ETIN35—Manual for RISC-V project

Love Bárány
love.barany@eit.lth.se

March 2025

Contents

1 Introduction, description of project, and some general tips

This document provides a more detailed manual regarding the second-phase project *Build your own RISC-V*, for the course ETIN35 IC-project 1 at LTH, Lund University. As the name suggests, the goal of the project is to build or create your own RISC-V processor that implements a subset of instructions and various performance enhancements. Exactly which instructions, and which enhancements, are listed in the next section.

Your pipeline should consist of 5 stages, be 32-bit, and have a control unit. It should be able to receive new programs over a serial link from a computer, to the FPGA the processor is implemented on. This is done through the UART protocol, which can be used with a USB cable. You can find more information about the UART protocol online, for instance here.

2 More detailed requirements per grade level

This section is accompanied by an annotated RISC-V reference card, where the instructions to implement are marked with a colour per grade level. This chart can be found at the end of this manual, but is also attached as a separate document. For all grade levels, the functionalities listed should be implemented and verified on an FPGA. By implemented, it means that you should be able to synthesize, implement/PnR, and generate a bitstream for the design without issue, and then transfer the bitstream to the FPGA and run it there. By verify, it means that you should be able to run various programs on the FPGA and be able to see that they have worked as expected. The programs should be transferred from a computer via a USB cable, using the UART protocol. The recommended way to verify if your programs are executing correctly is to use the Integrated Logic Analyzer IP Vivado comes with by default, `see Xilinx documentation here`. I recommend you place ILAs in your register file, program memory, and data memory, but you can also place them anywhere you want to easily debug.

There are certain exceptions that can occur based on the instructions you choose to implement, for instance: division by 0, or an incorrect instruction. Since your processor will not have any operating system to throw the exception at, you don't need to handle them in any particular way other than freezing the pipeline for example. Your processor should give some indication that an exception has occurred, for instance by turning on a specific LED on the board. It is up to you to decide exactly how you want to do it, an ILA is also fine for instance.

During approvals, I will run various programs that test the instructions and control units, and I will also go through your code and discuss it with you.

At the end of the term, all groups in each project should hold a presentation together, describing what has been done. It is also interesting to bring up differences between implementations here. Finally, a written report should be written where you describe what you've done in the project, where you also discuss why you chose to implement things in certain ways. More information about the presentation and report will come later, but it's a good idea to document your decisions as you go!

2.1 Grade 3

The instructions to be implemented are almost all of the ones in the basic integer instruction set RV32I, see the instructions marked in green in the accompanying reference card. Other than that, the following is required:

- Implement hazard detection and forwarding. If a hazard is detected which cannot be forwarded, the pipeline should be stalled.
- You should integrate the serial UART controller into your design. There is a finished controller that is provided to you (see later section), but it is only a controller: it doesn't write programs to your program memory.

- Implement a simple 2-bit saturation counter branch predictor. It should only be based on local history. You can decide the size of the branch history table/branch prediction buffer yourselves.

2.2 Grade 4

The instructions to be implemented are from the compressed instruction set RV32C, see the instructions marked in orange in the accompanying reference card. Other than that, the following is required:

- The compressed instructions are smaller than the other, so your decoding logic needs to be expanded.
- You should implement a more advanced branch predictor scheme, a 2-level/correlating predictor. There are many, I recommend `gshare`.

2.3 Grade 5

The instructions to be implemented are from the multiplication and division instruction set RV32M and the floating point (only single precision) instruction set RV32F, see the instructions marked in red in the accompanying reference card. Other than that, the following is required:

- Here the instructions are the hardest part, as many will take more than one cycle to complete. You therefore need to expand your pipeline to handle multi-cycle operations. Stalling the pipeline is enough.
- Floating point numbers are represented differently, and require their own registers.

Alternative grade 5: Instead of implementing instructions from RV32M and RV32F, you can instead implement and verify your design from grade 3 and 4 for ASIC. You should synthesize and PnR, and perform power analysis. You should also create a testbench with serial interface support. Please note that there will another TA who will help you if you choose to take this path.

3 Useful material to consider

There is a lot of information available by a quick Google search, of varying quality. There is a lot of good things that can be found, but to facilitate your searching slightly we recommend the following material. You should be able to complete the project solely based on these, but if you need further material there is nothing stopping you from seeking it out on your own!

3.1 Basic 5-stage pipeline implementations

A previous PhD student created a basic 5-stage pipeline RISC-V, which you can start out from! It provides the basic building blocks in the forms of various modules which should be expanded, and integrated with new ones. You can start from absolutely zero if you would rather do that, but these can still serve as good inspiration.

