Lab 5 Heart Rate Monitor

Deadlines & Grading

- **Group Selection**: Monday, April 7, 2014 at 11:59 PM on CMS
- **Prelab**
  - **Part A**: **Friday, April 11, 2014** at 11:59 PM – 9 points, **in groups of 2**
  - **Part B**: Thursday, April 17, 2014 at 11:59 PM – 12 points, **in groups of 2**
  - **Part C**: Thursday, April 24, 2014 at 11:59 PM – 5 points, **in groups of 2**
- **Lab Exercises**
  - **Part A**: Monday, April 14 / Tuesday, April 15 – 18 points, **in groups of 2**
  - **Part B**: Monday, April 21 / Tuesday, April 22 – 18 points, **in groups of 2**
  - **Part C**: Monday, April 28 / Tuesday, April 29 – 18 points, **in groups of 2**
- **Report**: Monday, May 5 / Tuesday, May 6 at 11:59 PM – 20 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: Background

Software programs have widely altered sport over time. Their use in sport was initially limited to sorting data collected by hand and presenting the relevant information in a useful manner.[1] Over time, computer-controlled videotaping and analysis provided more data gathering tools, which could be analyzed to target improved athletic performance.[1] One such example can be seen in swimming, where it was a long-standing belief that swimmers should pull their hands directly backwards in a straight-line to utilize drag forces and aid propulsion.[1] This idea was finally put to rest when video analysis of various pull patterns revealed that the 'S'-shaped pattern in the front crawl was more effective.[1]

As a result, a niche market of *performance analysis* software has emerged in recent decades. For example, Prozone provides dynamic extraction of important information from a recording, both during and after data capture.[2] Dartfish enables monitoring of two skiers side-by-side,[3] allowing for direct comparison and thus providing better insight into maximizing performance.

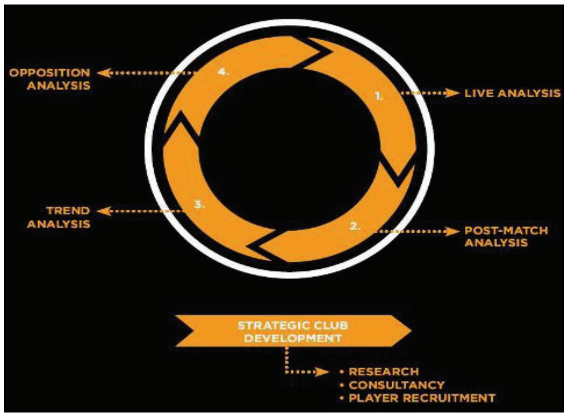

Figure 1. Prozone's real-time involvement in the coaching process using their analysis software.[2]

Videos, however, do not provide information on the inner workings of the body. Electronic sensor technology works around this limitation by providing a much larger data set for both coaches and athletes to pore over. For instance, sensors in skiers' apparel provide information about the athlete's motions such as forces, rotations, or accelerations.[1] A basketball player's movements during their best performance can be recorded by an armband, whose two sensors connect to a laptop to track the player's throws. The device plays a special tone when the athlete's arm matches the best performance movement.[4] Such real-time interaction can condition the athlete to recognize and mimic subtle muscle control for precise replication. This method was taken further by researchers at the University of Calgary, who used a helmet-mounted eye-tracker to record the gaze of athletes for examining neuro-motor psychology. [1]

Heart rate monitoring has proved to be an important application of sensor technology that has matured significantly over time. As early as 1872, Alexander Muirhead attached wires to a patient's wrist to record their heartbeat.[5] In the early 20th century, Augustus Waller created the

Figure 2. Polar WearLink® chest strap.[7]

first electrocardiograph (ECG/EKG).[5] Soon after, Willem Einthoven won the Nobel Prize in Medicine for his invention of a much more sensitive electrocardiograph technique in 1901.[5] Many years later, Polar Electro launched the first wireless heart rate monitor in 1977, as a training aid for the Finnish National Cross Country Ski team.[6] With advancements in various industries, such monitors began to be made out of conductive smart fabric with built-in microprocessors.[6] The sports bra released by Textronics in December 2005 was the first garment laden with heart sensors.[6] Today, heart rate monitors are ubiquitous in exercise equipment, and can even be linked wirelessly to smartphones, as seen in the Polar WearLink®+.[7]

Section III: Prelab, Part A

A. ALU Design & Operation

The first step to your processor design is to build an 8-bit ALU, as shown in Figure 3. A three-bit input **OP** indicates which operation the ALU should perform. Table A shows the operations you must implement.

