Lab 5: Processor Design

1 Introduction

A processor is a digital circuit that executes instructions that are stored in memory. In previous labs (and courses), you have had experience using the Nios II processor, both in terms of writing code to execute on it as well as integrating the processor into a larger system with memory and peripherals.

In this lab, you will design and build your own processor. You will be provided with a specification of its instruction set and 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. 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 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

Unlike the Nios II, which has a 32-bit architecture, the processor you are building has registers, instructions, and a datapath that are only 16 bits wide. Figure 1 shows an overview of the set of programmer-visible registers. There are eight general-purpose 16-bit registers named `R0-R7` that can be directly manipulated by instructions.

The Program Counter (PC) register contains the *byte address* of the currently executing instruction to fetch from memory, and is always an even number (the lowest bit is 0). The PC should automatically increment by 2 after each executed instruction, and certain instructions (jumps and calls) can further modify its value indirectly.

There are also two 1-bit *flags* registers, N and Z. These stand for **N**egative and **Z**ero. The flags are updated after every arithmetic instruction (addition, subtraction, or comparison) and are used by conditional jump instructions, discussed later.

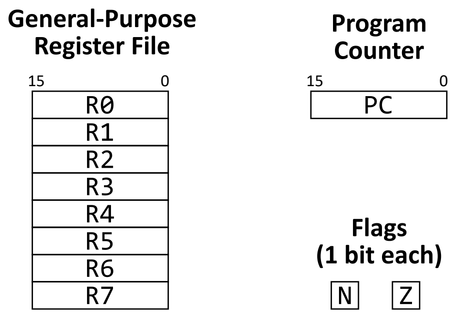

Figure 1: The processor's programmer-visible registers

Instruction	Operation Performed	Encoding				
		15 11	10 8	7 5	4	3 0
`mv Rx,Ry`	`[Rx] ← [Ry]`	*(unused)*	Ry	Rx	0	0000
`add Rx,Ry`	`[Rx] ← [Rx] + [Ry]` `update N,Z`		Ry	Rx	0	0001
`sub Rx,Ry`	`[Rx] ← [Rx] - [Ry]` `update N,Z`		Ry	Rx	0	0010
`cmp Rx,Ry`	`evaluate [Rx] - [Ry]` `update N,Z`		Ry	Rx	0	0011
`ld Rx,Ry`	`[Rx] ← mem[[Ry]]`		Ry	Rx	0	0100
`st Rx,Ry`	`mem[[Ry]] ← [Rx]`		Ry	Rx	0	0101
`mvi Rx,imm8`	`[Rx] ← s_ext(imm8)`	imm8		Rx	1	0000
`addi Rx,imm8`	`[Rx] ← [Rx] + s_ext(imm8)` `update N,Z`	imm8		Rx	1	0001
`subi Rx,imm8`	`[Rx] ← [Rx] - s_ext(imm8)` `update N,Z`	imm8		Rx	1	0010
`cmpi Rx,imm8`	`evaluate [Rx] - s_ext(imm8)` `update N,Z`	imm8		Rx	1	0011
`mvhi Rx,imm8`	`[Rx][15:8] ← imm8`	imm8		Rx	1	0110
`jr Rx`	`PC ← [Rx]`	*(unused)*		Rx	0	1000
`jzr Rx`	`PC ← [Rx] if Z==1`			Rx	0	1001
`jnr Rx`	`PC ← [Rx] if N==1`			Rx	0	1010
`callr Rx`	`R7 ← PC` `PC ← [Rx]`			Rx	0	1100
`j imm11`	`PC ← PC + 2*s_ext(imm11)`	imm11			1	1000
`jz imm11`	`PC ← PC + 2*s_ext(imm11) if Z==1`	imm11			1	1001
`jn imm11`	`PC ← PC + 2*s_ext(imm11) if N==1`	imm11			1	1010
`call imm11`	`R7 ← PC` `PC ← PC + 2*s_ext(imm11)`	imm11			1	1100

Table 1: The processor's instruction set

2.2 Instruction Set

Table 1 lists the 19 instructions that the processor can understand and execute. Each instruction is encoded as a 16-bit word. The first column gives each instruction's name and operands. Operands `Rx` or `Ry` are *placeholders* for the *names* of one of the eight general-purpose registers (eg. if `Ry` is `100` then it refers to register `R4`). Some instructions take their inputs not from the contents of registers, but from *immediate operands*, which are numbers stored within the instruction word itself. These come in two sizes: 8-bit (`imm8`) or 11-bit (`imm11`), and are sometimes sign-extended to 16 bits before use.