The basic RISC-V machines are available on GitHub, implemented in SystemVerilog and VHDL:

- `github.com/masoud-ata/riscv_sv` (SystemVerilog)[1]
- `github.com/masoud-ata/PH-RISC-V` (VHDL)

Something to note is that as previously mentioned you should be able to send programs to your RISC-V via UART over USB. The SystemVerilog codebase provides a UART controller, but the VHDL does not. If you choose to use VHDL, you can easily integrate the SystemVerilog file into your project. See for instance `Instantiating Verilog in VHDL`.

3.2 Computer Organization and Design RISC-V Edition, The Hardware Software Interface

This book provides implementation details of a RISC-V pipeline, and is extremely useful for this project. The first few chapters introduce some computer architecture concepts, assembly language and instructions, and some arithmetic for computers. The main chapter is Chapter 4 titled *The Processor*, which describes building a pipeline from the ground. The two basic implementations in Subsection 3.1 are based on this chapter. If you choose to start from zero, it is highly recommended that you follow this chapter. Even if you choose to start from the base implementations, you should read through this chapter to understand how the pipelines are implemented.

The remaining chapters are also interesting, but not as relevant to this project. Appendix A goes through basic digital logic design, and Appendix C describes how to go from control description on paper to digital implementations (this could be seen as a quick refresher of the first VLSI course).

This book is available for free through your LU-account. Either go to `lubsearch.lub.lu.se`, log in with your LU-account, and search for the book, or go to this direct link to its page: `handy direct link`.

3.3 Computer Architecture — A Quantitative Approach

This is the book that was used in the Computer Architecture course. It's much more focused on theory than the previous, but that doesn't mean it isn't useful! It is also written by the same authors, Hennessy and Patterson. I would

[1]A group has let me know that there are some errors in this implementation, which may be fixed, but discuss among yourselves otherwise!

recommend looking at this one when implementing the various branch predictors, since it is more theoretical in nature, and Computer Organization and Design doesn't go into as much detail regarding this. Section C.2 (in the 6th edition) talks of hazards in general, and describes a more simple dynamic branch predictor. Section 3.3 (in the 6th edition) talks about more advanced branch predictors. Section C.3 (in the 6th edition) covers some basic pipeline implementation details, which Computer Organization and Design also covers.

This book is also available for free through your LU-account. The 5th edition is available as an e-book, and various physical libraries around LTH also carry copies of the book. Again, go to `lubsearch.lub.lu.se` and search for it after logging in with your LU-account. Or, go to this `direct link`.

3.4 The RISC-V Reader: An Open Architecture Atlas

This book is also written by Patterson, but Hennessy was not involved with this one (are you starting to see a trend as to who knows their stuff in this field?). This book is great for practical details. It contains more detailed information about all of the instructions that you will implement. Its chapters are divided into the different ISAs, so depending on what grade you're going for you'll need more or less of this book. There are of course other sources for what the different instructions do, however this one is written well in my opinion. Appendix A lists all instructions of RV{I,C,M,F} and more, is easy to search and also explains through more software-like code what each instruction does. The website for this book is `www.riscvbook.com` where you can get free PDFs of the book in Chinese, Spanish, Portuguese, or Korean, but you could also just make a quick google search of the form *"the risc-v reader pdf"* and see what you find! The book is also available on Amazon if you so desire.

3.5 The RISC-V Instruction Set Manual Volume I: Unprivileged ISA

Probably the most technical "book" of the bunch. It is similar to the RISC-V Reader in that it lists all instructions, but this document lists *all* the instructions. This is not a document you scroll through leisurely, it is recommended to know what you're looking for and use the search function to find more. You can find it `at this URL`.