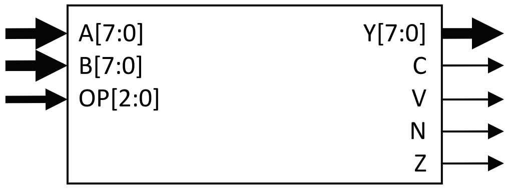

Figure 3. ALU block diagram.

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, and X represents *don't care*.

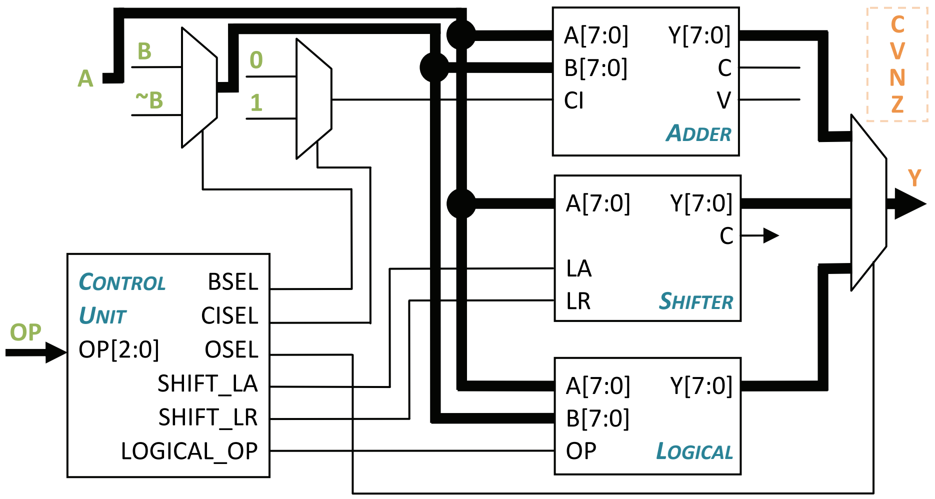

Figure 4. Incomplete ALU datapath. Inputs shown in green; outputs in orange.

The ALU functions will be split between three modules that will be placed inside the ALU. Figure 4 shows how the ALU operations are split among three modules. You will build two of them for prelab – for now, we will not build the **adder** module.

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).

The second module, **shifter**, can be found inside **shifter.v**. The SRA, SRL, and SLL instructions will be implemented here. Figure 5 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.)

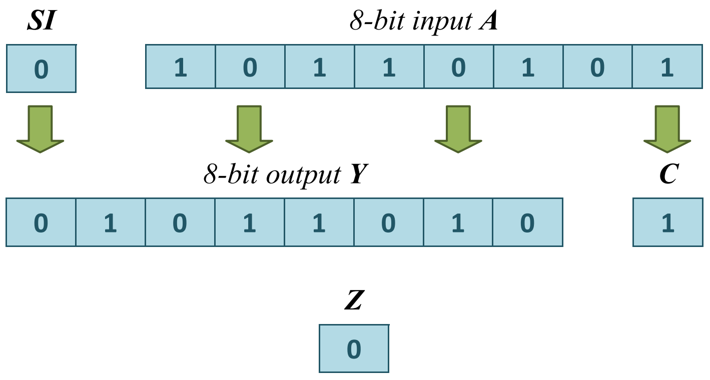

Figure 5. Visualization of the SRL operation (**LA** = 0, **LR** = 1).

Figure 4 also shows how you should interconnect these modules within the ALU. Note that there are two multiplexers to the left that control what values go into the **adder** and **logical** modules. One multiplexer controls whether **B** or **~B** is sent to these modules, while the other multiplex controls whether 0 or 1 is sent to **adder** as the carry in. The multiplexer to the right determines which of the modules you want to read your output **Y** from.

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 4. This unit reads in the **OP** input, and determines how to set the multiplexers. 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 an example multiplexer, **muxCI.v**, as well as the code inside **control.v** to determine which multiplexer input to select.

You are free to design any of the missing elements from the ALU in Figure 4 as you wish, but your final circuit must satisfy all of the requirements we have stated here. For the multiplexers, you must implement your own MUX modules in Verilog.

