] && !d_state_is_full[3]);
#endif
#if (NB_HART>4)
  c[4] = (f_state_is_full[4] && !d_state_is_full[4]);
  c[5] = (f_state_is_full[5] && !d_state_is_full[5]);
  c[6] = (f_state_is_full[6] && !d_state_is_full[6]);
  c[7] = (f_state_is_full[7] && !d_state_is_full[7]);
#endif
  ...
```

In a second step (see Listing 10.21), the *c[h]* Boolean values are ORed in a binary tree to form the is_selected value (i.e. if at least one hart *h* has its *c[h]* condition set, is_selected is set).

The selected_hart is set as the first one having its *c[h]* condition set. Hence, harts are priority ordered (hart 0 has the highest priority).

Listing 10.21 The fetch stage select_hart function: selected_hart and is_selected

```
  ...
#if   (NB_HART<2)
  *selected_hart =  0;
  *is_selected   =  c[0];
#elif (NB_HART<3)
  *selected_hart = (c[0])?0:1;
  *is_selected   = (c[0] || c[1]);
#elif (NB_HART<5)
  hart_num_t h01, h23;
  bit_t      c01, c23;
  h01 = (c[0])?0:1;
  c01 = (c[0] || c[1]);
  h23 = (c[2])?2:3;
  c23 = (c[2] || c[3]);
  *selected_hart = (c01)?h01:h23;
  *is_selected   = (c01 || c23);
#elif (NB_HART<9)
  hart_num_t h01, h23, h45, h67, h03, h47;
  bit_t      c01, c23, c45, c67, c03, c47;
  h01 = (c[0])?0:1;
  c01 = (c[0] || c[1]);
  h23 = (c[2])?2:3;
  c23 = (c[2] || c[3]);
  h45 = (c[4])?4:5;
  c45 = (c[4] || c[5]);
  h67 = (c[6])?6:7;
  c67 = (c[6] || c[7]);
  h03 = (c01)?h01:h23;
  h47 = (c45)?h45:h67;
  c03 = (c01 || c23);
  c47 = (c45 || c67);
  *selected_hart = (c03)?h03:h47;
  *is_selected   = (c03 || c47);
#endif
}
```

10.3.7 The Decode Stage

The decode function in the decode.cpp file (see Listing 10.22) works the same way as the fetch function. It starts by a hart selection (a call to the select_hart function; the selection is similar to the fetch stage one).

In parallel, the input from the fetch stage is saved in the state array (a call to the save_input_from_f function).

When a decoding hart has been selected, its instruction is decoded (in the stage_job function).

The fields of the output structures to be sent to the fetch stage and to the issue stage are filled (set_output_to_f and set_output_to_i functions).

The output to the fetch stage is valid if an instruction has been decoded and if it is neither a BRANCH nor a JALR.

The output to the issue stage is valid if an instruction has been decoded.

Listing 10.22 The decode function

```
void decode(
  from_f_to_d_t  d_from_f,
  bit_t         *i_state_is_full,
  d_state_t     *d_state,
  from_d_to_f_t *d_to_f,
  from_d_to_i_t *d_to_i,
  bit_t         *d_state_is_full){
  bit_t      is_selected;
  hart_num_t selected_hart;
  bit_t      is_decoding;
  hart_num_t decoding_hart;
  select_hart(d_state_is_full, i_state_is_full,
             &is_selected, &selected_hart);
  if (d_from_f.is_valid){
    d_state_is_full[d_from_f.hart] = 1;
    save_input_from_f(d_from_f, d_state);
  }
  is_decoding    =
    is_selected ||
   (d_from_f.is_valid && !i_state_is_full[d_from_f.hart]);
  decoding_hart =
   (is_selected)?selected_hart:d_from_f.hart;
  if (is_decoding){
    d_state_is_full[decoding_hart] = 0;
    stage_job(decoding_hart, d_state);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("hart %d: decoded  %04d: ",
                (int)decoding_hart,
                (int)(d_state[decoding_hart].fetch_pc<<2));
    disassemble(d_state[decoding_hart].fetch_pc,
                d_state[decoding_hart].instruction,
                d_state[decoding_hart].d_i);
    if (d_state[decoding_hart].d_i.is_jal)
      printf("      pc  = %16d (%8x)\n",
          (int)(d_state[decoding_hart].relative_pc<<2),
 (unsigned int)(d_state[decoding_hart].relative_pc<<2));
#endif
#endif
    set_output_to_f(decoding_hart, d_state, d_to_f);
    set_output_to_i(decoding_hart, d_state, d_to_i);
  }
  d_to_f->is_valid =
    is_decoding                                &&
   !d_state[decoding_hart].d_i.is_branch &&
   !d_state[decoding_hart].d_i.is_jalr;
  d_to_i->is_valid = is_decoding;
}
```

10.3.8 The Issue Stage

10.3.8.1 The Hart Selection in the Issue Function

The issue stage (see the issue function in Listing 10.23, located in the issue.cpp file) selects a hart to issue among the full entries in the i_state array (select_hart).

In parallel, it saves the input sent by the decode stage (save_input_from_d) if i_from_d.is_valid is set.

An instruction is issued if the hart selection is successful or else if the instruction input from the decode stage is issuable (i.e. the wait_12 field is clear, meaning that the two sources are unlocked; see Listing 10.27).

Listing 10.23 The issue function: hart selection

```
void issue(
  from_d_to_i_t   i_from_d,
  bit_t          *e_state_is_full,
  int             reg_file       [][NB_REGISTER],
  bit_t           is_reg_computed[][NB_REGISTER],
  i_state_t      *i_state,
  from_i_to_e_t  *i_to_e,
  bit_t          *i_state_is_full,
  bit_t          *is_lock,
  hart_num_t     *i_hart,
  reg_num_t      *i_destination){
  bit_t      is_selected;
  hart_num_t selected_hart;
  bit_t      is_issuing;
  hart_num_t issuing_hart;
  select_hart(i_state, i_state_is_full, e_state_is_full,
              is_reg_computed, &is_selected, &selected_hart);
```

```
    if (i_from_d.is_valid){
      i_state_is_full[i_from_d.hart] = 1;
      save_input_from_d(i_from_d, is_reg_computed, i_state);
    }
    is_issuing   =
      is_selected ||
     (i_from_d.is_valid && !e_state_is_full[i_from_d.hart] &&
     !i_state[i_from_d.hart].wait_12);
    issuing_hart =
     (is_selected)?selected_hart:i_from_d.hart;
    ...
```

10.3.8.2 The Issue Stage Job in the Issue Function

When an instruction is issued, its destination register is locked (see Listing 10.24). The locking is prepared in the issue function but it is done in the lock_unlock_update function.

The issue stage sends the issuing hart (i_hart) and the destination register (i_destination) to the lock_unlock_update function.

If an instruction is issued, the issuing hart i_state entry is emptied (i_state_is_full [issuing_hart] is cleared).

The issue stage_job function reads the sources from the register file and saves them into the i_state array entry.

The issuing hart sends the sources read in the register file to the execute stage (set_output_to_e function).

Listing 10.24 The issue stage job in the issue function

```
    ...
    *is_lock =
      is_issuing && !i_state[issuing_hart].d_i.has_no_dest;
    if (!i_state[issuing_hart].d_i.has_no_dest){
      *i_hart        = issuing_hart;
      *i_destination = i_state[issuing_hart].d_i.rd;
    }
    if (is_issuing){
      i_state_is_full[issuing_hart] = 0;
      stage_job(issuing_hart, i_state, reg_file);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
      printf("hart %d: issued   ", (int)issuing_hart);
      printf("%04d\n",
              (int)(i_state[issuing_hart].fetch_pc<<2));
#endif
#endif
      set_output_to_e(issuing_hart, i_state, i_to_e);
    }
    i_to_e->is_valid = is_issuing;
}
```

10.3.8.3 Selecting a Hart for Issue: The Select_hart Function Declarations

The issue stage select_hart function in the issue.cpp file (see Listing 10.25) defines four arrays of NB_HART Boolean values.

The is_locked_1 array indicates which are the harts holding an instruction with an unlocked rs1 source. The is_locked_2 array is related to the rs2 source.

The wait_12 array indicates which are the harts holding an instruction with no source locked, i.e. ready to be issued.

Listing 10.25 The issue stage select_hart function prototype and declarations

```
static void select_hart(
  i_state_t   *i_state,
  bit_t       *i_state_is_full,
  bit_t       *e_state_is_full,
  bit_t        is_reg_computed[][NB_REGISTER],
  bit_t       *is_selected,
  hart_num_t  *selected_hart){
  bit_t c          [NB_HART];
  bit_t is_locked_1[NB_HART];
  bit_t is_locked_2[NB_HART];
  bit_t wait_12    [NB_HART];
  ...
```

10.3.8.4 Selecting a Hart for Issue: The c Condition Computations

The c[*h*] condition is set (see Listing 10.26) if the hart *h* state entry is full, if the next stage hart *h* state entry is empty, and if the sources are not locked (i.e. wait_12[*h*] is clear).

Listing 10.26 The select_hart computation of the c condition for hart 0

```
  ...
  is_locked_1      [0]  =
    i_state        [0].d_i.is_rs1_reg   &&
    is_reg_computed[0][i_state[0].d_i.rs1];
  is_locked_2      [0]  =
    i_state        [0].d_i.is_rs2_reg   &&
    is_reg_computed[0][i_state[0].d_i.rs2];
  wait_12          [0]  =
    is_locked_1[0] || is_locked_2[0];
  c[0] = (i_state_is_full[0] && !e_state_is_full[0] && !wait_12[0])
      ;
  ...
```

If there are two harts, a second c[1] condition is computed the same way for hart 1.

If there are four harts, three more c[1] to c[3] conditions are computed the same way for harts 1 to 3.

If there are eight harts, seven more c[1] to c[7] conditions are computed the same way for harts 1 to 7.

The last part of the select_hart function in the issue stage computes the final is_selected and selected_hart values. The code is the same as the one in the select_hart function in the fetch stage (refer back to Listing 10.21).

10.3.8.5 Saving the Issue Stage Input

The save_input_from_d function in the issue.cpp file (see Listing 10.27) saves the instruction input from the decode stage into the i_state array.

It computes i_state[hart].wait_12. This Boolean value is set if the instruction input from the decode stage has some locked source register. If so, the instruction may not be selected for issue.

Listing 10.27 The save_input_from_d function

```
static void save_input_from_d(
  from_d_to_i_t i_from_d,
  bit_t         is_reg_computed[][NB_REGISTER],
  i_state_t    *i_state){
  hart_num_t hart;
  bit_t      is_locked_1;
  bit_t      is_locked_2;
  hart                        = i_from_d.hart;
  i_state[hart].fetch_pc      = i_from_d.fetch_pc;
  i_state[hart].d_i           = i_from_d.d_i;
  i_state[hart].relative_pc   = i_from_d.relative_pc;
  is_locked_1 =
    i_from_d.d_i.is_rs1_reg    &&
    is_reg_computed[hart][i_from_d.d_i.rs1];
  is_locked_2 =
    i_from_d.d_i.is_rs2_reg    &&
    is_reg_computed[hart][i_from_d.d_i.rs2];
  i_state[hart].wait_12       =
    is_locked_1 || is_locked_2;
#ifndef __SYNTHESIS__
  i_state[hart].instruction   = i_from_d.instruction;
#endif
}
```

10.3.9 The Execute Stage

The execute function in the execute.cpp file is shown in Listing 10.28.

It has the same structure as the other stage implementation functions: hart selection, input from the issue stage, stage job, and output filling.

Listing 10.28 The execute function

```
void execute(
  from_i_to_e_t  e_from_i,
  bit_t         *m_state_is_full,
#ifndef __SYNTHESIS__
  int            reg_file[][NB_REGISTER],
#endif
  e_state_t     *e_state,
  from_e_to_f_t *e_to_f,
  from_e_to_m_t *e_to_m,
  bit_t         *e_state_is_full){
  bit_t          is_selected;
  hart_num_t     selected_hart;
  bit_t          is_executing;
  hart_num_t     executing_hart;
  bit_t          bcond;
  int            result1;
  int            result2;
  code_address_t computed_pc;
  select_hart(e_state_is_full, m_state_is_full,
             &is_selected, &selected_hart);
  if (e_from_i.is_valid){
    e_state_is_full[e_from_i.hart] = 1;
    save_input_from_i(e_from_i, e_state);
  }
  is_executing   =
    is_selected ||
   (e_from_i.is_valid && !m_state_is_full[e_from_i.hart]);
  executing_hart =
   (is_selected)?selected_hart:e_from_i.hart;
  if (is_executing){
    e_state_is_full[executing_hart] = 0;
    compute  (executing_hart, e_state, &bcond, &result1,
             &result2, &computed_pc);
    stage_job(executing_hart, e_state,  bcond, computed_pc);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("hart %d: execute  ", (int)executing_hart);
    printf("%04d\n", (int)(e_state[executing_hart].fetch_pc<<2));
    if (e_state[executing_hart].d_i.is_branch ||
        e_state[executing_hart].d_i.is_jalr)
       emulate(executing_hart, reg_file,
               e_state[executing_hart].d_i,
               e_state[executing_hart].target_pc);
#endif
#endif
    set_output_to_f(executing_hart, e_state, e_to_f);
    set_output_to_m(executing_hart, result1, result2,
                    computed_pc, e_state, e_to_m);
  }
  //block fetch after last RET
  //(i.e. RET with 0 return address)
  e_to_f->is_valid =
   is_executing && e_state[executing_hart].is_target;
  e_to_m->is_valid = is_executing;
}
```

10.3.10 The Memory Access Stage

The mem_access function in the mem_access.cpp file which implements the memory access stage is shown in Listings 10.29 and 10.31.

10.3.10.1 The Hart Selection in the Mem_access Function

A hart is selected to process a ready instruction (see Listing 10.29). It is either the select_hart function selection or the instruction input from the execute stage (save_input_from_e).

Listing 10.29 The mem_access function: computing the is_accessing and accessing_hart values

```
void mem_access(
  from_e_to_m_t  m_from_e,
  bit_t         *w_state_is_full,
  int            data_ram[][HART_DATA_RAM_SIZE],
  m_state_t     *m_state,
  from_m_to_w_t *m_to_w,
  bit_t         *m_state_is_full){
  bit_t      is_selected;
  hart_num_t selected_hart;
  bit_t      is_accessing;
  hart_num_t accessing_hart;
  select_hart(m_state_is_full, w_state_is_full,
             &is_selected, &selected_hart);
  if (m_from_e.is_valid){
    m_state_is_full[m_from_e.hart] = 1;
    save_input_from_e(m_from_e, m_state);
  }
  is_accessing   =
    is_selected ||
   (m_from_e.is_valid && !w_state_is_full[m_from_e.hart]);
  accessing_hart =
   (is_selected)?selected_hart:m_from_e.hart;
  ...
```

10.3.10.2 The Save_input_from_e Function

The save_input_from_e function in the mem_access.cpp file (see Listing 10.30) saves the incoming instruction sent by the execute stage.

The function computes the accessed_h and is_local fields in the m_state array.

The accessed_h value is the accessed memory partition number.

The is_local bit indicates if the access is inside the memory partition of the accessing hart.

Listing 10.30 The save_input_from_e function: saving the input instruction and computing the accessed_h and is_local fields

```
static void save_input_from_e(
  from_e_to_m_t m_from_e,
  m_state_t    *m_state){
  hart_num_t hart;
  hart                      = m_from_e.hart;
  m_state[hart].rd          = m_from_e.rd;
  m_state[hart].has_no_dest = m_from_e.has_no_dest;
  m_state[hart].is_load     = m_from_e.is_load;
  m_state[hart].is_store    = m_from_e.is_store;
  m_state[hart].func3       = m_from_e.func3;
  m_state[hart].is_ret      = m_from_e.is_ret;
  m_state[hart].address     = m_from_e.address;
  m_state[hart].value       = m_from_e.value;
  m_state[hart].accessed_h  =
    (m_from_e.address>>(LOG_HART_DATA_RAM_SIZE+2)) + hart;
  m_state[hart].is_local    = (hart == m_state[hart].accessed_h);
#ifndef __SYNTHESIS__
  m_state[hart].fetch_pc    = m_from_e.fetch_pc;
  m_state[hart].instruction = m_from_e.instruction;
  m_state[hart].d_i         = m_from_e.d_i;
  m_state[hart].target_pc   = m_from_e.target_pc;
#endif
}
```

10.3.10.3 The Memory Access in the Mem_access Function

If an instruction has been selected, the mem_access function in the mem_access.cpp file (see Listing 10.31) calls stage_job.

Notice that the hart value transmitted to the stage_job function is not the accessing hart number, but the identifier of the accessed partition (m_state[accessing_hart].accessed_h).

The set_output_to_w function fills the m_to_w structure.

Listing 10.31 The mem_access function: the memory access in the stage_job function and the transmission of the instruction to the writeback stage with the set_output_to_w function

```
  ...
  if (is_accessing){
    m_state_is_full[accessing_hart] = 0;
    stage_job(m_state[accessing_hart].accessed_h,
              m_state[accessing_hart].is_load,
              m_state[accessing_hart].is_store,
              m_state[accessing_hart].address,
              m_state[accessing_hart].func3, data_ram,
             &m_state[accessing_hart].value);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("hart %d: mem       ", (int)accessing_hart);
    printf("%04d\n",
          (int)(m_state[accessing_hart].fetch_pc<<2));
#endif
#endif
    set_output_to_w(accessing_hart, m_state, m_to_w);
  }
  m_to_w->is_valid = is_accessing;
}
```

10.3.10.4 The Memory Access Stage_job Function

When the instruction is a load or a store, the stage_job function in the mem_access.cpp file (see Listing 10.32) calls either the mem_load or the mem_store function to access the memory. Otherwise, the stage_job function just returns.

Listing 10.32 The stage_job function

```
static void stage_job(
  hart_num_t       hart,
  bit_t            is_load,
  bit_t            is_store,
  b_data_address_t address,
  func3_t          func3,
  int              data_ram[][HART_DATA_RAM_SIZE],
  int             *value){
  if (is_load)
    *value = mem_load(hart, data_ram, address, func3);
  else if (is_store)
    mem_store(hart, data_ram, address, *value,
             (ap_uint<2>)func3);
}
```

10.3.10.5 The Mem_store Function

The mem_store function in the mem.cpp file (see Listing 10.33) selects the bytes that have to be written to memory according to the store size.

The address used for the access is the untransformed one (the relative address computed in the execute stage).

It is truncated to become an offset into a hart partition (a, a1, or a2 according to the size of the access, i.e. a byte for an SB instruction, a half word for an SH instruction or a full word for an SW instruction; the address is clipped to keep only the low order bits which address the accessed partition, e.g. LOG_HART_DATA_RAM_SIZE bits for a word access).

Then the accessed hart partition offset is added to form the final absolute address (e.g. (((b_data_address_t)hart)<<(LOG_HART_DATA_RAM_SIZE+2)) displacement for a byte access or data_ram[hart][a2] for a word access; remember that argument hart is not the accessing hart but the accessed partition).

Listing 10.33 The mem_store function in the mem.cpp file

```
void mem_store(
  hart_num_t        hart,
  int               data_ram[][HART_DATA_RAM_SIZE],
  b_data_address_t  address,
  int               value,
  ap_uint<2>        msize){
  w_hart_data_address_t a2  = address>>2;
  h_hart_data_address_t a1  = address>>1;
  b_hart_data_address_t a   = address;
  char                  value_0  = value;
  short                 value_01 = value;
  switch(msize){
    case SB:
      *((char*) (data_ram) +
              (((b_data_address_t)hart)<<
                (LOG_HART_DATA_RAM_SIZE+2)) + a)
                        = value_0;
      break;
    case SH:
      *((short*)(data_ram) +
              (((h_data_address_t)hart)<<
                (LOG_HART_DATA_RAM_SIZE+1)) + a1)
                        = value_01;
      break;
    case SW:
      data_ram[hart][a2] = value;
      break;
    case 3:
      break;
  }
}
```

10.3.10.6 The Mem_load Function

The mem_load function in the mem.cpp file (see Listing 10.34) reads a full word from the accessed partition (argument hart).

Listing 10.34 The mem_load function: reading the accessed word

```
int mem_load(
  hart_num_t       hart,
  int              data_ram[][HART_DATA_RAM_SIZE],
  b_data_address_t address,
  func3_t          msize){
  w_hart_data_address_t a2  = (address >> 2);
  ap_uint<2>            a01 =  address;
  bit_t                 a1  = (address >> 1);
  int                   result;
  char                  b, b0, b1, b2, b3;
  unsigned char         ub, ub0, ub1, ub2, ub3;
  short                 h, h0, h1;
  unsigned short        uh, uh0, uh1;
  int                   ib, ih;
  unsigned int          iub, iuh;
  int                   w;
  w = data_ram[hart][a2];
  ...
```

The mem_load function selects the requested bytes out of the read word (unchanged from rv32i_npp_ip; see Listing 6.11).

10.3.11 The Writeback Stage

10.3.11.1 The Hart Selection in the Write_back Function

The write_back function in the wb.cpp file (see Listing 10.35) selects the writing hart.

Listing 10.35 The write_back function: computing the is_writing and the writing_hart values

```
void write_back(
  from_m_to_w_t w_from_m,
  int            reg_file[][NB_REGISTER],
  w_state_t    *w_state,
  bit_t        *w_state_is_full,
  bit_t        *is_unlock,
  hart_num_t   *w_hart,
  reg_num_t    *w_destination,
  bit_t        *has_exited){
  bit_t      is_selected;
  hart_num_t selected_hart;
  bit_t      is_writing;
  hart_num_t writing_hart;
  select_hart(w_state_is_full,
             &is_selected, &selected_hart);
  if (w_from_m.is_valid){
    w_state_is_full[w_from_m.hart] = 1;
    save_input_from_m(w_from_m, w_state);
  }
  is_writing   =
    is_selected || w_from_m.is_valid;
  writing_hart =
   (is_selected)?selected_hart:w_from_m.hart;
  ...
```

10.3.11.2 The Register Unlocking in the Write_back Function

As the issue function, the write_back function (see Listing 10.36) sends to the lock_unlock_update function the hart and destination register to unlock through the is_unlock, w_hart, and w_destination variables.

Listing 10.36 The write_back function: computing the is_lock, w_hart, and w_destination values

```
  ...
  *is_unlock = is_writing &&
              !w_state[writing_hart].has_no_dest;
  if (!w_state[writing_hart].has_no_dest){
    *w_hart        = writing_hart;
    *w_destination = w_state[writing_hart].rd;
  }
  ...
```

10.3.11.3 The Write_back Function Job

The write_back function (see Listing 10.37) writes the value computed in the execute stage or loaded in the memory access stage into the destination register (stage_job function).

Listing 10.37 The write_back function: writing to the destination register in the stage_job function

```
  ...
  if (is_writing){
    w_state_is_full[writing_hart] = 0;
    stage_job(writing_hart, w_state, reg_file, has_exited);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("hart %d: wb       ", (int)writing_hart);
    printf("%04d\n", (int)(w_state[writing_hart].fetch_pc<<2));
    if (!w_state[writing_hart].d_i.is_branch &&
        !w_state[writing_hart].d_i.is_jalr)
      emulate(writing_hart, reg_file,
              w_state[writing_hart].d_i,
              w_state[writing_hart].target_pc);
#else
#ifdef DEBUG_FETCH
    printf("hart %d: %04d: %08x      ",
      (int)writing_hart,
      (int)(w_state[writing_hart].fetch_pc<<2),
            w_state[writing_hart].instruction);
#ifndef DEBUG_DISASSEMBLE
    printf("\n");
#endif
#endif
#ifdef DEBUG_DISASSEMBLE
    disassemble(w_state[writing_hart].fetch_pc,
                w_state[writing_hart].instruction,
                w_state[writing_hart].d_i);
#endif
#ifdef DEBUG_EMULATE
    printf("hart %d: ", (int)writing_hart);
    emulate(writing_hart, reg_file,
            w_state[writing_hart].d_i,
            w_state[writing_hart].target_pc);
#endif
#endif
#endif
  }
}
```

10.3.11.4 The Writeback Stage_job Function

In the stage_job function in the wb.cpp file (see Listing 10.38), the register file is updated.

The has_exited array is updated too. If the concerned hart has run a RET instruction with a null return address, the hart is exiting and its has_exited array entry is set.

Listing 10.38 The write_back stage stage_job function

```
static void stage_job(
  hart_num_t hart,
  w_state_t *w_state,
  int        reg_file[][NB_REGISTER],
  bit_t      *has_exited){
  if (!w_state[hart].has_no_dest)
    reg_file[hart][w_state[hart].rd] = w_state[hart].value;
  if (w_state[hart].is_ret && w_state[hart].value == 0)
    has_exited[hart] = 1;
}
```

10.3.12 The Lock_unlock_update Function

The lock_unlock_update function in the multihart_ip.cpp file (see Listing 10.39) groups the updates of the is_reg_computed array.

This is to simplify the job of the synthesizer. To furthermore help synthesis, the code is a bit redundant to emphasize the fact that the same entry of the array may not be locked and unlocked in the same cycle, i.e. the same register may not be locked and simultaneously unlocked (locking and unlocking the same register would write a 0 and a 1 in the same location).

Listing 10.39 The lock_unlock_update function

```
static void lock_unlock_update(
  bit_t      is_lock,
  hart_num_t i_hart,
  reg_num_t  i_destination,
  bit_t      is_unlock,
  hart_num_t w_hart,
  reg_num_t  w_destination,
  bit_t      is_reg_computed[][NB_REGISTER]){
  //complicate, but necessary to help the synthesizer
  //by excluding the possibility to write 1 and 0 in the same
  //array entry; drastically shortens the time for synthesis
  if (is_lock && !is_unlock)
    is_reg_computed[i_hart][i_destination] = 1;
  else if (is_unlock && !is_lock)
    is_reg_computed[w_hart][w_destination] = 0;
  else if (is_lock && is_unlock && ((i_hart != w_hart) ||
           (i_destination != w_destination))){
    is_reg_computed[i_hart][i_destination] = 1;
    is_reg_computed[w_hart][w_destination] = 0;
  }
}
```

10.3.13 The Running_cond_update Function

The running_cond_update function in the multihart_ip.cpp file is shown in Listing 10.40.

The run ends when all the harts have exited. The synthesizer is able to unroll the NB_HART iterations of the loop and to balance the bitwise OR expression, i.e. organize it as a perfect binary tree (when NB_HART is a power of 2).

Listing 10.40 The running_cond_update function

```
static void running_cond_update(
  bit_t *has_exited,
  bit_t *is_running){
  hart_num_p1_t h1;
  hart_num_t    h;
  bit_t         cond;
  cond = 0;
  for (h1=0; h1<NB_HART; h1++){
#pragma HLS UNROLL
    h = h1;
    cond = cond | !has_exited[h];
  }
  *is_running = cond;
}
```

10.4 Simulating the Multihart_ip

Experimentation

To simulate the multihart_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with multihart_ip. There are two testbench programs: testbench_seq_multihart_ip.cpp to run independent codes (one per hart) and testbench_par_multihart_ip.cpp to run a parallel sum of the elements of an array.
With testbench_seq_multihart_ip.cpp you can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder. You can also vary the number of harts.

Two different testbench files are provided. One is to run the set of test codes (from test_branch.s to test_sum.s). The other is to run a distributed parallel sum of the elements of an array.

The first testbench is in the testbench_seq_multihart_ip.cpp file. Each hart runs a copy of the selected test code. All the harts are set as active at the start of the run and they have the same start address (start_pc[h]=0).

The test codes can be built with the build_seq.sh shell script shown in Listing 10.41. The script needs the file name of the test code to build (argument $1; for example "./build_seq.sh test_mem").

Listing 10.41 The build_seq.sh shell script

```
$ cat build_seq.sh
riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 -Tdata 0 -o $1.elf
    $1.s
riscv32-unknown-elf-objcopy -O binary --only-section=.text $1.elf
    $1_text.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' $1_text.bin > $1_text.hex
$
```

10.4.1 Filling Harts with Independent Codes

The testbench_seq_multihart_ip.cpp file is shown in Listing 10.42.

Listing 10.42 The testbench_seq_multihart_ip.cpp file

```
#include <stdio.h>
#include "multihart_ip.h"
int          data_ram[NB_HART][HART_DATA_RAM_SIZE];
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_mem_text.hex"
};
unsigned int start_pc[NB_HART];
int main() {
  unsigned int nbi;
  unsigned int nbc;
  int          w;
  for (int i=0; i<NB_HART; i++) start_pc[i] = 0;
  multihart_ip((1<<NB_HART)-1,//start all harts
               start_pc, code_ram, data_ram, &nbi, &nbc);
  printf("%d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", nbi, nbc, ((float)nbi)/nbc);
  for (int h=0; h<NB_HART; h++){
    printf("hart %d data memory dump (non null words)\n", h);
    for (int i=0; i<HART_DATA_RAM_SIZE; i++){
      w = data_ram[h][i];
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
        (int)(4*(i+(((w_data_address_t)h)
             <<LOG_HART_DATA_RAM_SIZE))), w, (unsigned int)w);
    }
  }
  return 0;
}
```

For the run of two copies of test_mem.s (LOG_NB_HART is set to 1 in multihart_ip.h), the output is shown in Listing 10.43.

Listing 10.43 The output of the main function of the testbench_seq_multihart_ip.cpp file when test_mem_text.hex is included

```
hart 0: 0000: 00000513      li a0, 0
hart 0:       a0  =                0 (        0)
hart 1: 0000: 00000513      li a0, 0
hart 1:       a0  =                0 (        0)
hart 0: 0004: 00000593      li a1, 0
hart 0:       a1  =                0 (        0)
hart 1: 0004: 00000593      li a1, 0
hart 1:       a1  =                0 (        0)
...
```

```
hart 0: 0056: 00a62223      sw a0, 4(a2)
hart 0:       m[      2c] =               55 (      37)
hart 1: 0056: 00a62223      sw a0, 4(a2)
hart 1:       m[   2002c] =               55 (      37)
hart 0: 0060: 00008067      ret
hart 0:       pc  =                0 (       0)
hart 1: 0060: 00008067      ret
hart 1:       pc  =                0 (       0)
register file for hart 0
...
a0  =               55 (      37)
a1  =                0 (       0)
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
register file for hart 1
...
a0  =               55 (      37)
a1  =                0 (       0)
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
176 fetched and decoded instructions in 306 cycles (ipc = 0.58)
hart 0 data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
hart 1 data memory dump (non null words)
m[20000] =                1 (       1)
m[20004] =                2 (       2)
m[20008] =                3 (       3)
m[2000c] =                4 (       4)
m[20010] =                5 (       5)
m[20014] =                6 (       6)
m[20018] =                7 (       7)
m[2001c] =                8 (       8)
m[20020] =                9 (       9)
m[20024] =               10 (       a)
m[2002c] =               55 (      37)
```

10.4.2 Filling Harts with a Parallelized Code

The second testbench file is testbench_par_multihart_ip.cpp. It runs a distributed and parallel version of the test_mem.s program.

A set of NB_HART sub-arrays are filled in parallel by the running harts. After all the sub-arrays have been initialized, they are summed in parallel. The first hart reads the partial sums (remote memory accesses) and computes their overall sum (reduction operation).

The parallelized version of test_mem.s is named test_mem_par_2h.s for two harts, test_mem_par_4h.s for four harts, and test_mem_par_8h.s for eight harts. When you switch from *x* harts to *y* harts, you must not forget to do two updates: the LOG_NB_HART value in the multihart_ip.h file and the name of the included hex file in the testbench_par_multihart_ip.cpp file.

10.4.2.1 Building the Hex Files

The hex files can be built with the build_par.sh shell script shown in Listing 10.44.

Listing 10.44 The build_par.sh shell script

