Lab 5 Processor

Deadlines & Grading

▪ **Prelab A:**	Wednesday, Mar 28	8 points, **in groups of 2**
▪ **Lab Session A:**	Monday, Apr 9 / Tuesday, Apr 10	12 points, **in groups of 2**
▪ **Prelab B:**	Wednesday, Apr 11	10 points, **in groups of 2**
▪ **Lab Session B:**	Monday, Apr 23 / Tuesday, Apr 24	16 points, **in groups of 2**
▪ **Prelab C:**	Wednesday, Apr 25	6 points, **in groups of 2**
▪ **Lab Session C:**	Monday, April 30 / Tuesday, May 1	12 points, **in groups of 2**
▪ **Report**:	Monday, May 7 / Tuesday, May 8	16 points, **in groups of 2**

Section I: Overview

Lab 4 demonstrated how finite state machines provide a method of designing complex digital systems that are catered to a specific task. However, making modifications to the task (as you did for Part B) requires us to change the hardware design. Changing a chip in the real world is an expensive proposition. Thus, it is desirable to create a device whose functionality is easily altered without modifying the underlying hardware. A *microprocessor* provides these capabilities: a fixed sequential circuit that is general-purpose, which executes a series of simple tasks known as *instructions* to complete the desired task, and permits the execution of a different task by easily changing the series of instructions (a *program*) that the processor executes.

This lab will have you implement a processor from scratch, combining several concepts you have studied throughout this semester. First, you will build a combinational *arithmetic logic unit* (ALU). You will use this ALU in designing a processor that can execute sequential code and store meaningful results. After this basic processor has been created, you will then extend it to include branching and halting logic, which allows you to use loops in order to execute much more sophisticated programs.

Once complete, our microprocessor will be able to run a large number of different programs. For this lab, we will be providing you with several programs to execute. The final program is a heart rate monitor. The heart rate monitor observes an input that changes over time, and records the number of observed “pulses” within a certain window. This pulse count is then translated to an approximate heart rate, which is displayed to the user. While the lab uses a switch to simulate the heart pulse, this switch could easily be replaced with a chest probe that converts a sensed heartbeat into an electrical signal.

As you’ll see, the finished microprocessor can execute all of these programs without having to modify the processor circuitry. This is typically how advanced electronics are made in the real world – a microprocessor (for less complex applications, a simple processor known as a *microcontroller*) is the "brain" of the device, and runs software programs that orchestrate the entire device's operation. The ability to reuse the processor obviates the need to design a new chip for every new type of device manufactured, and also allows for updatability.

Section II: Part A

A. ALU Design & Operation

The first step to your processor design is to build an 8-bit ALU, as shown in Figure 1. A three-bit input **OP** indicates which operation the ALU should perform. Table A. shows the operations you must implement.

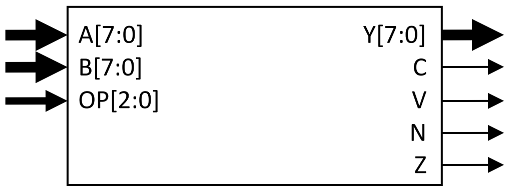

Figure 1. ALU block diagram.

Your ALU will also take in two 8-bit inputs **A** and **B**, and will output an 8-bit value **Y**. There is a 1-bit output, the carry out **C**. For our shift operations, we will use **C** to output the shift-out value, as the shifter does not carry out anything.

Three status (*flag*) bits provide information about the output **Y**. The **N** bit tells us if **Y** is negative, the **Z** bit tells us if **Y** is equal to zero, and the **V** bit informs us when overflow occurs. The expected values for all of these outputs are shown in Table A.

OP	INSTR	Y	C	V	N	Z
000	ADD	(A + B)[7:0]	(A + B)[8]	1 iff overflow	1 iff Y < 0	1 iff Y = 0
001	SUB	(A + (~B) + 1)[7:0]	(A + (~B) + 1)[8]	1 iff overflow	1 iff Y < 0	1 iff Y = 0
010	SRA	{A[7], A[7:1]}	A[0]	0	1 iff Y < 0	1 iff Y = 0
011	SRL	{0, A[7:1]}	A[0]	0	1 iff Y < 0	1 iff Y = 0
100	SLL	{A[6:0], 0}	A[7]	0	1 iff Y < 0	1 iff Y = 0
101	AND	A AND B	0	0	1 iff Y < 0	1 iff Y = 0
110	OR	A OR B	0	0	1 iff Y < 0	1 iff Y = 0

Table A. ALU operations that must be implemented. Note that ~B represents NOT B.

The ALU functions will be split between four modules that will be placed inside the ALU. Your task is to first implement each module, then compose them inside **alu.v**. **You should not have anything other than module instantiations and wire connections in alu.v**. Figure 2 shows how the ALU operations are split among three modules.

The interface of the first module, **logical**, can be found inside **logical.v**. You will implement the AND and OR instructions here. These instructions are bitwise, so you will AND or OR each bit in parallel. The one-bit input **OP** indicates if the desired operation is an AND (1) or OR (0).