NOTE 1 You are only allowed to use logical operators (&, |, ~, ^) for all of your Verilog modules in this part of the lab. Specifically, **the** + **and** – **operators are forbidden** for the ALU. Inside **alu.v**, place nothing but Verilog modules.

ASSIGNMENT 1 Implement the **shifter** and **logical** modules (do not implement **adder** yet).

1. Download **alu.zip** from CMS, unzip its contents into a folder called **lab5**, and open the project file **alu.qpf** inside Quartus. If you open **alu.v**, you will see empty modules already inserted for you.
2. Edit the files **shifter.v**, and **logical.v** to add your functionality for each module.
3. Save and compile these Verilog files.

ASSIGNMENT 2 Create the remaining two multiplexer modules.

1. Open the file **muxCI.v**.
2. Using this MUX design as a baseline, create new Verilog files that implement the multiplexers for B and the output To create a new file, click *File* → *New*. In the window that pops up, select *Verilog HDL File* under *Design File* and click OK.
3. Save the files.

ASSIGNMENT 3 Inside **control.v**, create a module called **control** that will implement the control unit shown in Figure 4.

B. Top-Level Assembly & ModelSim Testing

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

1. Open the file **alu.v**.
2. Insert your **control** module and multiplexer modules.
3. Connect these modules together, and to the input and output pins.
4. Save the file, and compile your project.

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 to thoroughly test the ALU for all non-adder operations. Make sure you add the delays after each case.

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

NOTE 2 When debugging, you should use *RTL simulation* to test your circuit. **However, when finished, make sure you use *gate-level simulation* one final time to verify that your circuit doesn't have any inferred latches or unusual delays.** These do *not* show up under RTL simulation.

C. Prelab Deliverables

DELIVERABLE 1: prelab5a.zip *due April 11*

Submit all Verilog files (including **shifter.v** and **logical.v**) from **Assignment 1**, all of your Verilog files from **Assignment 2**, the Verilog file **control.v** from **Assignment 3**, the Verilog file **alu.v** from **Assignment 4**, the Verilog test bench **alu_test.v** from **Assignment 5**, and a text file **readme.txt**, all as a ZIP file named **prelab5a.zip**. The **readme.txt** file should include your and your partner's names, your NetIDs, and any pertinent information for the graders.

NOTE 3 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.

Submit all of the above files to CMS (http://cms.csuglab.cornell.edu).

NOTE 4 Your prelab *must* be done in groups of two. **You must choose your partners on CMS by Friday, March 28.** This means both partners must log in and approve the group request. **Any ungrouped students will be paired by the instructors immediately after the deadline.**

Section IV: Lab Exercises, Part A

A. Building the Adder and Completing Your ALU

In the first lab session, we will check you off a fully-functioning version of the ALU. You will take your **corrected** prelab, and implement the **adder** module. Make sure you bring a digital copy of the **lab5** folder you created for Assignments 1 through 6 (again, **updated with corrections based on our feedback**) to lab. At the very least, the folder should contain all your Verilog files (**.v**) from your prelab submission, **alu.qpf**, and **alu.qsf**. You will need these updated files to complete the lab exercise.

The **adder** module can be found inside **adder.v**. Inside this module, you will implement the ADD and SUB instructions. For these operations, we will implement the arithmetic as follows: two eight-bit numbers **A** and **B** are combined as described in Table A to output a nine-bit number. 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 5 Again, you are only allowed to use logical operators (&, |, ~, ^) for all of your Verilog modules in this part of the lab. Specifically, **the + and – operators are forbidden** for the ALU. Inside **alu.v**, place nothing but Verilog modules.

ASSIGNMENT 7 Implement the **adder** module.

1. Open the project file **alu.qpf** (in the **lab5** folder from **Assignments 1 through 6**) inside Quartus.
2. Open **adder.v** and add your adder logic.
3. Connect the adder module properly inside the rest of your ALU code.
4. Save the file, and then compile your project. Correct any syntax errors.

ASSIGNMENT 8 Update the test bench to add cases that check the functionality of your adder. Your test bench should exhaustively test the *complete* functionality of the ALU, including your previously-implemented modules. Run the test bench in ModelSim to verify that your ALU is correct.

B. Preparing Your ALU for the DE2 Board

Before we download the design, we must first set up a few parameters before compiling a design. You will need a DE2 board, a plug, and a USB cable, all of which will be provided in lab.