```
$ cat build_par.sh
riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 -Tdata 0 -o
    test_mem_par_2h.elf test_mem_par_2h.s
riscv32-unknown-elf-objcopy -O binary --only-section=.text
    test_mem_par_2h.elf test_mem_par_2h_text.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' test_mem_par_2h_text.bin >
    test_mem_par_2h_text.hex
riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 -Tdata 0 -o
    test_mem_par_4h.elf test_mem_par_4h.s
riscv32-unknown-elf-objcopy -O binary --only-section=.text
    test_mem_par_4h.elf test_mem_par_4h_text.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' test_mem_par_4h_text.bin >
    test_mem_par_4h_text.hex
riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 -Tdata 0 -o
    test_mem_par_8h.elf test_mem_par_8h.s
riscv32-unknown-elf-objcopy -O binary --only-section=.text
    test_mem_par_8h.elf test_mem_par_8h_text.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' test_mem_par_8h_text.bin >
    test_mem_par_8h_text.hex
$
```

10.4.2.2 The Testbench_par_multihart_ip.cpp File

The testbench_par_multihart_ip.cpp file for a 2-hart processor is shown in Listing 10.45.

Listing 10.45 The testbench_par_multihart_ip.cpp file for a 2-hart processor

```
#include <stdio.h>
#include "multihart_ip.h"
#define OTHER_HART_START 0x74/4
int          data_ram[NB_HART][HART_DATA_RAM_SIZE];
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_mem_par_2h_text.hex"
};
unsigned int start_pc[NB_HART];
int main() {
  unsigned int nbi;
  unsigned int nbc;
  int          w;
  start_pc[0] = 0;
  for (int i=1; i<NB_HART; i++)
    start_pc[i] = OTHER_HART_START;
  multihart_ip((1<<NB_HART)-1,//start all harts
               start_pc, (unsigned int*)code_ram, data_ram,
                 &nbi, &nbc);
  printf("%d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", nbi, nbc, ((float)nbi)/nbc);
  for (int h=0; h<NB_HART; h++){
    printf("hart %d: data memory dump (non null words)\n", h);
    for (int i=0; i<HART_DATA_RAM_SIZE; i++){
      w = data_ram[h][i];
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
        (int)(4*(i+(((w_data_address_t)h)<<
                 LOG_HART_DATA_RAM_SIZE))), w, (unsigned int)w);
    }
  }
  return 0;
}
```

The RISC-V test program in the test_mem_par_2h.s file computes a parallelized sum of the elements of a distributed array. In the multihart IP, each hart computes the sum of its local elements of the array. The first hart then accumulates the partial results through remote accesses and saves the final sum in its local memory.

The RISC-V code is divided in two parts. The first part (see Listings 10.46 to 10.48) is run only by the first hart. The second part (see Listing 10.49) is run by all the other harts.

The full code is placed in the multihart_ip code memory at address 0.

10.4.2.3 The RISC-V Test Program for the First Hart

The first hart sets 10 elements of the distributed array in a first loop (.L1; see Listing 10.46).

Listing 10.46 The test_mem_par_2h.s file (RISC-V source code): initializing the hart part of the array to be summed

```
        .equ     LOG_NB_HART,1
        .equ     LOG_DATA_RAM_SIZE,(16+2)
        .equ     NB_HART,(1<<LOG_NB_HART)
        .equ     LOG_HART_DATA_RAM_SIZE,(LOG_DATA_RAM_SIZE-
            LOG_NB_HART)
        .equ     HART_DATA_RAM_SIZE,(1<<LOG_HART_DATA_RAM_SIZE)
        .globl   main
        .globl   other_hart_start
main:
        li       a0,0          /*a0=0*/
        li       a1,0          /*a1=0*/
        li       a2,0          /*a2=0*/
        addi     a3,a2,40      /*a3=40*/
.L1:
        addi     a1,a1,1       /*a1++*/
        sw       a1,0(a2)      /*t[a2]=a1*/
        addi     a2,a2,4       /*a2+=4*/
        bne      a2,a3,.L1     /*if (a3!=a2) goto .L1*/
...
```

Then, the first hart sums the elements in a second loop (.L2; see Listing 10.47). The local sum is saved at the local address 40.

Listing 10.47 The test_mem_par_2h.s file (RISC-V source code): summing the local partition

```
...
        li      a1,0        /*a1=0*/
        li      a2,0        /*a2=0*/
.L2:
        lw      a4,0(a2)    /*a4=t[a2]*/
        addi    a2,a2,4     /*a2+=4*/
        add     a0,a0,a4    /*a0+=a4*/
        bne     a2,a3,.L2   /*if (a3!=a2) goto .L2*/
        li      a2,40       /*a2=40*/
        sw      a0,0(a2)    /*t[a2]=a0*/
...
```

Eventually, the first hart adds the other harts partial sums to compute the total (see Listing 10.48). It uses accesses to other partitions (positive addresses greater than the partition size).

The synchronization between the writers (the other harts writing their partial sum in their local memory) and the reader (the first hart reading these partial sums from accesses to the external memory) is done in the inner .L3 loop. The first hart keeps reading while the memory word is null (the inner loop contains two instructions: lw and beq). As soon as the other hart has written its sum, the addressed word is non null anymore and the first harts exits from the inner loop and accumulates the value loaded into register a3.

Listing 10.48 The test_mem_par_2h.s file (RISC-V source code): accumulating the other partitions local sums

```
...
        li      a1,1        /*a1=1*/
        li      a2,NB_HART  /*a2=NB_HART*/
        li      a4,HART_DATA_RAM_SIZE
        mv      a5,a4       /*a5=a4*/
.L3:    lw      a3,40(a4)   /*a3=t[a4+40]*/
        beq     a3,zero,.L3 /*if (a3==0) goto .L3*/
        add     a0,a0,a3    /*a0+=a3*/
        add     a4,a4,a5    /*a4+=a5*/
        addi    a1,a1,1     /*a1++*/
        bne     a1,a2,.L3   /*if (a1!=a2) goto .L3*/
        li      a2,44       /*a2=44*/
        sw      a0,0(a2)    /*t[a2]=a0*/
        ret
```

10.4.2.4 The RISC-V Test Program for the Other Harts

All the other harts run the same second part of the code to initialize their share of the array (.L4 loop) and compute its sum (.L5 loop).

Listing 10.49 The test_mem_par_2h.s file (RISC-V source code): initializing other hart subarrays and summing them

```
...
other_hart_start:
        slli    t0,a1,3   /*t0=8*a1*/
        slli    t1,a1,1   /*t1=2*a1*/
        li      a0,0      /*a0=0*/
        add     a1,t0,t1  /*a1=t0+t1*/
        li      a2,0      /*a2=0*/
        addi    a3,a2,40   /*a3=40*/
.L4:
        addi    a1,a1,1    /*a1++*/
        sw      a1,0(a2)   /*t[a2]=a1*/
        addi    a2,a2,4    /*a2+=4*/
        bne     a2,a3,.L4  /*if (a2!=a3) goto .L4*/
        li      a1,0       /*a1=0*/
        li      a2,0       /*a2=0*/
.L5:
        lw      a4,0(a2)   /*a4=t[a2]*/
        addi    a2,a2,4    /*a2+=4*/
        add     a0,a0,a4   /*a0+=a4*/
        bne     a2,a3,.L5  /*if (a2!=a3) goto .L5*/
        li      a2,40      /*a2=40*/
        sw      a0,0(a2)   /*t[a2]=a0*/
        ret
```

The OTHER_HART_START constant definition sets the starting point of the code to be run by the harts other than 0 (OTHER_HART_START=0x74/4, i.e. instruction 29).

10.4.2.5 The Run Print

For the run of two harts (LOG_NB_HART is 1 and the included file is test_mem_par_2h_text.hex), the output is shown in Listing 10.50 (the final sum -of the 20 first integers- is 210; IPC is 0.63/CPI is 1.59):

Listing 10.50 The output of the main function of the testbench_par_multihart_ip.cpp file (test_mem_par_2h_text.hex code)

```
hart 0: 0000: 00000513      li a0, 0
hart 0:       a0  =                 0 (        0)
hart 1: 0116: 00359293      slli t0, a1, 3
hart 1:       t0  =                 8 (        8)
...
hart 0: 0104: 02c00613      li a2, 44
hart 0:       a2  =                44 (       2c)
hart 0: 0108: 00a62023      sw a0, 0(a2)
hart 0:       m[       2c] =               210 (       d2)
hart 0: 0112: 00008067      ret
hart 0:       pc  =                 0 (        0)
register file for hart 0
...
a0  =              210 (      d2)
a1  =                2 (       2)
a2  =               44 (      2c)
a3  =              155 (      9b)
a4  =           262144 (   40000)
a5  =           131072 (   20000)
...
register file for hart 1
...
a0  =              155 (      9b)
a1  =                0 (       0)
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               20 (      14)
...
192 fetched and decoded instructions in 305 cycles (ipc = 0.63)
hart 0: data memory dump (non null words)
m[     0] =                1 (        1)
```

```
m[    4] =                  2 (          2)
m[    8] =                  3 (          3)
m[    c] =                  4 (          4)
m[   10] =                  5 (          5)
m[   14] =                  6 (          6)
m[   18] =                  7 (          7)
m[   1c] =                  8 (          8)
m[   20] =                  9 (          9)
m[   24] =                 10 (          a)
m[   28] =                 55 (         37)
m[   2c] =                210 (         d2)
hart 1: data memory dump (non null words)
m[20000] =                 11 (          b)
m[20004] =                 12 (          c)
m[20008] =                 13 (          d)
m[2000c] =                 14 (          e)
m[20010] =                 15 (          f)
m[20014] =                 16 (         10)
m[20018] =                 17 (         11)
m[2001c] =                 18 (         12)
m[20020] =                 19 (         13)
m[20024] =                 20 (         14)
m[20028] =                155 (         9b)
```

10.5 Synthesizing the IP

Figure 10.8 shows the synthesis report for two harts. The IP cycle is two FPGA cycles (20 ns, 50 Mhz). This is also the case for four and eight harts.

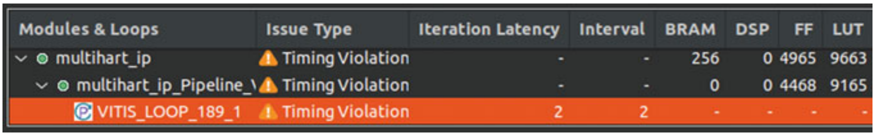
Figure 10.9 shows that the main loop iteration takes two cycles, hence the multihart IP cycle is 20 ns (for four and eight harts, the iteration takes three cycles but the II interval is two cycles).

10.6 The Vivado Project and the Implementation Report

Figure 10.10 shows the Vivado implementation cost of the multihart_ip design for two harts (4697 LUTs, 8.83%).

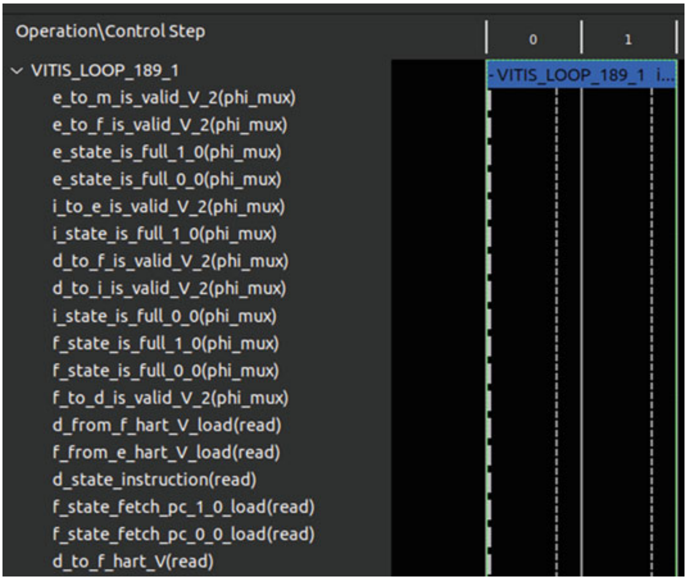
Figure 10.11 shows the Vivado implementation cost of the multihart_ip design for four harts (7537 LUTs, 14.17%).

Figure 10.12 shows the Vivado implementation cost of the multihart_ip design for eight harts (12,866 LUTs, 24.18%).



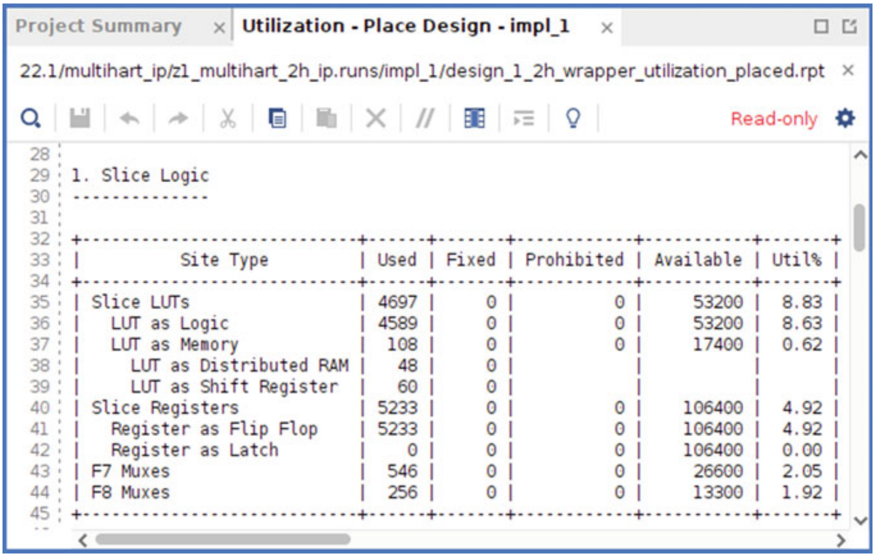
Modules & Loops	Issue Type	Iteration Latency	Interval	BRAM	DSP	FF	LUT
multihart_ip	⚠ Timing Violation	-	-	256	0	4965	9663
multihart_ip_Pipeline_\	⚠ Timing Violation	-	-	0	0	4468	9165
VITIS_LOOP_189_1	⚠ Timing Violation	2	2	-	-	-	-

Fig. 10.8 The multihart_ip synthesis report



```
VITIS_LOOP_189_1
    e_to_m_is_valid_V_2(phi_mux)
    e_to_f_is_valid_V_2(phi_mux)
    e_state_is_full_1_0(phi_mux)
    e_state_is_full_0_0(phi_mux)
    i_to_e_is_valid_V_2(phi_mux)
    i_state_is_full_1_0(phi_mux)
    d_to_f_is_valid_V_2(phi_mux)
    d_to_i_is_valid_V_2(phi_mux)
    i_state_is_full_0_0(phi_mux)
    f_state_is_full_1_0(phi_mux)
    f_state_is_full_0_0(phi_mux)
    f_to_d_is_valid_V_2(phi_mux)
    d_from_f_hart_V_load(read)
    f_from_e_hart_V_load(read)
    d_state_instruction(read)
    f_state_fetch_pc_1_0_load(read)
    f_state_fetch_pc_0_0_load(read)
    d_to_f_hart_V(read)
```

Fig. 10.9 The multihart_ip schedule



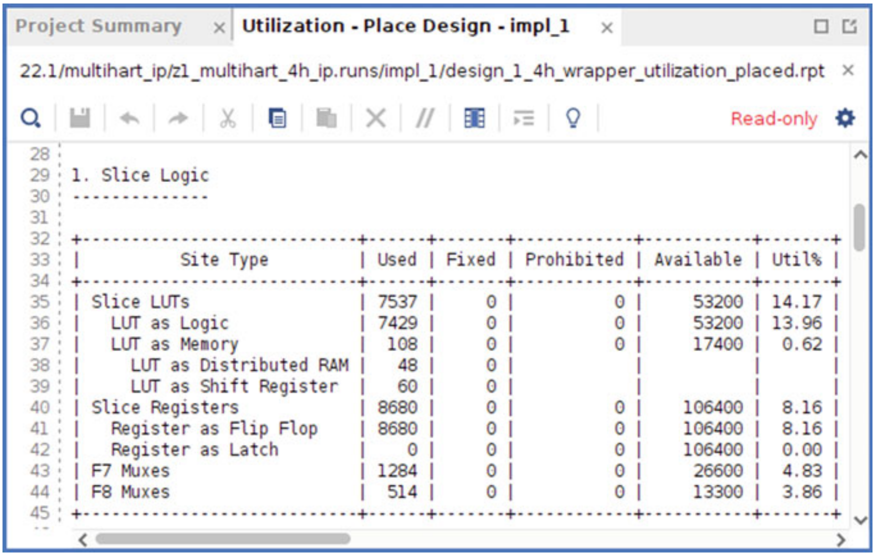
22.1/multihart_ip/z1_multihart_2h_ip.runs/impl_1/design_1_2h_wrapper_utilization_placed.rpt

```

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 4697 |     0 |          0 |     53200 |  8.83 |
|   LUT as Logic             | 4589 |     0 |          0 |     53200 |  8.63 |
|   LUT as Memory            |  108 |     0 |          0 |     17400 |  0.62 |
|     LUT as Distributed RAM |   48 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 5233 |     0 |          0 |    106400 |  4.92 |
|   Register as Flip Flop    | 5233 |     0 |          0 |    106400 |  4.92 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  546 |     0 |          0 |     26600 |  2.05 |
| F8 Muxes                   |  256 |     0 |          0 |     13300 |  1.92 |
+----------------------------+------+-------+------------+-----------+-------+
```

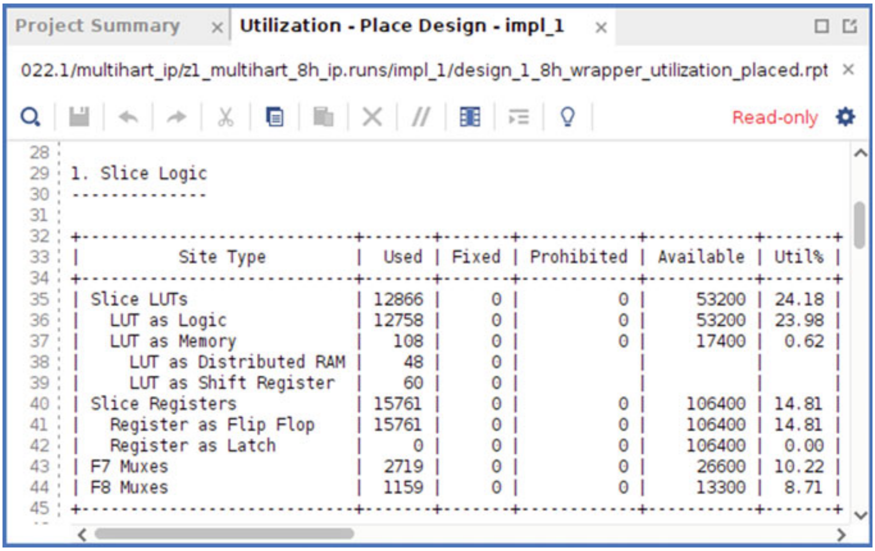
Fig. 10.10 The multihart_ip Vivado implementation report for two harts


Project Summary
Utilization - Place Design - impl_1
22.1/multihart_ip/z1_multihart_4h_ip.runs/impl_1/design_1_4h_wrapper_utilization_placed.rpt
Read-only

```
1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 7537 |     0 |          0 |     53200 | 14.17 |
|   LUT as Logic             | 7429 |     0 |          0 |     53200 | 13.96 |
|   LUT as Memory            |  108 |     0 |          0 |     17400 |  0.62 |
|     LUT as Distributed RAM |   48 |     0 |            |           |       |
|     LUT as Shift Register  |   60 |     0 |            |           |       |
| Slice Registers            | 8680 |     0 |          0 |    106400 |  8.16 |
|   Register as Flip Flop    | 8680 |     0 |          0 |    106400 |  8.16 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   | 1284 |     0 |          0 |     26600 |  4.83 |
| F8 Muxes                   |  514 |     0 |          0 |     13300 |  3.86 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 10.11 The multihart_ip Vivado implementation report for four harts


Project Summary
Utilization - Place Design - impl_1
022.1/multihart_ip/z1_multihart_8h_ip.runs/impl_1/design_1_8h_wrapper_utilization_placed.rpt
Read-only

```
1. Slice Logic
--------------

+----------------------------+-------+-------+------------+-----------+-------+
|          Site Type         |  Used | Fixed | Prohibited | Available | Util% |
+----------------------------+-------+-------+------------+-----------+-------+
| Slice LUTs                 | 12866 |     0 |          0 |     53200 | 24.18 |
|   LUT as Logic             | 12758 |     0 |          0 |     53200 | 23.98 |
|   LUT as Memory            |   108 |     0 |          0 |     17400 |  0.62 |
|     LUT as Distributed RAM |    48 |     0 |            |           |       |
|     LUT as Shift Register  |    60 |     0 |            |           |       |
| Slice Registers            | 15761 |     0 |          0 |    106400 | 14.81 |
|   Register as Flip Flop    | 15761 |     0 |          0 |    106400 | 14.81 |
|   Register as Latch        |     0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  2719 |     0 |          0 |     26600 | 10.22 |
| F8 Muxes                   |  1159 |     0 |          0 |     13300 |  8.71 |
+----------------------------+-------+-------+------------+-----------+-------+
```

Fig. 10.12 The multihart_ip Vivado implementation report for eight harts

10.7 Running the Multihart_ip on the Development Board

Experimentation

To run the multihart_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with multihart_ip.
There are two drivers: helloworld_seq_hart.c on which you can run independent programs on each hart and helloworld_par_hart.c on which you can run a parallel sum of the elements of an array.

10.7.1 Running Independent Codes

The code to drive the FPGA to run the test_mem.s program on two harts is shown in Listing 10.51 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script). To run another program, e.g. test_branch, test_jal_jalr, test_load_store, test_lui_auipc, test_op, test_op_imm, or test_sum, update the code_ram array initialization #include line. To increase the number of harts, change the LOG_NB_HART value.

Listing 10.51 The helloworld_seq_hart.c file driving the 2-hart multihart_ip

```
#include <stdio.h>
#include "xmultihart_ip.h"
#include "xparameters.h"
#define LOG_NB_HART             1
#define NB_HART                (1<<LOG_NB_HART)
#define LOG_CODE_RAM_SIZE      16
//size in words
#define CODE_RAM_SIZE          (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE      16
//size in words
#define DATA_RAM_SIZE          (1<<LOG_DATA_RAM_SIZE)
#define LOG_HART_DATA_RAM_SIZE (LOG_DATA_RAM_SIZE-LOG_NB_HART)
#define HART_DATA_RAM_SIZE     (1<<LOG_HART_DATA_RAM_SIZE)
XMultihart_ip_Config *cfg_ptr;
XMultihart_ip         ip;
word_type code_ram[CODE_RAM_SIZE] = {
#include "test_mem_text.hex"
};
word_type start_pc[NB_HART]={0};
int main(){
  unsigned int nbi, nbc;
  word_type    w;
  cfg_ptr = XMultihart_ip_LookupConfig(
      XPAR_XMULTIHART_IP_0_DEVICE_ID);
  XMultihart_ip_CfgInitialize(&ip, cfg_ptr);
  XMultihart_ip_Set_running_hart_set(&ip, (1<<NB_HART)-1);
  XMultihart_ip_Write_start_pc_Words(&ip, 0, start_pc, NB_HART);
  XMultihart_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XMultihart_ip_Start(&ip);
  while (!XMultihart_ip_IsDone(&ip));
  nbi = XMultihart_ip_Get_nb_instruction(&ip);
  nbc = XMultihart_ip_Get_nb_cycle(&ip);
  printf("%d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", nbi, nbc, ((float)nbi)/nbc);
  for (int h=0; h<NB_HART; h++){
    printf("hart %d data memory dump (non null words)\n", h);
    for (int i=0; i<HART_DATA_RAM_SIZE; i++){
      XMultihart_ip_Read_data_ram_Words
        (&ip, i+(((int)h)<<LOG_HART_DATA_RAM_SIZE), &w, 1);
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
                4*(i+(((int)h)<<LOG_HART_DATA_RAM_SIZE)),
                (int)w, (unsigned int)w);
    }
  }
}
```

The run of the RISC-V code in the test_mem.s file on the FPGA produces the print shown in Listing 10.52 to the putty window (run for two harts).

Listing 10.52 The helloworld.c print on the putty window

```
176 fetched and decoded instructions in 306 cycles (ipc = 0.58)
hart 0 data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
hart 1 data memory dump (non null words)
m[20000] =                1 (       1)
m[20004] =                2 (       2)
m[20008] =                3 (       3)
m[2000c] =                4 (       4)
m[20010] =                5 (       5)
m[20014] =                6 (       6)
m[20018] =                7 (       7)
m[2001c] =                8 (       8)
m[20020] =                9 (       9)
m[20024] =               10 (       a)
m[2002c] =               55 (      37)
```

10.7.2 Running a Parallelized Application

The code to drive the FPGA to run the test_mem_par_2h.s code on two harts is shown in Listing 10.53 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script). To increase the number of harts, change the LOG_NB_HART value and change the included hex file.

Listing 10.53 The helloworld_par_hart.c file driving the multihart_ip for the test_mem_par_2h.s code

```
#include <stdio.h>
#include "xmultihart_ip.h"
#include "xparameters.h"
#define LOG_NB_HART               1
#define NB_HART                  (1<<LOG_NB_HART)
#define LOG_CODE_RAM_SIZE        16
//size in words
#define CODE_RAM_SIZE            (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE        16
//size in words
#define DATA_RAM_SIZE            (1<<LOG_DATA_RAM_SIZE)
#define LOG_HART_DATA_RAM_SIZE   (LOG_DATA_RAM_SIZE-LOG_NB_HART)
#define HART_DATA_RAM_SIZE       (1<<LOG_HART_DATA_RAM_SIZE)
#define OTHER_HART_START         0x74/4
XMultihart_ip_Config *cfg_ptr;
XMultihart_ip         ip;
unsigned int code_ram[CODE_RAM_SIZE]={
#include "test_mem_par_2h_text.hex"
};
word_type start_pc[NB_HART];
int main(){
  unsigned int nbi;
  unsigned int nbc;
  word_type    w;
  cfg_ptr = XMultihart_ip_LookupConfig(
      XPAR_XMULTIHART_IP_0_DEVICE_ID);
  XMultihart_ip_CfgInitialize(&ip, cfg_ptr);
  XMultihart_ip_Set_running_hart_set(&ip, (1<<NB_HART)-1);
  for (int h=1; h<NB_HART; h++)
    start_pc[h]=OTHER_HART_START;
  start_pc[0] = 0;
  XMultihart_ip_Write_start_pc_Words(&ip, 0, start_pc, NB_HART);
  XMultihart_ip_Write_code_ram_Words(&ip, 0, code_ram,
      CODE_RAM_SIZE);
  XMultihart_ip_Start(&ip);
  while (!XMultihart_ip_IsDone(&ip));
  nbi = XMultihart_ip_Get_nb_instruction(&ip);
  nbc = XMultihart_ip_Get_nb_cycle(&ip);
  printf("%d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", nbi, nbc, ((float)nbi)/nbc);
  for (int h=0; h<NB_HART; h++){
    printf("hart %d data memory dump (non null words)\n", h);
    for (int i=0; i<HART_DATA_RAM_SIZE; i++){
      XMultihart_ip_Read_data_ram_Words
        (&ip, i+(((int)h)<<LOG_HART_DATA_RAM_SIZE), &w, 1);
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
                4*(i+(((int)h)<<LOG_HART_DATA_RAM_SIZE)),
                (int)w, (unsigned int)w);
    }
  }
}
```

The run of the RISC-V code in the test_mem_par_2h.s file on the FPGA produces the print shown in Listing 10.52 to the putty window (run for two harts).

Listing 10.54 The helloworld.c print on the putty window

```
192 fetched and decoded instructions in 305 cycles (ipc = 0.63)
hart 0: data memory dump (non null words)
m[    0] =                1 (        1)
m[    4] =                2 (        2)
m[    8] =                3 (        3)
m[    c] =                4 (        4)
m[   10] =                5 (        5)
m[   14] =                6 (        6)
m[   18] =                7 (        7)
m[   1c] =                8 (        8)
m[   20] =                9 (        9)
m[   24] =               10 (        a)
m[   28] =               55 (       37)
m[   2c] =              210 (       d2)
hart 1: data memory dump (non null words)
m[20000] =               11 (        b)
m[20004] =               12 (        c)
m[20008] =               13 (        d)
m[2000c] =               14 (        e)
m[20010] =               15 (        f)
m[20014] =               16 (       10)
m[20018] =               17 (       11)
m[2001c] =               18 (       12)
m[20020] =               19 (       13)
m[20024] =               20 (       14)
m[20028] =              155 (       9b)
```

10.7.3 Further Testing of the Multihart_ip

To pass the riscv-tests on the Vitis_HLS simulator, you just need to use the testbench_riscv_tests_multihart_ip.cpp program in the riscv-tests/my_isa/my_rv32ui folder as the testbench.

To pass the riscv-tests on the FPGA, you must use the helloworld_multihart_2h_ip.c, helloworld_multihart_4h_ip.c, or helloworld_multihart_8h_ip.c in the riscv-tests/my_isa/my_rv32ui folder. Normally, since you already ran the update_helloworld.sh shell script for the other processors, the helloworld_multihart_xh_ip.c file (*xh* stands for *2h*, *4h*, or *8h*) should have paths adapted to your environment. However if you did not, you must run the update_helloworld.sh shell script.

10.8 Comparing the Multihart_ip to the 4-stage Pipeline

To run a benchmark from the mibench suite, say my_dir/bench, you set the testbench as the testbench_bench_multihart_ip.cpp file found in the mibench/my_mibench/my_dir/bench folder. For example, to run basicmath, you set the testbench as testbench_basicmath_multihart_ip.cpp in the mibench/my_mibench/my_automotive/basicmath folder.

To run one of the official riscv-tests benchmarks, say bench, you set the testbench as the testbench_bench_multihart_ip.cpp file found in the riscv-tests/benchmarks/

bench folder. For example, to run median, you set the testbench as testbench_median_multihart_ip.cpp in the riscv-tests/benchmarks/median folder.

To run the same benchmarks on the FPGA, select helloworld_multihart_2h_ip.c to run on the z1_multihart_2h_ip Vivado project, helloworld_multihart_4h_ip.c to run on the z1_multihart_4h_ip Vivado project, or helloworld_multihart_8h_ip.c to run on the z1_multihart_8h_ip Vivado project.

10.8.1 Two Harts

Table 10.1 shows the execution time of the different benchmarks run on a 2-hart design as computed with equation 5.1 ($nmi * cpi * c$, where c = 20ns). The baseline time reference refers to two successive executions of the same test program on the rv32i_pp_ip design (the fastest design up to now).

Two harts are not enough to fill the pipeline. The CPI remains higher than on the 4-stage single hart pipeline (1.41 on the average versus 1.20; see 8.3). But the cycle reduction compensates the CPI degradation despite the six stages increased length of the pipeline.

On the benchmark suite, the 2-hart multihart_ip is 20% faster than the rv32i_pp_ip.