4 Instruction chart with grade levels marked

Instructions to implement

Base Integer Instructions: RV32I and RV64I

Category	Name	Fmt	RV32I Base	+RV64I
Shifts	Shift Left Logical	R	🟩 `SLL rd,rs1,rs2`	`SLLW rd,rs1,rs2`
	Shift Left Log. Imm.	I	🟩 `SLLI rd,rs1,shamt`	`SLLIW rd,rs1,shamt`
	Shift Right Logical	R	🟩 `SRL rd,rs1,rs2`	`SRLW rd,rs1,rs2`
	Shift Right Log. Imm.	I	🟩 `SRLI rd,rs1,shamt`	`SRLIW rd,rs1,shamt`
	Shift Right Arithmetic	R	🟩 `SRA rd,rs1,rs2`	`SRAW rd,rs1,rs2`
	Shift Right Arith. Imm.	I	🟩 `SRAI rd,rs1,shamt`	`SRAIW rd,rs1,shamt`
Arithmetic	ADD	R	🟩 `ADD rd,rs1,rs2`	`ADDW rd,rs1,rs2`
	ADD Immediate	I	🟩 `ADDI rd,rs1,imm`	`ADDIW rd,rs1,imm`
	SUBtract	R	🟩 `SUB rd,rs1,rs2`	`SUBW rd,rs1,rs2`
	Load Upper Imm	U	🟩 `LUI rd,imm`	
	Add Upper Imm to PC	U	🟩 `AUIPC rd,imm`	
Logical	XOR	R	🟩 `XOR rd,rs1,rs2`	
	XOR Immediate	I	🟩 `XORI rd,rs1,imm`	
	OR	R	🟩 `OR rd,rs1,rs2`	
	OR Immediate	I	🟩 `ORI rd,rs1,imm`	
	AND	R	🟩 `AND rd,rs1,rs2`	
	AND Immediate	I	🟩 `ANDI rd,rs1,imm`	
Compare	Set <	R	🟩 `SLT rd,rs1,rs2`	
	Set < Immediate	I	🟩 `SLTI rd,rs1,imm`	
	Set < Unsigned	R	🟩 `SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	🟩 `SLTIU rd,rs1,imm`	
Branches	Branch =	B	🟩 `BEQ rs1,rs2,imm`	
	Branch ≠	B	🟩 `BNE rs1,rs2,imm`	
	Branch <	B	🟩 `BLT rs1,rs2,imm`	
	Branch ≥	B	🟩 `BGE rs1,rs2,imm`	
	Branch < Unsigned	B	🟩 `BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	B	🟩 `BGEU rs1,rs2,imm`	
Jump & Link	J&L	J	🟩 `JAL rd,imm`	
	Jump & Link Register	I	🟩 `JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
Environment	CALL	I	`ECALL`	
	BREAK	I	`EBREAK`	
Control Status Register (CSR)				
	Read/Write	I	`CSRRW rd,csr,rs1`	
	Read & Set Bit	I	`CSRRS rd,csr,rs1`	
	Read & Clear Bit	I	`CSRRC rd,csr,rs1`	
	Read/Write Imm	I	`CSRRWI rd,csr,imm`	
	Read & Set Bit Imm	I	`CSRRSI rd,csr,imm`	
	Read & Clear Bit Imm	I	`CSRRCI rd,csr,imm`	
Loads	Load Byte	I	🟩 `LB rd,rs1,imm`	
	Load Halfword	I	🟩 `LH rd,rs1,imm`	
	Load Byte Unsigned	I	🟩 `LBU rd,rs1,imm`	
	Load Half Unsigned	I	🟩 `LHU rd,rs1,imm`	`LWU rd,rs1,imm`
	Load Word	I	🟩 `LW rd,rs1,imm`	`LD rd,rs1,imm`
Stores	Store Byte	S	🟩 `SB rs1,rs2,imm`	
	Store Halfword	S	🟩 `SH rs1,rs2,imm`	
	Store Word	S	🟩 `SW rs1,rs2,imm`	`SD rs1,rs2,imm`

RV Privileged Instructions

Category	Name	Fmt	RV mnemonic
Trap	Mach-mode trap return	R	`MRET`
	Supervisor-mode trap return	R	`SRET`
Interrupt	Wait for Interrupt	R	`WFI`
MMU	Virtual Memory FENCE	R	`SFENCE.VMA rs1,rs2`

Examples of the 60 RV Pseudoinstructions

Branch = 0 (`BEQ rs,x0,imm`)	J	`BEQZ rs,imm`
Jump (uses `JAL x0,imm`)	J	`J imm`
MoVe (uses `ADDI rd,rs,0`)	R	`MV rd,rs`
RETurn (uses `JALR x0,0,ra`)	I	`RET`