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

- Input **A** is mapped to toggle switches **SW15** (the MSB) to **SW8**.

- Input **B** is mapped to toggle switches **SW7** (the MSB) to **SW0**.
- The 3-bit **OP** input is connected to the *active-low* push-buttons **KEY2** (MSB) to **KEY0**. Remember that the push-buttons output a logic "1" when **not** pressed.

Circuit outputs are assigned as follows:

- Output **Y** is displayed on **HEX7** (MSB) and **HEX6**, as a hexadecimal number.
- Output **C** is displayed on **LEDG3**.
- Output **V** is displayed on **LEDG2**.
- Output **N** is displayed on **LEDG1**.
- Output **Z** is displayed on **LEDG0**.

1. Open the **alu.qpf** file.
2. Go to *Assignments* → *Devices…* and select the Cyclone II family. Under *Available Devices*, select *Specific device selected*, and choose the *EP2C35F672C6*. 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.
4. Compile your design. This will create the file **alu.sof**, which we use to program the FPGA.

C. Testing Your Design

1. Take the DE2 board and plug the power cable into the power port on the top left. 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 the LCD screen will say "Welcome to the Altera DE2 Board."
2. Connect a USB cable to the leftmost USB port on the DE2 board (the **USB Blaster Port**). Connect the other end to the front of your PC.
3. When you plug in the cable, Windows should tell you that the **Altera USB-Blaster** device was installed properly. If it wasn't, please let a TA know.
4. Turn the **RUN/PROG** switch on the left of the board to the **RUN** position.
5. Inside Quartus, go to *Tools* → *Programmer*. Under *Hardware Setup*, select *USB-Blaster*.
6. Inside the programmer tool window, you will see your **alu.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, and you can start testing your circuit.

Once you have verified that your circuit is working properly, request to be checked off by a TA. After you have successfully programmed your FPGA, we may ask questions related to the lab, which will count towards your lab exercise grade. **Remember to save your folder somewhere when you have finished – you will need these files to complete the second part of the lab!**

Before leaving the lab:

- Make sure a TA has checked off all parts of your lab, including parts that aren't fully working.
- If you have used the lab computer, make sure to **save all files remotely** and then log off. Upon logging off, the computer automatically deletes any files that you created, so be careful!
- Clean your bench.
- Get your lab partner's contact information if you don't have it already. You will be working with the same person on Parts B and C of the lab.

Section V: Prelab, Part B

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 (which will be provided), and will use a data RAM (which will also 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 6 shows the datapath that you will need to implement for this lab. All of the control signal inputs to the processor will be set based on the current instruction being executed. 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 will be used to retrieve data from the *register file*, perform an ALU operation, potentially write to the data RAM, and write a new value back to the register file. 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. You are responsible for implementing the rest of the processor. Details of how these structures should look are provided below.