Table 10.1 Execution time of the benchmarks on the multihart_ip processor (two active harts running the same program)

suite	benchmark	Cycles	nmi	cpi	Time (s)	4-stage time (s)	Improve (%)
mibench	basicmath	88,958,398	61,795,478	1.44	1.779167960	2.258589420	21
mibench	bitcount	88,133,658	65,306,478	1.35	1.762673160	2.274599940	23
mibench	qsort	18,756,398	13,367,142	1.40	0.375127960	0.485171460	23
mibench	stringsearch	1,720,638	1,098,326	1.57	0.034412760	0.038149800	10
mibench	rawcaudio	1,980,166	1,266,316	1.56	0.039603320	0.045490260	13
mibench	rawdaudio	1,383,490	936,598	1.48	0.027669800	0.033768060	18
mibench	crc32	960,042	600,028	1.60	0.019200840	0.021600960	11
mibench	fft	91,511,156	62,730,816	1.46	1.830223120	2.294015640	20
mibench	fft_inv	93,053,088	63,840,638	1.46	1.861061760	2.334520080	20
riscv-tests	median	75,636	55,784	1.36	0.001512720	0.002128260	29
riscv-tests	mm	461,940,400	315,122,748	1.47	9.238808000	11.604328500	20
riscv-tests	multiply	1,099,802	835,794	1.32	0.021996040	0.032401440	32
riscv-tests	qsort	736,402	543,346	1.36	0.014728040	0.019837800	26
riscv-tests	spmv	3,502,988	2,492,304	1.41	0.070059760	0.089876400	22
riscv-tests	towers	906,100	807,616	1.12	0.018122000	0.025583100	29
riscv-tests	vvadd	40,026	32,020	1.25	0.000800520	0.001080720	26

Table 10.2 Execution time of the program examples on the multihart_ip processor (four active harts running the same program)

suite	benchmark	Cycles	nmi	cpi	Time (s)	4-stage time (s)	Improve (%)
mibench	basicmath	136,057,928	123,590,956	1.10	2.721158560	4.517178840	40
mibench	bitcount	143,803,793	130,612,956	1.10	2.876075860	4.549199880	37
mibench	qsort	29,151,626	26,734,284	1.09	0.583032520	0.970342920	40
mibench	stringsearch	2,384,057	2,196,652	1.09	0.047681140	0.076299600	38
mibench	rawcaudio	2,873,992	2,532,632	1.13	0.057479840	0.090980520	37
mibench	rawdaudio	2,074,964	1,873,196	1.11	0.041499280	0.067536120	39
mibench	crc32	1,230,088	1,200,056	1.03	0.024601760	0.043201920	43
mibench	fft	138,090,940	125,461,632	1.10	2.761818800	4.588031280	40
mibench	fft_inv	140,447,968	127,681,276	1.10	2.808959360	4.669040160	40
riscv-tests	median	123,662	111,568	1.11	0.002231360	0.004256520	48
riscv-tests	mm	694,489,981	630,245,496	1.10	13.889799620	23.208657000	40
riscv-tests	multiply	1,894,335	1,671,588	1.13	0.037886700	0.064802880	42
riscv-tests	qsort	1,221,567	1,086,692	1.12	0.024431340	0.039675600	38
riscv-tests	spmv	5,430,195	4,984,608	1.09	0.108603900	0.179752800	40
riscv-tests	towers	1,615,420	1,615,232	1.00	0.032308400	0.051166200	37
riscv-tests	vvadd	66,076	64,040	1.03	0.001321520	0.002161440	39

10.8.2 Four Harts

Table 10.2 shows the execution time of the different program examples run on a 4-hart design as computed with equation 5.1 ($nmi * cpi * c$, where $c = 20$ ns). The baseline time reference refers to four successive executions of the same test program on the rv32i_pp_ip design.

With four harts, the CPI is lower than the rv32i_pp_ip CPI (1.09 vs. 1.20 on average). With the 20ns cycle, the multihart IP is 1.66 times faster than the rv32i_pp_ip on the average, when the four harts are running. However, the design uses 1.7 times more LUTs (7,537 LUTs versus 4,334 LUTs).

10.8.3 Eight Harts

Table 10.3 shows the execution time of the different program examples run on an 8-hart design as computed with equation 5.1 ($nmi * cpi * c$, where $c = 20$ns). The baseline time reference refers to eight successive executions of the same test program on the rv32i_pp_ip design.

The average CPI is 1.09, no better than with four harts. Running eight interleaved harts on the multihart IP is on the average 1.65 times faster than running the same eight programs successively on the rv32i_pp_ip. However, the 8-hart design uses 12,866 LUTs, i.e. 3 times the LUTs used to build the rv32i_pp_ip.

Table 10.3 Execution time of the program examples on the multihart_ip processor (eight active harts running the same program)

suite	benchmark	Cycles	nmi	cpi	Time (s)	4-stage time (s)	Improve (%)
mibench	basicmath	273,657,613	247,181,912	1.11	5.473152260	9.034357680	39
mibench	bitcount	285,828,811	261,225,912	1.09	5.716576220	9.098399760	37
mibench	qsort	58,457,350	53,468,568	1.09	1.169147000	1.940685840	40
mibench	stringsearch	4,782,512	4,393,304	1.09	0.095650240	0.152599200	37
mibench	rawcaudio	5,677,516	5,065,264	1.12	0.113550320	0.181961040	38
mibench	rawdaudio	4,138,785	3,746,392	1.10	0.082775700	0.135072240	39
mibench	crc32	2,479,691	2,400,112	1.03	0.049593820	0.086403840	43
mibench	fft	277,969,508	250,923,264	1.11	5.559390160	9.176062560	39
mibench	fft_inv	282,804,386	255,362,552	1.11	5.656087720	9.338080320	39
riscv-tests	median	247,080	223,136	1.11	0.004941600	0.008513040	42
riscv-tests	mm	1,399,160,540	1,260,490,992	1.11	27.983210800	46.417314000	40
riscv-tests	multiply	3,563,258	3,343,176	1.07	0.071265160	0.129605760	45
riscv-tests	qsort	2,391,276	2,173,384	1.10	0.047825520	0.079351200	40
riscv-tests	spmv	10,900,106	9,969,216	1.09	0.218002120	0.359505600	39
riscv-tests	towers	3,261,569	3,230,464	1.01	0.065231380	0.102332400	36
riscv-tests	vvadd	132,852	128,080	1.04	0.002657040	0.004322880	39

References

1. D.M. Tullsen, S.J. Eggers, H.M. Levy, Simultaneous multithreading: maximizing on-chip parallelism, in *22nd Annual International Symposium on Computer Architecture* (IEEE, 1995), pp. 392–403
2. F. Gabbay, *Speculative Execution based on Value Prediction*. EE Department TR 1080, Technion - Israel Institute of Technology (1996)
3. S. Mittal, A survey of value prediction techniques for leveraging value locality, in *Concurrency and Computation, Practice and Experience*, vol. 29, no. 21 (2017)
4. Y. Patt, W. Hwu, M. Shebanow, *HPS, A New Microarchitecture: Rationale and Introduction*, ACM SIGMICRO Newsletter, vol. 16, no. 4 (1985), pp. 103–108
5. R.M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Dev. **11**(1), 25–33 (1967)
6. J. Silc, B. Robic, T. Ungerer, *Processor Architecture, From Dataflow to Superscalar and Beyond* (Springer, Heidelberg, 1999)

Part II
Multiple Core Processors

In this second part, I present multicore systems (the replicated cores are based on the six-stage multicycle pipeline presented in Chap. 9, either single or multi-threaded). The cores are interconnected through the AXI standard interconnection system. On a Pynq-Z1/Pynq-Z2, up to eight harts can be implemented (e.g., eight single hart cores, two cores with four harts each, or four cores with two harts each).

Connecting IPs 11

Abstract

This chapter presents the AXI interconnection system. You will build two multi-IP components. The different IPs are connected via the AXI interconnect IP provided by the Vivado component library. The first design connects a rv32i_npp_ip processor (presented in Chap. 6) to two block memories, one for code and the other for data. This design is intended to show how the AXI interconnection system works. The second design connects two IPs sharing two data memory banks. It is intended to show how multiple memory blocks are shared by multiple IPs, using the AXI interconnection to exchange data.

11.1 The AXI Interconnection System

> The Advanced eXtensible Interface (AXI) part of the ARM Advanced Microcontroller Bus Architecture 3 (AXI3) and 4 (AXI4) specifications, is a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface, mainly designed for on-chip communication. (wikipedia)

The Vivado design suite provides ready-to-use IPs to build an AXI-based System-On-Chip (SOC). The central component is the AXI interconnect IP (see Fig. 11.1). It is used to interconnect multiple masters to multiple slaves identifying themselves with memory mapping. In a memory mapping system, each IP is identified by its memory address in a virtual memory address space.

A *master* is an IP which can initiate a transaction on the AXI bus. A *slave* is an IP which responds to a master's request. A *transaction* is a round-trip data transmission from a master to a slave (the *request*) and from the slave back to the master (the *response*). The request can be a single or multiple word read, or a single or multiple word write. The response is either an acknowledgement (response to a write request, meaning that the write is done) or a single or multiple data words (response to a read request).

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_11

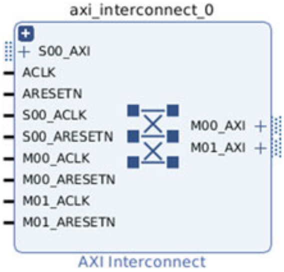


Fig. 11.1 The AXI interconnect IP

Figure 11.2 depicts a basic construction interconnecting the Zynq7 processing system with a processor IP and a memory.

Through this basic interconnection, the Zynq7 writes data to the BRAM IP. Then, it sends a start signal to the CPU IP. When the run is done, the Zynq7 reads data from the BRAM IP.

There is also a direct link between the CPU IP and the BRAM IP. Through this link, the CPU can directly access the memory to load and store data.

From this scheme, a design is built (see Fig. 11.3), connecting together a memoryless rv32i_npp_ip CPU with two external memories, i.e. a memory to host a RISC-V program (the upper Block Memory Generator IP) and a memory to host its data (the lower Block Memory Generator IP).

The SoC is composed of the Zynq7 IP, the AXI interconnect IP, the rv32i_npp_ip, two Block Memory Generator IPs (representing the memories), and two AXI BRAM controllers.

The Block Memory Generator IPs provided by the Vivado design suite are not directly connectable to the AXI bus as they do not have any AXI-related pins. The Vivado design suite provides an AXI BRAM controller IP which is to be placed between the AXI interconnect and a Block Memory Generator IP.

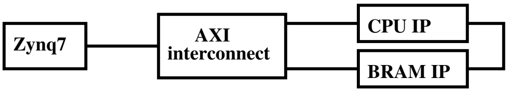


Fig. 11.2 Connecting the Zynq7 master to a CPU IP and a BRAM IP slaves

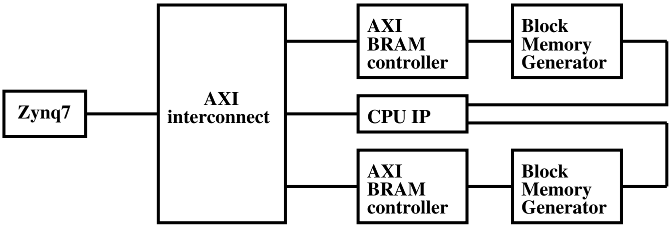


Fig. 11.3 A CPU IP with separated code and data memories

11.2 The Non Pipelined RISC-V Processor with External Memory IPs

All the source files related to the rv32i_npp_bram_ip can be found in the rv32i_npp_bram_ip folder.

11.2.1 The Top Function with a Bram Interface

Experimentation

To simulate the rv32i_npp_bram_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with rv32i_npp_bram_ip.
You can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

To experiment this construction, I choose the CPU IP presented in Chap. 6 and named rv32i_npp_ip. The Vitis_HLS project name is rv32i_npp_bram_ip. The top function (in the rv32i_npp_ip.cpp file) has to be updated to provide the AXI interface pins through the pragma HLS INTERFACE, as shown in Listing 11.1.

Listing 11.1 The rv32i_npp_ip top function prototype

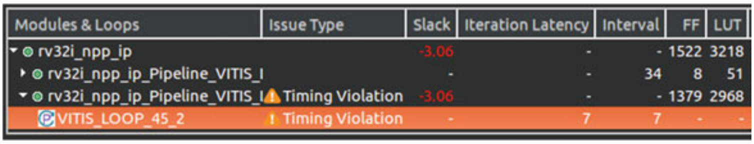
```
void rv32i_npp_ip(
  unsigned int  start_pc,
  unsigned int  code_ram[CODE_RAM_SIZE],
  int           data_ram[DATA_RAM_SIZE],
  unsigned int *nb_instruction){
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE bram      port=code_ram
#pragma HLS INTERFACE bram      port=data_ram
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=return
  ...
```

The s_axilite ports provide the slave AXI interface with which the Zynq7 can send the start run order and arguments to the rv32i_npp_ip (i.e. the start pc and the nb_instruction counter).

The code_ram and data_ram bram ports are the CPU IP private accesses to the Block Memory Generator IPs. They are used by the LOAD/STORE RISC-V instructions run by the rv32i_npp_ip processor.

The remaining of the rv32i_npp_ip top function is unchanged (refer back to Sect. 6.1).

The other files are unchanged, including the testbench and the RISC-V test programs.



Modules & Loops	Issue Type	Slack	Iteration Latency	Interval	FF	LUT
rv32i_npp_ip		-3.06	-	-	1522	3218
rv32i_npp_ip_Pipeline_VITIS_l		-	-	34	8	51
rv32i_npp_ip_Pipeline_VITIS_l	Timing Violation	-3.06	-	-	1379	2968
VITIS_LOOP_45_2	Timing Violation	-	7	7	-	-

Fig. 11.4 The rv32i_npp_bram_ip synthesis report

11.2.2 The IP Synthesis

Figure 11.4 is the synthesis report showing that the IP cycle is unchanged (seven FPGA cycles like in the original rv32i_npp_ip). The Timing Violation is not important. The design will be finely routed by Vivado.

11.2.3 The Vivado Project

The IPs are connected together in the z1_rv32i_npp_bram_ip Vivado project.

Once the z1_rv32i_npp_bram_ip project and the design_1 have been created, in the **Diagram** frame, the Zynq7 Processing System IP is added and the automatic connection is run (**Run Block Automation**), resulting in the construction shown in Fig. 11.5.

Then, the AXI Interconnect IP is added (do not click on the proposed **Run Connection Automation** yet). It is a generic IP which can be parameterized to fit the future design. Select the IP, right-click and select the **Customize Block** to open a **Re-customize IP** dialog box. In the **Top Level Settings** tab, update the **Number of Master Interfaces** entry and set it to 3. The **Diagram** frame now contains what

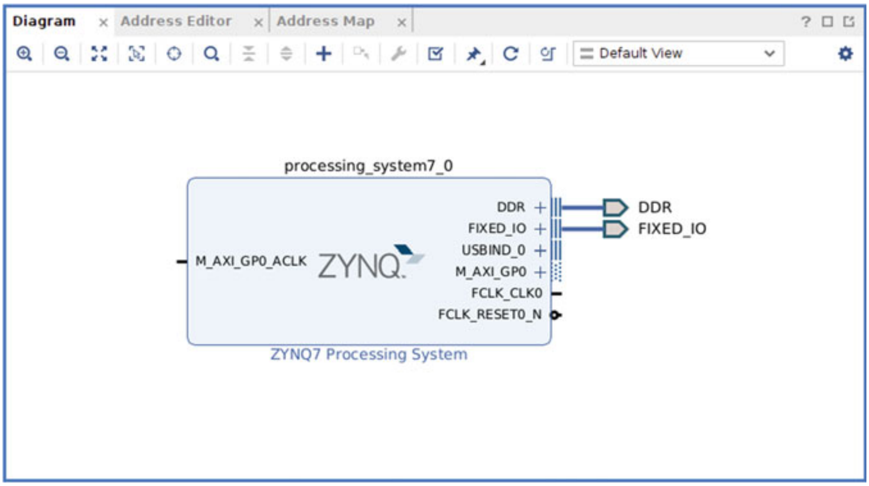


Fig. 11.5 The rv32i_npp_bram_ip construction (first step)

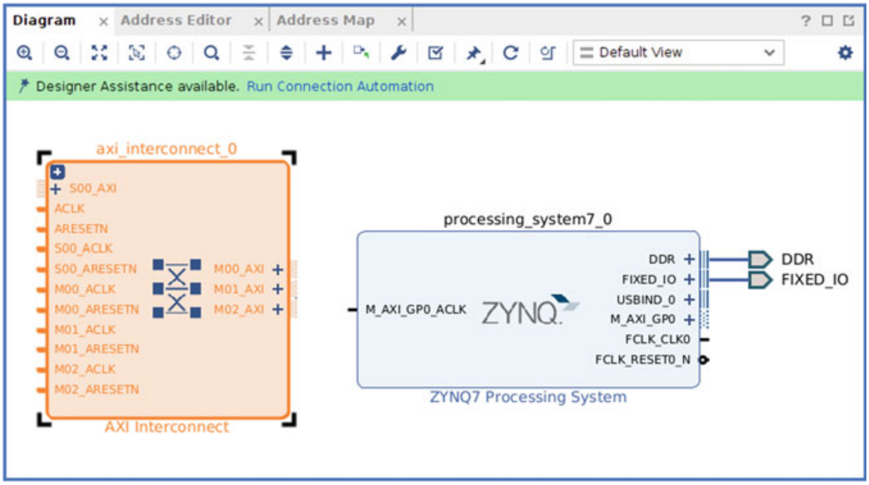


Fig. 11.6 The rv32i_npp_bram_ip construction (second step)

is shown in Fig. 11.6 (notice the three M0x_AXI pins on the right edge of the AXI Interconnect IP).

The *master* and *slave* namings may be confusing because an IP may be seen as a master and, in the same time, as a slave. In the design shown in Fig. 11.3, the Zynq7 IP is a master. The AXI interconnect IP is its slave. It is also the master of the three other IPs (the CPU and the two AXI BRAM controllers). As a slave of the Zynq7, it serves its transactions and as a master of the CPU and BRAMs, it propagates to them the transactions initiated by the Zynq7.

The third step is to add the two AXI BRAM Controller IPs (still no click on the proposed **Run Connection Automation**). They should be customized too (select the IP, right-click and select **Customize Block**). In the **Re-customize IP** dialog box, set the **Number of BRAM interfaces** to 1. Also set the AXI Protocol to AXI4LITE. The **Diagram** frame now contains what is shown in Fig. 11.7.

The fourth step is to add two **Block Memory Generator** IPs and customize them (the **Memory Type** should be set to True Dual Port RAM). You may also rename the IPs as code_ram and data_ram (after selecting a memory block IP, edit the **Block Properties/Name** entry). The **Diagram** frame now contains what is shown in Fig. 11.8.

The fifth step is to add the rv32i_npp_bram_ip CPU built in Sect. 11.2.1. The rv32i_npp_bram_ip folder in which the IP has been defined, synthesized, and exported should be added to the list of visible IPs (on the main window menu, tab **Tools**, select **Settings**, then expand **IP** in the **Project Settings** frame and select **Repository**; in the **IP repositories** frame, click on "+" and navigate to the folder containing your IP). Back on the Vivado main window, **Diagram** frame, you can add your rv32i_npp_bram_ip component. The **Diagram** frame now contains what is shown in Fig. 11.9.

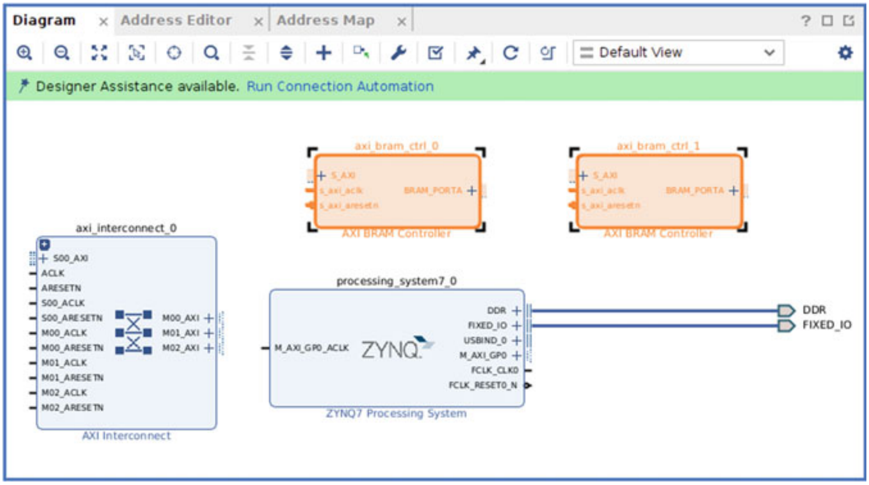


Fig. 11.7 The rv32i_npp_bram_ip construction (third step)

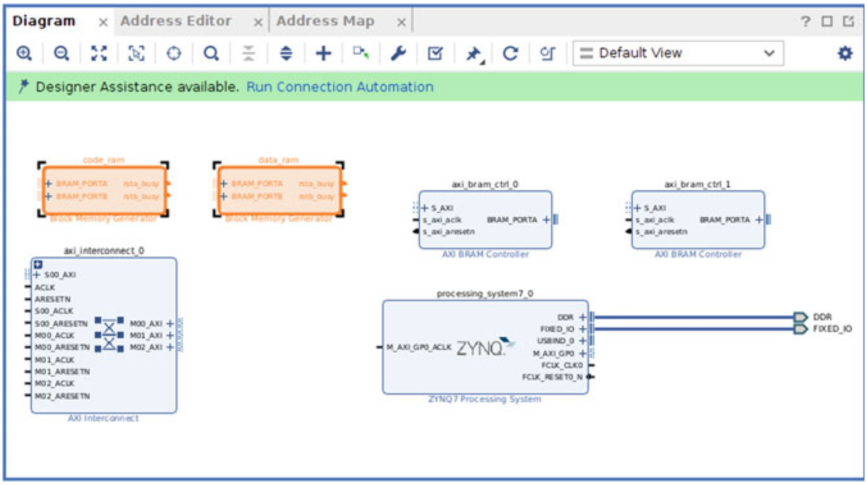


Fig. 11.8 The rv32i_npp_bram_ip construction (fourth step)

The sixth step is to connect all the IPs together. You can use the tool to do *hand made* connections (the connections are a bit too complex to have the automatic connection system find the connections that you need). To draw a connection, move the mouse over a pin and you should see a pen appear, suggesting that you can pull a line when you click. Pull the line until you reach the pin to be connected (if you pulled a line from a wrong starting point, you can escape and cancel the line drawing with the return key).

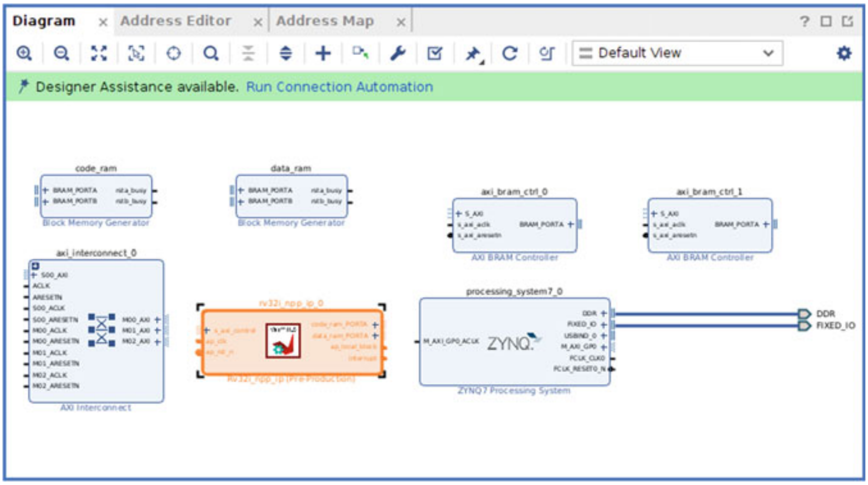


Fig. 11.9 The rv32i_npp_bram_ip construction (fifth step)

The CPU code_ram_PORTA pin should be connected to the code_ram BRAM_PORTA pin. The CPU data_ram_PORTA pin should be connected to the data_ram BRAM_PORTA pin.

The axi_bram_ctrl_0 BRAM_PORTA pin is connected to the code_ram BRAM_PORTB pin. The axi_bram_ctrl_1 BRAM_PORTA pin is connected to the data_ram BRAM_PORTB pin.

The AXI interconnect M00_AXI pin is connected to the rv32i_npp_ip s_axi_control pin. The M01_AXI pin is connected to the axi_bram_ctrl_0 S_AXI pin. The M02_AXI pin is connected to the axi_bram_ctrl_1 S_AXI pin.

The remaining connections can be done with the automatic connection (click on **Run Connection Automation** and select **All Automation**). The **Diagram** frame now contains what is shown in Fig. 11.10 (after clicking on the **Regenerate Layout** button on top of the **Diagram** frame (second button from the right, showing a sort of loop)).

Before creating the HDL wrapper, you must set the address mapping (2^{16} words or 256KB for the code memory and the same for the data memory, as defined in the rv32i_npp_ip.h file).

First, in the **Address Editor** frame (at the right of the **Diagram** frame) apply a default assignment (right-click on **Network 0** and select **Assign All**).

Then edit upward, starting with the bottom line. Update the **Master Base Address**, **Range** and **Master High Address** as shown in Fig. 11.11 (0x4000_0000, 256K for axi_bram_ctrl_0, 0x4004_0000, 256K for axi_bram_ctrl_1, and 0x4008_0000, 64K for rv32i_npp_ip_0).

In the **Diagram** frame, validate your design (right-click and select **Validate Design**).

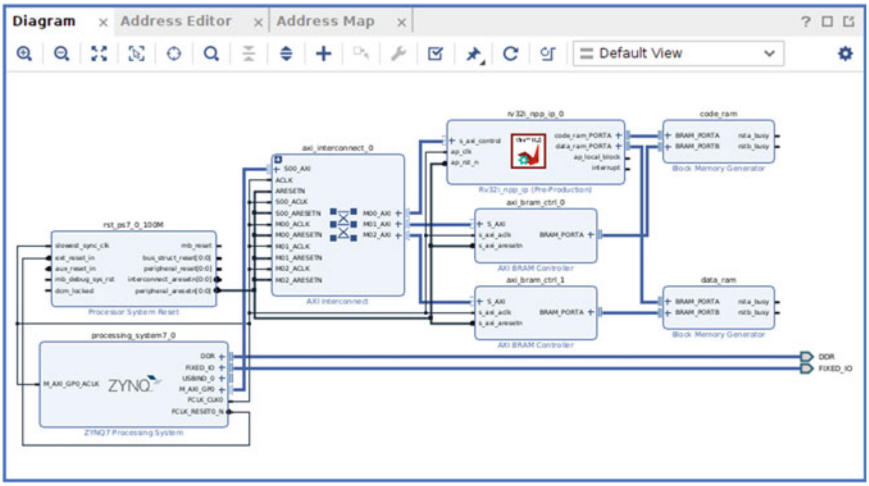
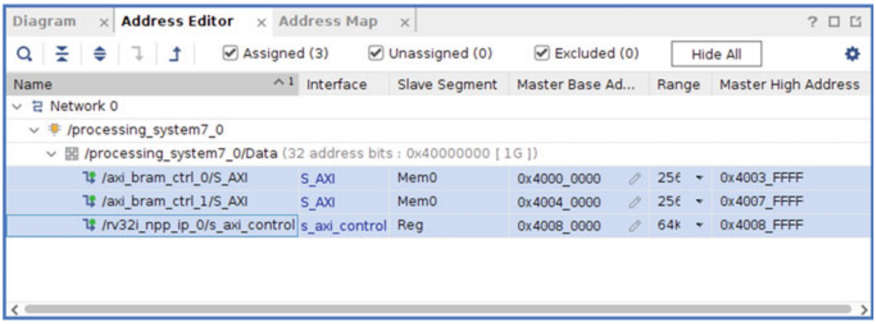


Fig. 11.10 The rv32i_npp_bram_ip construction (sixth step)



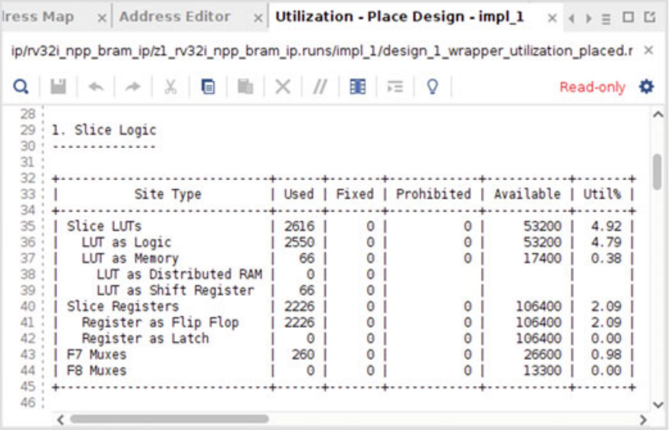
☑ Assigned (3) ☑ Unassigned (0) ☑ Excluded (0)

Name	Interface	Slave Segment	Master Base Ad...	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	256	0x4003_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x4004_0000	256	0x4007_FFFF
/rv32i_npp_ip_0/s_axi_control	s_axi_control	Reg	0x4008_0000	64k	0x4008_FFFF

Fig. 11.11 The address mapping

In the **Block Design** frame, **Source** tab, right-click on design_1 (design_1.bd)(1) and select **Create HDL Wrapper**. Then, generate the bitstream (down the left panel, **PROGRAM AND DEBUG** menu, **Generate Bitstream** button). Among the produced reports, the implementation utilization one shows the FPGA resources used (LUTs, FFs, and BRAM blocks).

Figure 11.12 shows the Vivado implementation cost of the rv32i_npp_bram_ip design (2616 LUTs, 4.92% of the FPGA, instead of 4091 LUTs in the original rv32i_npp_ip design; the two projects are identical except for the HLS INTERFACE for the two ram arguments, however, their respective resource utilizations shown on the design_1_wrapper_utilization_placed.rpt report files in the rv32i_npp_ip/z1_rv32i_npp_ip.runs/impl_1 and rv32i_npp_bram_ip/z1_rv32i_npp_bram_ip.runs/impl_1 folders are very different; unfortunately, I have no explanation for this).


Iress Map
Address Editor
Utilization - Place Design - impl_1
ip/rv32i_npp_bram_ip/z1_rv32i_npp_bram_ip.runs/impl_1/design_1_wrapper_utilization_placed.r
Read-only

```
1. Slice Logic
--------------


+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 2616 |     0 |          0 |     53200 |  4.92 |
|   LUT as Logic             | 2550 |     0 |          0 |     53200 |  4.79 |
|   LUT as Memory            |   66 |     0 |          0 |     17400 |  0.38 |
|     LUT as Distributed RAM |    0 |     0 |            |           |       |
|     LUT as Shift Register  |   66 |     0 |            |           |       |
| Slice Registers            | 2226 |     0 |          0 |    106400 |  2.09 |
|   Register as Flip Flop    | 2226 |     0 |          0 |    106400 |  2.09 |
|   Register as Latch        |    0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |  260 |     0 |          0 |     26600 |  0.98 |
| F8 Muxes                   |    0 |     0 |          0 |     13300 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+
```

Fig. 11.12 The rv32i_npp_bram_ip Vivado implementation report

11.2.4 Running the IP on the Development Board

Experimentation

To run the rv32i_npp_bram_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with rv32i_npp_bram_ip.
You can play with your IP, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder.

The code to be run on the Zynq7 master is shown in Listing 11.2 (do not forget to adapt the path to the hex file to your environment with the update_helloworld.sh shell script). The external code_ram and data_ram arrays are mapped as defined by Vivado (i.e. 0x4000_0000 and 0x4004_0000, to be checked with the **Address Editor** in Vivado).