The second column describes the operations performed by the instruction. `[Rx]` refers to the contents of the register named by the 3-bit index `Rx` in the general-purpose register file, and this is the convention used throughout this document. For example, the instruction '`mv R3,R5`', according to Table 1, copies the contents of `R5` into `R3`. `s_ext()` refers to sign-extending an 8-bit or 11-bit immediate value to a full 16 bits.

Finally, the right-hand side of the table gives the encoding of the instruction. The least-significant 5 bits specify the *opcode* that identifies the type of the instruction, with the remaining 11 bits encoding the operands, which are some combination of register names and/or immediate values. As an example, the word `1111001011010000` (hex `0xF2D0`) decodes to '`mvi R6,0xF2`', and would store the value `0xFFF2` (`-14` decimal) in register `R6` due to sign extension. The rest of this section describes the instructions in more detail.

2.2.1 Arithmetic Instructions

The six arithmetic instructions (`add sub cmp addi subi cmpi`) perform either addition or subtraction, and are the only instructions that update the flags registers `N` and `Z` based on the result of the arithmetic operation. The comparison instructions `cmp` and `cmpi` are equivalent to `sub` and `subi` except they do not write the result back to the register file – they just update the flags.

- `N` is set to bit 15 of the result of the arithmetic operation, which signifies that the result is **N**egative when interpreted as a twos-complement signed number.
- `Z` is set to 1 only if the entire 16-bit arithmetic result is **Z**ero.

2.2.2 Loads and Stores

The `ld` and `st` instructions are the only ones that directly access memory. Both take two register operands `Rx` and `Ry`, and use `[Ry]` as a **byte address** into memory. To simplify the processor design, all loads and stores are forced word-aligned, meaning **bit 0 of the address used to access memory is always set to 0 regardless of the value of bit 0 of `[Ry]`** . For example, both of the store instructions in the code below store the value `0x0012` into memory location `0x006c`:

```
mvi R0, 0x12
mvi R1, 0x6c
st  R0, R1
mvi R1, 0x6d
st  R0, R1
```

2.2.3 Control Flow Instructions

The last eight instructions in Table 1 use either a single register operand `Rx` or an 11-bit immediate operand `imm11` to change the value of the `PC` and thus alter the control flow of the program. When the operand is `Rx`, then `PC` is simply set to `[Rx]`. When the operand is `imm11`, the 16-bit sign-extended value of `imm11` is treated as a **signed relative offset** (in units of 16-bit instructions, not bytes) so the `PC` is *incremented* by *twice* this value. The `imm11` mode gives your processor the ability to jump up to 1024 instructions forwards or backwards relative to the current `PC` without having to load a jump address into a register first.

Important: the signed offset shall be applied after the PC has already been incremented by 2, which is the PC's common behavior for all instructions. This will simplify the design of your processor. Here is an example of a jump instruction that creates an infinite loop:

```
j 0x7FF
```

This is because the 11-bit immediate value (`0x7FF`), sign-extended to 16 bits, is `0xFFFF` or `-1`. When the instruction executes, `PC` first advances by 2 (as it does for every instruction) and then changes to `PC + 2*(-1) = PC-2` by adding twice the `-1` jump offset to `PC`. `PC` is now back to what it originally was.

Conditional Jumps

The conditional jumps `jz/jzr` and `jn/jnr` behave just like `j/jr` except that they only change the `PC` if the corresponding flag (`Z` or `N` respectively) is set to 1. Otherwise, the `PC` advances by 2 to the next instruction.