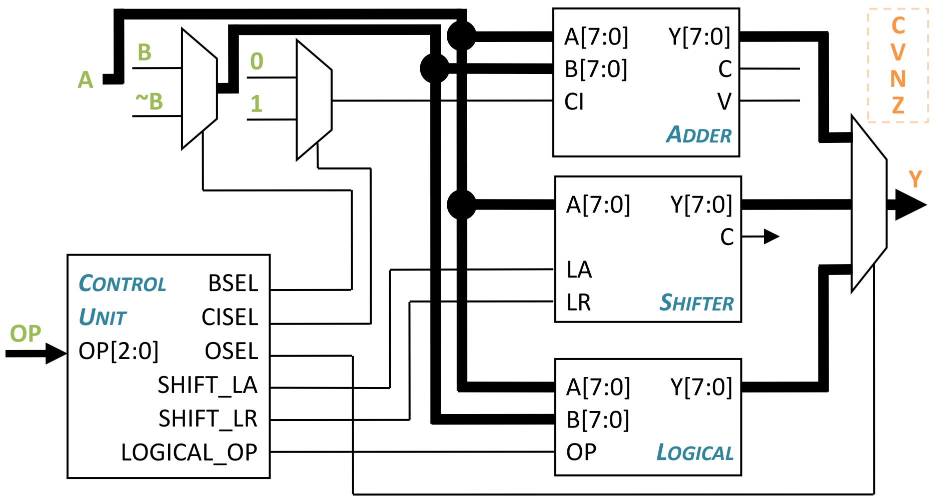

Figure 2. Incomplete ALU datapath. Inputs shown in green; outputs in orange. Not all bit widths are shown.

The interface of the second module, **shifter**, can be found inside **shifter.v**. The SRA, SRL, and SLL instructions will be implemented here. Figure 3 illustrates how the SRL instruction works, as it logically shifts in a 0 (internal signal **SI**). The shifter takes two additional one-bit inputs – **LA**, which tells the shifter if the desired instruction is a logical shift (0) or arithmetic shift (1), and **LR**, which indicates the direction of the shift (0 = left). (Note that there is no arithmetic left shift instruction, so when **LR** = 0, **LA** is ignored).

The interface of the last module, **adder**, can be found inside **adder.v**. Here you will implement the ADD and SUB instructions. These instructions combine two eight-bit numbers **A** and **B** to output a nine-bit number as described in Table A. Bits 7 through 0 of this nine-bit number are used to produce the value **Y**, while bit 8, the MSB, is used for the output **C** (carry out). Inputs **A** and **B**, and output **Y**, are two's complement numbers. The overflow detection logic inside the **adder** module sets **V**=1 whenever overflow occurs.

Note that there are two multiplexers on the left of Figure 2 that control what values go into the **adder** and **logical** modules. One multiplexer switches between **B** and **~B**, while the other switches between 0 and 1. A third multiplexer to the right determines which modules you want to read the output from.

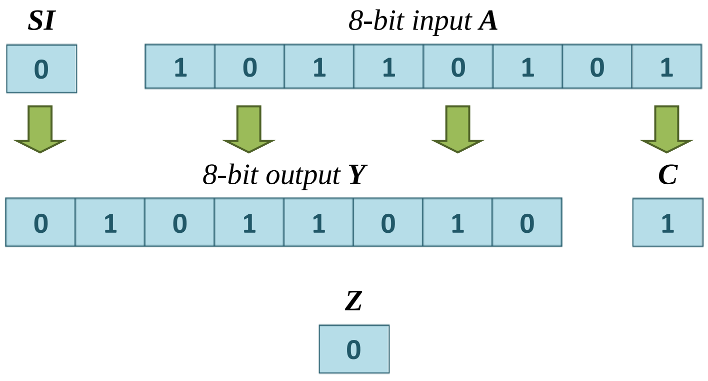

Figure 3. Visualization of the SRL operation (**LA** = 0, **LR** = 1).

The three multiplexers select their values to send by using the select signals **BSEL**, **CISEL**, and **OSEL**. These signals are generated by a *control unit*, shown in Figure 2. This unit reads in the **OP** input, and determines how to set the muxes. You should implement this in its own module, called **control**. If you would like to use other control signals within your ALU, feel free to implement them in here as well. This separation, between the control logic and the data itself, is an important concept in designing more complex circuits and processors, as we will see later.

As an example, let's say that **OP** receives 3'b001 as an input. In order to implement the SUB instruction, the control unit should set **BSEL** to pass **~B** to the adder, **CISEL** to pass 1 to the adder, and **OSEL** to pass the output of the adder to wire **Y**. Each computation module (**adder**, **logical**, and **shifter**) will *always* be outputting an answer. It is the job of the **OSEL** multiplexer to pick which of the three outputs is the correct one. Other control signals beyond what we have illustrated (which can be sent into the modules) might be useful.

Your control unit should not pass **OP** directly to the other modules. Instead, it should read the value of **OP** and determine what information the modules need. For example, the shifter needs the **LA** signal to tell it which direction to shift, which should only be a single bit. The control unit must implement logic that uses the value of **OP** to output the correct value of **LA**, which is then connected by a wire to the shifter. To get you started, we have provided some of the code inside **control.v**. You are free to design any of the missing elements from the ALU in Figure 2 as you wish, but your final circuit must satisfy all of the requirements we have stated here. For this lab there is no need to use separate MUX modules, you can simply use the ternary operator, "?".

NOTE 1 You are only allowed to use logical (`&`, `|`, `~`, `^`), equality (`==`), and conditional (`?`) operators for all of your Verilog modules in this part of the lab. Specifically, **the +, −, <, >, <<, and >> operators are forbidden** for all modules inside the ALU.

B. Prelab Deliverables

ASSIGNMENT 1 For the prelab, implement the **adder** module and configure the **alu** to perform ADD.