As the arrays are external to the rv32i_npp_bram_ip, they are not accessed through XRv32i_npp_ip_Write_code_ram_Words and XRv32i_npp_ip_Read_data_ram_Words functions but directly using the AXI addresses assigned to the code_ram and data_ram pointers. These addresses are used by the AXI interconnect IP to route the Zynq7 read and write requests to the concerned AXI BRAM controller IP.

Listing 11.2 The helloworld.c file

```
#include <stdio.h>
#include "xrv32i_npp_ip.h"
#include "xparameters.h"
#define LOG_CODE_RAM_SIZE 16
//size in words
#define CODE_RAM_SIZE     (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE 16
```

```
//size in words
#define DATA_RAM_SIZE     (1<<LOG_DATA_RAM_SIZE)
XRv32i_npp_ip_Config *cfg_ptr;
XRv32i_npp_ip ip;
int *code_ram = (int *)(0x40000000);
int *data_ram = (int *)(0x40040000);
word_type input_code_ram[CODE_RAM_SIZE]={
#include "test_mem_0_text.hex"
};
int main(){
  word_type w;
  cfg_ptr = XRv32i_npp_ip_LookupConfig(
      XPAR_XRV32I_NPP_IP_0_DEVICE_ID);
  XRv32i_npp_ip_CfgInitialize(&ip, cfg_ptr);
  XRv32i_npp_ip_Set_start_pc(&ip, 0);
  for (int i=0; i<CODE_RAM_SIZE; i++)
    code_ram[i] = input_code_ram[i];
  XRv32i_npp_ip_Start(&ip);
  while (!XRv32i_npp_ip_IsDone(&ip));
  printf("%d fetched and decoded instructions\n",
    (int)XRv32i_npp_ip_Get_nb_instruction(&ip));
  printf("data memory dump (non null words)\n");
  for (int i=0; i<DATA_RAM_SIZE; i++){
    w = data_ram[i];
    if (w != 0)
      printf("m[%5x] = %16d (%8x)\n", 4*i, (int)w,
            (unsigned int)w);
  }
  return 0;
}
```

If the RISC-V code in the test_mem.h file is run, the helloworld driver outputs what is shown in Listing 11.3.

Listing 11.3 The helloworld run output

```
88 fetched and decoded instructions
data memory dump (non null words)
m[    0] =              1 (       1)
m[    4] =      2 (       2)
m[    8] =              3 (       3)
m[    c] =              4 (       4)
m[   10] =      5 (       5)
m[   14] =      6 (       6)
m[   18] =      7 (       7)
m[   1c] =      8 (       8)
m[   20] =              9 (       9)
m[   24] =         10 (       a)
m[   2c] =             55 (      37)
```

11.3 Connecting Multiple CPUs and Multiple RAMs Through an AXI Interconnect

All the source files related to the multi_core_multi_ram_ip can be found in the multi_core_multi_ram_ip folder.

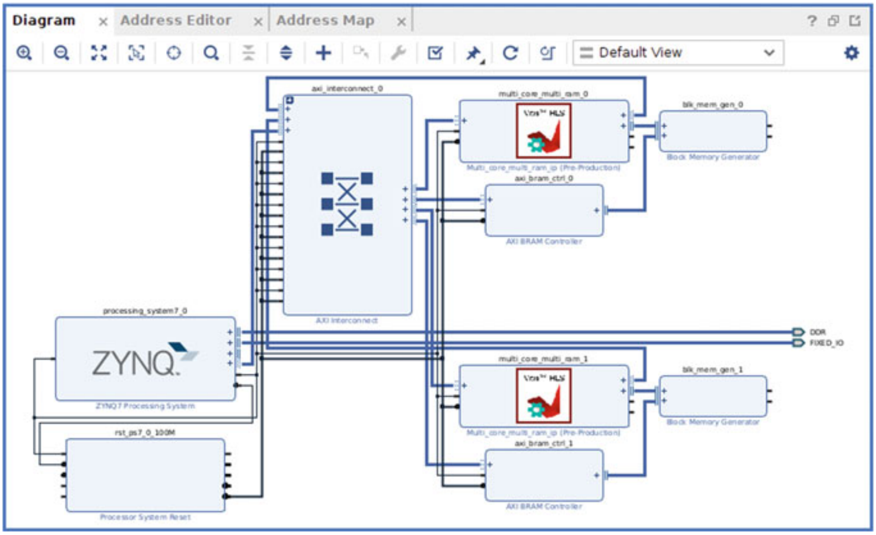


Fig. 11.13 Two CPUs and two RAMs interconnected with an AXI Interconnect IP

11.3.1 The Multiple IPs Design

The second design connects a set of CPUs and a set of RAM blocks. They communicate through the AXI Interconnect IP. Each CPU accesses its local memory bank (direct connection) and also any of the other memory banks (through the AXI interconnect).

Figure 11.13 shows the Vivado design. The design shown includes two CPUs but can be extended up to the AXI Interconnect IP extensibility (i.e. up to 16 slaves and 16 masters on a single AXI Interconnect IP).

Each CPU is an AXI slave to receive its input data from the Zynq IP. It is also an AXI master to access non local memory banks. So, the AXI Interconnect IP has four master ports and three slave ports ($2n$ master ports and $n + 1$ slave ports for n CPUs and n memory banks). The master ports are on the right side of the AXI interconnect IP and the slave ports are on the upper part of the left side.

11.3.2 The CPU Top Function

The code in the multi_core_multi_ram_ip.cpp file which defines the CPUs is shown in Listing 11.4.

The top function receives its identity (ip_num) through the axilite interface. It has an access to its local part of the shared memory (local_ram). It also has an AXI master port to access the full shared memory (data_ram with an m_axi interface).

Listing 11.4 The multi_core_multi_ram_ip.cpp file defining the IP chips

```
#include "ap_int.h"
#include "multi_core_multi_ram_ip.h"
void multi_core_multi_ram_ip(
  int ip_num,
  int local_ram[LOCAL_RAM_SIZE],
  int data_ram [RAM_SIZE]){
#pragma HLS INTERFACE s_axilite port=ip_num
#pragma HLS INTERFACE bram      port=local_ram
#pragma HLS INTERFACE m_axi     port=data_ram offset=slave
#pragma HLS INTERFACE s_axilite port=return
  ip_num_t              ip;
  ip_num_p1_t           next_ip;
  int                   local_value  = 18;
  int                   global_value = 19;
  ap_uint<5>            i;
  ap_uint<1>            i0;
  ap_uint<LOG_RAM_SIZE> offset, i_div_2;
  unsigned int          global_address, local_address;
  ip      = ip_num;
  next_ip = (ip_num_p1_t)ip + 1;
  offset  =
   (((ap_uint<LOG_RAM_SIZE>)next_ip)<<LOG_LOCAL_RAM_SIZE) + 8;
  for (i=0; i<16; i++){
#pragma HLS PIPELINE II=10
    i0             = i;
    i_div_2        = i >> 1;
    global_address = (unsigned int)(offset + i_div_2);
    local_address  = (unsigned int)(i_div_2);
    if (i0 == 0){
      local_ram[local_address]  = local_value;
      data_ram [global_address] = global_value;
    }
    else{
      local_value  = local_ram[local_address];
      global_value = data_ram [global_address];
    }
  }
}
```

The Initiation Interval has been set to 10 to avoid overlapping ("#pragma HLS PIPELINE II=10").

Every even CPU cycle, the top function writes to the local memory bank ("local_ram[local_address] = local_value") and to the memory of the CPU next to it ("data_ram [global_address] = global_value").

Every odd cycle, the CPU regenerates local_value and global_value from what was written in the preceding even cycle (which proves the write is effective).

11.3.3 The CPU Header File and the Testbench Code

The header file contains the definitions of the constants. It is shown in Listing 11.5.

Listing 11.5 The multi_core_multi_ram_ip.h file

```
#include "ap_int.h"
#define LOG_NB_RAM         1  //2^LOG_NB_RAM ram blocks
#define LOG_NB_IP          LOG_NB_RAM
```

```
#define LOG_RAM_SIZE       16 //2^LOG_RAM_SIZE words
#define NB_RAM             (1<<LOG_NB_RAM)
#define RAM_SIZE           (1<<LOG_RAM_SIZE)
#define LOG_LOCAL_RAM_SIZE (LOG_RAM_SIZE - LOG_NB_RAM)
#define LOCAL_RAM_SIZE     (1<<LOG_LOCAL_RAM_SIZE)
typedef ap_uint<LOG_NB_IP+1> ip_num_p1_t;
typedef ap_uint<LOG_NB_IP> ip_num_t;
```

When the design is reduced to two CPUs (i.e. LOG_NB_IP is set to 1), the first one writes to bank 0 (local access) and to bank 1 (AXI access) and the second one accesses bank 1 (local access) and bank 0 (AXI access), as shown on the testbench code in Listing 11.6.

Listing 11.6 The testbench_multi_core_multi_ram_ip.cpp file

```
#include "multi_core_multi_ram_ip.h"
int  ram[RAM_SIZE];
int *ram0 = ram;
int *ram1 = &ram[LOCAL_RAM_SIZE];
void multi_core_multi_ram_ip(
  int ip_num,
  int local_ram[LOCAL_RAM_SIZE],
  int data_ram [RAM_SIZE]
);
int main(){
  multi_core_multi_ram_ip(0, ram, ram);
  multi_core_multi_ram_ip(1, &ram[LOCAL_RAM_SIZE], ram);
  printf("ram0 dump\n");
  for (int i=0; i<LOCAL_RAM_SIZE; i++){
    if (ram0[i]!=0)
      printf("ram0[%4d] = %2d\n", 4*i, ram0[i]);
  }
  printf("ram1 dump\n");
  for (int i=0; i<LOCAL_RAM_SIZE; i++){
    if (ram1[i]!=0)
      printf("ram1[%4d] = %2d\n", 4*i, ram1[i]);
  }
  return 0;
}
```

11.4 Simulating, Synthesizing, and Running a Multiple IP Design

Experimentation

To simulate the multi_core_multi_ram_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with multi_core_multi_ram_ip.
The testbench_multi_core_multi_ram_ip.cpp program runs a test to check the possibility for the two cores to access the two memory blocks.

11.4.1 Simulation

The simulation does not behave like the run on the FPGA. On the FPGA, the two CPUs are run in parallel. Global accesses from one CPU are done while the other CPU is running. In the simulation, the first CPU is fully run before the second CPU starts its own run. In the multi_core_multi_ram_ip example, there is no difference in the output though.

The testbench prints what is shown in Listing 11.7.

Listing 11.7 The testbench print

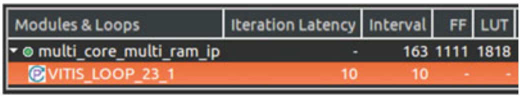
```
ram0 dump
ram0[   0] = 18
ram0[   4] = 18
ram0[   8] = 18
ram0[  12]= 18
ram0[  16] = 18
ram0[  20] = 18
ram0[  24] = 18
ram0[  28] = 18
ram0[  32] = 19
ram0[  36] = 19
ram0[  40] = 19
ram0[  44] = 19
ram0[  48] = 19
ram0[  52] = 19
ram0[  56] = 19
ram0[  60] = 19
ram1 dump
ram1[   0] = 18
ram1[   4] = 18
ram1[   8] = 18
ram1[  12] = 18
ram1[  16] = 18
ram1[  20] = 18
ram1[  24] = 18
ram1[  28] = 18
ram1[  32] = 19
ram1[  36] = 19
ram1[  40] = 19
ram1[  44] = 19
ram1[  48] = 19
ram1[  52] = 19
ram1[  56] = 19
ram1[  60] = 19
```

11.4.2 Synthesis

The synthesis report in Fig. 11.14 shows that the II interval is 10 and the iteration latency is 10 FPGA cycles, because of the external memory access duration through the AXI interconnect.

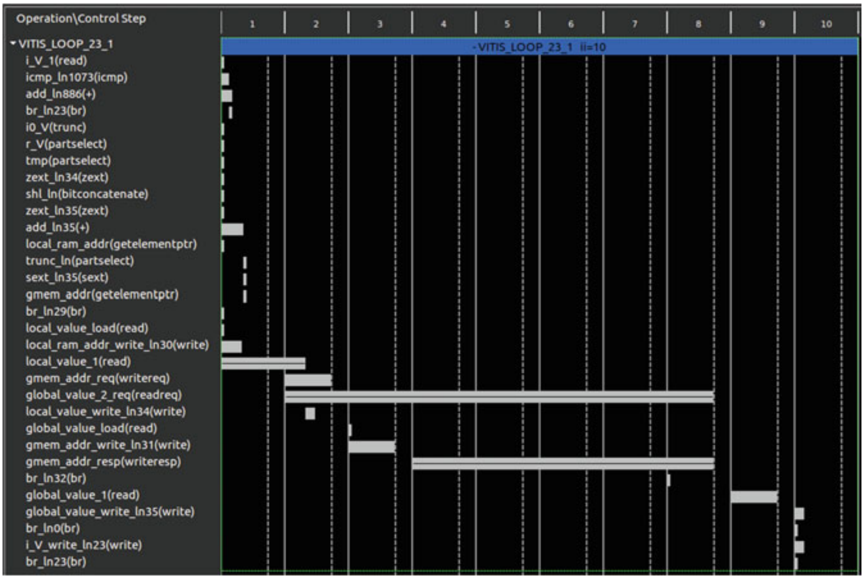
The Schedule Viewer (see Fig. 11.15) shows that the local memory write takes one FPGA cycle (local_ram_addr(getelementptr) and local_ram_addr_write_ln30(write) at cycle 1).

The local memory read takes two cycles (local_value_1(read)).



Modules & Loops	Iteration Latency	Interval	FF	LUT
multi_core_multi_ram_ip	-	163	1111	1818
VITIS_LOOP_23_1	10	10	-	-

Fig. 11.14 Synthesis analysis for the multi core multi RAM design



VITIS_LOOP_23_1
i_V_1(read)
icmp_ln1073(icmp)
add_ln886(+)
br_ln23(br)
i0_V(trunc)
r_V(partselect)
tmp(partselect)
zext_ln34(zext)
shl_ln(bitconcatenate)
zext_ln35(zext)
add_ln35(+)
local_ram_addr(getelementptr)
trunc_ln(partselect)
sext_ln35(sext)
gmem_addr(getelementptr)
br_ln29(br)
local_value_load(read)
local_ram_addr_write_ln30(write)
local_value_1(read)
gmem_addr_req(writereq)
global_value_2_req(readreq)
local_value_write_ln34(write)
global_value_load(read)
gmem_addr_write_ln31(write)
gmem_addr_resp(writeresp)
br_ln32(br)
global_value_1(read)
global_value_write_ln35(write)
br_ln0(br)
i_V_write_ln23(write)
br_ln23(br)

Fig. 11.15 Schedule of the multi_core_multi_ram IP main loop

The global memory write request takes 1+5 cycles (gmem_addr_resp(writeresp) after gmem_addr(getelementptr)).

The read request takes 1+7 cycles (global_value_2_req(readreq) after gmem_addr (getelementptr)).

11.4.3 The Vivado Project

To build the design in Vivado, place on the **Diagram** frame the Zynq7 Processing System IP, **Run Block Automation**, add the AXI Interconnect IP and **Run Connection Automation** to automatically add and connect the Processor System Reset IP. You obtain the Diagram shown in Fig. 11.16.

Then, you must add to the **Diagram** frame two multi_core_multi_ram IPs, two AXI BRAM Controller IPs and two Block Memory Generator IPs, as shown in Fig. 11.17.

The AXI Interconnect IP must be customized to offer four master ports and three slave ports (select the IP and right click, then **customize block**). Set the number of slave interfaces to 3 and the number of master interfaces to 4.

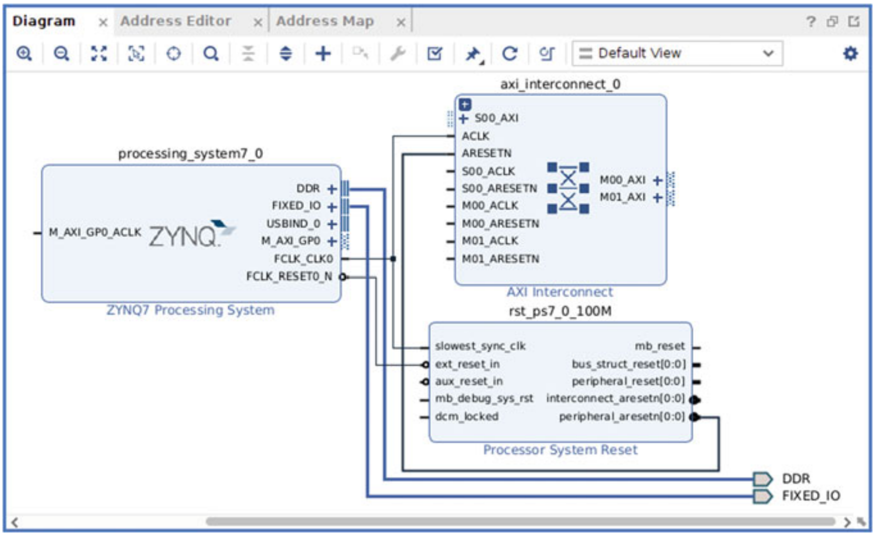


Fig. 11.16 The system IPs to start building the design

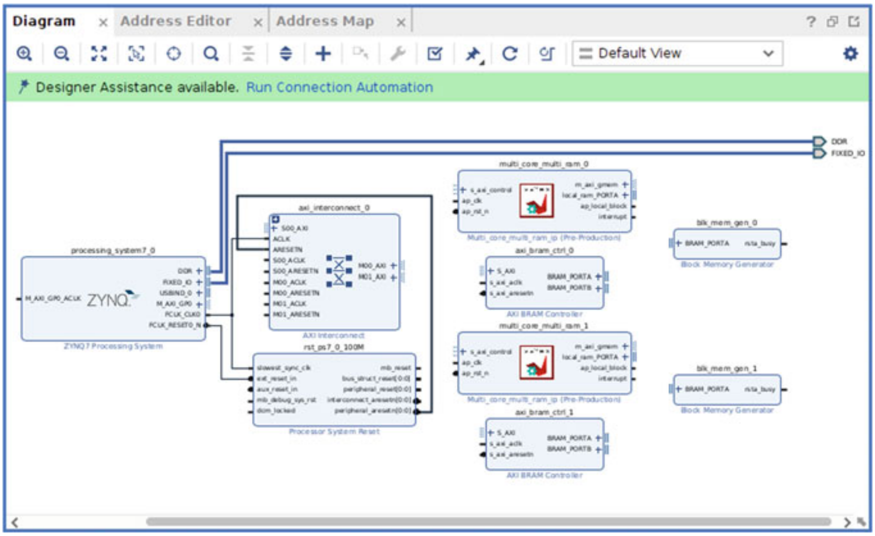

Designer Assistance available. Run Connection Automation

Fig. 11.17 The set of IPs to build the design

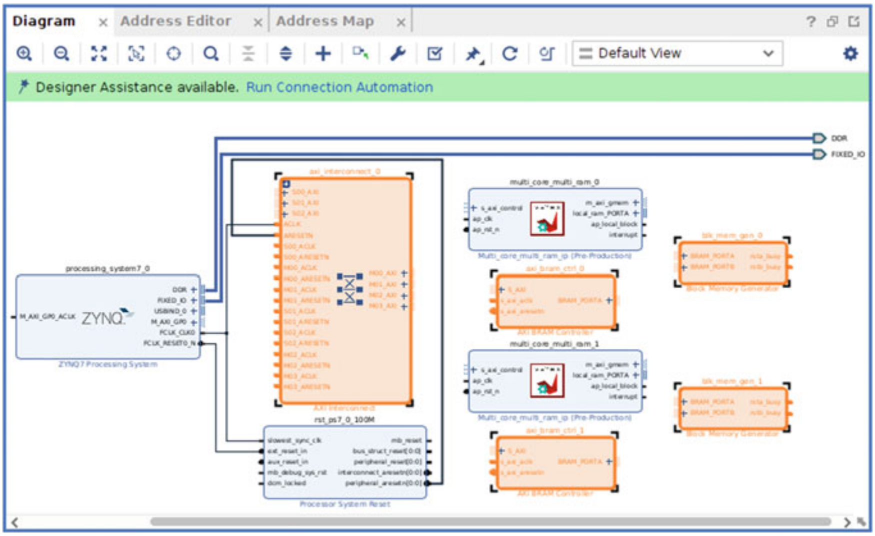


Fig. 11.18 The set of customized IPs to build the design

The two Block Memory Generator IPs should also be customized. Set the **Memory Type** to **True Dual Port RAM**.

The two AXI BRAM Controller IPs are also customized. Set the AXI Protocol to **AXI4LITE** and the **Number of BRAM Interfaces** to 1.

The updated **Diagram** frame is shown in Fig. 11.18.

The next step is to connect the slave AXI Interconnect links as shown in Fig. 11.19.

The next step is to connect the master AXI Interconnect links as shown in Fig. 11.20.

The next step is to connect the Block Memory Generator IPs links as shown in Fig. 11.21.

Lastly, you can let the automatic system finish the wiring (**Run Connection Automation**) to obtain the design shown in Fig. 11.13.

The components connected to the AXI interconnect must be placed in the memory mapped address space. Each component is assigned to a base address. You do this by opening the **Address Editor** frame as shown in Fig. 11.22.

Select all the unassigned lines and proceed to their default assignment (right-click on the line and select **Assign**; you can select all the lines and assign them globally). You obtain the default assignment as shown in Fig. 11.23.

You may exclude the unused address spaces.

For example, the multi_core_multi_ram_0 IP does not need an external (AXI) access to its own multi_core_multi_ram_0 internal code memory, neither to the multi_core_multi_ram_1 IP code memory.

Right-click on multi_core_multi_ram_0 and on multi_core_multi_ram_1 for the multi_core_multi_ram_0 IP and do the same for the multi_core_multi_ram_1 IP. Select **Exclude**.

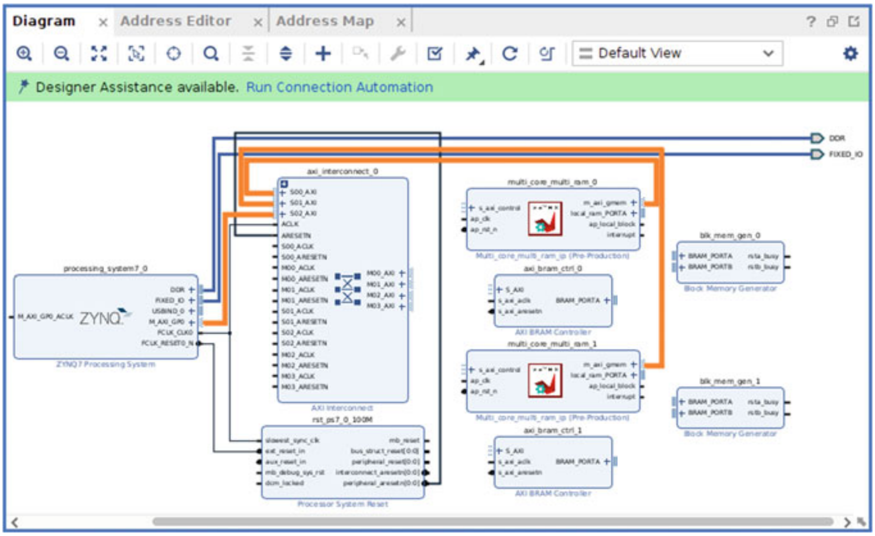

Designer Assistance available. Run Connection Automation

Fig. 11.19 The slave AXI Interconnect links

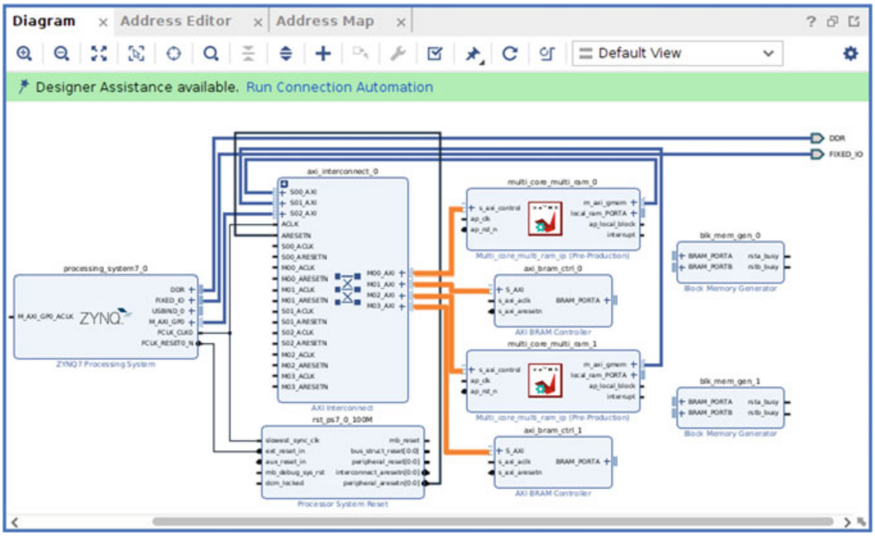

Designer Assistance available. Run Connection Automation

Fig. 11.20 The master AXI Interconnect links

Figure 11.24 shows the updated address space after exclusion.

When the addresses have been edited, you can validate your design (right-click on the **Diagram** frame, select **Validate Design**). You can create the HDL wrapper and generate the bitstream.

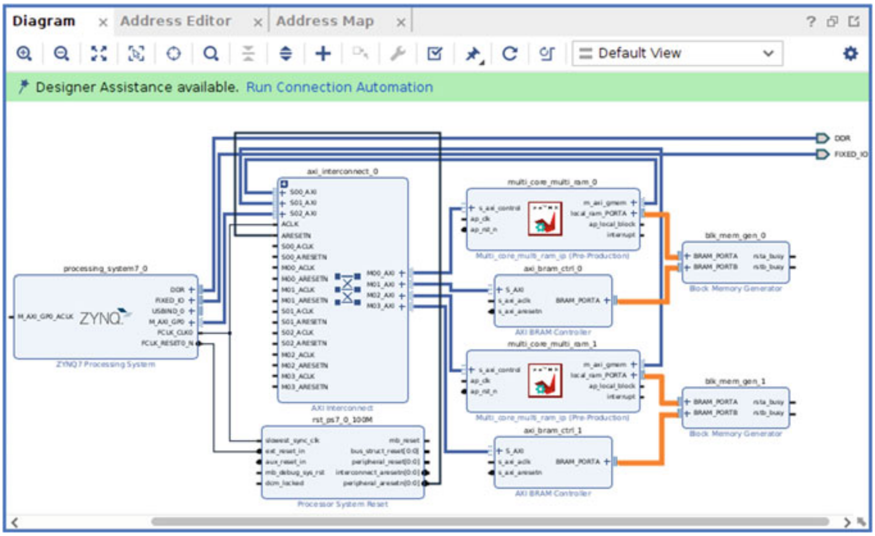
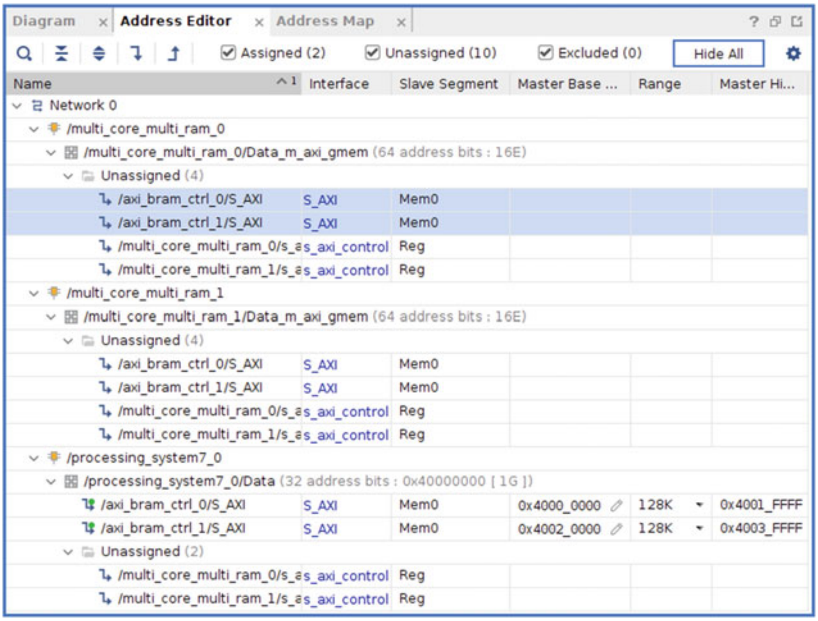


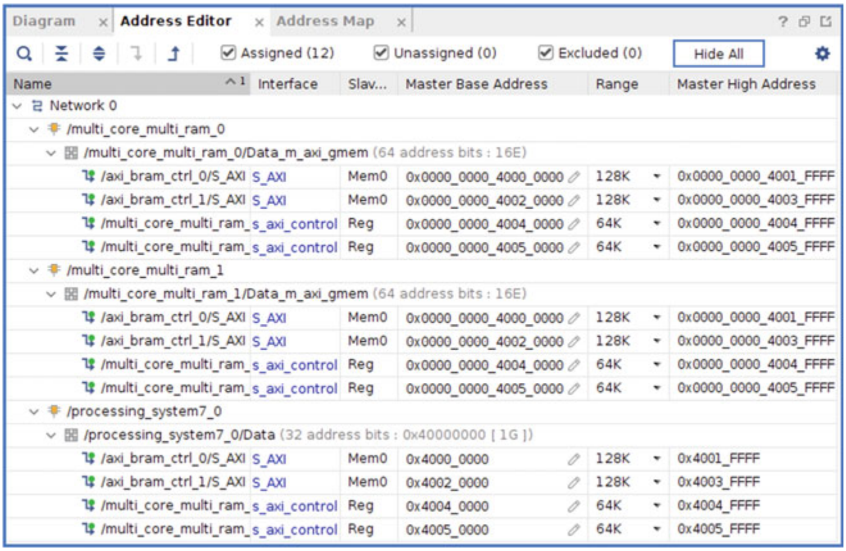
Fig. 11.21 The Block Memory Generator IPs links



☑ Assigned (2) ☑ Unassigned (10) ☑ Excluded (0)

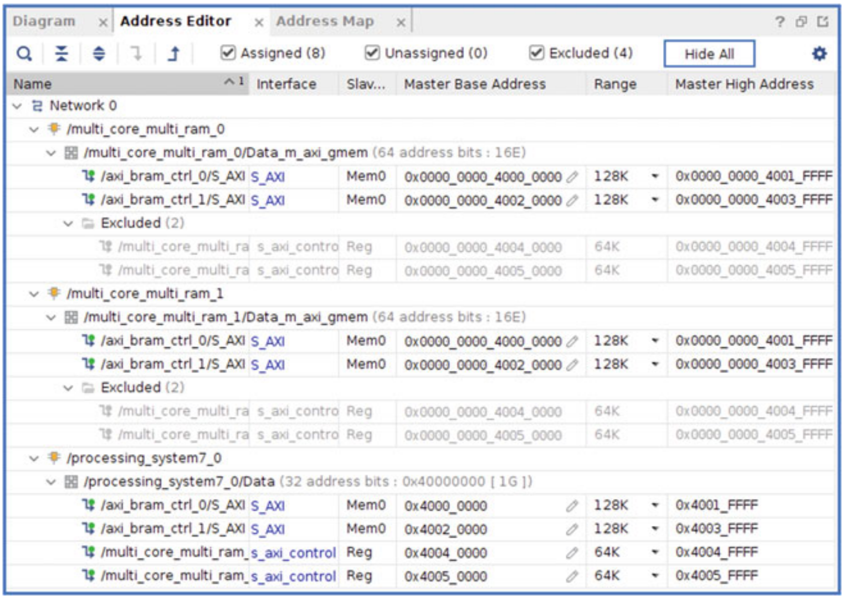
Name	Interface	Slave Segment	Master Base ...	Range	Master Hi...
Network 0					
/multi_core_multi_ram_0					
/multi_core_multi_ram_0/Data_m_axi_gmem (64 address bits : 16E)					
Unassigned (4)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0			
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0			
/multi_core_multi_ram_0/s_a	s_axi_control	Reg			
/multi_core_multi_ram_1/s_a	s_axi_control	Reg			
/multi_core_multi_ram_1					
/multi_core_multi_ram_1/Data_m_axi_gmem (64 address bits : 16E)					
Unassigned (4)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0			
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0			
/multi_core_multi_ram_0/s_a	s_axi_control	Reg			
/multi_core_multi_ram_1/s_a	s_axi_control	Reg			
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	128K	0x4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x4002_0000	128K	0x4003_FFFF
Unassigned (2)					
/multi_core_multi_ram_0/s_a	s_axi_control	Reg			
/multi_core_multi_ram_1/s_a	s_axi_control	Reg			

Fig. 11.22 The **Address Editor** frame



Name	Interface	Slav...	Master Base Address	Range	Master High Address
Network 0					
/multi_core_multi_ram_0					
/multi_core_multi_ram_0/Data_m_axi_gmem (64 address bits : 16E)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000_4000_0000	128K	0x0000_0000_4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x0000_0000_4002_0000	128K	0x0000_0000_4003_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x0000_0000_4004_0000	64K	0x0000_0000_4004_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x0000_0000_4005_0000	64K	0x0000_0000_4005_FFFF
/multi_core_multi_ram_1					
/multi_core_multi_ram_1/Data_m_axi_gmem (64 address bits : 16E)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000_4000_0000	128K	0x0000_0000_4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x0000_0000_4002_0000	128K	0x0000_0000_4003_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x0000_0000_4004_0000	64K	0x0000_0000_4004_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x0000_0000_4005_0000	64K	0x0000_0000_4005_FFFF
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	128K	0x4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x4002_0000	128K	0x4003_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x4004_0000	64K	0x4004_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x4005_0000	64K	0x4005_FFFF

Fig. 11.23 The default assignment



Name	Interface	Slav...	Master Base Address	Range	Master High Address
Network 0					
/multi_core_multi_ram_0					
/multi_core_multi_ram_0/Data_m_axi_gmem (64 address bits : 16E)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000_4000_0000	128K	0x0000_0000_4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x0000_0000_4002_0000	128K	0x0000_0000_4003_FFFF
Excluded (2)					
/multi_core_multi_ra	s_axi_contro	Reg	0x0000_0000_4004_0000	64K	0x0000_0000_4004_FFFF
/multi_core_multi_ra	s_axi_contro	Reg	0x0000_0000_4005_0000	64K	0x0000_0000_4005_FFFF
/multi_core_multi_ram_1					
/multi_core_multi_ram_1/Data_m_axi_gmem (64 address bits : 16E)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000_4000_0000	128K	0x0000_0000_4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x0000_0000_4002_0000	128K	0x0000_0000_4003_FFFF
Excluded (2)					
/multi_core_multi_ra	s_axi_contro	Reg	0x0000_0000_4004_0000	64K	0x0000_0000_4004_FFFF
/multi_core_multi_ra	s_axi_contro	Reg	0x0000_0000_4005_0000	64K	0x0000_0000_4005_FFFF
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	128K	0x4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x4002_0000	128K	0x4003_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x4004_0000	64K	0x4004_FFFF
/multi_core_multi_ram_	s_axi_control	Reg	0x4005_0000	64K	0x4005_FFFF

Fig. 11.24 The updated address assignment

11.4.4 Running the Multiple IP Design

Experimentation

To run the multi_core_multi_ram_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with multi_core_multi_ram_ip.