Calls and Subroutines

The `call` and `callr` functions are similar to the unconditional jump instructions `j` and `jr`, but in addition to changing the `PC`, they first store the current value of `PC` (**after it has been incremented by 2**) into register `R7`. This effectively remembers the address of the next instruction after the `call`, allowing a return to that location later by performing a `'jr R7'`. This provides a mechanism to implement subroutines.

2.3 Operation

Upon reset, the contents of registers `PC`, `R0-R7`, `N`, and `Z` shall be set to 0. After coming out of reset, the processor will continue to execute instructions forever. This involves:

1. Fetching the 16-bit word from memory at address = `PC`.
2. Incrementing `PC` by 2.
3. Based on the encoding of the fetched 16-bit word, performing the corresponding operation(s) in Table 1.
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 `ld`, may require additional clock cycles 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 1. You can do as you wish.

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 2 lists the required signals.

Signal	**Direction**	**Width**	**Description**
`clk`	input	1	Clock
`reset`	input	1	Active-high reset
`o_mem_addr`	output	16	Address (in bytes)
`o_mem_rd`	output	1	Read enable
`i_mem_rddata`	input	16	Read data
`o_mem_wr`	output	1	Write enable
`o_mem_wrdata`	output	16	Write data

Table 2: Processor signal interface

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. Figure 2 illustrates the timing for reads and writes.

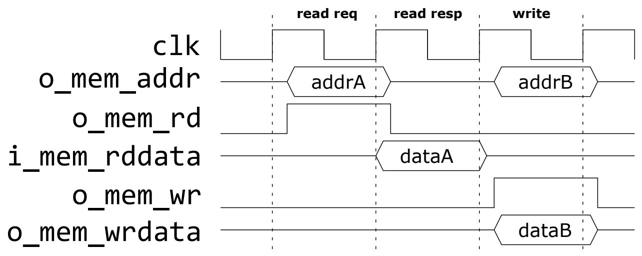

Figure 2: Memory read and write timing

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. Three programs are included but you can compile your own too. Change the `HEX_FILE` parameter at the top of the testbench to change the program.

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 and then terminate the simulation. 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, also terminating the simulation afterwards. Unlike strings in C, each character of a string is 16 bits, with the upper 8 bits unused.

3.2 Included Programs

Three programs are included to help you test all of the instructions of your processor. Each has source code in a `.s` file, and an assembled version ready to be used by the testbench in a corresponding `.hex` file.

- **sum10:** Calculates the sum of the integers from 1 to 10. It should display the result 55 (`0x37` hex) when it finishes. Try this one first.
- **capital:** Takes the string stored in the data section of the program, converts it to uppercase, and prints it out using the "write to address 0x1002" functionality of the testbench. This tests the ability to call and return from subroutines using the `call` and `jr` instructions.
- **regjump:** This is a more abstract test that exercises the `jnr` and `jnz` conditional register jump instructions. It simply prints out "Success" or "Fail".

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 1. 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. It also outputs a `.mif` version of the same program, which is required for Lab 6.

The assembler is called `asm.exe` and can run in Windows. The source code, and a Makefile, are provided for compilation on other platforms like Linux. Ideally, you would run `asm.exe` from a Windows Command Prompt, so that you can see it report any syntax errors in your code. You can also drag and drop a `.s` file onto the executable, but you won't be able to see any messages from the assembler.

4 Lab Instructions and Marking Scheme

This is a two-week lab and the following items will all be marked during the second week. You are encouraged to also attend the first week lab session to receive design assistance from your TA.

- **Preparation:** Draw the datapath for your processor. Include the names of all control signals (2 marks)
- **Preparation:** Write a Verilog/SystemVerilog implementation of your processor and name the top-level module `cpu`. Its input/output signals should be the ones from Table 2 (4 marks)
- **In-Lab:** Demonstrate that your processor functions correctly by showing your TA the simulated execution of the three included test programs in ModelSim, or your own program(s) that exercise all the processor's instructions (6 marks)

Frequently Asked Questions

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