CprE 3810, Computer Organization and Assembly Level Programming
Lab 2

Note

In this lab you will create the bones of a RISC-V datapath and hand-generate control signals for it. The result will form the basis for your project. This lab will require greater autonomy and will likely take you longer than Lab 1——please plan accordingly.

1 Prelab ✅

Read this lab manual. In addition, read *Free Range VHDL* Chapters 5, 6, and 7: pages 51–68, 71–86. Describe the provided `dffg.vhd` in terms of edge sensitivity and reset type (active low/high and synchronous/asynchronous).

2 Debug Report

~~Throughout this portion of the assignment, you must post at least one bug report to your lab section channel. If you have no bugs, practice by inventing one and posting it. You must also make at least one substantive comment on a lab mate's bug report, and once you solve the bug, post which solution worked.~~

3 Register File

The RISC-V(RV32) register file contains 32, 32-bit registers, along with two ports for reading (corresponding to rs1 and rs2 in a RISC-V ISA R-type instruction) and one port for writing (corresponding to rd in a RISC-V ISA R-type instruction). Provide your solution (VHDL code, simulation waveforms) in a folder called `RegFile/`.

a. Draw the interface description (i.e., the "symbol" or high-level blackbox) for the RISC-V register file. Which ports do you think are necessary, and how wide (in bits) do they need to be? [*Some things to think about when answering this question: will this need to be a synchronous (i.e. with a clock) circuit? Are you going to have a reset signal? How would one be able to differentiate between read and write requests? If you are unsure about the answers to these questions from lectures, some answers can be found in P&H chapter 4. Verify with your TA before proceeding.*]

b. The first component you will need is an individual register. A behavioral VHDL edge-triggered flip-flop (with parallel access and reset) is provided in file `dffg.vhd`. Create an N-bit register using this flip-flop as your basis. [This should now be simple using structural VHDL with generics.]

c. Create a testbench to test your register design to make sure it is working as expected, and include a waveform screenshot in your report PDF. A sample testbench that also incorporates a clock generator can be found in `tb_dff.vhd`. [*You will need to modify this slightly for your N-bit version.*]

d. A decoder is a logic structure that takes in an N-bit value, and sets one bit out of a 2^N-bit output value based on the corresponding decimal representation (e.g. a `3:8` decoder will output "`00000010`" when the input is "`001`", will output "`00010000`" when the input is "`100`", and will generally set only bit I when the binary input corresponds to I in decimal). What type of decoder would be required by the RISC-V register file and why?

e. Implement the appropriate decoder using either structural or dataflow VHDL. Create a testbench and use QuestaSim to test your decoder design to make sure it is working as expected. [*It is not easy (and not worth it) to make this a general N:2N decoder. Free Range VHDL provides a dataflow representation of a decoder using "with-select-when" that will greatly simplify this task.*]

f. For a given read request on the register file, we only want a single register's output value, even though all 32 registers can and will be read in parallel. A 32-bit `32:1` multiplexer can be used to accomplish this task. Based on the dataflow and structural VHDL we have already learned, there are several different ways to implement a 32-bit 32:1 multiplexer. In your write-up, describe and defend the design you intend on implementing for the next part.

g. Based on your answer, implement your 32-bit 32:1 multiplexer. Create a testbench and use QuestaSim to test your design to make sure it is working as expected. *[You can design this component however you want. Generally a dataflow implementation will be easier to code and faster to simulate. On the flip side, a structural implementation provides more detail about the underlying logic layout. Hybrid approaches are also valid.]*

h. Draw a (simplified) schematic (i.e., components within the high-level blackbox) for the RISC-V register file, using the same top-level interface ports as in your solution describe above and using only the register, decoder, and mux VHDL components you have created. Keep in mind that although there are 32 32-bit registers in the RISC-V register file, register `zero` is required by the ISA to return the value of zero at all times. [*To keep things simple, you do not need to draw out each of the 32 registers and corresponding control signals. Instead I recommend simplifying the structure as `zero, 1, 2, ..., 31`.*]

i. Based on your answer register file schematic, fully implement the RISC-V register file and create a testbench for it. Use QuestaSim to test your design to make sure it is working as expected. [*This should be mostly structural code. For register `zero`, you can set a signal's value to all '0's using the somewhat counterintuitive value <= (others => '0'); VHDL assignment statement. An alternative strategy is to set the i_RST port to '1' at all times.*]

4 My First RISC-V Datapath

Given the components you have already created in this and the previous week's lab, you can now implement the core datapath of a RISC-V-like processor. Provide your solution (VHDL code, simulation waveforms) in a folder `MyFirstRISCVDatapath/`.