The code to drive the FPGA is shown in Listing 11.8.

Once the two CPUs have been initialized, their arguments ip_num and data_ram must be set (XMulti_core_multi_ram_ip_Set_ip_num and XMulti_core_multi_ram_ip_Set_data_ram).

Then, the CPUs are started. The driver waits until they are both done before it prints the non null memory words.

Listing 11.8 The helloworld.c file driving the FPGA

```
#include <stdio.h>
#include "xmulti_core_multi_ram_ip.h"
#include "xparameters.h"
#define LOG_NB_RAM         1  //2^LOG_NB_RAM ram blocks
#define LOG_RAM_SIZE       16 //2^LOG_RAM_SIZE words
#define LOG_LOCAL_RAM_SIZE (LOG_RAM_SIZE - LOG_NB_RAM)
#define LOCAL_RAM_SIZE     (1<<LOG_LOCAL_RAM_SIZE)
#define BASE_RAM 0x40000000 int *ram0 = (int *)(BASE_RAM + 0); int
    *ram1 = (int
*)(BASE_RAM + 0x20000); XMulti_core_multi_ram_ip_Config *cfg_ptr0;
XMulti_core_multi_ram_ip_Config *cfg_ptr1; XMulti_core_multi_ram_ip
ip0; XMulti_core_multi_ram_ip ip1; int main(){
  cfg_ptr0 = XMulti_core_multi_ram_ip_LookupConfig(
      XPAR_XMULTI_CORE_MULTI_RAM_IP_0_DEVICE_ID);
  XMulti_core_multi_ram_ip_CfgInitialize(&ip0, cfg_ptr0);
  XMulti_core_multi_ram_ip_Set_ip_num(&ip0, 0);
  XMulti_core_multi_ram_ip_Set_data_ram(&ip0, BASE_RAM);
  cfg_ptr1 = XMulti_core_multi_ram_ip_LookupConfig(
      XPAR_XMULTI_CORE_MULTI_RAM_IP_1_DEVICE_ID);
  XMulti_core_multi_ram_ip_CfgInitialize(&ip1, cfg_ptr1);
  XMulti_core_multi_ram_ip_Set_ip_num(&ip1, 1);
  XMulti_core_multi_ram_ip_Set_data_ram(&ip1, BASE_RAM);
  XMulti_core_multi_ram_ip_Start(&ip0);
  XMulti_core_multi_ram_ip_Start(&ip1);
  while (!XMulti_core_multi_ram_ip_IsDone(&ip0));
  while (!XMulti_core_multi_ram_ip_IsDone(&ip1));
  printf("ram0 dump\n");
  for (int i=0; i<LOCAL_RAM_SIZE; i++){
    if (ram0[i]!=0)
      printf("ram0[%2d] = %2d\n", 4*i, ram0[i]);
  }
  printf("ram1 dump\n");
  for (int i=0; i<LOCAL_RAM_SIZE; i++){
    if (ram1[i]!=0)
      printf("ram1[%2d] = %2d\n", 4*i, ram1[i]);
  }
  return 0;
}
```

The execution on the FPGA prints the same lines as in the simulation.

A Multicore RISC-V Processor

12

Abstract

This chapter will make you build your first multicore RISC-V CPU. The processor is built from multiple IPs, each being a copy of the multicycle_pipeline_ip presented in Chap. 9. Each core has its own code and data memories. The data memory banks are interconnected with an AXI interconnect IP. An example of a parallelized matrix multiplication is used to measure the speedup when increasing the number of cores from one to eight.

12.1 An Adaptation of the Multicycle_pipeline_ip to Multicore

All the source files related to the multicore_multicycle_ip can be found in the multicore_multicycle_ip folder.

Figure 12.1 presents the design of a 4-core IP. Each core has a local internal code memory, filled with RISC-V code by the Zynq through the AXI interconnect.

The cores implement the multicycle pipeline design presented in Chap. 9.

Each core has a direct access to an external data memory bank (four cores, four data memory banks). It also has an indirect access to the other data banks through the AXI interconnect.

The Zynq also has access to the data memory banks, either to initialize arguments before run or to dump results after.

The memory model is the one described in Sect. 10.2 with cores instead of harts.

The codes presented in the chapter describe the implementation of anyone of the multiple cores composing the processor, not all the processor.

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics
in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_12

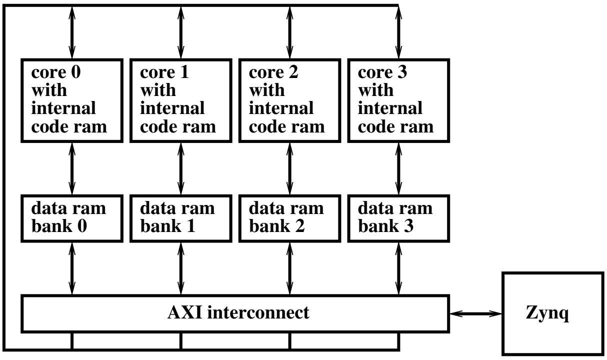

core 0 with internal code ram
core 1 with internal code ram
core 2 with internal code ram
core 3 with internal code ram
data ram bank 0
data ram bank 1
data ram bank 2
data ram bank 3
AXI interconnect
Zynq

Fig. 12.1 A 4-core IP

12.1.1 Adding an IP Number to the Top Function Prototype

The multicycle_pipeline_ip top function is located in the multicycle_pipeline_ip.cpp file.

Its prototype is shown in Listing 12.1.

The ip_num argument, with an axilite interface, is the core identification number sent by the Zynq.

The data ram is accessible through two arguments.

The first one named ip_data_ram is an access to the local data ram bank. It has a bram interface, which implies a direct connection between the core IP and a Block Memory Generator IP (the final Vivado design for two cores is shown in Fig. 12.3). Through ip_data_ram, the core can read and write bytes or words from/to its local memory bank (LOAD/STORE RISC-V instructions with an access address local to the IP data ram).

The second data ram argument is named data_ram. It has a master AXI interface as explained in the preceding chapter. Through data_ram, the core can read and write words from/to any remote memory bank (LOAD/STORE RISC-V instructions with an access address external to the IP data ram).

According to the word access address, the memory access is a local one (when 0 <= address < IP_DATA_RAM_SIZE) or a remote one (when address < 0 or address >= IP_DATA_RAM_SIZE).

Addresses are relative to the core. Before a remote access, the address is turned to an absolute value into the data_ram address space (DATA_RAM_SIZE) by adding ip_num * IP_DATA_RAM_SIZE. The AXI interconnect IP routes the absolute address to the target memory bank through an AXI BRAM Controller where the access is done using the memory block port B.

Listing 12.1 The transformed multicycle_pipeline_ip top function for a multicore design

```
void multicycle_pipeline_ip(
  unsigned int  ip_num,
  unsigned int  start_pc,
  unsigned int  ip_code_ram[IP_CODE_RAM_SIZE],
  int           ip_data_ram[IP_DATA_RAM_SIZE],
  int           data_ram    [NB_IP][IP_DATA_RAM_SIZE],
  unsigned int *nb_instruction,
  unsigned int *nb_cycle){
#pragma HLS INTERFACE s_axilite port=ip_num
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=ip_code_ram
#pragma HLS INTERFACE bram      port=ip_data_ram
#pragma HLS INTERFACE m_axi     port=data_ram offset=slave
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=nb_cycle
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INLINE recursive
  ...
```

For the meaning of the m_axi and bram interfaces, please refer back to Sect. 11.2.1.

The memory arrays have sizes depending on the number of IPs (see Listing 12.2), to keep the total memory within the limit of 512 KB (to avoid going beyond the 540 KB available on the FPGA).

For a 2-core IP, each core has a 128 KB (32K instructions) code memory and a 128 KB (32K words) data memory. The size is 64 KB + 64 KB for a 4-core IP and 32 KB + 32 KB for an 8-core IP.

Listing 12.2 The size of the code and data memories defined in the multicycle_pipeline_ip.h file

```
...
#define LOG_NB_IP               1
#define NB_IP                  (1<<LOG_NB_IP)
#define LOG_CODE_RAM_SIZE      16
#define CODE_RAM_SIZE          (1<<LOG_CODE_RAM_SIZE)
#define LOG_DATA_RAM_SIZE      16
#define DATA_RAM_SIZE          (1<<LOG_DATA_RAM_SIZE)
#define LOG_IP_CODE_RAM_SIZE   (LOG_CODE_RAM_SIZE-LOG_NB_IP)//in
     word
#define IP_CODE_RAM_SIZE       (1<<LOG_IP_CODE_RAM_SIZE)
#define LOG_IP_DATA_RAM_SIZE   (LOG_DATA_RAM_SIZE-LOG_NB_IP)//in
     words
#define IP_DATA_RAM_SIZE       (1<<LOG_IP_DATA_RAM_SIZE)
...
```

12.1.2 The IP Top Function Declarations

Like in the multihart_ip design, the IP top function local declarations (see Listing 12.3) add _from_ variables to the multicycle_pipeline_ip.

For example, the f_to_d variable has a matching d_from_f variable.

Listing 12.3 The top function declarations of _from_ and _to_ variables

```
  ...
  int               reg_file        [NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file          dim=1
    complete
  bit_t             is_reg_computed[NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=is_reg_computed   dim=1
    complete
  from_f_to_f_t f_from_f;
  from_d_to_f_t f_from_d;
  from_e_to_f_t f_from_e;
  from_f_to_f_t f_to_f;
  from_f_to_d_t f_to_d;
  from_f_to_d_t d_from_f;
  from_d_to_f_t d_to_f;
  from_d_to_i_t d_to_i;
  from_d_to_i_t i_from_d;
  bit_t         i_wait;
  i_safe_t      i_safe;
  from_i_to_e_t i_to_e;
  from_i_to_e_t e_from_i;
  from_e_to_f_t e_to_f;
  from_e_to_m_t e_to_m;
  from_e_to_m_t m_from_e;
  from_m_to_w_t m_to_w;
  from_m_to_w_t w_from_m;
  bit_t         is_running;
  counter_t     nbi;
  counter_t     nbc;
  ...
```

12.1.3 The IP Top Function Initializations

Listing 12.4 shows the top function initializations. The inter-stage links valid bits are all cleared except the f_to_f one. The main loop starts as if the fetch stage would send the start_pc to itself.

Listing 12.4 The top function initializations

```
  ...
  init_reg_file (ip_num, reg_file, is_reg_computed);
  f_to_f.is_valid   = 1;
  f_to_f.next_pc    = start_pc;
  f_to_d.is_valid   = 0;
  d_to_f.is_valid   = 0;
  d_to_i.is_valid   = 0;
  i_to_e.is_valid   = 0;
  e_to_f.is_valid   = 0;
  e_to_m.is_valid   = 0;
  m_to_w.is_valid   = 0;
  i_wait            = 0;
  i_safe.is_full    = 0;
  nbi               = 0;
  nbc               = 0;
  ...
```

12.1.4 The IP Top Function Main Loop

The do ... while loop (see Listing 12.5) is slightly modified to incorporate the new ip_num, ip_code_ram, ip_data_ram, and data_ram arguments into the fetch, mem_access, and write_back function calls.

Eventually, like in the multihart_ip design the order of the calls goes from the fetch stage to the writeback stage. This is to ensure that the synthesis is able to stand the II=2 constraint.

Listing 12.5 The do ... while loop

```
  ...
  do{
#pragma HLS PIPELINE II=2
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("=============================================\n");
    printf("cycle %d\n", (int)nbc);
#endif
#endif
    new_cycle(f_to_f, d_to_f, e_to_f, f_to_d, d_to_i, i_to_e,
              e_to_m, m_to_w, &f_from_f, &f_from_d, &f_from_e,
             &d_from_f, &i_from_d, &e_from_i, &m_from_e,
             &w_from_m);
    fetch(f_from_f, f_from_d, f_from_e, i_wait, ip_code_ram,
         &f_to_f, &f_to_d);
    decode(d_from_f, i_wait, &d_to_f, &d_to_i);
    issue(i_from_d, reg_file, is_reg_computed, &i_safe,
         &i_to_e, &i_wait);
    execute(
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
            ip_num,
#endif
#endif
            e_from_i,
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
            reg_file,
#endif
#endif
           &e_to_f, &e_to_m);
    mem_access(ip_num, m_from_e, ip_data_ram, data_ram, &m_to_w);
    write_back(
#ifndef __SYNTHESIS__
               ip_num,
#endif
               w_from_m, reg_file, is_reg_computed);
    statistic_update(w_from_m, &nbi, &nbc);
    running_cond_update(w_from_m, &is_running);
  } while (is_running);
  ...
```

12.1.5 The Register File Initialization

The register file initialization (the init_reg_file function defined in the multicycle_pipeline_ip.cpp file; see Listing 12.6) sets register a0 (x10) with the core identification number received as the first argument.

Moreover, it initializes the sp register of each core with the address of the first word of the next core memory bank. When places are allocated on the stack by decreasing the sp register, the locations are within the core's memory bank. As a result, each core has its own local stack.

Listing 12.6 The init_reg_file function

```
//a0/x10 is set with the IP number
static void init_reg_file(
  ip_num_t ip_num,
  int      *reg_file,
  bit_t    *is_reg_computed){
  reg_num_p1_t r;
  for (r=0; r<NB_REGISTER; r++){
#pragma HLS UNROLL
    is_reg_computed[r] = 0;
    if (r==10)
      reg_file      [r] = ip_num;
    else if (r==SP)
      reg_file      [r] = (1<<(LOG_IP_DATA_RAM_SIZE+2));
    else
      reg_file      [r] = 0;
  }
}
```

12.1.6 The Memory Access

The fetch (in the fetch.cpp file), decode (in the decode.cpp file), issue (in the issue.cpp file), execute (in the execute.cpp file), and write_back (in the wb.cpp file) functions are mostly unchanged from the multicycle_pipeline_ip implementation.

The mem_access function (in the mem_access.cpp file) is shown in Listing 12.7.

As the memory is partitioned, the mem_access function determines, from the access address, if the access is local (is_local) and otherwise, which is the accessed partition (accessed_ip).

The mem_access function calls the stage_job function and the set_output_to_w function (defined in the same file).

Listing 12.7 The mem_access function

```
void mem_access(
  ip_num_t         ip_num,
  from_e_to_m_t    m_from_e,
  int             *ip_data_ram,
  int              data_ram[][IP_DATA_RAM_SIZE],
  from_m_to_w_t   *m_to_w){
  int       value;
  ip_num_t accessed_ip;
  bit_t     is_local;
  if (m_from_e.is_valid){
    value           = m_from_e.value;
    accessed_ip     =
      (m_from_e.address>>(LOG_IP_DATA_RAM_SIZE+2)) + ip_num;
    is_local        = (ip_num == accessed_ip);
    stage_job(accessed_ip, is_local, m_from_e.is_load,
              m_from_e.is_store, m_from_e.address,
              m_from_e.func3, ip_data_ram, data_ram, &value);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("mem       ");
    printf("%04d\n", (int)(m_from_e.pc<<2));
#endif
#endif
    set_output_to_w(m_from_e.rd, m_from_e.has_no_dest,
                    m_from_e.is_load, m_from_e.is_ret,
                    m_from_e.value, value,
#ifndef __SYNTHESIS__
                    m_from_e.pc,  m_from_e.instruction,
                    m_from_e.d_i, m_from_e.target_pc,
#endif
                    m_to_w);
  }
  m_to_w->is_valid = m_from_e.is_valid;
}
```

The stage_job function (see Listing 12.8) calls either mem_load or mem_store according to the memory access type.

Listing 12.8 The stage_job function in the mem_access.cpp file

```
static void stage_job(
  ip_num_t          accessed_ip,
  bit_t             is_local,
  bit_t             is_load,
  bit_t             is_store,
  b_data_address_t  address,
  func3_t           func3,
  int              *ip_data_ram,
  int               data_ram[][IP_DATA_RAM_SIZE],
  int              *value){
  if (is_load)
    *value =
      mem_load (accessed_ip, is_local,
                ip_data_ram, data_ram, address, func3);
  else if (is_store)
      mem_store(accessed_ip, is_local,
                ip_data_ram, data_ram, address, *value,
               (ap_uint<2>)func3);
}
```

The memory load and store functions (defined in the mem.cpp file) are unchanged except for an if statement with the is_local condition to either access the local memory (ip_data_ram) or the global one (data_ram).

In the mem_load function (see Listing 12.9), the addressed word is fully read from ip_data_ram (local access) or from data_ram (global access). After the access, the mem_load function selects the accessed bytes. The code is similar to the one shown in Listing 12.11.

The local access takes one processor cycle and the global access takes five. Hence, the variable *w* is filled after a variable latency.

Listing 12.9 The beginning of the mem_load function

```
int mem_load(
  ip_num_t          ip,
  bit_t             is_local,
  int              *ip_data_ram,
  int               data_ram[][IP_DATA_RAM_SIZE],
  b_data_address_t  address,
  func3_t           msize){
  ap_uint<2>           a01 = address;
  bit_t                a1  = address>>1;
  w_ip_data_address_t  a2  = address>>2;
  int                  result;
  char                 b, b0, b1, b2, b3;
  unsigned char        ub, ub0, ub1, ub2, ub3;
  short                h, h0, h1;
  unsigned short       uh, uh0, uh1;
  int                  w, ib, ih;
  unsigned int         iub, iuh;
  if (is_local)
    w =  ip_data_ram[a2];
  else
    w = data_ram[ip][a2];
...
```

There is a possible problem on two back-to-back accesses to an external memory partition, i.e. a store immediately followed by a load at the same address (see Listing 12.10: the accessed address is negative, i.e. external). In this case, the load starts its external access while the store is still in progress in the AXI interconnect IP (the arbitration policy in the AXI interconnect IP is complex as described in [1]: it is difficult to know at which cycles the store and the load accesses are done in the memory bank). After testing on the FPGA, it turns out that the loaded value is not the stored one as it should be.

Listing 12.10 A remote store followed by a remote load at the same address

```
 li    a0,-4
/*access prior IP memory partition*/
sw    t1,0(a0)
lw    t0,0(a0)
```

To serialize the load after the store the pipeline should be frozen with a wait condition sent from the memory access stage, in a similar way to the wait condition sent by the issue stage when a locked source is detected.

However, this would impact the CPI for every remote store just because of unoptimized codes (the load after the store is useless).

Nevertheless, the succession of a store and a load at the same address does occur, for example when a compilation at optimization level 0 is done ("-O0"). The compiler does produce such unoptimized code, but the store and load accessed address in this case is local (i.e. in the stack), not remote.

I decided to apply the same relaxed scheduling policy as I did for the issue stage (refer back to Sect. 9.3.3). The programmer is warned that he/she should not use back-to-back store and load to the same external address (there should be at least four instructions in between to ensure correct scheduling).

If a strict scheduling is wished, the memory access stage should be equipped with a safe and a wait signal, like for the issue stage. The wait should be raised when a remote store is processed. It should be maintained during four processor cycles (use a counter to control this). The memory stage wait condition should be added to all the preceding stages (fetch, decode, issue, and execute).

In the mem_store function (see Listing 12.11), the access is done according to the size encoded in the memory access instruction. The data_ram and ip_data_ram word pointers are casted into either char (i.e. byte) or short (i.e. half word) pointers. The access address (i.e. an offset in the selected ram) is also turned into a char (a), short (a1), or word (a2) displacement, with an added IP offset if the access is not local.

Listing 12.11 The mem_store function

```
void mem_store(
  ip_num_t          ip,
  bit_t             is_local,
  int              *ip_data_ram,
  int               data_ram[][IP_DATA_RAM_SIZE],
  b_data_address_t  address,
  int               rv2,
  ap_uint<2>        msize){
  b_ip_data_address_t a      = address;
  h_ip_data_address_t a1     = address>>1;
  w_ip_data_address_t a2     = address>>2;
  char                rv2_0  = rv2;
  short               rv2_01 = rv2;
  switch(msize){
    case SB:
      if (is_local)
        *((char*) (ip_data_ram) + a)
        = rv2_0;
      else
        *((char*) (data_ram) +
                  (((b_data_address_t)ip)<<
                    (LOG_IP_DATA_RAM_SIZE+2)) + a)
        = rv2_0;
      break;
    case SH:
      if (is_local)
        *((short*)(ip_data_ram) + a1)
        = rv2_01;
      else
        *((short*)(data_ram) +
                  (((h_data_address_t)ip)<<
                    (LOG_IP_DATA_RAM_SIZE+1)) + a1)
        = rv2_01;
      break;
    case SW:
      if (is_local)
        ip_data_ram [a2] = rv2;
      else
        data_ram[ip][a2] = rv2;
      break;
    case 3:
      break;
  }
}
```

12.2 Simulating the IP

Experimentation

To simulate the multicore_multicycle_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with multicore_multicycle_ip. There are two testbench programs: testbench_seq_multicore_multicycle_ip.cpp to run independent codes (one per core) and testbench_par_multicore_multicycle_ip.cpp to run a parallel sum of the elements of an array.
With testbench_seq_multicore_multicycle_ip.cpp you can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder. You can also vary the number of cores.

Like in the multihart_ip project, two testbench files are provided, one to run fully independent codes (testbench_seq_multicore_multicycle_ip.cpp) and the other to run codes sharing a distributed array (testbench_par_multicore_multicycle_ip.cpp).

12.2.1 Simulating Independent Programs on the Different IPs

The first testbench file is testbench_seq_multicore_multicycle_ip.cpp. It runs a set of programs distributed in the core code_ram arrays. As an example, I provide a version in which the codes are all built on the same source file (the multicore processor IP runs NB_IP copies of the same program).

The example uses the test_mem.s code presented in Sect. 6.5 as the source.

The hex files used to initialize the code_ram arrays are built with the build_seq.sh shell script.

The script (see Listing 12.12) has one argument which is the name of the test file to be built, e.g. "./build_seq.sh test_mem" to build the test_mem_text.hex file. The data section is based at address 0, which means that the addresses out of the compiler are unchanged by the linker. Hence, the addresses in the code are relative to the running core. For example, address 0 is the start of the running core memory partition, i.e. address 0 of the shared memory if the running core is core 0 or IP_DATA_RAM_SIZE if the running core is core 1.

Listing 12.12 The build_seq.sh shell script

```
$ cat build_seq.sh
riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 -Tdata 0 -o $1.elf
    $1.s
riscv32-unknown-elf-objcopy -O binary --only-section=.text $1.elf
    $1_text.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' $1_text.bin > $1_text.hex
$
```

Once the hex files have been built, they can be used to initialize the code_ram arrays. In the testbench example shown in Listing 12.13, the code_ram array is initialized with the test_mem_text.hex produced by the build_seq.sh shell script from the test_mem.s source file.

The main function runs NB_IP successive calls to multicycle_pipeline_ip. Each call runs one execution of the test_mem.s RISC-V code.

Listing 12.13 The testbench_seq_multicore_multicycle_ip.cpp file

```
#include <stdio.h>
#include "multicycle_pipeline_ip.h"
unsigned int code_ram[IP_CODE_RAM_SIZE]={
#include "test_mem_text.hex"
};
int          data_ram[NB_IP][IP_DATA_RAM_SIZE];
int main(){
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  int          w;
  for (int i=0; i<NB_IP; i++)
    multicycle_pipeline_ip(i, 0, code_ram, &data_ram[i][0],
                           data_ram, &nbi[i], &nbc[i]);
  for (int i=0; i<NB_IP; i++){
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i],
      ((float)nbi[i])/nbc[i]);
    printf("data memory dump (non null words)\n");
    for (int j=0; j<IP_DATA_RAM_SIZE; j++){
      w = data_ram[i][j];
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
          (i*IP_DATA_RAM_SIZE + j)*4, w, (unsigned int)w);
    }
  }
  return 0;
}
```

To run the test_mem.s code on the Vitis_HLS simulation of two cores, you must first set LOG_NB_IP as 1 in the multicycle_pipeline_ip.h file. Then, you must build the .hex files by running "./build_seq.sh test_mem". Eventually, you can start the simulation.

For the run of two copies of test_mem.s, the output is composed of the list of the instructions run in the first core followed by its register file final state and the list of the instructions run in the second core with its register file final state (see Listing 12.14).

Listing 12.14 The output of the main function of the testbench_seq_multicore_multicycle_ip.cpp file: the code run and the final register file state

```
0000: 00000513      li a0, 0
      a0  =                0 (       0)
0004: 00000593      li a1, 0
      a1  =                0 (       0)
...
0056: 00a62223      sw a0, 4(a2)
      m[   2c] =               55 (      37)
0060: 00008067      ret
      pc  =                0 (       0)
...
sp  =           131072 (   20000)
...
a0  =               55 (      37)
```

```
...
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
0000: 00000513      li a0, 0
      a0  =                0 (       0)
0004: 00000593      li a1, 0
      a1  =                0 (       0)
...
0056: 00a62223      sw a0, 4(a2)
      m[2002c] =               55 (      37)
0060: 00008067      ret
      pc  =                0 (       0)
...
sp  =           262144 (   40000)
...
a0  =               55 (      37)
...
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
```

After that, the run outputs for each core the number of instructions run, the number of cycles of the run, and the dump of the data memory bank (non null values) (see Listing 12.15).

Listing 12.15 The output of the main function of the testbench_seq_multicore_multicycle_ip.cpp file: the memory dump

```
core 0: 88 fetched and decoded instructions in 279 cycles (ipc =
    0.32)
data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
core 1: 88 fetched and decoded instructions in 279 cycles (ipc =
    0.32)
data memory dump (non null words)
m[20000] =                1 (       1)
m[20004] =                2 (       2)
m[20008] =                3 (       3)
m[2000c] =                4 (       4)
m[20010] =                5 (       5)
m[20014] =                6 (       6)
m[20018] =                7 (       7)
m[2001c] =                8 (       8)
m[20020] =                9 (       9)
m[20024] =               10 (       a)
m[2002c] =               55 (      37)
```

12.2.2 Simulating a Parallelized Program

The second testbench file is testbench_par_multicore_multicycle_ip.cpp. It runs a distributed and parallel version of the test_mem.s program composed of test_mem_par_ip0.s and test_mem_par_otherip.s.

The code in the test_mem_par_ip0.s file is the core 0 job followed by the sum reduction job. It is the same code as the one already presented in Listings 10.46 to 10.48.

The first core initializes its part of the array (loop .L1), computes its local sum (loop .L2), and after, computes the total sum through remote memory accesses (loop .L3 with a synchronization internal loop).

All the other cores run the code in the test_mem_par_otherip.s file. It is the same code as the one already presented in Listing 10.49.

Each core computes its local sum.

The hex files used to initialize the code_ram arrays are built with the build_par.sh shell script shown in Listing 12.6.

Listing 12.16 The build_par.sh shell script

```
$ cat build_par.sh
./build_par_ip0.sh
./build_par_otherip.sh
$ cat build_par_ip0.sh
riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 -Tdata 0 -o
    test_mem_par_ip0.elf test_mem_par_ip0.s
riscv32-unknown-elf-objcopy -O binary --only-section=.text
    test_mem_par_ip0.elf test_mem_par_ip0_text.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' test_mem_par_ip0_text.bin >
    test_mem_par_ip0_text.hex
$ cat build_par_otherip.sh
riscv32-unknown-elf-gcc -nostartfiles -Ttext 0 -Tdata 0 -o
    test_mem_par_otherip.elf test_mem_par_otherip.s
riscv32-unknown-elf-objcopy -O binary --only-section=.text
    test_mem_par_otherip.elf test_mem_par_otherip_text.bin
hexdump -v -e '"0x" /4 "%08x" ",\n"' test_mem_par_otherip_text.bin
    > test_mem_par_otherip_text.hex
$
```

Once the test_mem_par_ip0_text.hex and test_mem_par_otherip_text.hex files have been built for the number of cores defined both in test_mem_par_ip0.s and in multicycle_pipeline.h, they can be used to initialize the code_ram_0 and code_ram arrays.