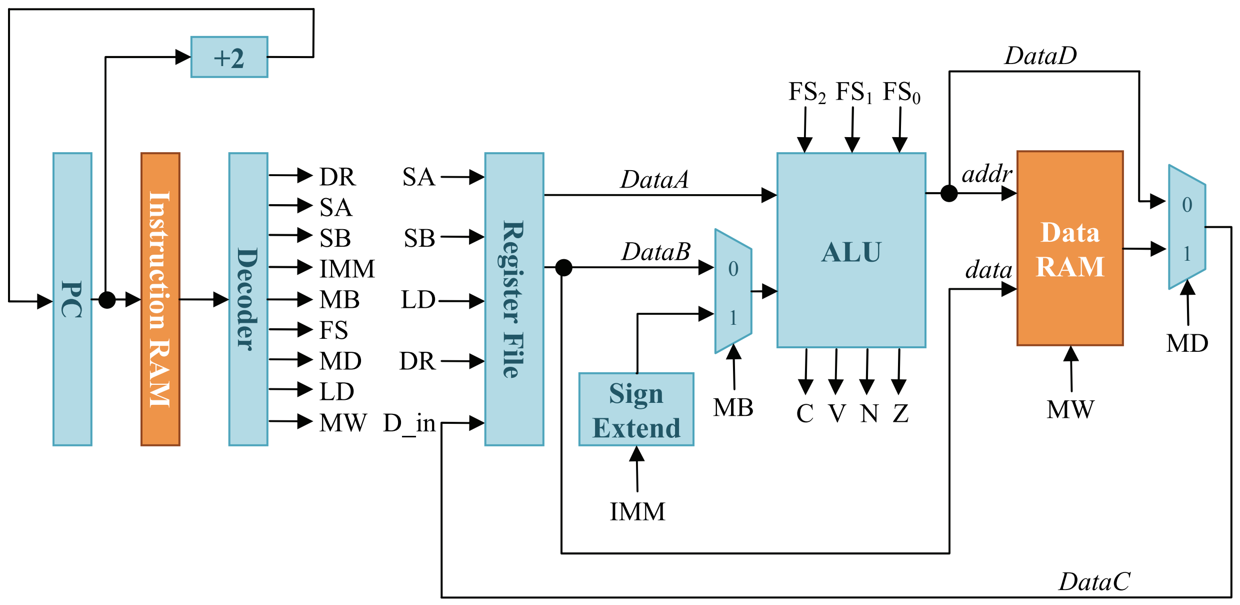

Figure 6. Processor datapath. Boxes in blue must be implemented, while boxes in orange are provided for you.

Register File

The register file inside your microprocessor should contain eight registers, each of which are eight bits wide. Since you will have eight registers, the **SA**, **SB**, and **DR** signals must be three bits wide each. When **RESET** = 1, set all eight registers to 0.

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 7. 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 7. 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 6.

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

You will be designing all of the microprocessor components shown in Figure 6, *except for the instruction and data RAMs*. We have provided both RAM blocks for you, and have already wired them up in the circuit for you.

The instruction RAM takes in an 8-bit address, **addr**, that selects which line 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 6.

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 DE2 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 six addresses in memory to output pins **IOC**, **IOD**, …, **IOG**, and **IOH**. These are connected to addresses 250 through 255, respectively, inside the data RAM. These are known as *memory-mapped* outputs, since they always output the value of a certain location. As detailed in Section VI-A, we take these outputs and connect them to either LEDs or seven-segment displays on the DE2 board. For example, data stored at memory address 252 will be sent on output pin **IOE**, which we can then read on seven-segment displays **HEX1** and **HEX0**. We will be using these outputs to check if your processors are working properly.

We have also mapped 16 input switches to memory locations 248 and 249. These *memory-mapped inputs* are called **IOA** and **IOB**. Essentially, reading from memory address 248 and 249 in the data RAM will provide your program with the switch values in **IOA** and **IOB**, respectively. Section VI-A has details on which switches are connected to **IOA** and **IOB**.

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.

We **will** be using these outputs for grading, so please make sure that they are properly connected.

Provided Instruction RAM Modules

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.

Refer to Assignment 11 for instructions on how to change between programs.

NOTE 6 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 can check if you have any by going to the *Warning* tab in Quartus when you compile your code.

ASSIGNMENT 9 Implement the processor as the Verilog module **cpu**.

1. Download **lab5.zip** from CMS, unzip its contents into your **lab5** folder from Part A, and open the project file **lab5.v** inside Quartus.
2. Create and edit modules that perform the functionality described above. All of your work should be inside the **cpu** module, provided in **cpu.v**. Compile your design.

NOTE 7 **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. If you make a new module (e.g., `RegisterFile`), make sure it gets its own file with a properly-capitalized filename (e.g., **RegisterFile.v**)!

ASSIGNMENT 10 Run the test bench, **lab5_test.v**. Feel free to modify **lab5iram.v** to test out all of the required instructions. (You will **not** be submitting either of these files.) Make sure you follow NOTE 2.

ASSIGNMENT 11 Use the other IRAM 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.

C. Prelab Deliverables

DELIVERABLE 2: prelab5b.zip *due April 17*

Submit *all* of your Verilog files, including **lab5.v** and **cpu.v**, from **Assignment 9**, and a text file **readme.txt**, all as a ZIP file named **prelab5b.zip**. The **readme.txt** file should include your and your partner's names, your NetIDs, and any pertinent information for the graders.

NOTE 8 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.

Submit all of the above files to CMS (http://cms.csuglab.cornell.edu).

Section VI: Lab Exercises, Part B

In lab, we will work with you to fix your processor. Make sure you bring a digital copy of the **lab5** folder you created for Assignments 1 through 11 to lab. At the very least, the folder should contain all your Verilog files (**.v**) from your prelab submission and from Part A, **lab5.qpf**, and **lab5.qsf**. You will need these files to complete the lab exercise.