1. Download **lab5a_release.zip** from CMS, unzip its contents, and open the project inside Quartus. If you open **alu.v**, you will see empty modules already inserted for you.
2. Edit the file **adder.v** to add your adder functionality.
3. Edit the file **alu.v** to instantiate the adder as well as to generate the **Y**, **C**, **V**, **N**, and **Z** outputs correctly. You do not need **control** or muxes as the alu will only perform ADD for the prelab.
4. For the prelab only you should connect the **A**, **B**, and **Y** signals in the **alu** module directly to the adder (without muxes).
5. We have provided a testbench with only a trivial test case. You should add your own test cases to verify your adder and document each test case in your **readme** file.

DELIVERABLE: prelab5a.zip

Submit your Verilog code and testbench, and a text file **readme.txt**, all as a ZIP file named **prelab5a.zip**. Please use the exact same names of the signals we have specified. The **readme.txt** file should include your and your partner's names, your NetIDs, and any pertinent information for the graders. Submit the ZIP file to CMS (http://cms.csuglab.cornell.edu).

NOTE 2 If you have created any sub-modules that you are using inside any of the required files, please remember to include the **.v** files for those as well. When in doubt, just submit all **.v** files inside your directory. You will get zero credit for missing files.

C. In-Lab Assignments

ASSIGNMENT 2 Create the **shifter**, **logical**, and **control** modules inside the appropriate Verilog file. You may also create other modules as needed.

ASSIGNMENT 3 Create the multiplexers and connect all the modules together.

D. Top-Level Assembly & ModelSim Testing

ASSIGNMENT 4 Wire together your modules in **alu.v**, adding any necessary logic needed to make your ALU work. This should include the three multiplexers and any associated logic gates. **Check for inferred latches!**

ASSIGNMENT 5 Open the test bench file **alu_test.v**. This time, we have only provided a couple of test cases. Add your own test cases and include at least one test case for each operation. **The lab TAs will check for proper test cases.**

ASSIGNMENT 6 Run ModelSim to verify that your ALU is working correctly. Refer to *Tutorial C* on how to examine internal module values.

E. Preparing Your ALU for the DE0 Board

For this lab, the input pins have already been assigned for you. They are mapped as follows:

- Inputs **A** and **B** are mapped to toggle switches **SW7** (the MSB) through **SW0**. **SW8** selects whether the switches control **A** or **B**. Press **KEY3** to write the value on **SW7-SW0** to the selected input.
- The button **KEY0** cycles through different values of **OP**. When **OP** reaches 110_2, the next button press will reset it to 000_2.

Circuit outputs are assigned as follows:

- Input **A** is displayed on **HEX5** (MSB) and **HEX4**, as a hexadecimal number.
- Input **B** is displayed on **HEX3** (MSB) and **HEX2**, as a hexadecimal number.
- Output **Y** is displayed on **HEX1** (MSB) and **HEX0**, as a hexadecimal number.
- Output **C** is displayed on **LEDR7**.
- Output **V** is displayed on **LEDR6**.
- Output **N** is displayed on **LEDR5**.
- Output **Z** is displayed on **LEDR4**.
- Input **OP** is displayed on **LEDR2** (MSB) through **LEDR0**.

If you want to check where the pin assignments are located:

1. Open the **lab5a.qpf** file.
2. Go to *Assignments* → *Devices…* and select the Cyclone V family. Under *Available Devices*, select *Specific device selected*, and choose the *5CEBA4F23C7*. Click *OK*.
3. Check pin assignments by going to *Assignment* → *Assignment Editor*. This will show you a list of all pins. For this lab, the pins should have already been set for you.

Compile your design. This will create a file **ECE2300.sof**, which we use to program the FPGA.

F. Testing Your Design

1. Take the DE0-CV board and connect one end of the USB cable to the leftmost port (USB Blaster Port) and other end to your system.
2. Make sure that the red power switch on the left is pushed in to turn the board on. You will see the board light up and various things displayed on the 7-segment displays and the LEDs. This indicates the board is ready to be programed.
3. When you plug in the cable, if you are on Windows, it should tell you that the **Altera USB-Blaster** device was installed properly while if you are on Quartus-VM, you need to go the Devices -> USB Devices in the Virtual-Box menu and select Altera USB-Blaster device.
4. Make sure the **RUN/PROG** switch on the left of the board is set to the **RUN** position.
5. Inside Quartus, go to *Tools* → *Programmer*. Under *Hardware Setup*, select *USB-Blaster*. If you only see *No Hardware* you likely have an issue with the USB-Blaster driver.
6. Inside the programmer tool window, you will see your ***.sof** file listed. Make sure that the check box under *Program/Configure* is checked, and click *Start*. Once this completes, your design will now be downloaded to your FPGA.

Section III: Part B – Basic Processor

A. Basic Processor Overview

For this part of the lab, you will implement a single-cycle microprocessor, based on the design that we have studied in class. This processor will read its instructions from an instruction RAM and will use a data RAM (both of which will be provided) to read and write register data. You will implement the functionality for thirteen instructions in this part of the lab. In Part C, you will eventually add in branch and jump support. For now, you will be implementing ALU and memory instructions.

B. Microprocessor Design

Figure 4 shows the datapath that you will need to implement for this lab. All of the control signals are produced by the *decoder* based on the current instruction. Your processor will initialize on **RESET** = 1 by setting **PC**, the *program counter*, to 0. Every cycle, when **RESET** is low, the processor will use **PC** to look up the instruction stored at that address inside the *instruction RAM*. This instruction is decoded by the decoder into the various control signals that will perform the computation. These control signals can read/write the *register file*, perform an ALU operation, or read/write the *data RAM*. At each positive clock edge, the **PC** will be advanced, and a new instruction will be read.

The instruction and data RAM modules are provided for you (**lab5dram.v** and **lab5iram.v**). You are responsible for implementing the rest of the processor inside **cpu.v**. Details of how the CPU should look is provided below.