a. Until we have a memory module, the only way to store any non-zero values in the register file is to make use of immediate-type arithmetic instructions. Add an additional 32-bit 2:1

multiplexer to the second input of your adder/subtractor design, using a new control signal called `ALUSrc`. Your design should now behave according to the following control table [note that this is not strictly correct RISC-V, which does not need or have a `subi` instruction]:

$nAdd\_Sub$	$ALUSrc$	$Operation$
0	0	$C \leftarrow A + B$
0	1	$C \leftarrow A +$ immediate
1	0	$C \leftarrow A - B$
1	1	$C \leftarrow A -$ immediate

b. Draw a symbol for this RISC-V-like datapath. [*This is the second step in the 381 design cycle – ports. Think about all of the signals you will need: control input signals, data input signals, clock, reset, etc.*]

c. Draw a schematic of the simplified RISC-V processor datapath consisting only of the component described in part a and the register file from part 1.

d. Implement this simplified RISC-V-like datapath using structural VHDL. Create a VHDL testbench to demonstrate that your datapath can support the following code. Include in your report waveform screenshots that demonstrate your properly functioning design. Annotate what the final register file state should be. [*You do not have to assemble these instructions into their proper RISC-V machine language equivalents. Instead, determine what values inputs to your design would correspond to for these instructions. For example, for the first instruction, `rs1=zero`, `rs2=x`, `rd=1`, `nAddSub='0'`, `ALUSrc='1'`, `regWrite='1'`. Each instruction should complete within a single cycle. When writing your testbench, it can be helpful to change your inputs on the non-active edge of the clock to clearly show which cycle the datapath is operating in.*]

```
addi 1,  zero,  1     # Place "1" in 1
addi 2,  zero,  2     # Place "2" in 2
addi 3,  zero,  3     # Place "3" in 3
addi 4, zero, 4       # Place "4" in 4
addi 5, zero, 5       # Place "5" in 5
addi 6, zero, 6       # Place "6" in 6
addi 7, zero, 7       # Place "7" in 7
addi 8, zero, 8       # Place "8" in 8
addi 9, zero, 9       # Place "9" in 9
addi 10, zero, 10     # Place "10" in 10
add 11, 1, 2          # 11 = 1 + 2
sub 12, 11, 3         # 12 = 11 - 3
add 13, 12, 4         # 13 = 12 + 4
sub 14, 13, 5         # 14 = 13 - 5
add 15, 14, 6         # 15 = 14 + 6
sub 16, 15, 7         # 16 = 15 - 7
add 17, 16, 8         # 17 = 16 + 8
sub 18, 17, 9         # 18 = 17 - 9
add 19, 18, 10        # 19 = 18 + 10
addi 20, zero, -35    # Place "-35" in 20
add  21, 19, 20   # 21 = 19 + 20
```

5 Memory

Memory. The way memory structures are designed is typically very technology-dependent, and consequently while there are simple methods for implementing generic (behavioral) memories in VHDL, for a specific memory design most computer architects either follow technology-dependent coding styles so that synthesis tools can infer the underlying structure, or they select and configure a pre-designed soft Intellectual Property (IP) core. For our RISC-V instruction and data memories we will follow the latter approach. Provide your solution to this problem (VHDL code, simulation waveforms) in a folder called '`Memory/`.

a. During this lab you will use a memory IP that is similar to one synthesizable by Quartus for the Cyclone IV. As you will find out, even though the majority of the code is provided for you, working with IP is not always a trivial task. The first issue is that this module comes with limited documentation (unfortunate, but the price was too good to pass up). Read through the `mem.vhd` file, and based on your understanding of the VHDL implementation, provide a 2-3 sentence description of each of the individual ports (both generic and regular). For example, what is port `''q''` for and when does it become valid.

b. The `dmem.hex` file provided as part of this lab contains only a few simple memory initialization values. Before continuing, modify this file to contain the following 32- bit values first starting at address 0x0 in memory: -1, 2, -3, 4, 5, 6, -7, -8, 9, -10. [*The included `dmem.hex` file contains some hints, but be aware that the `mem.vhd` RAM module we have is word-addressable and not byte-addressable as RISC-V expects.*]

c. Create a VHDL testbench `tb_dmem.vhd` that:

   (a) instantiates the `mem` module as datamemory (labeled as `dmem`)

   (b) reads the initial 10 values stored in memory.

   (c) writes those same values back to consecutive locations in memory starting at 0x100, and then.

   (d) reads those new values back to ensure they were written properly.