Remember that when you change the number of cores, you should do two updates of LOG_NB_IP: a first one in the test_mem_par_ip0.s source code and a second one in the multicycle_pipeline.h file. Then, you must rebuild the hex files with the build_par.sh script.

Listing 12.7 shows the testbench_par_multicore_multicycle_ip.cpp file.

Listing 12.17 The testbench_par_multicore_multicycle_ip.cpp file

```
#include <stdio.h>
#include "multicycle_pipeline_ip.h"
unsigned int code_ram_0[IP_CODE_RAM_SIZE]={
#include "test_mem_par_ip0_text.hex"
};
unsigned int code_ram  [IP_CODE_RAM_SIZE]={
#include "test_mem_par_otherip_text.hex"
};
int data_ram[NB_IP][IP_DATA_RAM_SIZE];
int main(){
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  int          w;
  for (int i=1; i<NB_IP; i++)
    multicycle_pipeline_ip(i, 0, code_ram, &data_ram[i][0],
                           data_ram, &nbi[i], &nbc[i]);
  multicycle_pipeline_ip(0, 0, code_ram_0, &data_ram[0][0],
                         data_ram, &nbi[0], &nbc[0]);
  for (int i=0; i<NB_IP; i++){
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i],
      ((float)nbi[i])/nbc[i]);
    printf("data memory dump (non null words)\n");
    for (int j=0; j<IP_DATA_RAM_SIZE; j++){
      w = data_ram[i][j];
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
          (i*IP_DATA_RAM_SIZE + j)*4, w, (unsigned int)w);
    }
  }
  return 0;
}
```

As I already mentioned in the last chapter, the Vitis_HLS simulation does not work like the run on the FPGA. The simulation runs the core IPs sequentially, i.e. core 0 is first fully run before core 1 starts. If the code run on core 0 reads some memory word written by the code run on core 1, the read misses the written value on the simulation (e.g. because the write is done after the read) but not necessarily on the FPGA (e.g. if in fact the write is done before the read when the two cores are run simultaneously).

This is a general limitation to the simulation of multiple IPs in the Vitis_HLS tool.

To keep the simulation identical to the run on the FPGA, I have organized the testbench to run the writing IPs before the reading one. All the cores except core 0 are first simulated. At the end of each simulation, the result is written to the shared data_ram array.

Then, core 0 is simulated. At the end of the RISC-V program run, remote accesses read the data_ram values written by the other cores.

It should be noticed that this solution does not work when there are multiple RAW dependencies, some from core 0 to core 1 and others from core 1 to core 0. In this case, there is no general solution. You have to blindly try your SoC IP directly on the FPGA without any preliminary simulation check.

The RISC-V code to be run on core 0 successively fills a 10 elements array (loop .L1 in Listing 10.46), sums the elements (loop .L2 in Listing 10.47), saves the sum to memory, gathers and accumulates the sums computed by the other cores (loop .L3 in Listing 12.18), and saves the final sum to memory.

The lw a3,40(a4) load at label .L3 is a remote access (see Listing 12.8). The beq branch after the load is a security to ensure that core 0 waits until the other cores have dumped their local sum to their local memory (then the loaded memory word is not null).

Listing 12.18 The .L3 loop

```
       ...
.L3:   lw      a3,40(a4)    /*a3=t[a4+40]*/
       beq     a3,zero,.L3  /*if (a3==0) goto .L3*/
       add     a0,a0,a3     /*a0+=a3*/
       add     a4,a4,a5     /*a4+=a5*/
       addi    a1,a1,1      /*a1++*/
       bne     a1,a2,.L3    /*if (a1!=a2) goto .L3*/
       ...
```

To run the test_mem_par_ip0.s and test_mem_par_otherip.s codes on the Vitis_HLS simulation of two cores, you must first set LOG_NB_IP as 1 in the test_mem_par_ip0.s and in the multicycle_pipeline_ip.h file. Then, you must build the .hex files by running the build_par.sh shell script. Eventually, you can start the simulation.

For the run on two cores, the output (see Listing 12.9) contains core 1 results followed by core 0 results (as core 1 is run before core 0).

Listing 12.19 The output of the main function of the testbench_par_multicore_multicycle_ip.cpp file: the code run and the register file final state

```
0000: 00351293      slli t0, a0, 3
      t0  =                8 (       8)
0004: 00151313      slli t1, a0, 1
      t1  =                2 (       2)
...
0064: 00a62023      sw a0, 0(a2)
      m[20028] =            155 (       9b)
0068: 00008067      ret
      pc  =                0 (       0)
...
sp  =            262144 (   40000)
...
t0  =                 8 (       8)
t1  =                 2 (       2)
...
a0  =               155 (      9b)
...
a2  =                40 (      28)
a3  =                40 (      28)
a4  =                20 (      14)
...
0000: 00000513      li a0, 0
      a0  =                0 (       0)
0004: 00000593      li a1, 0
      a1  =                0 (       0)
...
0108: 00a62023      sw a0, 0(a2)
      m[  2c] =            210 (       d2)
0112: 00008067      ret
      pc  =                0 (       0)
```

```
...
sp   =           131072 (    20000)
...
a0   =              210 (       d2)
a1   =                2 (        2)
a2   =               44 (       2c)
a3   =              155 (       9b)
a4   =           262144 (    40000)
a5   =           131072 (    20000)
...
```

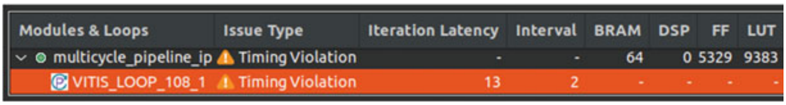
Then (see Listing 12.10) the run dumps the memory banks (the sum of the 20 first integers is 210).

Listing 12.20 The output of the main function of the testbench_par_multicore_multicycle_ip.cpp file: the memory dump

```
...
core 0: 101 fetched and decoded instructions in 273 cycles (ipc =
    0.37)
data memory dump (non null words)
m[     0] =                1 (        1)
m[     4] =                2 (        2)
m[     8] =                3 (        3)
m[     c] =                4 (        4)
m[    10] =                5 (        5)
m[    14] =                6 (        6)
m[    18] =                7 (        7)
m[    1c] =                8 (        8)
m[    20] =                9 (        9)
m[    24] =               10 (        a)
m[    28] =               55 (       37)
m[    2c] =              210 (       d2)
core 1: 90 fetched and decoded instructions in 243 cycles (ipc =
    0.37)
data memory dump (non null words)
m[20000] =               11 (        b)
m[20004] =               12 (        c)
m[20008] =               13 (        d)
m[2000c] =               14 (        e)
m[20010] =               15 (        f)
m[20014] =               16 (       10)
m[20018] =               17 (       11)
m[2001c] =               18 (       12)
m[20020] =               19 (       13)
m[20024] =               20 (       14)
m[20028] =              155 (       9b)
```

12.3 Synthesizing the IP

Figure 12.2 shows that the II=2 constraint is satisfied. The iteration latency, set by the global memory access one, is 13 FPGA cycles. The number of IPs declared in the multicycle_pipeline_ip.h file is two (notice that the number of used resources is for one IP, not for two).



Modules & Loops	Issue Type	Iteration Latency	Interval	BRAM	DSP	FF	LUT
multicycle_pipeline_ip	⚠ Timing Violation	-	-	64	0	5329	9383
VITIS_LOOP_108_1	⚠ Timing Violation	13	2	-	-	-	-

Fig. 12.2 Synthesis report for a 2-core processor

12.4 The Vivado Project

You should separate the design in three Vivado projects: z1_multicore_multicycle_2c_ip for two cores, z1_multicore_multicycle_4c_ip for four cores and z1_multicore_multicycle_8c_ip for eight cores.

The Vivado Design is shown in Fig. 12.3 (example for two cores; you need eight masters and five slaves on the interconnect for four cores and 16 masters and nine slaves for eight cores; do not forget to customize the AXI BRAM controller with one port and the Block Memory Generator with two true dual ports).

Before you build the wrapper, you must set the address map with the Address Editor.

For a 2-core design, the axi_bram_ctrl range is 128 KB and the addresses are 0x4000_0000 and 0x4002_0000, as shown in Fig. 12.4. For a 4-core design the range is 64 KB and the addresses are 0x4000_0000, 0x4001_0000, 0x4002_0000, and 0x4003_0000. For an 8-core design, the range is 32 KB and the addresses are 0x4000_0000, 0x4000_8000, ..., 0x4003_0000, and 0x4003_8000.

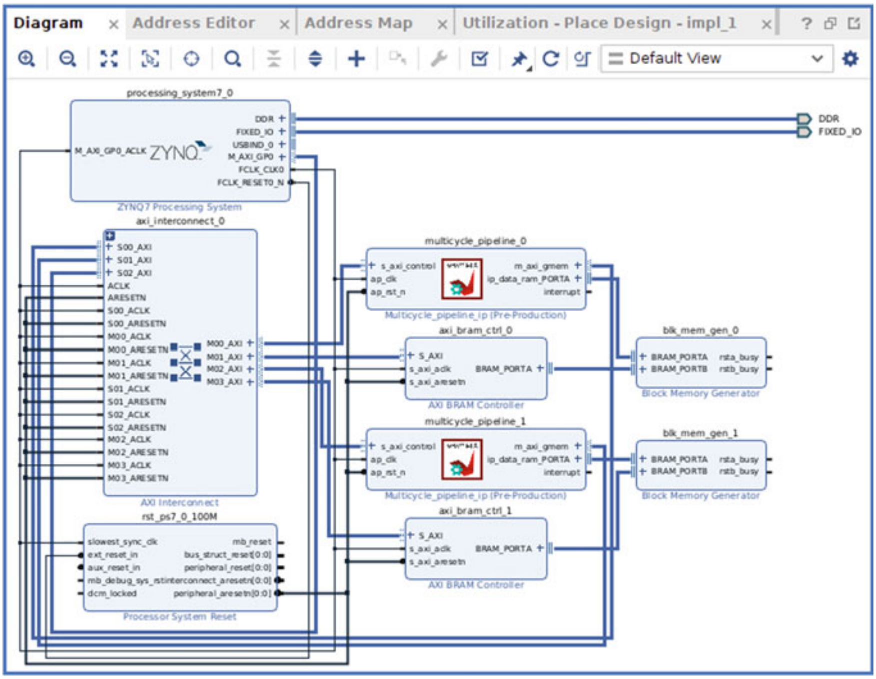
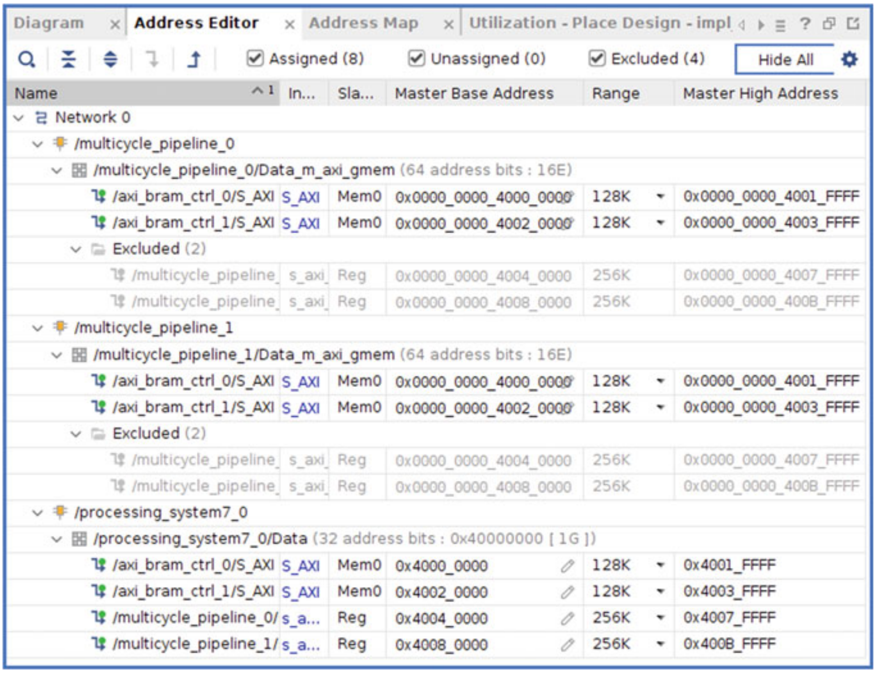


Fig. 12.3 The 2-core design



☑ Assigned (8) ☑ Unassigned (0) ☑ Excluded (4)

Name	In...	Sla...	Master Base Address	Range	Master High Address
Network 0					
/multicycle_pipeline_0					
/multicycle_pipeline_0/Data_m_axi_gmem (64 address bits : 16E)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000_4000_0000	128K	0x0000_0000_4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x0000_0000_4002_0000	128K	0x0000_0000_4003_FFFF
Excluded (2)					
/multicycle_pipeline_	s_axi	Reg	0x0000_0000_4004_0000	256K	0x0000_0000_4007_FFFF
/multicycle_pipeline_	s_axi	Reg	0x0000_0000_4008_0000	256K	0x0000_0000_400B_FFFF
/multicycle_pipeline_1					
/multicycle_pipeline_1/Data_m_axi_gmem (64 address bits : 16E)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000_4000_0000	128K	0x0000_0000_4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x0000_0000_4002_0000	128K	0x0000_0000_4003_FFFF
Excluded (2)					
/multicycle_pipeline_	s_axi	Reg	0x0000_0000_4004_0000	256K	0x0000_0000_4007_FFFF
/multicycle_pipeline_	s_axi	Reg	0x0000_0000_4008_0000	256K	0x0000_0000_400B_FFFF
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	128K	0x4001_FFFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x4002_0000	128K	0x4003_FFFF
/multicycle_pipeline_0/	s_a...	Reg	0x4004_0000	256K	0x4007_FFFF
/multicycle_pipeline_1/	s_a...	Reg	0x4008_0000	256K	0x400B_FFFF

Fig. 12.4 Address mapping for the 2-core design

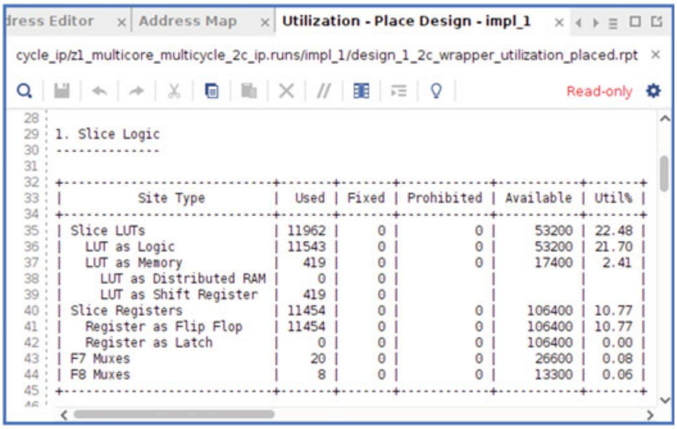
For two cores, the s_axi_control range is 256 KB and the addresses are 0x4004_0000 and 0x4008_0000. For four cores, the range is 128 KB and the addresses are 0x4004_0000, 0x4006_0000, 0x4008_0000, and 0x400A_0000. For eight cores, the range is 64 KB and the addresses are 0x4004_0000, 0x4005_0000, ..., and 0x400B_0000.

The Vivado bitstream generation for a 2-core processor produces the implementation report in Fig. 12.5, showing that it uses 11,962 LUTs (22.48%; the 4-core processor uses 22,155 LUTs; the 8-core processor uses 43,731 LUTs).

12.5 Running the IPs on the Development Board

Experimentation

To run the multicore_multicycle_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with multicore_multicycle_ip. There are two drivers: helloworld_seq.c on which you can run independent programs on each core and helloworld_par.c on which you can run a parallel sum of the elements of an array.



cycle_ip/z1_multicore_multicycle_2c_ip.runs/impl_1/design_1_2c_wrapper_utilization_placed.rpt

```
1. Slice Logic
--------------

+----------------------------+-------+-------+------------+-----------+-------+
|          Site Type         |  Used | Fixed | Prohibited | Available | Util% |
+----------------------------+-------+-------+------------+-----------+-------+
| Slice LUTs                 | 11962 |     0 |          0 |     53200 | 22.48 |
|   LUT as Logic             | 11543 |     0 |          0 |     53200 | 21.70 |
|   LUT as Memory            |   419 |     0 |          0 |     17400 |  2.41 |
|     LUT as Distributed RAM |     0 |     0 |            |           |       |
|     LUT as Shift Register  |   419 |     0 |            |           |       |
| Slice Registers            | 11454 |     0 |          0 |    106400 | 10.77 |
|   Register as Flip Flop    | 11454 |     0 |          0 |    106400 | 10.77 |
|   Register as Latch        |     0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |    20 |     0 |          0 |     26600 |  0.08 |
| F8 Muxes                   |     8 |     0 |          0 |     13300 |  0.06 |
+----------------------------+-------+-------+------------+-----------+-------+
```

Fig. 12.5 Vivado implementation report for a 2-core processor

12.5.1 Running Independent Programs

The code to drive the FPGA to run the eight RISC-V test programs is shown in Listing 12.21 (test_branch, test_jal_jalr, test_load_store, test_lui_auipc, test_mem, test_op, test_op_imm, and test_sum). You must update the paths of the included code files. Use the update_helloworld.sh shell script provided in the same folder.

Listing 12.21 The helloworld_seq.c file driving the multicore_multicycle_ip

```
#include <stdio.h>
#include "xmulticycle_pipeline_ip.h"
#include "xparameters.h"
#define LOG_NB_IP                1
#define NB_IP                   (1<<LOG_NB_IP)
#define LOG_IP_CODE_RAM_SIZE    (16-LOG_NB_IP)//in word
#define IP_CODE_RAM_SIZE        (1<<LOG_IP_CODE_RAM_SIZE)
#define LOG_IP_DATA_RAM_SIZE    (16-LOG_NB_IP)//in words
#define IP_DATA_RAM_SIZE        (1<<LOG_IP_DATA_RAM_SIZE)
#define DATA_RAM                 0x40000000
int *data_ram = (int*)DATA_RAM;
XMulticycle_pipeline_ip_Config *cfg_ptr[NB_IP];
XMulticycle_pipeline_ip ip[NB_IP];
word_type code_ram[IP_CODE_RAM_SIZE]={
#include "test_mem_text.hex"
};
int main(){
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  int          w;
  for (int i=0; i<NB_IP; i++){
    cfg_ptr[i] = XMulticycle_pipeline_ip_LookupConfig(i);
    XMulticycle_pipeline_ip_CfgInitialize(&ip[i], cfg_ptr[i]);
    XMulticycle_pipeline_ip_Set_ip_num    (&ip[i], i);
    XMulticycle_pipeline_ip_Set_start_pc  (&ip[i], 0);
    XMulticycle_pipeline_ip_Write_ip_code_ram_Words(&ip[i], 0,
        code_ram, IP_CODE_RAM_SIZE);
    XMulticycle_pipeline_ip_Set_data_ram  (&ip[i], DATA_RAM);
  }
  for (int i=0; i<NB_IP; i++)
    XMulticycle_pipeline_ip_Start(&ip[i]);
  for (int i=NB_IP-1; i>=0; i--)
    while (!XMulticycle_pipeline_ip_IsDone(&ip[i]));
  for (int i=0; i<NB_IP; i++){
    nbc[i] = (int)XMulticycle_pipeline_ip_Get_nb_cycle
                   (&ip[i]);
    nbi[i] = (int)XMulticycle_pipeline_ip_Get_nb_instruction
                   (&ip[i]);
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i],
      ((float)nbi[i])/nbc[i]);
    printf("data memory dump (non null words)\n");
    for (int j=0; j<IP_DATA_RAM_SIZE; j++){
      w = data_ram[i*IP_DATA_RAM_SIZE + j];
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
          (i*IP_DATA_RAM_SIZE + j)*4, w, (unsigned int)w);
    }
  }
  return 0;
}
```

For LOG_NB_IP = 1 (i.e. two cores), the run of the RISC-V code in the test_mem.h file should print on the putty window what is shown in Listing 12.22.

Listing 12.22 The helloworld_seq.c prints

```
core 0: 88 fetched and decoded instructions in 279 cycles (ipc =
    0.32)
data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
core 1: 88 fetched and decoded instructions in 279 cycles (ipc =
    0.32)
data memory dump (non null words)
m[20000] =                1 (       1)
m[20004] =                2 (       2)
m[20008] =                3 (       3)
m[2000c] =                4 (       4)
m[20010] =                5 (       5)
m[20014] =                6 (       6)
m[20018] =                7 (       7)
m[2001c] =                8 (       8)
m[20020] =                9 (       9)
m[20024] =               10 (       a)
m[2002c] =               55 (      37)
```

12.5.2 Running a Parallelized Program

The code to drive the FPGA to run the distributed sum is shown in Listing 12.23 (you must update the paths of the included code files; use the update_helloworld.sh shell script provided in the same folder).

Listing 12.23 The helloworld_par.c file driving the multicore_multicycle_ip

```
#include <stdio.h>
#include "xmulticycle_pipeline_ip.h"
#include "xparameters.h"
#define LOG_NB_IP              1
#define NB_IP                 (1<<LOG_NB_IP)
#define LOG_IP_CODE_RAM_SIZE  (16-LOG_NB_IP)//in word
#define IP_CODE_RAM_SIZE      (1<<LOG_IP_CODE_RAM_SIZE)
#define LOG_IP_DATA_RAM_SIZE  (16-LOG_NB_IP)//in words
#define IP_DATA_RAM_SIZE      (1<<LOG_IP_DATA_RAM_SIZE)
#define LOG_DATA_RAM_SIZE     16
#define DATA_RAM_SIZE         (1<<LOG_DATA_RAM_SIZE)
#define DATA_RAM               0x40000000
int *data_ram = (int*)DATA_RAM;
XMulticycle_pipeline_ip_Config *cfg_ptr[NB_IP];
XMulticycle_pipeline_ip ip[NB_IP];
word_type code_ram_0[IP_CODE_RAM_SIZE]={
#include "test_mem_par_ip0_text.hex"
};
word_type code_ram[IP_CODE_RAM_SIZE]={
#include "test_mem_par_otherip_text.hex"
};
int main(){
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  int          w;
  for (int i=0; i<NB_IP; i++){
    cfg_ptr[i] = XMulticycle_pipeline_ip_LookupConfig(i);
    XMulticycle_pipeline_ip_CfgInitialize(&ip[i], cfg_ptr[i]);
    XMulticycle_pipeline_ip_Set_ip_num   (&ip[i], i);
    XMulticycle_pipeline_ip_Set_start_pc (&ip[i], 0);
    XMulticycle_pipeline_ip_Set_data_ram (&ip[i], DATA_RAM);
  }
  for (int i=1; i<NB_IP; i++)
    XMulticycle_pipeline_ip_Write_ip_code_ram_Words(&ip[i], 0,
        code_ram, IP_CODE_RAM_SIZE);
  XMulticycle_pipeline_ip_Write_ip_code_ram_Words(&ip[0], 0,
      code_ram_0, IP_CODE_RAM_SIZE);
  for (int i=1; i<NB_IP; i++)
    XMulticycle_pipeline_ip_Start(&ip[i]);
  XMulticycle_pipeline_ip_Start(&ip[0]);
  for (int i=NB_IP-1; i>=0; i--)
    while (!XMulticycle_pipeline_ip_IsDone(&ip[i]));
  for (int i=0; i<NB_IP; i++){
    nbc[i] = (int)XMulticycle_pipeline_ip_Get_nb_cycle(&ip[i]);
    nbi[i] = (int)XMulticycle_pipeline_ip_Get_nb_instruction(&ip[i
        ]);
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i],
      ((float)nbi[i])/nbc[i]);
    printf("data memory dump (non null words)\n");
    for (int j=0; j<IP_DATA_RAM_SIZE; j++){
      w = data_ram[i*IP_DATA_RAM_SIZE + j];
      if (w != 0)
        printf("m[%5x] = %16d (%8x)\n",
          (i*IP_DATA_RAM_SIZE + j)*4, w, (unsigned int)w);
    }
  }
  return 0;
}
```

The run should print on the putty window (for a 2-core IP) what is shown in Listing 12.24.

Listing 12.24 The helloworld prints

```
core 0: 101 fetched and decoded instructions in 273 cycles (ipc =
    0.37)
data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   28] =               55 (      37)
m[   2c] =              210 (      d2)
core 1: 90 fetched and decoded instructions in 243 cycles (ipc =
    0.37)
data memory dump (non null words)
m[20000] =               11 (       b)
m[20004] =               12 (       c)
m[20008] =               13 (       d)
m[2000c] =               14 (       e)
m[20010] =               15 (       f)
m[20014] =               16 (      10)
m[20018] =               17 (      11)
m[2001c] =               18 (      12)
m[20020] =               19 (      13)
m[20024] =               20 (      14)
m[20028] =              155 (      9b)
```

12.6 Evaluating the Parallelism Efficiency of the Multicore IP

Instead of comparing the multicore design to the 4-stage pipeline baseline, I prefer to compare the different versions of the multicore (i.e. 2-core, 4-core, and 8-core).

To measure the efficiency of the parallelization, I compare the times to run a distributed matrix multiplication on an increasing number of cores.

The matrix multiplication is a computation example which tests the capability of the shared memory interconnection to route an intense traffic. Each core uses a large amount of external memory data (the more the cores, the higher the proportion of external data accesses).

The code of the matrix multiplication can be found in the mulmat.c file in the multicore_multicycle_ip folder. The code contains a definition of the LOG_NB_IP constant. It should be adapted to the LOG_NB_IP constant defined in the multicore_multicycle_ip.h file.

Table 12.1 Execution time of the parallelized matrix multiplication on the multicore_multicycle_ip processor and speedup from the sequential run

Number of cores	Cycles	nmi	CPI	Time (s)	Speedup
1	6,236,761	3,858,900	1.62	0.124735220	–
2	5,162,662	4,545,244	2.27	0.103253240	1.21
4	2,618,162	4,601,284	2.28	0.052363240	2.38
8	1,351,309	4,728,936	2.29	0.027026180	4.62

Then, the mulmat_text.hex file is to be built with the build_mulmat.sh script.

The run can be simulated in Vitis_HLS with the testbench_mulmat_par_multicore_multicycle_ip.cpp testbench.

The matrix multiplication can be run on the FPGA with the helloworld_mulmat.c driver.

Table 12.1 shows the execution times of the matrix multiplication on a multicore design with the number of cores ranging from two to eight. They are compared to the baseline time of a sequential version of the same program run on the single core multicycle pipeline design presented in Chap. 9 and the speedup is given (the speedup is the ratio of the sequential time to the parallel time).

All the executions multiply a 64 * 96 matrix X by a 96 * 48 matrix Y into a 64 * 48 matrix Z for a total of 13,824 integer values, i.e. 55,296 bytes (the size of the data memory is 256 KB; the remaining of the data memory is used to host the stacks).

The three matrices are interleaved in the shared memory (for example on four cores, the core 0 memory holds the first quarter of matrix X, followed by the first quarter of matrix Y, the first quarter of matrix Z, and ended by the core 0 stack; the core 1 memory holds the succession of the second quarters of the matrices and the core 1 stack; it is the same structuration for core 2 and core 3 memories).

The X and Y matrices are initialized with value 1 everywhere. Hence, matrix Z has value 96 everywhere.

As the speedup grows linearly with the number of cores, the experience shows that the AXI interconnection system is fast enough to sustain the succession of requests even with eight interconnected cores.

However, the speedup is far from optimal (it should be 8 for 8 cores instead of 4.62), which means that the runs are slowered by the external access latencies and by the low CPI performance of the multicycle pipeline.

The design in the next chapter tries to better fill the pipelines and hide the remote access latencies through the multithreading technique already employed in Chap. 10.

Reference

1. https://developer.arm.com/documentation/102202/0300/Transfer-behavior-and-transaction-ordering

A Multicore RISC-V Processor with Multihart Cores

13

Abstract

This chapter will make you build your second multicore RISC-V CPU. The processor is built from multiple IPs, each being a copy of the multihart_ip presented in Chap. 10. Each core runs multiple harts. Each core has its own code and data memories. The code memory is common to all the harts of the core. The data memory of the core is partitioned between the implemented harts. Hence, a *c* core with *h* hart processor has *h***c* data memory partitions embedded in *c* memory IPs. The data memory banks are interconnected with an AXI interconnect IP. Any hart has a private access to its data memory partition and any other partition of the same core, and a remote access to any partition of any other core. An example of a parallelized matrix multiplication is used to measure the speedup when moving the number of cores from one to four and the number of harts from one to eight with a maximum of 16 harts in the whole IPs for simulation and a maximum of eight implementable harts on the FPGA.

13.1 An Adaptation of the Multihart_ip to Multicore

All the source files related to the multicore_multihart_ip can be found in the multicore_multihart_ip folder.

Like for the multicore_multicycle_ip design presented in Chap. 12, the codes presented in this chapter describe the implementation of anyone of the multiple cores composing the processor, not all the processor. The concerned core is identified by the ip_num argument sent to the IP when it is started.

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_13

13.1.1 The Multicore Multihart IP Top Function Prototype and Local Declarations

The multihart_ip top function defining the CPUs is located in the multihart_ip.cpp file.

Its prototype (see Listing 13.1) is a mix of the multihart_ip one (refer back to Sect. 10.3.4) and the multicore_multicycle_ip one (refer back to Sect. 12.1.1). The ip_num argument is the IP number. The running_hart_set argument is the set of running harts in the core. The start_pc argument is the array used to set the starting pc of the running harts in the core code memory.

Listing 13.1 The multihart_ip top function for a multicore and multihart design

```
void multihart_ip(
  unsigned int  ip_num,
  unsigned int  running_hart_set,
  unsigned int  start_pc   [NB_HART],
  unsigned int  ip_code_ram[IP_CODE_RAM_SIZE],
  int           ip_data_ram     [NB_HART][HART_DATA_RAM_SIZE],
  int           data_ram [NB_IP][NB_HART][HART_DATA_RAM_SIZE],
  unsigned int *nb_instruction,
  unsigned int *nb_cycle){
#pragma HLS INTERFACE s_axilite port=ip_num
#pragma HLS INTERFACE s_axilite port=running_hart_set
#pragma HLS INTERFACE s_axilite port=start_pc
#pragma HLS INTERFACE s_axilite port=ip_code_ram
#pragma HLS INTERFACE bram      port=ip_data_ram storage_type=
    ram_1p
#pragma HLS INTERFACE m_axi     port=data_ram offset=slave
#pragma HLS INTERFACE s_axilite port=nb_instruction
#pragma HLS INTERFACE s_axilite port=nb_cycle
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INLINE recursive
  ...
```

The local declarations (see Listing 13.2) are vectorized like in the Chap. 10 multihart_ip top function.

Listing 13.2 The multihart_ip top function declarations

```
  ...
  int    reg_file       [NB_HART][NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=reg_file        dim=0 complete
  bit_t  is_reg_computed[NB_HART][NB_REGISTER];
#pragma HLS ARRAY_PARTITION variable=is_reg_computed dim=0 complete
  from_d_to_f_t f_from_d;
  from_e_to_f_t f_from_e;
  bit_t         f_state_is_full[NB_HART];
#pragma HLS ARRAY_PARTITION variable=f_state_is_full dim=1 complete
  f_state_t     f_state        [NB_HART];
#pragma HLS ARRAY_PARTITION variable=f_state         dim=1 complete
  from_f_to_d_t f_to_d;
  ...
```

The do ... while loop is shown in Listing 13.3.