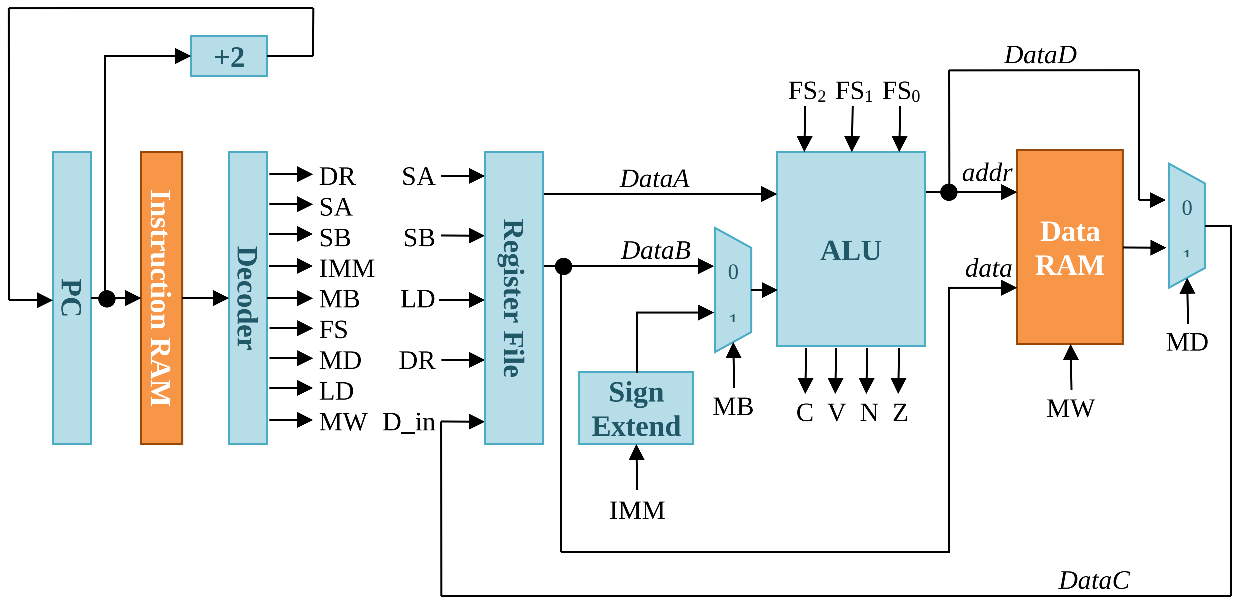

Figure 4. Processor datapath. Boxes in blue must be implemented, while boxes in orange are provided for you. All blue boxes are part of the **CPU** module, while the orange boxes are external modules.

Control Signals

Below is a description of the control signals produced by the decoder. A control signal may not be required for certain instructions, in which case it can be set to a don't care value.

▪ **DR**:	register file write address
▪ **SA**:	register file read address A
▪ **SB**:	register file read address B
▪ **IMM**:	6-bit signed immediate
▪ **MB**:	ALU input mux select (0=DataB, 1=IMM)
▪ **FS**:	ALU function
▪ **MD**:	output mux select (0=ALU, 1=data ram)
▪ **LD**:	register file write enable
▪ **MW**:	data ram write enable

Design Hierarchy

The file **lab5.v** (provided for you) instantiates the CPU, instruction RAM, and data RAM. **Your task is to implement the other blocks inside cpu.v**. Different IRAMs containing different programs are provided, and you can change the program to run by changing the IRAM in **lab5.v**.

Register File

The register file inside your microprocessor should contain eight registers, each of which are eight bits wide. The signals **SA** and **SB** select which of the 8 registers will be output to **DataA** and **DataB** respectively. **LD** controls whether the register file will be written to, with **DR** specifying the write address and *D_in* the write data. On a reset, set all eight registers to 8'b0. **Look at how the RAMs are implemented** to see the best way to implement the register file.

Instruction Format & Operations

In this processor, we support two types of 16-bit instructions: *register-to-register* and *immediate*. You can see the fields that we will use for this in Figure 5. Each instruction has a 4-bit opcode **OP**, and has register fields that are 3 bits wide (**RS**, **RT**, and in the case of register-to-register, **RD**). The register-to-register instructions also include a 3-bit function field, **FUNCT**, which is sent to the ALU. Immediate instructions contain a 6-bit constant, **IMM**, instead.

REGISTER-TO-REGISTER

15 14 13 12	11 10 9	8 7 6	5 4 3	2 1 0
OP	**RS**	**RT**	**RD**	**FUNCT**

IMMEDIATE

15 14 13 12	11 10 9	8 7 6	5 4 3 2 1 0
OP	**RS**	**RT**	**IMM**

Figure 5. 16-bit instruction formats.

Real processors contain many instructions that perform various operations, known collectively as an instruction set. We will only be implementing a small subset of a full instruction set for our processor. Table B shows the instructions that you must implement for the processor in this part of the lab. As part of this, you must use the ALU that you implemented in Part A of the lab. Also, keep in mind that IMM must be sign extended, as shown in Figure 4.