As always, include waveform screenshots in your report PDF. [*Note that in order to load the `dmem.hex` file, you will need to start a simulation in `QuestaSim` and use the `tcl command mem load -infile dmem.hex -format hex /tb_dmem/dmem/ram`. This assumes you are currently in the same directory as `dmem.hex`, and you've labeled your memory `dmem` and your testbench `tb_dmem`.*]

6 Bit-width Extenders

Several instructions in the RISC-V ISA implicitly require sign or zero extension of 16-bit values such that these values can be operated on using a 32-bit ALU and/or stored in a 32-bit register file. Provide your solution to this problem (VHDL code, simulation waveforms) in a folder called `Extenders/`. [*This is "more of the same" from what you've been doing both in terms of VHDL, QuestaSim simulation, and should not take you more than 30 minutes.*]

a. What are the RISC-V instructions that require some value to be sign extended? What are the RISC-V instructions that require some value to be zero extended? [*We've covered this in class, but if you are unsure, the answers can be found in P&H chapter 2.*]

b. Given your solution to part a), what are the different 16-bit to 32-bit "extender" components that would be required by a RISC-V processor implementation? *[Note that to implement a full RISC-V processor there would also be 8-bit to 32-bit extender; however, we will use software to emulate this behavior in our term project so you are not required to implement these.]*

c. Implement the RISC-V extender components using any VHDL style you prefer. These can be implemented as two separate entities with no control signals, or as one separate entity with a control bit specifying "sign" versus "zero" extension. *[This is a relatively simple task using either "`for/generate`" or the "`others`" VHDL construct learned in previous lab parts. There are many different ways to do this; if your code looks complex for this part you are probably overthinking the problem.]*

d. Use QuestaSim to test your extender components to make sure they are working as expected, and include waveform screenshots in your report PDF.

7 My Second RISC-V Datapath

Given the components you have created or worked with in this week's lab, you can now add support for load and store instructions to the RISC-V-like datapath from the previous week's lab. Provide your solution to this problem (VHDL code, simulation waveforms) in a folder called `MySecondRISC-VDatapath/`.

a. Incorporating a data memory module into the RISC-V processor requires additional control signals associated with store and load instructions. In order to support the two word-granularity integer load and store instructions as listed in P&H chapter 2, what control signals will need to be added to the simple processor from part 3? How do these control signals correspond to the ports on the `mem.vhd` component analyzed in part 4?

b. Draw a schematic of a simplified RISC-V processor consisting only of multiplexers from Lab 1, the base components used in part 3, the extender component described in part 5, and the data memory from part 4. This design should only require inputs for a clock, the control signals, the three register address ports, and a **16-bit immediate value.**

c. Implement this simplified RISC-V processor using structural VHDL. Create a VHDL testbench to demonstrate that your design can generate the correct value when "running" the following code. Include the waveform in your report. *[Similar to your previous datapath testbench, you do not have to assemble these instructions into their proper RISC-V machine language equivalents. Instead, determine what values the control inputs to your datapath design would correspond to for these instructions.]*

```
addi 25, zero, 0        # Load &A into 25
addi 26, zero, 256      # Load &B into 26
lw   1,  0(25)          # Load A[0] into 1
lw   2,  4(25)          # Load A[1] into 2
add  1,  1, 2           # 1 = 1 + 2
sw   1,  0(26)          # Store 1 into B[0]
lw 2, 8(25)             # Load A[2] into 2
add 1, 1, 2             # 1 = 1 + 2
sw 1, 4(26)             # Store 1 into B[1]
lw 2, 12(25)            # Load A[3] into 2
add 1, 1, 2             # 1 = 1 + 2
sw 1, 8(26)             # Store 1 into B[2]
```

```
lw 2, 16(25)            # Load A[4] into 2
add 1, 1, 2             # 1 = 1 + 2
sw 1, 12(26)            # Store 1 into B[3]
lw 2, 20(25)            # Load A[5] into 2
add 1, 1, 2             # 1 = 1 + 2
sw 1, 16(26)            # Store 1 into B[4]
lw 2, 24(25)            # Load A[6] into 2
add 1, 1, 2             # 1 = 1 + 2
addi 27, zero, 512      # Load &B[64] into 27
sw 1, -4(27)            # Store 1 into B[63]
sw   1, -4(27)          # Store 1 into B[63]
```

8 Submission Instructions 🗂️

Below is a list of the required turn-in components for the lab to be submitted to Canvas. Naming conventions for each file are also provided below.

8.1 Turn-In Checklist

- Complete the lab report using template (`Lab2_report.doc` in `Lab2.zip`).
- Create a zip file `Lab2_submit.zip` with the zip file, you must also submit a pdf report.

8.2 File Structure

```
Canvas
├── Lab2_report.pdf
└── Lab2_submit.zip/tgz
    ├── RegFile/
    ├── MyFirstRISC-VDatapath/
    ├── Memory/
    ├── Extenders/
    ├── MySecondRISC-VDatapath/
    └── Lab2_report.doc
```

Credit: Portions of this lab were originally created by Dr. Joe Zambreno.