Lab 5: Processor Design

1 Introduction

A processor is a digital circuit that executes instructions that are stored in memory. In lab 4, you used the Nios II processor to execute code as well as to control peripherals such as memory and the floating point multiplier.

In this lab, you will design and build your own processor that uses the base RISC-V instruction set (RV32I). You will be provided with a specification of this Instruction Set Architecture (ISA) and an external signal interface. Your task will be to implement the processor as a set of Verilog/SystemVerilog modules according to these specifications. You will *not* be required to synthesize your processor with Quartus Prime on the DE1-SoC board in this lab but your code should still be synthesizable[1]. Instead, you will simulate it in a provided ModelSim testbench, which will contain a (simulated) memory block holding the program code for the processor to execute. The source code and assembled hex files for several small programs will be available, which you can use to test your processor, along with an assembler that can compile these and other programs.

2 Processor Architecture

2.1 Registers

Like the Nios II, the processor you are building has registers, instructions, and a datapath that are 32 bits wide. Figure 1 shows an overview of the set of programmer-visible registers. There are 32 32-bit general-purpose registers namely `x0-x31` that can be directly manipulated by instructions. Register `x0` however is always zero.

The Program Counter (`PC`) register contains the *byte address* of the currently executing instruction, and is always an even number (the lowest bit is 0). The `PC` should automatically increment by 4 after each executed instruction unless the instruction is a jump or taken branch which adds an immediate value to the `PC`.

2.2 Instruction Set

In the base RV32I ISA, there are six core instruction formats (R/I/S/B/U/J), as shown in Table 1. All are a fixed 32 bits in length and must be aligned on a four-byte (32-bits) boundary in memory. Each of the formats corresponds to different types of instructions. All instructions of the same format share the same encoding with regard to the placement of their values in the 32 bits.

For simplicity in the decoding process, the opcode is in the same place `[6:0]` across all formats. The same is true for the source (`rs1`, `rs2`) and destination (`rd`) operands in the formats that use them. Operands `rd`, `rs1` and `rs2` are *placeholders* for the *names* of one of the 32 general-purpose registers (i.e. if `rs1` is `00001` then

[1] You will use your code for lab 7 as well, where you **will** be required to synthesize it.

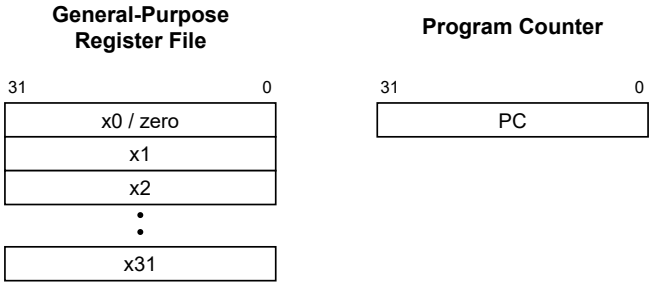

Figure 1: The processor's registers.

it refers to register `x1`). The `funct3` and `funct7` values serve the purpose of differentiating instructions that have the same opcode.

Immediate values. The place of the immediate value (`imm`) differs across formats and may require some manipulation before it can be used. All immediate values are sign-extended to 32 bits.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Table 1: Instruction Formats

Table 2[1] lists the 37 instructions that the processor can understand and execute[2]. The first column is the **instruction** itself, while the second column is its **name**. The third and fourth columns give the instruction **format** and the **opcode** respectively. The fifth and sixth columns show the **`funct3`** and **`funct7`** values. You can see that `funct3` values repeat across instructions belonging to different formats, but are unique for same-format ones. The second-to-last column provides a **description** of the operation performed by the instruction. The last column provides **notes** about converting the result of specific instructions to 32 bits (e.g., a Load Half Unsigned instruction loads a 16-bit value from memory but is padded with zeroes in the upper 16 bits so as to create a 32-bit value).

As an example, the word `000000000001|00101|000|00111|0010011` (hex `0x00128393`) decodes to '`addi x7, x5, 0x001`', and would add the immediate value `0x00000001` to the value in register `x5` and store the result in register x7. The rest of this section describes the instructions in more detail.

2.2.1 Arithmetic Instructions

There are ten arithmetic instructions (`add sub xor or and sll srl sra slt sltu`) that use register values as operands and their equivalents that substitute one register with an immediate value, resulting in a total of nineteen instructions (there is no `subi` instruction). These are the only instructions that make use of the `funct7` value to differentiate among them. The result of these instructions is stored in a destination register (`rd`). You should be **careful** when implementing the different shift instructions, especially the distinction between logical shift and arithmetic shift. The `sll(i)` instruction is a logical left shift (zeros are shifted into the lower bits); `srl(i)` is a logical right shift (zeros are shifted into the upper bits); and `sra(i)` is an

[1] RISC-V Card source

[2] The full RV32I instruction set includes three other instructions (`fence ecall ebreak`) which you do **not** need to implement.