OP	FUNCT	Instruction		Operation	Z	V	N
0000	000	NOP		all write enables = 0	N/A	N/A	N/A
0010	N/A	LB	RT, IMM(RS)	*RT = MEM[*RS + IMM]	N/A	N/A	N/A
0100	N/A	SB	RT, IMM(RS)	MEM[*RS + IMM] = *RT	N/A	N/A	N/A
0101	N/A	ADDI	RT, RS, IMM	{C, *RT} = *RS + IMM	1 iff RT = 0	1 iff overflow	1 iff RT < 0
0110	N/A	ANDI	RT, RS, IMM	*RT = *RS & IMM	1 iff RT = 0	0	1 iff RT < 0
0111	N/A	ORI	RT, RS, IMM	*RT = *RS | IMM	1 iff RT = 0	0	1 iff RT < 0
1111	000	ADD	RD, RS, RT	{C, *RD} = *RS + *RT	1 iff RD = 0	1 iff overflow	1 iff RD < 0
1111	001	SUB	RD, RS, RT	{C, *RD} = *RS – *RT	1 iff RD = 0	1 iff overflow	1 iff RD < 0
1111	010	SRA	RD, RS	{*RD, C} = {*RS[7], *RS}	1 iff RD = 0	0	1 iff RD < 0
1111	011	SRL	RD, RS	{*RD, C} = {0, *RS}	1 iff RD = 0	0	1 iff RD < 0
1111	100	SLL	RD, RS	{C, *RD} = {*RS, 0}	1 iff RD = 0	0	1 iff RD < 0
1111	101	AND	RD, RS, RT	*RD = *RS & *RT	1 iff RD = 0	0	1 iff RD < 0
1111	110	OR	RD, RS, RT	*RD = *RS | *RT	1 iff RD = 0	0	1 iff RD < 0

Table B. Processor operations to be implemented. Instructions with **OP** = 0000_2 or **OP** = 1111_2 use the register-to-register format, while all others use the immediate format. Under the *Operation* column, *RS refers to the value stored in register RS, *RT refers to the value stored in register RT, and *RD to the value stored in register RD.

Instruction RAM & Data RAM

The instruction RAM takes in an 8-bit address, **addr**, which selects the line that is output on the 16-bit **data** pin. The RAM is *byte-addressable* – every byte has its own row number – and our instructions are 16 bits wide (the first instruction is at address 0, the second at address 2, etc.). As a result, when we "increment" the **PC**, we actually need to add 2 to go to the next instruction.

The data RAM has a **clock** input, an 8-bit address input **addr**, an 8-bit input **data**, a 1-bit *write enable* input **mw**, and an 8-bit data output **q**. When we read from memory, changing **addr** loads the data we request on **q**. When writing, we must set **mw** high, provide the bits to write on **data**, and make sure the data is held until the positive edge of the next clock cycle. When the processor is started, most of the RAM is uninitialized (and shows up as x's in RTL simulation).

In order to communicate with the RAMs, you must assert the proper signals, as shown in the datapath in Figure 4. The instruction RAM accepts an address from the program counter and outputs the data at that address to the decoder. The data RAM accepts an address from the ALU and accepts data from the register file (to implement writes) and outputs data at the given address (to implement reads).

Clock & I/O Pins

There are two input switches wired to the circuit that we care about. The one-bit input **RESET** is used to synchronously reset the **PC** of the processor to 0 while **RESET** = 1. As long as

RESET = 1, the processor will not execute anything (alternatively, it can execute the NOP instruction). The second input, **CLK_SEL**, allows us to choose what frequency to run the processor at. We input a 50 MHz clock, **CLK50**, to the system. We then pass this through the **var_clk** module, which generates a slower **CLK** signal for our use. We will keep **CLK_SEL** set to 1, both for simulation and running on the DE0 board. This will run the processor at 10 MHz.

Since the data RAM is inside the processor, we aren't able to observe its contents from outside the FPGA. To get around this, we have connected the last four addresses (252 to 255) in memory to output pins **IOD**, **IOE**, **IOF**, and **IOG**. These are known as *memory-mapped* outputs, since they always output the value of a certain location. We take these outputs and connect them to either LEDs or seven-segment displays on the DE0 board. We will be using these outputs to check if your processors are working properly.

We have also mapped the input switches to memory locations 249 and 250. These *memory-mapped inputs* are called **IOA** and **IOB**. Essentially, reading from memory address 249 and 250 in the data RAM will provide your program with the switch values in **IOA** and **IOB**, respectively. Section III-E has details on how the switches are connected to **IOA** and **IOB**. Similarly, **KEY0** and **KEY1** are mapped to the two lowest bits of **IOC**, at memory location 251.

In order to facilitate test benches while you are simulating your circuits, we have also provided the following output pins:

- **PC**: the current value of the **PC** register,
- **NextPC**: the value that will be saved to the **PC** register at the end of the clock cycle,
- **Iin**: the 16-bit output of the instruction RAM,
- **Din**: the 8-bit output of the data RAM,
- **DataA**: the content of the register specified by the **RS** field of the current instruction,
- **DataB**: the content of the register specified by the **RT** field of the current instruction,
- **DataC**: the 8-bit input **D_in** to the register file, and
- **DataD**: the 8-bit output from the ALU.

Provided Programs and Testbench

We are providing you with three example programs which you can use to test your processor. We have placed them in three separate modules, so you can easily interchange them:

- **lab5iram**: This program stores a sequence of numbers between 0 and 5 to memory address 255.
- **lab5iram1A**: This program takes the inputs from **IOA** and **IOB**, XORs the two 8-bit values together, and then counts the number of ones and zeros in the result. The number of ones is stored in memory address 255, and the number of zeros is stored to address 254.
- **lab5iram1B**: This program reads the lower four bits of **IOA** and **IOB**, and multiplies them together, storing the result in memory address 255.

The file **lab5_test.v** also contains a testbench which compares your processor to a *functional processor model* as it executes the chosen program. The testbench cannot test everything (it cannot look inside the regfile or memory, for example). You may write your own programs to use with the testbench.