A. Preparing Your Processor for the DE2 Board

Before we download the design, we must first set up a few parameters before compiling a design. You will need a DE2 board, a plug, and a USB cable, all of which will be provided in lab.

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

- The **RESET** input uses toggle switch **SW0**.
- The **CLK_SEL** input uses toggle switch **SW1**. Set this to 1 to execute the processor at 10 MHz (which we will usually do), and to 0 for 1 Hz execution.
- Input **IOA** (memory location 248) uses toggle switches **SW17** (MSB) through **SW10**.
- Input **IOB** (memory location 249) uses toggle switches **SW9** (MSB) through **SW2**.

Circuit outputs are assigned as follows:

- Output **IOC** (memory location 250) is shown on **LEDR7** (MSB) through **LEDR0**.
- Output **IOD** (memory location 251) is shown on **LEDG7** (MSB) through **LEDG0**.
- Output **IOE** (memory location 252) is shown on **HEX1** and **HEX0**.
- Output **IOF** (memory location 253) is shown on **HEX3** and **HEX2**.
- Output **IOG** (memory location 254) is shown on **HEX5** and **HEX4**.
- Output **IOH** (memory location 255) is shown on **HEX7** and **HEX6**.

1. Open the **lab5.qpf** file.
2. Go to *Assignments* → *Devices…* and select the Cyclone II family. Under *Available Devices*, select *Specific device selected*, and choose the *EP2C35F672C6*. 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.
4. Compile your design. This will create the file **lab5.sof**, which we use to program the FPGA.

B. Testing Your Design

7. Take the DE2 board and plug the power cable into the power port on the top left. 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 the LCD screen will say "Welcome to the Altera DE2 Board."
8. Connect a USB cable to the leftmost USB port on the DE2 board (the **USB Blaster Port**). Connect the other end to the front of your PC.
9. When you plug in the cable, Windows should tell you that the **Altera USB-Blaster** device was installed properly. If it wasn't, please let a TA know.
10. Turn the **RUN/PROG** switch on the left of the board to the **RUN** position.
11. Inside Quartus, go to *Tools* → *Programmer*. Under *Hardware Setup*, select *USB-Blaster*.
12. Inside the programmer tool window, you will see your **lab5.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, and you can start testing your circuit.

Once you have verified that your circuit is working properly, request to be checked off by a TA. After you have successfully programmed your FPGA, we may ask questions related to the lab, which will count towards your lab exercise grade. **Remember to save your folder somewhere when you have finished – you will need these files to complete the second part of the lab!**

Section VII: Prelab, Part C

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.