Inst	Name	FMT	Opcode	funct3	funct7	Description	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2[4:0]	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2[4:0]	
sra	Shift Right Arithmetic	R	0110011	0x5	0x20	rd = rs1 >> rs2[4:0]	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2) ? 1 : 0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2) ? 1 : 0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[11:5]=0x00	rd = rs1 << imm[4:0]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[11:5]=0x00	rd = rs1 >> imm[4:0]	
srai	Shift Right Arithmetic Imm	I	0010011	0x5	imm[11:5]=0x20	rd = rs1 >> imm[4:0]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][7:0]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][15:0]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][31:0]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][7:0]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][15:0]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][7:0] = rs2[7:0]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][15:0] = rs2[15:0]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][31:0] = rs2[31:0]	
lui	Load Upper Immediate	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
beq	Branch ==	B	1100011	0x0		PC = (rs1 == rs2) ? PC+imm : PC+4	
bne	Branch !=	B	1100011	0x1		PC = (rs1 != rs2) ? PC+imm : PC+4	
blt	Branch <	B	1100011	0x4		PC = (rs1 < rs2) ? PC+imm : PC+4	
bge	Branch $\geq$	B	1100011	0x5		PC = (rs1 >= rs2) ? PC+imm : PC+4	
bltu	Branch < (U)	B	1100011	0x6		PC = (rs1 < rs2) ? PC+imm : PC+4	zero-extends
bgeu	Branch $\geq$ (U)	B	1100011	0x7		PC = (rs1 >= rs2) ? PC+imm : PC+4	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	

Table 2: Base Integer Instructions

arithmetic right shift (the original sign bit is copied into the vacated upper bits). Lastly, note that the `slt(i)` and `slt(i)u` store a 1 or a 0 in the destination register depending on the result of the comparison of the first source operand to either the second source operand or an immediate value. The difference between them is that `slt(i)u` assumes that the values are always unsigned (the immediate is sign extended as normal before being treated as unsigned).

2.2.2 Loads and Stores

The load (`lb lh lw lbu lhu`) and store (`sb sh sw`) instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores in the S-type. The effective address is obtained by adding register `rs1` to the sign-extended 12-bit offset. Loads copy a value from memory to register `rd`. Stores copy the value in register `rs2` to memory. **Important: be careful when copying values that are less than 32-bit long. Depending on the instruction you may need to either sign-extend (e.g., `lh`) or zero-extend them (e.g., `lhu`).**

2.2.3 LUI and AUIPC

The `lui` instruction does **not** access memory. It uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register `rd`, filling the lowest 12 bits with zeros.

The `auipc` instruction (add upper immediate to `PC`) is used to build `PC`-relative addresses and uses the U-type format. It forms a 32-bit offset by combining the 20-bit U-immediate with 12 trailing zero bits and adds this offset to the address of itself. The result of this is stored in register `rd`. It does **not** modify the `PC`.

2.2.4 Control Flow Instructions

The last eight instructions in Table 2 change the value of the `PC` and thus alter the control flow of the program. All branch instructions use the B-type instruction format. They compare the two source registers (`rs1` and `rs2`) and depending on the result and the specific branch instruction they either increment the `PC` by 4 or they add the immediate value to the PC. For example, **`beq`** branches only when the values of `rs1` and `rs2` match. The 12-bit B-immediate encodes signed offsets. The offset is sign-extended and added to the address of the branch instruction to give the target address.

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset. The offset is sign-extended and added to the address of the jump instruction to form the jump target address. JAL stores the address of the instruction following the jump (`PC`+4) into register `rd`. The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the sign-extended 12-bit I-immediate to the register `rs1`, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (`PC`+4) is written to register `rd`.

2.3 Operation

Upon reset, the contents of registers `PC` and `x0`-`x31` shall be set to 0. After coming out of reset, the processor will continue to execute instructions forever. This involves:

1. Fetching the 32-bit word from memory at address = `PC`.
2. Incrementing `PC` by 4 if the instruction is not a jump or taken branch.
3. Based on the encoding of the fetched 32-bit word, performing the corresponding operation(s) in Table 2.
4. Going back to step 1.