The HLS DEPENDENCE pragma is used to provide information to the synthesizer. To avoid unnecessary serializations, it is possible through an HLS DEPENDENCE pragma to eliminate a parasitic dependence with the dependent=false option.

In Listing 13.3, the pragma orders the synthesizer to eliminate any RAW dependency on the data_ram variable between successive iterations of the loop (type=inter option).

In other words, the data_ram variable read in the mem_load function to perform a remote load is not serialized after the data_ram variable write in the mem_store function to perform a remote store (the RAW dependency between the two accesses to the data_ram variable in successive iterations of the do ... while loop is eliminated by the synthesizer). Hence, a back-to-back remote load starts while the just preceding remote store is still in progress.

A consequence of this choice is that a remote store (write to the data_ram variable) followed by a load at the same address (read from the data_ram variable) would not behave correctly (refer back to Listing 12.10).

However, the elimination of inter iteration RAW dependencies on the data_ram variable is necessary to keep the processor cycle set as two FPGA cycles.

Listing 13.3 The do ... while loop

```
label
  ...
  do {
#pragma HLS DEPENDENCE dependent=false direction=RAW type=inter
    variable=data_ram
#pragma HLS PIPELINE II=2
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("=============================================\n");
    printf("cycle %d\n", (unsigned int)nbc);
#endif
#endif
    new_cycle(f_to_d, d_to_f, d_to_i, i_to_e, e_to_f, e_to_m,
              m_to_w, &f_from_d, &f_from_e, &d_from_f,
             &i_from_d, &e_from_i, &m_from_e, &w_from_m);
    statistic_update(e_from_i, &nbi, &nbc);
    running_cond_update(has_exited, &is_running);
    fetch(f_from_d, f_from_e, d_state_is_full,
          ip_code_ram, f_state, &f_to_d, f_state_is_full);
    decode(d_from_f, i_state_is_full, d_state, &d_to_f,
          &d_to_i, d_state_is_full);
    issue(i_from_d, e_state_is_full, reg_file,
          is_reg_computed, i_state, &i_to_e, i_state_is_full,
         &is_lock, &i_hart, &i_destination);
    execute(
#ifndef __SYNTHESIS__
            ip_num,
#endif
            e_from_i, m_state_is_full,
#ifndef __SYNTHESIS__
            reg_file,
#endif
            e_state, &e_to_f, &e_to_m, e_state_is_full);
    mem_access(ip_num, m_from_e, w_state_is_full,
               ip_data_ram, data_ram, m_state, &m_to_w,
               m_state_is_full);
    write_back(
#ifndef __SYNTHESIS__
               ip_num,
#endif
               w_from_m, reg_file, w_state, w_state_is_full,
              &is_unlock, &w_hart, &w_destination,
               has_exited);
    lock_unlock_update(is_lock, i_hart, i_destination,
                       is_unlock, w_hart, w_destination,
                       is_reg_computed);
  } while (is_running);
  ...
```

The fetch (in the fetch.cpp file), decode (in the decode.cpp file), issue (in the issue.cpp file), execute (in the execute.cpp file), and write_back (in the wb.cpp file) functions have the same code as in the Chap. 10. The lock_unlock_update function (in the multihart_ip.cpp file) is unchanged too.

The init_file function in the multihart_ip.cpp file (see Listing 13.4) sets registers a0, a1, and sp (core identification number, hart identification number, and hart stack pointer respectively).

Listing 13.4 The init_file function

```
//a0/x10 is set with the IP number
//a1/x11 is set with the hart number
static void init_file(
  ip_num_t ip_num,
  int       reg_file        [][NB_REGISTER],
  bit_t     is_reg_computed[][NB_REGISTER]){
  hart_num_p1_t h1;
  hart_num_t    h;
  reg_num_p1_t  r1;
  reg_num_t     r;
  for (h1=0; h1<NB_HART; h1++){
#pragma HLS UNROLL
    h = h1;
    for (r1=0; r1<NB_REGISTER; r1++){
#pragma HLS UNROLL
      r = r1;
      is_reg_computed[h][r] = 0;
      if (r==10)
        reg_file     [h][r] = ip_num;
      else if (r==11)
        reg_file     [h][r] = h;
      else if (r==SP)
        reg_file     [h][r] = ((int)(ip_num+1))<<(
            LOG_IP_DATA_RAM_SIZE+2);
      else
        reg_file     [h][r] = 0;
    }
  }
}
```

13.1.2 The Data Memory Accesses

The mem_access function code implementing the memory access stage is presented in Listing 13.5.

A hart is selected in two steps. The select_hart function returns the highest priority ready hart number. In parallel, the input from the execute stage is saved in the m_state array (save_input_from_e function). The selection process then keeps the select_hart function selection, or if no ready hart was found, selects the just input hart.

The access is done in the stage_job function. The accessed_ip and the accessed_h hart numbers are choosen according to the is_local_ip bit computed in the save_input_from_e function, when an instruction is input from the execute stage. These three values are pre-computed at instruction input to shorten the critical path of the memory access.

The mem_access function in the mem_access.cpp file fills the output structure to the writeback stage (set_output_to_w function).

Listing 13.5 The mem_access function

```
void mem_access(
  ip_num_t       ip_num,
  from_e_to_m_t  m_from_e,
  bit_t         *w_state_is_full,
  int            ip_data_ram[][HART_DATA_RAM_SIZE],
  int            data_ram   [][NB_HART][HART_DATA_RAM_SIZE],
  m_state_t     *m_state,
  from_m_to_w_t *m_to_w,
  bit_t         *m_state_is_full){
  bit_t       is_selected;
  hart_num_t  selected_hart;
  bit_t       is_accessing;
  hart_num_t  accessing_hart;
  bit_t       input_is_selectable;
  input_is_selectable =
    m_from_e.is_valid && !w_state_is_full[m_from_e.hart];
  select_hart(m_state_is_full, w_state_is_full,
              &is_selected, &selected_hart);
  if (m_from_e.is_valid){
    m_state_is_full[m_from_e.hart] = 1;
    save_input_from_e(ip_num, m_from_e, m_state);
  }
  is_accessing   =
    is_selected || input_is_selectable;
  accessing_hart =
   (is_selected)?selected_hart:m_from_e.hart;
  if (is_accessing){
    m_state_is_full[accessing_hart] = 0;
    stage_job(m_state[accessing_hart].accessed_ip,
              m_state[accessing_hart].accessed_h,
              m_state[accessing_hart].is_local_ip,
              m_state[accessing_hart].is_load,
              m_state[accessing_hart].is_store,
              m_state[accessing_hart].address,
              m_state[accessing_hart].func3,
              ip_data_ram, data_ram,
             &m_state[accessing_hart].value);
#ifndef __SYNTHESIS__
#ifdef DEBUG_PIPELINE
    printf("hart %d: mem      ", (int)accessing_hart);
    printf("%04d\n",
          (int)(m_state[accessing_hart].fetch_pc<<2));
#endif
#endif
    set_output_to_w(accessing_hart, m_state, m_to_w);
  }
  m_to_w->is_valid = is_accessing;
}
```

Listing 13.6 shows the code of the save_input_from_e function, located in the mem_access.cpp file.

The accessed absolute_hart number is computed from the address (which gives the accessed hart number relative to the accessing IP), from the ip_num accessing IP and from the accessing hart.

The accessed IP (m_state[hart].accessed_ip) is the IP number of the memory accessed IP partition. It is the upper part of the absolute_hart number. The accessed hart (m_state[hart].accessed_h) is the hart number, within the accessed IP, of the memory accessed hart partition. It is the lower part of the absolute_hart number.

The m_state[hart].is_local_ip bit is set if the accessed IP is the accessing IP.

Listing 13.6 The save_input_from_e function

```
static void save_input_from_e(
  ip_num_t      ip_num,
  from_e_to_m_t m_from_e,
  m_state_t    *m_state){
  hart_num_t                     hart;
  ap_uint<LOG_NB_IP+LOG_NB_HART> absolute_hart;
  hart                      = m_from_e.hart;
  m_state[hart].rd          = m_from_e.rd;
  m_state[hart].has_no_dest = m_from_e.has_no_dest;
  m_state[hart].is_load     = m_from_e.is_load;
  m_state[hart].is_store    = m_from_e.is_store;
  m_state[hart].func3       = m_from_e.func3;
  m_state[hart].is_ret      = m_from_e.is_ret;
  m_state[hart].address     = m_from_e.address;
  m_state[hart].value       = m_from_e.value;
  m_state[hart].result      = m_from_e.value;
  absolute_hart =
   (m_from_e.address>>
   (LOG_HART_DATA_RAM_SIZE+2)) + (((ap_uint<LOG_NB_IP+LOG_NB_HART>)
       ip_num)<<LOG_NB_HART) + hart;
  m_state[hart].accessed_ip = absolute_hart>>LOG_NB_HART;
  m_state[hart].accessed_h  = absolute_hart;
  m_state[hart].is_local_ip  =(m_state[hart].accessed_ip == ip_num
     );
#ifndef __SYNTHESIS__
  m_state[hart].fetch_pc    = m_from_e.fetch_pc;
  m_state[hart].instruction = m_from_e.instruction;
  m_state[hart].d_i         = m_from_e.d_i;
  m_state[hart].target_pc   = m_from_e.target_pc;
#endif
}
```

Listing 13.7 shows the code of the stage_job function, located in the mem_access.cpp file.

The ip_num argument refers to the accessed IP, not to the accessing one. The hart argument refers to the accessed hart partition.

If the instruction is neither a load nor a store, the stage_job function does nothing (the instruction just transits through the memory access stage).

A load accesses the accessed IP and accessed hart memory partition in the mem_load function. A store accesses the memory in the mem_store function.

Listing 13.7 The stage_job function

```
static void stage_job(
  ip_num_t         ip_num,
  hart_num_t       hart,
  bit_t            is_local_ip,
  bit_t            is_load,
  bit_t            is_store,
  b_data_address_t address,
  func3_t          func3,
  int              ip_data_ram[][HART_DATA_RAM_SIZE],
  int              data_ram   [][NB_HART][HART_DATA_RAM_SIZE],
  int             *value){
  if (is_load)
    *value =
      mem_load (ip_num, is_local_ip, hart, ip_data_ram, data_ram,
                address, func3);
  else if (is_store)
      mem_store(ip_num, is_local_ip, hart, ip_data_ram, data_ram,
                address, *value, (ap_uint<2>)func3);
}
```

Listing 13.8 shows the start of the code of the mem_load function, located in the mem.cpp file.

A local load reads from the local IP data ram (ip_data_ram). A remote load reads from the hart in the ip within the data_ram. After a full word has been loaded, the addressed bytes are selected and returned as the result of the load (this part of the mem_load is not shown as it is unchanged from preceding designs; refer back to Listing 6.11).

Listing 13.8 The mem_load function

```
int mem_load(
  ip_num_t         ip,
  bit_t            is_local,
  hart_num_t       hart,
  int              ip_data_ram[][HART_DATA_RAM_SIZE],
  int              data_ram   [][NB_HART][HART_DATA_RAM_SIZE],
  b_data_address_t address,
  func3_t          msize){
  ap_uint<2>            a01 =  address;
  bit_t                 a1  = (address >> 1);
  w_hart_data_address_t a2  = (address >> 2);
  int                   result;
  char                  b, b0, b1, b2, b3;
  unsigned char         ub, ub0, ub1, ub2, ub3;
  short                 h, h0, h1;
  unsigned short        uh, uh0, uh1;
  int                   w, ib, ih;
  unsigned int          iub, iuh;
  if (is_local)
    w =  ip_data_ram[hart][a2];
  else
    w = data_ram[ip][hart][a2];
...
```

Listing 13.9 shows the code of the mem_store function, located in the mem.cpp file.

A local store writes to the local IP ram bank (i.e. in the hart partition of the ip_data_ram array).

A remote store writes to the accessed IP memory bank (i.e. writes in the hart partition of the ip core in the data_ram array) through the AXI interconnect.

Listing 13.9 The mem_store function

```
void mem_store(
  ip_num_t         ip,
  bit_t            is_local,
  hart_num_t       hart,
  int              ip_data_ram[][HART_DATA_RAM_SIZE],
  int              data_ram   [][NB_HART][HART_DATA_RAM_SIZE],
  b_data_address_t address,
  int              rv2,
  ap_uint<2>       msize){
  b_hart_data_address_t a      = address;
  h_hart_data_address_t a1     = address>>1;
  w_hart_data_address_t a2     = address>>2;
  char                  rv2_0  = rv2;
  short                 rv2_01 = rv2;
  switch(msize){
  case SB:
    if (is_local)
     *((char*) (ip_data_ram) +
               ((((b_ip_data_address_t)hart)<<
               (LOG_HART_DATA_RAM_SIZE+2)) | a))
      = rv2_0;
    else
     *((char*) (data_ram) +
             ((((b_data_address_t)ip)<<
              (LOG_IP_DATA_RAM_SIZE+2))   |
              (((b_ip_data_address_t)hart)<<
              (LOG_HART_DATA_RAM_SIZE+2)) | a))   = rv2_0;
    break;
  case SH:
    if (is_local)
     *((short*)(ip_data_ram) +
           ((((h_ip_data_address_t)hart)<<
           (LOG_HART_DATA_RAM_SIZE+1)) | a1))
      = rv2_01;
    else
     *((short*)(data_ram) +
             ((((h_data_address_t)ip)<<
              (LOG_IP_DATA_RAM_SIZE+1))   |
              (((h_ip_data_address_t)hart)<<
              (LOG_HART_DATA_RAM_SIZE+1)) | a1)) = rv2_01;
    break;
  case SW:
    if (is_local)
      ip_data_ram [hart][a2] = rv2;
    else
      data_ram[ip][hart][a2] = rv2;
    break;
  case 3:
    break;
  }
}
```

13.2 Simulating the IP

Experimentation

To simulate the multicore_multihart_ip, operate as explained in Sect. 5.3.6, replacing fetching_ip with multicore_multihart_ip. There are two testbench programs: testbench_seq_multihart_ip.cpp to run independent codes (one per hart in each core) and testbench_par_multihart_ip.cpp to run a parallel sum of the elements of an array.

With testbench_seq_multihart_ip.cpp you can play with the simulator, replacing the included test_mem_0_text.hex file with any other .hex file you find in the same folder. You can also vary the number of cores and the number of harts (two cores and two, four, or eight harts per core; four cores and two or four harts per core; eight cores and two harts per core; in any case, no more than 16 harts per processor).

13.2.1 Simulating Independent Programs

Like in the multihart_ip and multicore_multicycle_ip projects, two testbench files are provided, one to run fully independent codes (testbench_seq_multihart_ip.cpp) and the other to run independent codes sharing a distributed array (testbench_par_multihart_ip.cpp).

The first testbench (see Listing 13.10) runs NB_IP calls to the multihart_ip function. The NB_IP cores are run sequentially (on the FPGA, they are run in parallel; however, as the cores do not interact, the simulation behaviour is identical to the real FPGA one).

To build the hex files, you can use the build_seq.sh script file, identical to the one presented in Sect. 12.2.1 (e.g. "./build_seq.sh test_mem" to build test_mem_text.hex from test_mem.s).

Listing 13.10 The testbench_seq_multihart_ip.cpp file

```
#include <stdio.h>
#include "multihart_ip.h"
unsigned int code_ram[IP_CODE_RAM_SIZE]={
#include "test_mem_text.hex"
};
int          data_ram[NB_IP][NB_HART][HART_DATA_RAM_SIZE];
unsigned int start_pc[NB_HART];
int main() {
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  int          w;
  for (int h=0; h<NB_HART; h++) start_pc[h] = 0;
  for (int i=0; i<NB_IP; i++){
    multihart_ip(i, (1<<NB_HART)-1, start_pc, code_ram,
     &data_ram[i][0], &data_ram[0], &nbi[i], &nbc[i]);
  }
  for (int i=0; i<NB_IP; i++){
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i],
    ((float)nbi[i])/nbc[i]);
    for (int h=0; h<NB_HART; h++){
      printf("hart: %d data memory dump (non null words)\n", h);
      for (int j=0; j<HART_DATA_RAM_SIZE; j++){
        w = data_ram[i][h][j];
        if (w != 0)
          printf("m[%5x] = %16d (%8x)\n",
            4*((i<<LOG_IP_DATA_RAM_SIZE)   +
               (h<<LOG_HART_DATA_RAM_SIZE) + j),
            w, (unsigned int)w);
      }
    }
  }
  return 0;
}
```

For the run of four copies (two cores of two harts) of test_mem.h, the output is shown in Listings 13.11 to 13.13.

Listing 13.11 The output of the main function of the testbench_seq_multicore_multihart_ip.cpp file: core 0

```
hart 0: 0000: 00000513      li a0, 0
hart 0:       a0  =                0 (       0)
hart 1: 0000: 00000513      li a0, 0
hart 1:       a0  =                0 (       0)
...
hart 0: 0056: 00a62223      sw a0, 4(a2)
hart 0:       m[   2c] =                55 (      37)
hart 1: 0056: 00a62223      sw a0, 4(a2)
hart 1:       m[1002c] =                55 (      37)
hart 0: 0060: 00008067      ret
hart 0:       pc  =                0 (       0)
hart 1: 0060: 00008067      ret
hart 1:       pc  =                0 (       0)
register file for hart 0
...
sp  =           131072 (   20000)
...
a0  =               55 (      37)
...
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
register file for hart 1
...
sp  =           131072 (   20000)
...
a0  =               55 (      37)
...
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
```

Listing 13.12 The output of the main function of the testbench_seq_multicore_multihart_ip.cpp file: core 1

```
hart 0: 0000: 00000513      li a0, 0
hart 0:       a0  =                0 (       0)
hart 1: 0000: 00000513      li a0, 0
hart 1:       a0  =                0 (       0)
...
hart 0: 0056: 00a62223      sw a0, 4(a2)
hart 0:       m[2002c] =               55 (      37)
hart 1: 0056: 00a62223      sw a0, 4(a2)
hart 1:       m[3002c] =               55 (      37)
hart 0: 0060: 00008067      ret
hart 0:       pc  =                0 (       0)
hart 1: 0060: 00008067      ret
hart 1:       pc  =                0 (       0)
register file for hart 0
...
sp  =           262144 (   40000)
...
a0  =               55 (      37)
...
sp  =           262144 (   40000)
...
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
register file for hart 1
...
a0  =               55 (      37)
a1  =                0 (       0)
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               10 (       a)
...
```

Listing 13.13 The output of the main function of the testbench_seq_multicore_multihart_ip.cpp file: memory dump

```
core 0: 176 fetched and decoded instructions in 306 cycles (ipc =
    0.58)
hart: 0 data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
hart: 1 data memory dump (non null words)
m[10000] =                1 (       1)
m[10004] =                2 (       2)
m[10008] =                3 (       3)
m[1000c] =                4 (       4)
m[10010] =                5 (       5)
m[10014] =                6 (       6)
m[10018] =                7 (       7)
m[1001c] =                8 (       8)
```

```
m[10020] =                9 (       9)
m[10024] =               10 (       a)
m[1002c] =               55 (      37)
core 1: 176 fetched and decoded instructions in 306 cycles (ipc = 0.58)
hart: 0 data memory dump (non null words)
m[20000] =                1 (       1)
m[20004] =                2 (       2)
m[20008] =                3 (       3)
m[2000c] =                4 (       4)
m[20010] =                5 (       5)
m[20014] =                6 (       6)
m[20018] =                7 (       7)
m[2001c] =                8 (       8)
m[20020] =                9 (       9)
m[20024] =               10 (       a)
m[2002c] =               55 (      37)
hart: 1 data memory dump (non null words)
m[30000] =                1 (       1)
m[30004] =                2 (       2)
m[30008] =                3 (       3)
m[3000c] =                4 (       4)
m[30010] =                5 (       5)
m[30014] =                6 (       6)
m[30018] =                7 (       7)
m[3001c] =                8 (       8)
m[30020] =                9 (       9)
m[30024] =               10 (       a)
m[3002c] =               55 (      37)
```

13.2.2 Simulating a Parallelized Program

The second testbench file is testbench_par_multihart_ip.cpp, shown in Listing 13.14. It runs a distributed and parallel version of the test_mem.s program. A set of NB_HART sub-arrays per core are filled in parallel by the running harts. They are summed in parallel. The first hart of the first core reads the partial sums (remote memory accesses) and computes their sum.

The hex files can be built with the build_par.sh script, identical to the one presented in Sect. 12.2.2. It is necessary to set the LOG_NB_IP and LOG_NB_HART constants in the test_mem_par_ip0.s and test_mem_par_otherip.s files according to their values in multicore_multihart_ip.h.

The IPs in the processor are run sequentially. To obtain an identical behaviour to the parallel run on the FPGA, the first core is the last one to be simulated, to be able to read the partial sums computed by the simulation of the other cores.

Listing 13.14 The testbench_par_multihart_ip.cpp file

```
#include <stdio.h>
#include "multihart_ip.h"
#define OTHER_HART_START 0x78/4
unsigned int code_ram_0[IP_CODE_RAM_SIZE]={
#include "test_mem_par_ip0_text.hex"
};
unsigned int code_ram  [IP_CODE_RAM_SIZE]={
#include "test_mem_par_otherip_text.hex"
};
int data_ram[NB_IP][NB_HART][HART_DATA_RAM_SIZE];
unsigned int start_pc  [NB_HART]={0};
unsigned int start_pc_0[NB_HART];
int main(){
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  int          w;
  start_pc_0[0] = 0;
  for (int i=1; i<NB_HART; i++)
    start_pc_0[i] = OTHER_HART_START;
  for (int i=1; i<NB_IP; i++)
    multihart_ip(i, (1<<NB_HART)-1, start_pc, code_ram,
     &data_ram[i][0], data_ram, &nbi[i], &nbc[i]);
  multihart_ip(0, (1<<NB_HART)-1, start_pc_0, code_ram_0,
   &data_ram[0][0], data_ram, &nbi[0], &nbc[0]);
  for (int i=0; i<NB_IP; i++){
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i],
    ((float)nbi[i])/nbc[i]);
    for (int h=0; h<NB_HART; h++){
      printf("hart %d: data memory dump (non null words)\n", h);
      for (int j=0; j<HART_DATA_RAM_SIZE; j++){
        w = data_ram[i][h][j];
        if (w != 0)
          printf("m[%5x] = %16d (%8x)\n",
            4*((i<<LOG_IP_DATA_RAM_SIZE)   +
               (h<<LOG_HART_DATA_RAM_SIZE) + j),
            w, (unsigned int)w);
      }
    }
  }
  return 0;
}
```

The code run is the one already presented in Sect. 12.2.2. All the cores except the first one run the test_mem_par_otherip.s code. The first core runs the test_mem_par_ip0.s code.

In the testbench code, the call related to the first core is placed in the last position to ensure correct simulation.

For the run on two cores of two harts, the output is shown in Listings 13.15 to 13.17 (the final sum of the 40 first integers is 820).

Listing 13.15 The output of the main function of the testbench_par_multicore_multihart_ip.cpp file: core 1

```
hart 0: 0000: 00359293      slli t0, a1, 3
hart 0:       t0  =                0 (       0)
hart 1: 0000: 00359293      slli t0, a1, 3
hart 1:       t0  =                8 (       8)
...
hart 0: 0088: 00a62023      sw a0, 0(a2)
hart 0:       m[20028] =              255 (      ff)
hart 1: 0088: 00a62023      sw a0, 0(a2)
hart 1:       m[30028] =              355 (     163)
hart 0: 0092: 00008067      ret
hart 0:       pc  =                0 (       0)
hart 1: 0092: 00008067      ret
hart 1:       pc  =                0 (       0)
register file for hart 0
...
```

```
sp  =           262144 (   40000)
...
t0  =                8 (       8)
t1  =                2 (       2)
...
a0  =              255 (      ff)
...
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               30 (      1e)
...
register file for hart 1
...
sp  =           262144 (   40000)
...
t0  =                8 (       8)
t1  =                2 (       2)
...
a0  =              355 (     163)
...
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               40 (      28)
...
```

Listing 13.16 The output of the main function of the testbench_par_multicore_multihart_ip.cpp file: core 0

```
hart 0: 0000: 00000513      li a0, 0
hart 0:       a0  =                0 (       0)
hart 1: 0120: 00359293      slli t0, a1, 3
hart 1:       t0  =                8 (       8)
...
hart 0: 0112: 00a62023      sw a0, 0(a2)
hart 0:       m[   2c] =              820 (     334)
hart 0: 0116: 00008067      ret
hart 0:       pc  =                0 (       0)
register file for hart 0
...
sp  =           131072 (   20000)
...
a0  =              820 (     334)
a1  =                4 (       4)
a2  =               44 (      2c)
a3  =              355 (     163)
a4  =           262144 (   40000)
a5  =            65536 (   10000)
...
register file for hart 1
...
sp  =           131072 (   20000)
...
a0  =              155 (      9b)
a1  =                0 (       0)
a2  =               40 (      28)
a3  =               40 (      28)
a4  =               20 (      14)
...
```

Listing 13.17 The output of the main function of the testbench_par_multicore_multihart_ip.cpp file: memory dump

```
core 0: 212 fetched and decoded instructions in 357 cycles (ipc =
     0.59)
hart 0: data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   28] =               55 (      37)
m[   2c] =              820 (     334)
hart 1: data memory dump (non null words)
m[10000] =               11 (       b)
m[10004] =               12 (       c)
m[10008] =               13 (       d)
m[1000c] =               14 (       e)
m[10010] =               15 (       f)
m[10014] =               16 (      10)
m[10018] =               17 (      11)
m[1001c] =               18 (      12)
m[10020] =               19 (      13)
m[10024] =               20 (      14)
m[10028] =              155 (      9b)
core 1: 192 fetched and decoded instructions in 290 cycles (ipc =
     0.66)
hart 0: data memory dump (non null words)
m[20000] =               21 (      15)
m[20004] =               22 (      16)
m[20008] =               23 (      17)
m[2000c] =               24 (      18)
m[20010] =               25 (      19)
m[20014] =               26 (      1a)
m[20018] =               27 (      1b)
m[2001c] =               28 (      1c)
m[20020] =               29 (      1d)
m[20024] =               30 (      1e)
m[20028] =              255 (      ff)
hart 1: data memory dump (non null words)
m[30000] =               31 (      1f)
m[30004] =               32 (      20)
m[30008] =               33 (      21)
m[3000c] =               34 (      22)
m[30010] =               35 (      23)
m[30014] =               36 (      24)
m[30018] =               37 (      25)
m[3001c] =               38 (      26)
m[30020] =               39 (      27)
m[30024] =               40 (      28)
m[30028] =              355 (     163)
```

13.2.3 Synthesizing the IP

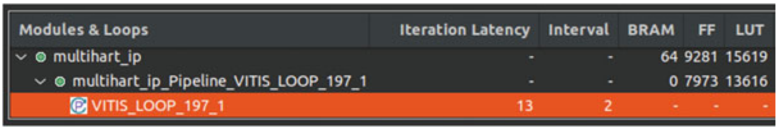
Figure 13.1 shows that the II = 2 constraint is satisfied for a 2-core and two harts per core processor (the II = 2 constraint is also satisfied for a 2-core and four harts per core design and for a 4-core and two harts per core design). The iteration latency, imposed by the global memory access one, is 13 FPGA cycles.

13.2.4 The Vivado Project

The Vivado Design is shown in Fig. 13.2.

Before you build the wrapper, you must set the address map with the Address Editor.

For a 2-core design, the axi_bram_ctrl range is 128 KB and the addresses are 0x4000_0000 and 0x4002_0000. For a 4-core design the range is 64 KB and the addresses are 0x4000_0000, 0x4001_0000, 0x4002_0000, and 0x4003_0000.



Modules & Loops	Iteration Latency	Interval	BRAM	FF	LUT
⌄ ● multihart_ip	-	-	64	9281	15619
⌄ ● multihart_ip_Pipeline_VITIS_LOOP_197_1	-	-	0	7973	13616
VITIS_LOOP_197_1	13	2	-	-	-

Fig. 13.1 Synthesis report for a 2-core of two harts processor

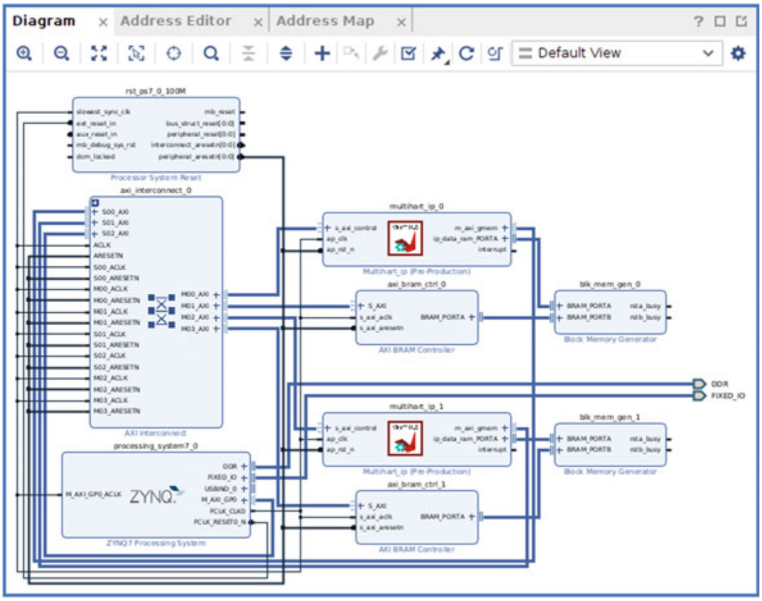
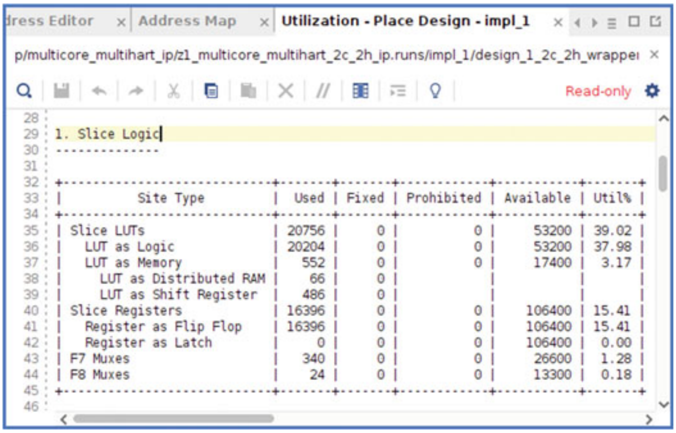


Fig. 13.2 The 2-core 2-hart design


dress Editor
Address Map
Utilization - Place Design - impl_1
p/multicore_multihart_ip/z1_multicore_multihart_2c_2h_ip.runs/impl_1/design_1_2c_2h_wrappe
Read-only

```
1. Slice Logic
--------------


+----------------------------+-------+-------+------------+-----------+-------+
|          Site Type         |  Used | Fixed | Prohibited | Available | Util% |
+----------------------------+-------+-------+------------+-----------+-------+
| Slice LUTs                 | 20756 |     0 |          0 |     53200 | 39.02 |
|   LUT as Logic             | 20204 |     0 |          0 |     53200 | 37.98 |
|   LUT as Memory            |   552 |     0 |          0 |     17400 |  3.17 |
|     LUT as Distributed RAM |    66 |     0 |            |           |       |
|     LUT as Shift Register  |   486 |     0 |            |           |       |
| Slice Registers            | 16396 |     0 |          0 |    106400 | 15.41 |
|   Register as Flip Flop    | 16396 |     0 |          0 |    106400 | 15.41 |
|   Register as Latch        |     0 |     0 |          0 |    106400 |  0.00 |
| F7 Muxes                   |   340 |     0 |          0 |     26600 |  1.28 |
| F8 Muxes                   |    24 |     0 |          0 |     13300 |  0.18 |
+----------------------------+-------+-------+------------+-----------+-------+
```

Fig. 13.3 Vivado implementation report for a 2-core of two harts processor

For two cores, the s_axi_control range is 256 KB and the addresses are 0x4004_0000 and 0x4008_0000. For four cores, the range is 128KB and the addresses are 0x4004_0000, 0x4006_0000, 0x4008_0000, and 0x400A_0000.