C. Prelab Deliverables

ASSIGNMENT 7

For the prelab, implement the **decoder** module. We've provided a **decoder** project and starting testbench. You must complete the module and add your own tests.

1. Download **lab5.zip** from CMS and unzip its contents. Leave the **lab5** project for now and work in the **decoder** project.
2. Implement the decoder logic. You **cannot modify** the ports we provided for the submission. You must determine whether each port is an input or output.
3. Add test cases to the testbench (**decoder_test.v**). Describe your test cases in the **readme** file. **Proper testing will be part of your grade**.

NOTE 3

Your Verilog should contain no **inferred latches**. Inferred latches are `reg` variables inside combinational `always` blocks that aren't always written to, and often cause serious errors. You should for inferred latch messages in the **Warning** tab in Quartus when you compile your code.

DELIVERABLE: prelab5b.zip

Submit your Verilog code and testbench, and a text file **readme.txt**, all as a ZIP file named **prelab5b.zip**. Please use the exact same names of the signals we have specified. The **readme.txt** file should include your and your partner's names, your NetIDs, and any pertinent information for the graders. Submit the ZIP file to CMS (http://cms.csuglab.cornell.edu).

NOTE 4

If you have created any sub-modules that you are using inside any of the required files, please remember to include the **.v** files for those as well. When in doubt, just submit all **.v** files inside your directory. You will get zero credit for missing files.

D. In-Lab Assignments

ASSIGNMENT 8

Create the other modules in the Figure 4, then instantiate and wire them up in **cpu.v**. Feel free to modify **lab5iram.v** to test out all of the required instructions.

ASSIGNMENT 9

Use the other instruction RAM files provided inside **lab5.zip** to test different programs on your processor.

1. Inside **lab5.v**, find the instantiation of the **lab5iram** module. Change this to use either module **lab5iram1A** or **lab5iram1B**.
2. Re-compile your project, and then simulate it.

NOTE 5

Do not modify the files **lab5.v**, **lab5dram.v**, **var_clk.v**, or **hex_to_seven_seg.v**. You can create as many new modules as you please, but make sure each modules gets its own file.

E. Preparing Your Processor for the DE0 Board

For this lab, the input pins have already been assigned for you. They are mapped as follows:

- The **RESET** input uses the debounced pushbutton **FPGA_RESET**.
- The **CLK_SEL** input uses toggle switch **SW9**. Set this to 1 to execute the processor at 10 MHz (which we will usually do), and to 0 for 1 Hz execution.
- Inputs **IOA** (memory location 249) and **IOB** (memory location 250) are mapped to toggle switches **SW7** (the MSB) through **SW0**. **SW8** selects whether the switches control **IOA** or **IOB**. Press **KEY3** to write the value on **SW7**-**SW0** to the selected input.
- Input **IOC** (memory location 251) contains the current states of **KEY1** and **KEY0** in its two lowest bits. The remaining bits are all 0.

Circuit outputs are assigned as follows:

- Output **IOD** (memory location 252) is shown on **LEDR7** (MSB) through **LEDR0**.
- Output **IOE** (memory location 253) is shown on **HEX1** and **HEX0**.
- Output **IOF** (memory location 254) is shown on **HEX3** and **HEX2**.
- Output **IOG** (memory location 255) is shown on **HEX5** and **HEX4**.

If you want to check where the pin assignments are located:

1. Open the **lab5a.qpf** file.
2. Go to *Assignments → Devices…* and select the Cyclone V family. Under *Available Devices*, select *Specific device selected*, and choose the *5CEBA4F23C7*. Click *OK*.
3. Check pin assignments by going to *Assignment → Assignment Editor*. This will show you a list of all pins. For this lab, the pins should have already been set for you.

Compile your design. This will create a file **ECE2300.sof**, which we use to program the FPGA.

F. Testing Your Design

1. Take the DE0-CV board and connect one end of the USB cable to the leftmost port (USB Blaster Port) and other end to your system.
2. Make sure that the red power switch on the left is pushed in to turn the board on. You will see the board light up and various things displayed on the 7-segment displays and the LEDs. This indicates the board is ready to be programed.
3. When you plug in the cable, if you are on Windows, it should tell you that the **Altera USB-Blaster** device was installed properly while if you are on Quartus-VM, you need to go the Devices -> USB Devices in the Virtual-Box menu and select Altera USB-Blaster device.
4. Make sure the **RUN/PROG** switch on the left of the board is set to the **RUN** position.
5. Inside Quartus, go to *Tools → Programmer*. Under *Hardware Setup*, select *USB-Blaster*. If you only see *No Hardware* you likely have an issue with the USB-Blaster driver.
6. Inside the programmer tool window, you will see your ***.sof** file listed. Make sure that the check box under *Program/Configure* is checked, and click *Start*. Once this completes, your design will now be downloaded to your FPGA.

Section IV: Part C – Full Processor

A. Extending the Processor

The single-cycle processor that you implemented in Part B of this lab can only process relatively simplistic code. This is because we can only tell the program to start at PC = 0, and it keeps running forever, executing all of the instructions in order. For this part, you will add in an instruction that can pause the processor, and later we will also add in support for branches. Once this is complete, you will then run a program for a heart rate monitor on your processor.