Optional Compressed (16-bit) Instruction Extension: RV32C

Category	Name	Fmt	RVC	RISC-V equivalent
Loads	Load Word	CL	🟧 `C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Float Load Word SP	CL	`C.FLW rd',rs1',imm`	`FLW rd',rs1',imm*8`
	Float Load Word	CI	`C.FLWSP rd,imm`	`FLW rd,sp,imm*8`
	Float Load Double	CL	`C.FLD rd',rs1',imm`	`FLD rd',rs1',imm*16`
	Float Load Double SP	CI	`C.FLDSP rd,imm`	`FLD rd,sp,imm*16`
Stores	Store Word	CS	🟧 `C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Float Store Word	CS	`C.FSW rs1',rs2',imm`	`FSW rs1',rs2',imm*8`
	Float Store Word SP	CSS	`C.FSWSP rs2,imm`	`FSW rs2,sp,imm*8`
	Float Store Double	CS	`C.FSD rs1',rs2',imm`	`FSD rs1',rs2',imm*16`
	Float Store Double SP	CSS	`C.FSDSP rs2,imm`	`FSD rs2,sp,imm*16`
Arithmetic	ADD	CR	🟧 `C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Immediate	CI	🟧 `C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	SUB	CR	🟧 `C.SUB rd,rs1`	`SUB rd,rd,rs1`
	AND	CR	🟧 `C.AND rd,rs1`	`AND rd,rd,rs1`
	AND Immediate	CI	🟧 `C.ANDI rd,imm`	`ANDI rd,rd,imm`
	OR	CR	🟧 `C.OR rd,rs1`	`OR rd,rd,rs1`
	eXclusive OR	CR	🟧 `C.XOR rd,rs1`	`AND rd,rd,rs1`
	MoVe	CR	🟧 `C.MV rd,rs1`	`ADD rd,rs1,x0`
	Load Immediate	CI	🟧 `C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	🟧 `C.LUI rd,imm`	`LUI rd,imm`
Shifts	Shift Left Imm	CI	🟧 `C.SLLI rd,imm`	`SLLI rd,rd,imm`
	Shift Right Ari. Imm.	CI	🟧 `C.SRAI rd,imm`	`SRAI rd,rd,imm`
	Shift Right Log. Imm.	CI	🟧 `C.SRLI rd,imm`	`SRLI rd,rd,imm`
Branches	Branch=0	CB	🟧 `C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	🟧 `C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	🟧 `C.J imm`	`JAL x0,imm`
	Jump Register	CR	🟧 `C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	🟧 `C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	🟧 `C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

Optional Compressed Extention: RV64C

All RV32C (except `C.JAL`, 4 word loads, 4 word strores) plus:

ADD Word (`C.ADDW`)	Load Doubleword (`C.LD`)
ADD Imm. Word (`C.ADDIW`)	Load Doubleword SP (`C.LDSP`)
SUBtract Word (`C.SUBW`)	Store Doubleword (`C.SD`)
	Store Doubleword SP (`C.SDSP`)

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)	+RV64M
Multiply	MULtiply	R	🟥 `MUL rd,rs1,rs2`	`MULW rd,rs1,rs2`
	MULtiply High	R	`MULH rd,rs1,rs2`	
	MULtiply High Sign/Uns	R	`MULHSU rd,rs1,rs2`	
	MULtiply High Uns	R	`MULHU rd,rs1,rs2`	
Divide	DIVide	R	🟥 `DIV rd,rs1,rs2`	`DIVW rd,rs1,rs2`
	DIVide Unsigned	R	🟥 `DIVU rd,rs1,rs2`	
Remainder	REMainder	R	🟥 `REM rd,rs1,rs2`	`REMW rd,rs1,rs2`
	REMainder Unsigned	R	🟥 `REMU rd,rs1,rs2`	`REMUW rd,rs1,rs2`

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)	+RV64A
Load	Load Reserved	R	`LR.W rd,rs1`	`LR.D rd,rs1`
Store	Store Conditional	R	`SC.W rd,rs1,rs2`	`SC.D rd,rs1,rs2`
Swap	SWAP	R	`AMOSWAP.W rd,rs1,rs2`	`AMOSWAP.D rd,rs1,rs2`
Add	ADD	R	`AMOADD.W rd,rs1,rs2`	`AMOADD.D rd,rs1,rs2`
Logical	XOR	R	`AMOXOR.W rd,rs1,rs2`	`AMOXOR.D rd,rs1,rs2`
	AND	R	`AMOAND.W rd,rs1,rs2`	`AMOAND.D rd,rs1,rs2`
	OR	R	`AMOOR.W rd,rs1,rs2`	`AMOOR.D rd,rs1,rs2`
Min/Max	MINimum	R	`AMOMIN.W rd,rs1,rs2`	`AMOMIN.D rd,rs1,rs2`
	MAXimum	R	`AMOMAX.W rd,rs1,rs2`	`AMOMAX.D rd,rs1,rs2`
	MINimum Unsigned	R	`AMOMINU.W rd,rs1,rs2`	`AMOMINU.D rd,rs1,rs2`
	MAXimum Unsigned	R	`AMOMAXU.W rd,rs1,rs2`	`AMOMAXU.D rd,rs1,rs2`