The Vivado bitstream generation produces the implementation report in Fig. 13.3, showing that the 2-core IP with two harts uses 20,756 LUTs (39.02%). The 2-core 4-hart version uses 36,204 LUTs (68.05%). The 4-core 2-hart IP uses 37,520 LUTs (70.53%).

13.3 Running the IP on the Development Board

Experimentation

To run the multicore_multihart_ip on the development board, proceed as explained in Sect. 5.3.10, replacing fetching_ip with multicore_multihart_ip. There are two drivers: helloworld_seq.c on which you can run independent programs on each hart of each core and helloworld_par.c on which you can run a parallel sum of the elements of an array.

13.3.1 Running Independent Programs

The code to drive the FPGA to run the eight copies of the test_mem.h program is shown in Listing 13.8 (the same code can be used to run the other test programs:

test_branch, test_jal_jalr, test_load_store, test_lui_auipc, test_op, test_op_imm, and test_sum). You must adapt the path to the test_mem_text.hex file to your own installation with the update_helloworld.sh script.

Listing 13.18 The helloworld_seq.c file driving the multicore_multicycle_ip

```
#include <stdio.h>
#include "xmultihart_ip.h"
#include "xparameters.h"
#define LOG_NB_IP                       1
#define NB_IP                          (1<<LOG_NB_IP)
#define LOG_NB_HART                     1
#define NB_HART                        (1<<LOG_NB_HART)
#define LOG_CODE_RAM_SIZE              16
#define LOG_DATA_RAM_SIZE              16
#define LOG_IP_CODE_RAM_SIZE           (LOG_CODE_RAM_SIZE-LOG_NB_IP)//in
    word
#define IP_CODE_RAM_SIZE               (1<<LOG_IP_CODE_RAM_SIZE)
#define LOG_IP_DATA_RAM_SIZE           (LOG_DATA_RAM_SIZE-LOG_NB_IP)//in
    words
#define LOG_HART_DATA_RAM_SIZE         (LOG_IP_DATA_RAM_SIZE-LOG_NB_HART)
#define HART_DATA_RAM_SIZE             (1<<LOG_HART_DATA_RAM_SIZE)
#define DATA_RAM                       0x40000000
int *data_ram = (int*)DATA_RAM;
XMultihart_ip_Config *cfg_ptr[NB_IP];
XMultihart_ip ip[NB_IP];
word_type code_ram[IP_CODE_RAM_SIZE]={
#include "test_mem_text.hex"
};
word_type start_pc[NB_HART];
int main(){
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  word_type    w;
  for (int h=0; h<NB_HART; h++)
    start_pc[h] = 0;
  for (int i=0; i<NB_IP; i++){
    cfg_ptr[i] = XMultihart_ip_LookupConfig(i);
    XMultihart_ip_CfgInitialize(&ip[i], cfg_ptr[i]);
    XMultihart_ip_Set_ip_num(&ip[i], i);
    XMultihart_ip_Set_running_hart_set(&ip[i], (1<<NB_HART)-1);
    XMultihart_ip_Write_start_pc_Words(&ip[i], 0, start_pc, NB_HART
        );
    XMultihart_ip_Write_ip_code_ram_Words(&ip[i], 0, code_ram,
        IP_CODE_RAM_SIZE);
    XMultihart_ip_Set_data_ram(&(ip[i]), DATA_RAM);
  }
  for (int i=0; i<NB_IP; i++) XMultihart_ip_Start(&ip[i]);
  for (int i=NB_IP-1; i>=0; i--)
    while (!XMultihart_ip_IsDone(&ip[i]));
  for (int i=0; i<NB_IP; i++){
    nbc[i] = (int)XMultihart_ip_Get_nb_cycle(&ip[i]);
    nbi[i] = (int)XMultihart_ip_Get_nb_instruction(&ip[i]);
  }
  for (int i=0; i<NB_IP; i++){
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i], ((float)nbi[i])/
     nbc[i]);
    for (int h=0; h<NB_HART; h++){
      printf("hart %d data memory dump (non null words)\n", h);
      for (int j=0; j<HART_DATA_RAM_SIZE; j++){
        w = data_ram[(i<<LOG_IP_DATA_RAM_SIZE)    +
                     (h<<LOG_HART_DATA_RAM_SIZE) + j];
        if (w != 0)
          printf("m[%5x] = %16d (%8x)\n",
            4*((i<<LOG_IP_DATA_RAM_SIZE)  +
               (h<<LOG_HART_DATA_RAM_SIZE) + j), (int)w, (unsigned
                   int)w);
      }
    }
  }
}
```

The run should print on the putty window what is shown in Listing 13.9.

Listing 13.19 The helloworld_seq.c prints

```
core 0: 176 fetched and decoded instructions in 306 cycles (ipc =
    0.58)
hart 0 data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   2c] =               55 (      37)
hart 1 data memory dump (non null words)
m[10000] =                1 (       1)
m[10004] =                2 (       2)
m[10008] =                3 (       3)
m[1000c] =                4 (       4)
m[10010] =                5 (       5)
m[10014] =                6 (       6)
m[10018] =                7 (       7)
m[1001c] =                8 (       8)
m[10020] =                9 (       9)
m[10024] =               10 (       a)
m[1002c] =               55 (      37)
core 1: 176 fetched and decoded instructions in 306 cycles (ipc =
    0.58)
hart 0 data memory dump (non null words)
m[20000] =                1 (       1)
m[20004] =                2 (       2)
m[20008] =                3 (       3)
m[2000c] =                4 (       4)
m[20010] =                5 (       5)
m[20014] =                6 (       6)
m[20018] =                7 (       7)
m[2001c] =                8 (       8)
m[20020] =                9 (       9)
m[20024] =               10 (       a)
m[2002c] =               55 (      37)
hart 1 data memory dump (non null words)
m[30000] =                1 (       1)
m[30004] =                2 (       2)
m[30008] =                3 (       3)
m[3000c] =                4 (       4)
m[30010] =                5 (       5)
m[30014] =                6 (       6)
m[30018] =                7 (       7)
```

```
m[3001c] =                8 (        8)
m[30020] =                9 (        9)
m[30024] =               10 (        a)
m[3002c] =               55 (       37)
```

13.3.2 Running a Parallelized Program

The parallelized code is distributed in the core code memories.

The code to drive the FPGA to run the distributed sum is shown in Listing 13.20. The paths to test_mem_par_ip0_text.hex and test_mem_par_otherip_text.hex should be adapted to your environment with the update_helloworld.sh script.

Listing 13.20 The helloworld_par.c file driving the multicore_multihart_ip

```
#include <stdio.h>
#include "xmultihart_ip.h"
#include "xparameters.h"
#define LOG_NB_IP                  1
#define NB_IP                     (1<<LOG_NB_IP)
#define LOG_NB_HART                1
#define NB_HART                   (1<<LOG_NB_HART)
#define LOG_CODE_RAM_SIZE         16
#define LOG_DATA_RAM_SIZE         16
#define LOG_IP_CODE_RAM_SIZE      (LOG_CODE_RAM_SIZE-LOG_NB_IP)//in
    word
#define IP_CODE_RAM_SIZE          (1<<LOG_IP_CODE_RAM_SIZE)
#define LOG_IP_DATA_RAM_SIZE      (LOG_DATA_RAM_SIZE-LOG_NB_IP)//in
    words
#define LOG_HART_DATA_RAM_SIZE    (LOG_IP_DATA_RAM_SIZE-LOG_NB_HART)
#define HART_DATA_RAM_SIZE        (1<<LOG_HART_DATA_RAM_SIZE)
#define DATA_RAM                  0x40000000
#define OTHER_HART_START          0x78/4
int *data_ram = (int*)DATA_RAM;
XMultihart_ip_Config *cfg_ptr[NB_IP];
XMultihart_ip ip[NB_IP];
word_type code_ram_0[IP_CODE_RAM_SIZE]={
#include "test_mem_par_ip0_text.hex"
};
word_type code_ram[IP_CODE_RAM_SIZE]={
#include "test_mem_par_otherip_text.hex"
};
word_type start_pc[NB_HART];
int main(){
  unsigned int nbi[NB_IP];
  unsigned int nbc[NB_IP];
  word_type    w;
  for (int h=0; h<NB_HART; h++)
    start_pc[h] = 0;
  for (int i=0; i<NB_IP; i++){
    cfg_ptr[i] = XMultihart_ip_LookupConfig(i);
    XMultihart_ip_CfgInitialize(&ip[i], cfg_ptr[i]);
    XMultihart_ip_Set_ip_num(&ip[i], i);
    XMultihart_ip_Set_running_hart_set(&ip[i], (1<<NB_HART)-1);
    XMultihart_ip_Set_data_ram(&(ip[i]), DATA_RAM);
  }
  for (int i=1; i<NB_IP; i++){
    XMultihart_ip_Write_start_pc_Words(&ip[i], 0, start_pc, NB_HART
        );
    XMultihart_ip_Write_ip_code_ram_Words(&ip[i], 0, code_ram,
        IP_CODE_RAM_SIZE);
  }
  for (int h=1; h<NB_HART; h++)
    start_pc[h]=OTHER_HART_START;
  XMultihart_ip_Write_start_pc_Words(&ip[0], 0, start_pc, NB_HART);
  XMultihart_ip_Write_ip_code_ram_Words(&ip[0], 0, code_ram_0,
      IP_CODE_RAM_SIZE);
  for (int i=0; i<NB_IP; i++) XMultihart_ip_Start(&ip[i]);
  for (int i=NB_IP-1; i>=0; i--)
    while (!XMultihart_ip_IsDone(&ip[i]));
  for (int i=0; i<NB_IP; i++){
    nbc[i] = (int)XMultihart_ip_Get_nb_cycle(&ip[i]);
    nbi[i] = (int)XMultihart_ip_Get_nb_instruction(&ip[i]);
  }
  for (int i=0; i<NB_IP; i++){
    printf("core %d: %d fetched and decoded instructions\
 in %d cycles (ipc = %2.2f)\n", i, nbi[i], nbc[i], ((float)nbi[i])/
     nbc[i]);
    for (int h=0; h<NB_HART; h++){
      printf("hart %d data memory dump (non null words)\n", h);
      for (int j=0; j<HART_DATA_RAM_SIZE; j++){
        w = data_ram[(i<<LOG_IP_DATA_RAM_SIZE)   +
                     (h<<LOG_HART_DATA_RAM_SIZE) + j];
        if (w != 0)
          printf("m[%5x] = %16d (%8x)\n",
            4*((i<<LOG_IP_DATA_RAM_SIZE)   +
               (h<<LOG_HART_DATA_RAM_SIZE) + j), (int)w, (unsigned
                   int)w);
      }
    }
  }
}
```

The run should print on the putty window what is shown in Listing 13.21.

Listing 13.21 The helloworld_par.c prints

```
core 0: 212 fetched and decoded instructions in 357 cycles (ipc =
    0.59)
hart 0 data memory dump (non null words)
m[    0] =                1 (       1)
m[    4] =                2 (       2)
m[    8] =                3 (       3)
m[    c] =                4 (       4)
m[   10] =                5 (       5)
m[   14] =                6 (       6)
m[   18] =                7 (       7)
m[   1c] =                8 (       8)
m[   20] =                9 (       9)
m[   24] =               10 (       a)
m[   28] =               55 (      37)
m[   2c] =              820 (     334)
hart 1 data memory dump (non null words)
m[10000] =               11 (       b)
m[10004] =               12 (       c)
m[10008] =               13 (       d)
m[1000c] =               14 (       e)
m[10010] =               15 (       f)
m[10014] =               16 (      10)
m[10018] =               17 (      11)
m[1001c] =               18 (      12)
m[10020] =               19 (      13)
```

```
m[10024] =              20 (       14)
m[10028] =             155 (       9b)
core 1: 192 fetched and decoded instructions in 290 cycles (ipc =
    0.66)
hart 0 data memory dump (non null words)
m[20000] =              21 (       15)
m[20004] =              22 (       16)
m[20008] =              23 (       17)
m[2000c] =              24 (       18)
m[20010] =              25 (       19)
m[20014] =              26 (       1a)
m[20018] =              27 (       1b)
m[2001c] =              28 (       1c)
m[20020] =              29 (       1d)
m[20024] =              30 (       1e)
m[20028] =             255 (       ff)
hart 1 data memory dump (non null words)
m[30000] =              31 (       1f)
m[30004] =              32 (       20)
m[30008] =              33 (       21)
m[3000c] =              34 (       22)
m[30010] =              35 (       23)
m[30014] =              36 (       24)
m[30018] =              37 (       25)
m[3001c] =              38 (       26)
m[30020] =              39 (       27)
m[30024] =              40 (       28)
m[30028] =             355 (      163)
```

13.4 Evaluating the Parallelism Efficiency of the Multicore Multihart IP

Table 13.1 shows the execution time of the matrix multiplication on a multicore and multihart design. They are compared to the baseline time of the sequential version run on the multicycle pipeline design and the speedup is given. The conditions of the run are the same as the ones presented in Sect. 12.6.

The code of the matrix multiplication can be found in the mulmat_xc_yh.c file in the multicore_multihart_ip folder (xc ranges from 2c to 8c, i.e. from two cores to eight cores and xh ranges from 2h to 8h, i.e. from two harts to eight harts; the total number of harts should not be more than 16).

Then, the mulmat_xc_yh_text.hex files are to be built with the build_mulmat_xc_yh.sh script. The run can be simulated in Vitis_HLS with the testbench_mulmat_par_multihart_ip.cpp testbench. The matrix multiplication can be run on the FPGA with the helloworld_mulmat_xc_yh.c driver (xc is 2c or 4c and xh is 2h or 4h with a total number of harts no more than eight).

The lines starting with a star (*) correspond to simulations only as the matching designs cannot be implemented on the XC7Z020 FPGA.

The fastest design to run the matrix multiplication is the 8-core 2-hart (speedup of 7.34). The fastest implementable design is the 8-core 1-hart evaluated in the preceding chapter (4.62 times faster than a single core single hart processor; however, the design

Table 13.1 Execution time of the parallelized matrix multiplication on the multicore_multihart_ip processor and speedup from the sequential run

Number of cores	Number of harts	Cycles	nmi	cpi	Time (s)	Speedup
1	1	6,236,761	3,858,900	1.62	0.124735220	–
2	2	3,072,198	4,601,284	1.34	0.061443960	2.03
2	4	2,466,429	4,728,936	1.04	0.049328580	2.53
*2	8	2,694,867	5,057,744	1.07	0.053897340	2.31
4	2	1,581,360	4,728,936	1.34	0.031627200	3.94
*4	4	1,326,897	5,057,744	1.05	0.026537940	4.70
*8	2	849,410	5,057,744	1.34	0.016988200	7.34

uses 43,731 LUTs to be compared to the 4,111 LUTs used by the single core single hart IP, i.e. nearly 11 times more).

The experience shows that with two harts on a core, the remote memory access latency is hidden by the multithreading mechanism as the speedup is super-optimal (2.03 for two cores) or close to optimal (3.94 for four cores and 7.34 for eight cores) (the speedup can be more than optimal as we run more threads than cores; the optimal speedup related to the number of threads run on a 2-core 2-hart processor is 4).

With four harts per core, the speedup related to the number of cores is super-optimal (2.53 for two cores and 4.70 for four cores).

Conclusion: Playing with the Pynq-Z1/Z2 Development Board Leds and Push Buttons

14

Abstract

This chapter makes you play with the leds and push buttons of the development board. In a first step, an experience is built from a driver run on the Zynq Processing System and directly interacting with the board buttons and leds. Then, the driver is modified to interact with a multicore_multicycle_ip processor presented in Chap. 12. The processor runs a RISC-V program which accesses the board buttons and leds. From the general organization of the multicore_multicycle_ip processor design shown in this chapter, you can develop any RISC-V application to access the resources on the development board (switches, buttons and leds, DDR3 DRAM, SD card), including the expansion connectors (USB, HDMI, Ethernet RJ45, Pmods and Arduino shield).

14.1 A Zynq Design to Access Buttons and Leds on the Development Board

All the source files related to the button/led IP can be found in the pynq_io folder.

All the development boards include a set of buttons and leds. These resources can be accessed from the FPGA, either directly from its PS part (the Zynq Processing System) or from the PL part (the Programmable Logic implementing your RISC-V processors).

In a first step, you will build a design containing a Zynq Processing System and two GPIO IPs (GPIO stands for General Purpose I/O) interconnected with the AXI interconnect IP. One of the two GPIO IPs will be connected to the four button pads out of the FPGA and connected to the push buttons on the board. The other GPIO IP will be connected to the four led pads.

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1_14

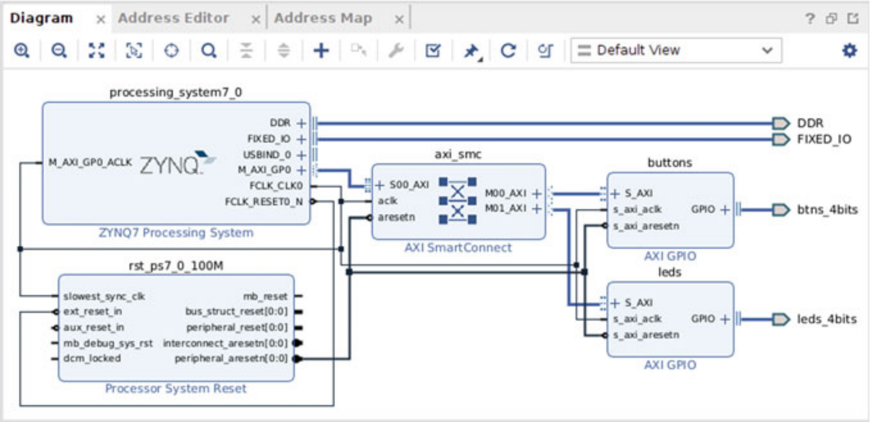


Fig. 14.1 Buttons and leds access design based on the Zynq Processing System

Figure 14.1 shows the GPIO design to be built in Vivado. You must add three IPs: the Zynq7 Processing System and two AXI_GPIO IPs. The AXI smart connect IP is automatically added when you run connect automation. The two GPIO IPs have been renamed buttons and leds. On the connection dialog box, right-click on the buttons GPIO. In the Options/Select Board Part Interface box, select btns_4bits (4 Buttons). For the leds GPIO, select leds_4bits (4 leds).

With the address editor, you can check the addresses allocated by the AXI interconnection system to the two GPIO IPs (0x41200000 for buttons and 0x41210000 for leds).

Once the bitstream has been generated and the hardware has been exported, on Vitis IDE, you should run the helloworld_button_led.c driver shown in Listing 14.1 (all the software resources are located in the pynq_io folder).

Listing 14.1 The GPIO buttons and leds driver

```
#include <stdio.h>
#include "xparameters.h"
#include "xgpio.h"
#define BTN_CHANNEL 1 //AXI_GPIO can be configured with 1 or 2
    channels
#define led_CHANNEL 1 //The Vivado Project configuration is for one
     channel
int main() {
  XGpio_Config *cfg_ptr;
  XGpio leds_device, buttons_device;
  u32 data;
  cfg_ptr = XGpio_LookupConfig(XPAR_LEDS_DEVICE_ID);
  XGpio_CfgInitialize(&leds_device, cfg_ptr, cfg_ptr->BaseAddress);
  cfg_ptr = XGpio_LookupConfig(XPAR_BUTTONS_DEVICE_ID);
  XGpio_CfgInitialize(&buttons_device, cfg_ptr, cfg_ptr->
      BaseAddress);
  //unpressed button = 1 ; pressed button = 0 ; init as unpressed
  XGpio_SetDataDirection(&buttons_device, BTN_CHANNEL, 0xf);
  //off led = 0 ; on led = 1 ; init as off
  XGpio_SetDataDirection(&leds_device, LED_CHANNEL, 0);
  while (1){
    //data is the bitmap of the four buttons with 0/pressed, 1/
        unpressed
    data = XGpio_DiscreteRead(&buttons_device, BTN_CHANNEL);
    XGpio_DiscreteWrite(&leds_device, LED_CHANNEL, data);
  }
}
```

When the driver is running, you light led LD*x* by pressing on button BTN*x* (with *x* ranging from 0 to 3).

14.2 A Design to Access Buttons and Leds from a RISC-V Processor

The RISC-V processor can access the board resources through the AXI interconnection.

The RISC-V processor accesses the external resources through memory addresses out of its internal space (i.e. with loads and stores to addresses beyond the IP_DATA_RAM_SIZE limit).

Hence, the GPIO address spaces must be mapped after the RISC-V processor one and the code run on the RISC-V processor must address these external spaces to access the buttons and leds.

Figure 14.2 shows the design mixing the RISC-V processor and the GPIO IPs.

Figure 14.3 shows the memory mapping of the AXI interconnection. The RISC-V processor data RAM is 128 KB large and ranges from address 0x40000000 to 0x4001ffff. The buttons GPIO space starts at address 0x40020000 and the leds GPIO space starts at address 0x40030000.

The helloworld_button_led_multicore_multicycle.cpp Vitis IDE driver shown in Listing 14.2 runs the multicycle_pipeline_ip which runs the RISC-V code to access the leds and buttons.

It is the same code as the one to drive the multicore_multicycle_ip design presented in Sect. 12.5.1 (do not forget to update the .hex file paths to your environment with the update_helloworld shell script). As the code run by the RISC-V processor is a forever loop, the call to XMulticycle_pipeline_ip_IsDone never returns.

Listing 14.2 The RISC-V GPIO buttons and leds driver

```
#include <stdio.h>
#include "xmulticycle_pipeline_ip.h"
#include "xparameters.h"
#define LOG_NB_IP                1
#define NB_IP                   (1<<LOG_NB_IP)
#define LOG_IP_CODE_RAM_SIZE    (16-LOG_NB_IP)//in word
#define IP_CODE_RAM_SIZE        (1<<LOG_IP_CODE_RAM_SIZE)
#define LOG_IP_DATA_RAM_SIZE    (16-LOG_NB_IP)//in words
#define IP_DATA_RAM_SIZE        (1<<LOG_IP_DATA_RAM_SIZE)
#define DATA_RAM                 0x40000000
int *data_ram = (int*)DATA_RAM;
XMulticycle_pipeline_ip_Config *cfg_ptr;
XMulticycle_pipeline_ip ip;
```

```
int       data     [IP_DATA_RAM_SIZE]={
#include "button_led_data.hex"
};
word_type code_ram[IP_CODE_RAM_SIZE]={
#include "button_led_text.hex"
};
int main(){
  cfg_ptr = XMulticycle_pipeline_ip_LookupConfig(
      XPAR_MULTICYCLE_PIPELINE_0_DEVICE_ID);
  XMulticycle_pipeline_ip_CfgInitialize(&ip, cfg_ptr);
  XMulticycle_pipeline_ip_Set_ip_num    (&ip, 0);
  XMulticycle_pipeline_ip_Set_start_pc  (&ip, 0);
  XMulticycle_pipeline_ip_Write_ip_code_ram_Words(&ip, 0, code_ram,
       IP_CODE_RAM_SIZE);
  XMulticycle_pipeline_ip_Set_data_ram  (&ip, DATA_RAM);
  for (int i=0; i<IP_DATA_RAM_SIZE; i++) data_ram[i] = data[i];
  XMulticycle_pipeline_ip_Start(&ip);
  while (!XMulticycle_pipeline_ip_IsDone(&ip));
  return 0;
}
```

The RISC-V program to be run (button_led.c shown in Listing 14.3) is apparently the same as the driver shown in Listing 14.1.

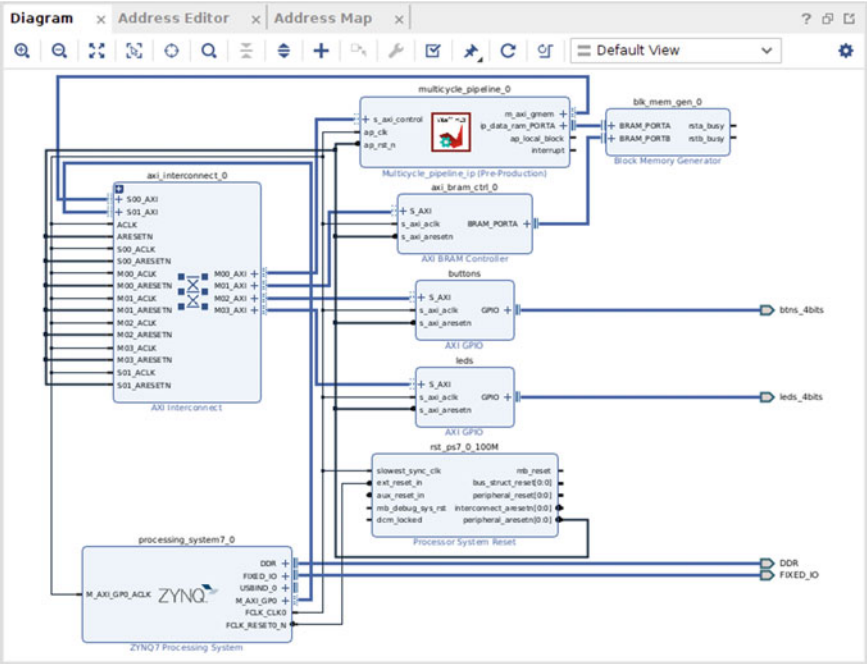


Fig. 14.2 Buttons and leds access design based on the RISC-V multicore_multicycle_ip

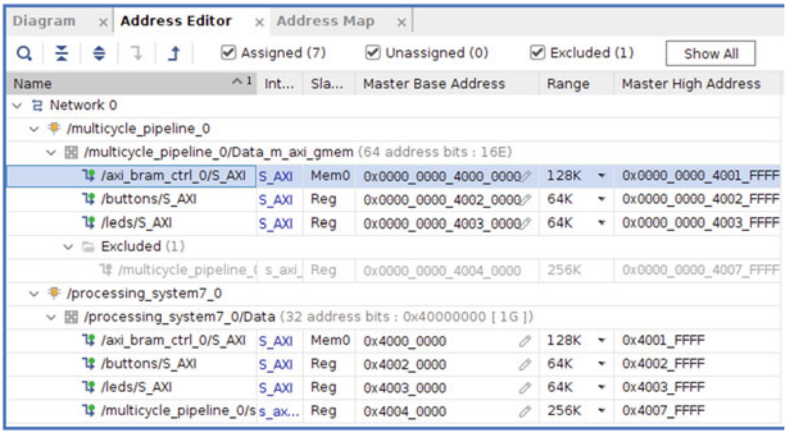
Listing 14.3 The RISC-V GPIO buttons and leds driver

```
#include <stdio.h>
#include "gpio_utils/xparameters.h"
#include "gpio_utils/xgpio.h"
#define BTN_CHANNEL 1 //AXI_GPIO can be configured with 1 or 2
     channels
#define LED_CHANNEL 1 //The Vivado Project configuration is for one
      channel
int main() __attribute__((section(".text.main")));
int main(){
  XGpio_Config *cfg_ptr;
  XGpio leds_device, buttons_device;
  u32 data;
  cfg_ptr = XGpio_LookupConfig(XPAR_LEDS_DEVICE_ID);
  XGpio_CfgInitialize(&leds_device, cfg_ptr, cfg_ptr->BaseAddress);
  cfg_ptr = XGpio_LookupConfig(XPAR_BUTTONS_DEVICE_ID);
  XGpio_CfgInitialize(&buttons_device, cfg_ptr, cfg_ptr->
      BaseAddress);
  //unpressed button = 1 ; pressed button = 0 ; init as unpressed
  XGpio_SetDataDirection(&buttons_device, BTN_CHANNEL, 0xf);
  //off led = 0 ; on led = 1 ; init as off
  XGpio_SetDataDirection(&leds_device, LED_CHANNEL, 0);
  while (1){
    //data is the bitmap of the four buttons with 0/pressed, 1/
        unpressed
    data = XGpio_DiscreteRead(&buttons_device, BTN_CHANNEL);
    XGpio_DiscreteWrite(&leds_device, LED_CHANNEL, data);
  }
}
```

However, the Vivado/Vitis IDE XGpio_ functions must be adapted to the RISC-V processor.

This adaptation only consists in gathering the necessary header files to compile the driver. A few source files are also needed. The original files have been slighly modified to comment some unnecessary inclusions and limit the number of imported files.

The gpio_utils folder contains the C code and header files which are these adaptations of the XGpio_ functions to the RISC-V processor. These files are the modified versions of the original files which can be found in the /opt/Xilinx/Vitis/2022.1/data/embeddedsw/XilinxProcessorIPLib/drivers/gpio_v4_9/src folder.



Name	Int...	Sla...	Master Base Address	Range	Master High Address
Network 0					
/multicycle_pipeline_0					
/multicycle_pipeline_0/Data_m_axi_gmem (64 address bits : 16E)					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x0000_0000_4000_0000	128K	0x0000_0000_4001_FFFF
/buttons/S_AXI	S_AXI	Reg	0x0000_0000_4002_0000	64K	0x0000_0000_4002_FFFF
/leds/S_AXI	S_AXI	Reg	0x0000_0000_4003_0000	64K	0x0000_0000_4003_FFFF
Excluded (1)					
/multicycle_pipeline_(	s_axi_	Reg	0x0000_0000_4004_0000	256K	0x0000_0000_4007_FFFF
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	128K	0x4001_FFFF
/buttons/S_AXI	S_AXI	Reg	0x4002_0000	64K	0x4002_FFFF
/leds/S_AXI	S_AXI	Reg	0x4003_0000	64K	0x4003_FFFF
/multicycle_pipeline_0/s	s_ax...	Reg	0x4004_0000	256K	0x4007_FFFF

Fig. 14.3 AXI interconnect address mapping

To build the RISC-V code, you can use the build.sh shell script (it builds the .hex text and data files according to the GPIO sources and the driver shown in Listing 14.3).

14.3 Conclusion

The technique employed to interface the development board leds and buttons with the RISC-V processor can be applied to all the other devices. In the Xilinx resources, you can find driver examples for the connectors of the various available boards (at https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers;
e.g. the GPIO files can be found at https://github.com/Xilinx/embeddedsw/tree/master/XilinxProcessorIPLib/drivers/gpio/src; a documentation is available at https://xilinx.github.io/embeddedsw.github.io/gpio/doc/html/api/files.html).

The processor designs presented in this book are basic unoptimized hardwares. They can be improved either by expanding the capabilities (i.e. adding the RISC-V ISA expansions), or by optimizing the low-level VHDL or Verilog code which can be derived from HLS when exporting RTL.

Their target is bare-metal but OS-based targets like Linux can also be reached when implementing the RISC-V privileged ISA.

A full machine, with DRAM, keyboard, mouse, and screen could be developed around a development board, using the USB, HDMI, and Ethernet connectors. Only a SATA interface for hard drives is missing but there is an SD-card which can serve as a permanent memory.

All these developments would deserve a new volume including experimentations to build a full computer. It could be combined with a renewed version of the Douglas Comer Xinu implementation of Unix, oriented to Linux kernels.

Index

B. Goossens, *Guide to Computer Processor Architecture*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-18023-1