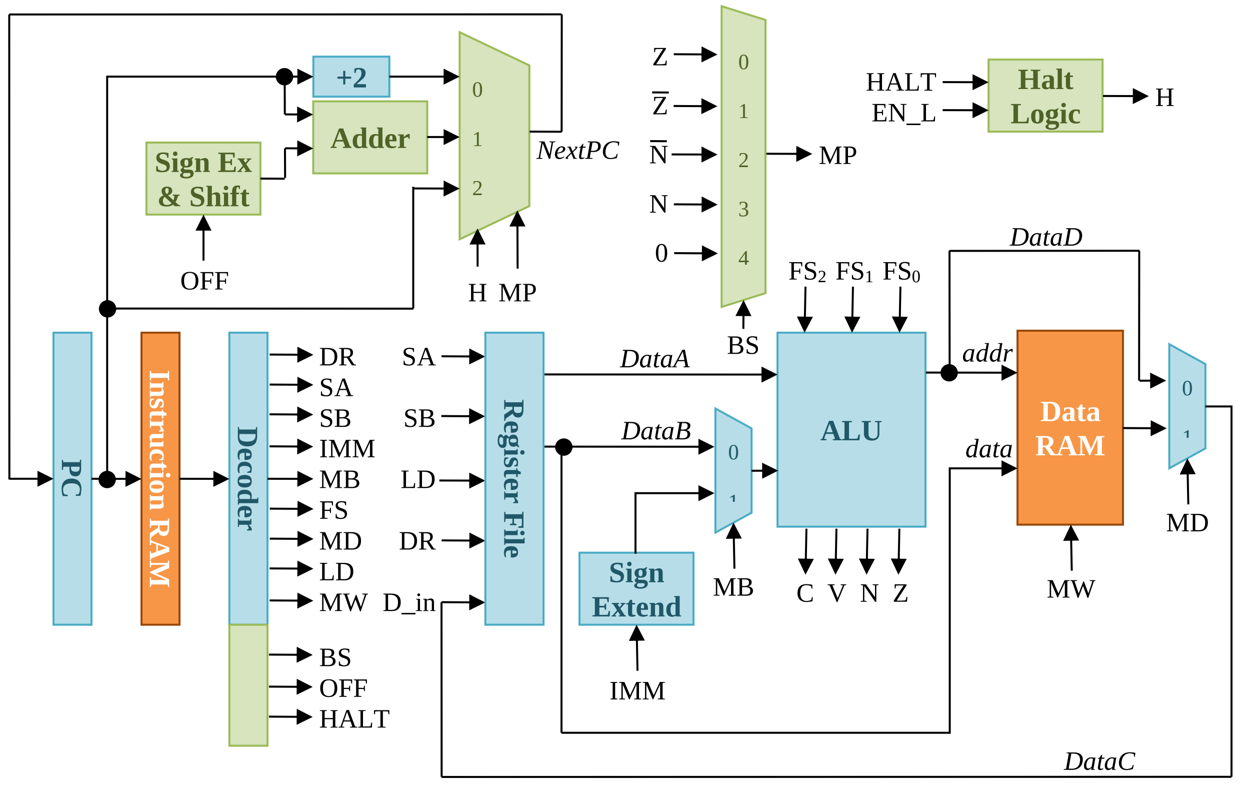

Figure 6. Extended processor datapath. Boxes in green are new modules that you must implement. Boxes in blue were implemented in Part B, but may need modification, while boxes in orange are provided for you. The green boxes are still part of the **CPU** module.

B. Design Updates

Figure 6 shows the datapath that you will need to implement for this lab, with all new additions in green. If you have not done so already, please make sure that your processor from the first part of the lab is working properly before adding in any modifications for this lab. This will make debugging a lot more straight-forward for you. We will again provide instruction RAMs and data RAMs for you.

OP	FUNCT	Instruction	Operation	Z	V	N
0000	001	HALT	all write enables = 0, stall until EN_L = 0	*N/A*	*N/A*	*N/A*
1000	*N/A*	BEQ RT, RS, IMM	if(*RS == *RT) PC += {IMM, 0}	1 iff *RS – *RT = 0	*N/A*	*N/A*
1001	*N/A*	BNE RT, RS, IMM	if(*RS != *RT) PC += {IMM, 0}	1 iff *RS – *RT = 0	*N/A*	*N/A*
1010	*N/A*	BGEZ RS, IMM	if(*RS ≥ 0) PC += {IMM, 0}	*N/A*	*N/A*	1 iff *RS < 0
1011	*N/A*	BLTZ RS, IMM	if(*RS < 0) PC += {IMM, 0}	*N/A*	*N/A*	1 iff *RS < 0

Table C. New processor operations that must be implemented. The **HALT** instruction uses the register-to-register format, while all other instructions use the immediate format. Under the *Operation* column, *RS refers to the value stored in register RS, *RT refers to the value stored in register RT, and *RD to the value stored in register RD.

Halting the Processor

You will be responsible for adding in a HALT instruction to the processor, as shown in Table C. The behavior of HALT is similar to the NOP instruction. However, there is one key difference. Once the processor reaches the HALT instruction, it does not advance the **PC** until it detects a transition from 1 to 0 on the **EN_L** signal. **EN_L** is an active-low signal that tells the processor it can continue executing. We will detect changes on **EN_L** by comparing the current and previous values, and update the previous value on each positive clock edge.

I/O Pins

In addition to our previous I/O, we will also use the debounced pushbutton switch **KEY2** for the active low input **EN_L**, which will tell the processor that it can continue to the next instruction if it is currently on a HALT instruction. You will need to add a new input to your **cpu** module, called **EN_L** (note that it is in all capital letters), that takes in this signal from the input port (which already exists).

As was the case before, we still have our memory-mapped input pins **IOA** and **IOB**, as well as our memory-mapped outputs **IOC** through **IOH**.

ASSIGNMENT 10 Update your processor to include the HALT instruction. Update the instruction RAM file (**lab5iram.v**) and use ModelSim to verify that HALT works properly. **Proper testing will be part of your grade**.