The number of cycles required to perform each step is an implementation detail, and up to you to decide. Some instructions, such as `lw`, may require an additional clock cycle(s) to complete. For this lab, it is recommended to only perform one step at a time, and to have only one instruction being executed at a time. A simple design can execute one instruction every 3 or 4 clock cycles. To simplify the processor's design, the behavior is *undefined* when trying to execute an instruction that does not exist in Table 2. You can do as you wish. **Tip: In lab 7, you will be asked to pipeline the processor you design in this lab to have 4 stages. Thus you may want to design your processor with that in mind.**

2.4 Signal Interface

Your processor can not work in isolation – it must be connected to a memory bus in order to fetch instructions and to access data (and in the next lab, I/O devices) via loads and stores. Table 3 lists the required signals. Your processor writes to memory by asserting address, writedata, and the write enable. For reads, read data will be returned to you one cycle after you assert address and the read enable. Take this timing into account when fetching instructions or performing loads.

Signal	Direction	Width	Description
clk	input	1	Clock
reset	input	1	Active-high reset
o_pc_addr	output	32	Address (in bytes)
o_pc_rd	output	1	PC Read enable
i_pc_rddata	input	32	PC Read data
o_ldst_addr	output	32	Address (in bytes)
o_ldst_rd	output	1	Load/Store Read enable
i_ldst_rddata	input	32	Load/Store Read data
o_ldst_wr	output	1	Load/Store Write enable
o_ldst_wrdata	output	32	Load/Store Write data
o_tb_regs	output	32 x 32	Register values

Table 3: Processor signal interface

3 Testing Your Processor

3.1 Testbench

Included with the starter kit is a SystemVerilog testbench (`tb.sv`) that instantiates your processor and connects it to a memory block. The memory block is initialized with a `.hex` file containing the machine code and data for a compiled program. Five programs are included but you can compile your own too. You can add other programs by modifying the `PROGRAMS` array. You select which of these programs to run by invoking the `do_test` function (with the index of the test in the `PROGRAMS` array) in the initial block at the bottom of the testbench file.

The testbench treats writes to certain memory addresses in a special way. When the program writes a value to address `0x1000`, the testbench will print out this number in the ModelSim console. When the program writes a value to address `0x1002`, the value is treated as a pointer to a null-terminated string, and the string is printed out.

3.2 Included Programs

Five programs are included to help you test all of the instructions of your processor. These are reduced versions of the programs that will be used to test the correctness of your implementation. Each has source code in a `.s` file, and an assembled version ready to be used by the testbench in a corresponding `.hex` file.

- **0_basic:** Contains a sequence of basic independent instructions. Try this one first.
- **1_arithdep:** Contains A sequence of dependent arithmetic instructions.
- **2_branch_nottaken:** This program tests conditional branches that are not taken.
- **3_branch_taken:** This program test branches, conditional or not, that are always taken.
- **4_memdep:** This program test loads and stores from/to memory.

3.3 Writing Your Own Programs

We provide an assembler with the starter kit that can compile programs for your processor, written with the instruction set from Table 2. It outputs a `.hex` memory initialization file that can be used by Quartus Prime or ModelSim to initialize the contents of a Verilog memory block. The assembler comes with a README file that guides you through setting everything up and using it.

You are not required to write or submit your own programs, but keep in mind that the provided programs do not test all instructions and cases. The automarker will use additional programs that may test additional instructions so you are expected to test out all the instructions you implement.

4 Submission

To help you move forward in stages, this lab is split into three parts, with each part requiring you to implement a subset of the instructions presented in Table 2.

- **Part 1:** Implement the arithmetic instructions (first two sets in Table 2) along with `lui` and `auipc`. Your processor should be able to pass tests **0_basic** and **1_arithdep**. (2 marks)
- **Part 2:** Implement branch and jump instructions. Your processor should be able to pass tests **2_branch_nottaken** and **3_branch_taken**. (2 marks)
- **Part 3:** Implement load and store instructions. Your processor should be able to pass the **4_memdep** test. (2 marks)

This is just a suggestion so feel free to implement as you see fit. For the automarker you can simply submit a single file which is your processor. All the test programs will be run for that processor.

Important: You do not need to implement three different versions of your processor. You can design a single processor and add functionality as you work on each new part. In the end you will submit just a single file for marking.

You will submit a single file with all your code using the following command: `submitece342s 5 part1.sv`

Frequently Asked Questions

In this section, questions from previous years have been collected and answered. Please check here first before posting a question on Piazza or Quercus.

Q1: Should we take care of overflows in the ADD/SUB instructions?

A1: No, you can just let them overflow. Commercial processors have a separate overflow flag for this, but you do not need to implement this.

Q2: Do we need to optimize this processor to use the minimum number of cycles possible?

A2: No. For this lab, your processor can be a single-stage multi-cycle processor.