Two Optional Floating-Point Instruction Extensions: RVF & RVD

Category	Name	Fmt	RV32{F|D} (SP,DP Fl. Pt.)	+RV64{F|D}
Move	Move from Integer	R	`FMV.W.X rd,rs1`	`FMV.D.X rd,rs1`
	Move to Integer	R	`FMV.X.W rd,rs1`	`FMV.X.D rd,rs1`
Convert	ConVerT from Int	R	🟥 `FCVT.S.W rd,rs1`	`FCVT.{S|D}.L rd,rs1`
	ConVerT from Int Unsigned	R	🟥 `FCVT.S.WU rd,rs1`	`FCVT.{S|D}.LU rd,rs1`
	ConVerT to Int	R	🟥 `FCVT.W.S rd,rs1`	`FCVT.L.{S|D} rd,rs1`
	ConVerT to Int Unsigned	R	🟥 `FCVT.WU.S rd,rs1`	`FCVT.LU.{S|D} rd,rs1`
Load	Load	I	🟥 `FLW rd,rs1,imm`	
Store	Store	S	🟥 `FSW rs1,rs2,imm`	
Arithmetic	ADD	R	🟥 `FADD.S rd,rs1,rs2`	
	SUBtract	R	🟥 `FSUB.S rd,rs1,rs2`	
	MULtiply	R	🟥 `FMUL.S rd,rs1,rs2`	
	DIVide	R	🟥 `FDIV.S rd,rs1,rs2`	
	SQuare RooT	R	`FSQRT.{S|D} rd,rs1`	
Mul-Add	Multiply-ADD	R	`FMADD.{S|D} rd,rs1,rs2,rs3`	
	Multiply-SUBtract	R	`FMSUB.{S|D} rd,rs1,rs2,rs3`	
	Negative Multiply-SUBtract	R	`FNMSUB.{S|D} rd,rs1,rs2,rs3`	
	Negative Multiply-ADD	R	`FNMADD.{S|D} rd,rs1,rs2,rs3`	
Sign Inject	SiGN source	R	`FSGNJ.{S|D} rd,rs1,rs2`	
	Negative SiGN source	R	`FSGNJN.{S|D} rd,rs1,rs2`	
	Xor SiGN source	R	`FSGNJX.{S|D} rd,rs1,rs2`	
Min/Max	MINimum	R	`FMIN.{S|D} rd,rs1,rs2`	
	MAXimum	R	`FMAX.{S|D} rd,rs1,rs2`	
Compare	compare Float =	R	`FEQ.{S|D} rd,rs1,rs2`	
	compare Float <	R	`FLT.{S|D} rd,rs1,rs2`	
	compare Float ≤	R	`FLE.{S|D} rd,rs1,rs2`	
Categorize	CLASSify type	R	`FCLASS.{S|D} rd,rs1`	
Configure	Read Status	R	`FRCSR rd`	
	Read Rounding Mode	R	`FRRM rd`	
	Read Flags	R	`FRFLAGS rd`	
	Swap Status Reg	R	`FSCSR rd,rs1`	
	Swap Rounding Mode	R	`FSRM rd,rs1`	
	Swap Flags	R	`FSFLAGS rd,rs1`	
	Swap Rounding Mode Imm	I	`FSRMI rd,imm`	

Calling Convention

Register	ABI Name	Saver
`x0`	`zero`	---
`x1`	`ra`	Caller
`x2`	`sp`	Callee
`x3`	`gp`	---
`x4`	`tp`	---
`x5-7`	`t0-2`	Caller
`x8`	`s0/fp`	Callee
`x9`	`s1`	Callee
`x10-11`	`a0-1`	Caller
`x12-17`	`a2-7`	Caller
`x18-27`	`s2-11`	Callee
`x28-31`	`t3-t6`	Caller
`f0-7`	`ft0-7`	Caller
`f8-9`	`fs0-1`	Callee
`f10-11`	`fa0-1`	Caller
`f12-17`	`fa2-7`	Caller
`f18-27`	`fs2-11`	Callee
`f28-31`	`ft8-11`	Caller

`zero`	Hardwired zero
`ra`	Return address
`sp`	Stack pointer
`gp`	Global pointer
`tp`	Thread pointer
`t0-6,ft0-11`	Temporaries
`s0-11,fs0-11`	Saved registers

🟩	*Grade 3*
🟧	*Grade 4*
🟥	*Grade 5*

Note that only single-precision floating point operations should be implemented for Grade 5!