DELIVERABLE: prelab5c.zip
Submit all of your Verilog files from **Lab 5 Part B** and **Part C prelab**, as well as a text file **readme.txt**, all as a ZIP file named **lab5c.zip**. The **readme.txt** file should include your and your partner's names, your NetIDs, and information on your testing procedure. Submit the ZIP file to CMS (http://cms.csuglab.cornell.edu).

C. Branching Support

We will be adding four branch instructions to our CPU. These instructions are listed in Table C. You will need to add a branch select multiplexer that chooses which condition of the ALU to look for. In order to do this, you will have to compute certain values in your ALU. If the conditions that we are looking for are true, we will take the immediate **IMM**, sign extend the value and shift it left by 1, and then add it to the **PC** (i.e., the next value of **PC** should be **PC** + {**IMM**, 0}). This will give us the new *target address*, from which we will now start executing.

For the BGEZ and BLTZ instructions, you may find it easier to add some functionality to your ALU. However, this is certainly not necessary in order to implement these instructions. (Think about which ALU functions preserve the sign of **DataA**, thus outputting the correct status bits.)

D. Testing Your Design

For this lab, the pins have already been assigned for you. New I/O since Part B is as follows:
- The **EN_L** input uses the *active-low* debounced pushbutton **KEY2**.

Refer to Section III-E for instructions on how to program your modified processor onto the DE0 board.

First, use the newly-provided instruction RAMs to test the new branch instructions. Once you confirm that your branching logic is working fine, you will use your processor to run the **heart rate monitor**, using the provided instruction RAM (**lab5iramHRM.v**). The heart rate monitor will watch for pulses coming in on the least significant bit of **IOC** (**KEY0** on the DE0 board).

ASSIGNMENT 11 Update your processor to include branching logic. Test your updated processor and make sure it can run the heart rate monitor code.

Keep your final processor code, as we will need it for Lab 6!

Section V: Lab Report

DELIVERABLE: lab5report.pdf

You will submit a single report for your group. **Each partner must share in the writing of the report.** Please refer to the Lab Report Guidelines for general information on what to include. The report must be a PDF file, named **lab5report.pdf**, and should be submitted through CMS (http://cms.csuglab.cornell.edu).

Make sure to refer to the Lab Report Guidelines. For this lab report, you need to determine what needs to be included to describe the specification, your design, and key findings. Make sure that your lab report is self-contained and properly explains the processor design (*all three parts* of the lab). Make sure that we can reproduce your processor strictly from reading your lab report. **Do not include Verilog code to do this.**

For this particular lab report, you should include the following:

- **Introduction:**
  - Discuss what a single-cycle processor is.
- **Design:**
  - Describe the incremental design processor which took place over 3 lab sessions.
  - Include a diagram of the final processor datapath. Describe the purpose of each module in the datapath in executing a program.
- **Implementation and Testing:**
  - Describe the testing you did using ModelSim on the prelabs and lab sessions. Summarize your test cases and whether you believe they provide good test coverage.
  - Describe the testbench provided for part C of the lab. How does it work?
- **Modifications:**
  - In this lab you should answer the questions below in addition to describing any in-lab modifications. Your proposed changes cannot impact correct functionality.
  - Discuss one change to your processor design that could reduce the average number of instructions needed to write a program.
  - Discuss one change to your processor design (not implementation) which could increase the processor's maximum clock frequency.
- **Conclusion and Work Distribution**

Appendix A: Frequently Asked Questions

Can we make a one-bit adder module to use in our adder module?

Definitely. While it is not required, you can make a one-bit full adder module (don't forget to create this in its own file). Inside the **adder** module, you can then instantiate eight full adders to perform your addition. If you do this, remember to include the Verilog file for the for the full adder module when you submit your processor files!

In the table of instructions for the CPU (Table B), under the "Operation" column, why are some of the destination registers concatenated with "C?" For example, the ADDI instruction stores the result in {C,*RT}. What does that mean?

It's not that they are concatenated, but rather it's saying that the carry out bit is the most significant bit of the 9-bit result of the adder. In other words, it's not that you need to concatenate them, but if you did, {C, *RT} would be equal to the result of that addition.

References

[1] Dabnichki, P. (2008). *Computers in Sport* (pp. 105, 218, 251). Southampton: WIT Press.

[2] *Services*. (2009). Prozone. Retrieved October 26, 2013, from http://www.prozonesports.com/services.html

[3] Spino, M. (2010). *Innovative Digital Technology Increases Coach's Insight into Player Performance*. The Sport Digest. Retrieved November 25, 2012, from http://thesportdigest.com/archive/article/innovative-digital-technology-increases-coachs-insight-player-performance

[4] Sauser, B. (2008, April 10). *A Training Tool for Athletes*. MIT Technology Review. Retrieved November 25, 2012, from http://www.technologyreview.com/news/409885/a-training-tool-for-athletes/

[5] Jenkins, D., and Gerred, S. (2009, May 11). *A (Not so) Brief History of Electrocardiography*. ECG Library. Retrieved October 26, 2013, from http://www.ecglibrary.com/ecghist.html

[6] Parker, S. (2007, November 6). *History of Heart Rate Monitors*. ArticlesBase. Retrieved November 25, 2012, from http://www.articlesbase.com/health-articles/history-of-heart-rate-monitors-253755.html

[7] *Polar WearLink®+ Transmitter with Bluetooth®* (2013). Polar Electro. http://www.polar.com/us-en/products/accessories/Polar_WearLink_transmitter_with_Bluetooth