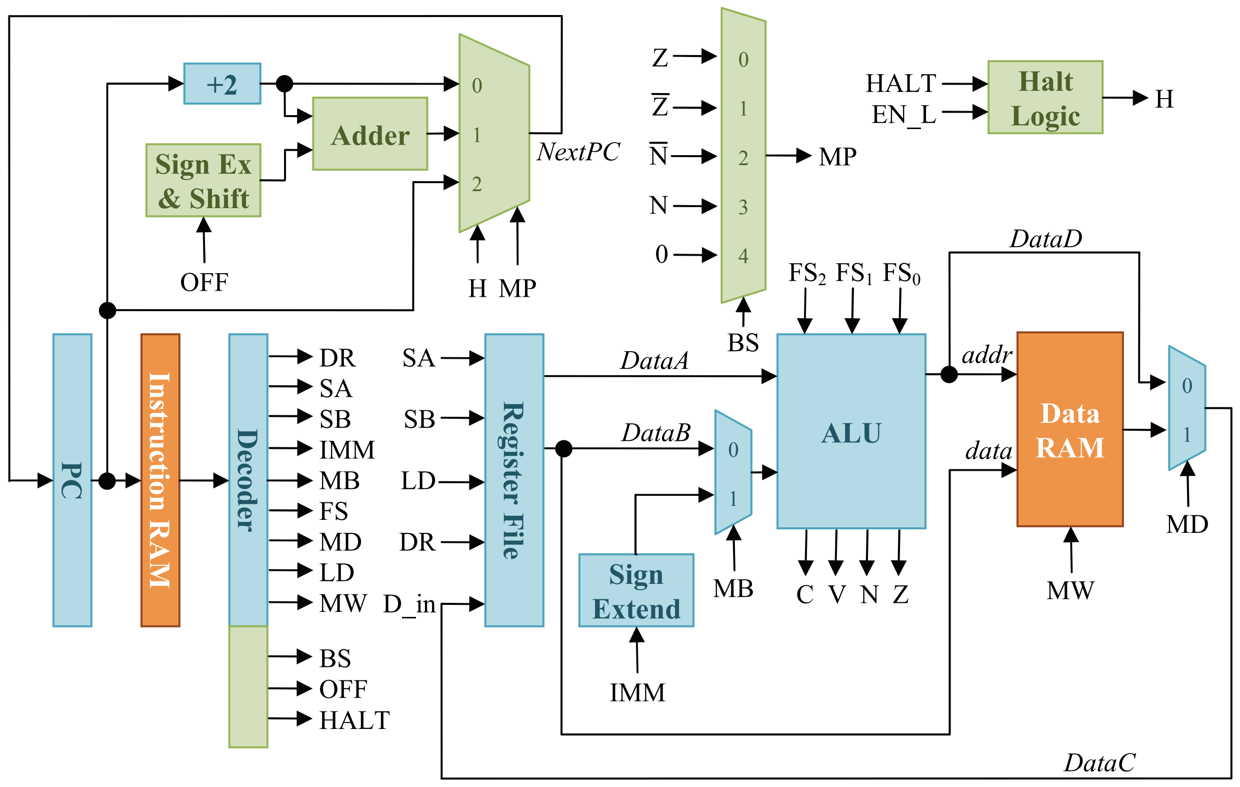

Figure 8. Extended processor datapath. Boxes in green are new modules that you must implement. Boxes in blue were implemented in Part B, but may need modification, while boxes in orange are provided for you.

B. Design Updates

Figure 8 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 at each positive clock edge.

For now, we will ignore the four branching instructions. We will add these in later.

I/O Pins

In addition to our previous I/O, we will also use one of the debounced pushbutton switches 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 12 Update your processor to include halting logic.

1. Edit modules that perform the functionality described above. All of your work should again be inside the **cpu** module, provided in **cpu.v**.
2. Compile your design.

ASSIGNMENT 13 Update the instruction RAM file, **lab5iram.v**, so that it thoroughly tests your circuit. Use ModelSim to verify the processor's functionality.

C. Prelab Deliverables

DELIVERABLE 3: prelab5c.zip *due April 24*

Submit *all* of your Verilog files, including **lab5.v** and **cpu.v**, from **Assignment 12**, and a text file **readme.txt**, all as a ZIP file named **prelab5c.zip**. The **readme.txt** file should include your and your partner's names, your NetIDs, and any pertinent information for the graders.

NOTE 9 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.

Submit all of the above files to CMS (http://cms.csuglab.cornell.edu).

Section VIII: Lab Exercises, Part C

A. 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** *after it has been incremented* (i.e., the next value of **PC** should be **PC** + 2 + {**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.)

Make sure you bring a digital copy of the **lab5** folder you created for Assignments 1 through 13 to lab. At the very least, the folder should contain all your **updated** Verilog files (**.v**) from your prelab submission and from Parts A and B, **lab5.qpf**, and **lab5.qsf**. You will need these files to complete the lab exercise. **Make sure you fix all prelab errors before coming to lab!**

ASSIGNMENT 14 Update your processor to include branching logic.

1. Edit modules that perform the functionality described above. All of your work should again be inside the **cpu** module, provided in **cpu.v**.
2. Compile your design and correct any compilation errors.
3. Use the new instruction RAMs to test the new functionality of your processor in ModelSim. Details about these instruction RAMs will be provided when the code is uploaded.

B. 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 **KEY0**.

Refer to Section VI for instructions on how to program your modified processor onto the DE2.

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 (**lab5iramHBM.v**). This will require you to also use a new data RAM, **lab5dramHBM.v**. The heart rate monitor will watch for pulses coming in on the least significant bit of **IOB** (**SW2** on the DE2 board). More details will be posted when the code is uploaded.

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

Section IX: Lab Report

DELIVERABLE 4: lab5report.pdf *due* ***seven days after lab***

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 on Blackboard 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).

Groups in the Monday lab sessions must submit their report by **Monday, April 28 at 11:59 PM**. Groups in the Tuesday evening lab session have until **Tuesday, April 29 at 11:59 PM**.

In general, lab reports for ENGRD 2300 should do the following:

- Assume that the reader has a general ECE knowledge, but has never seen this handout (and does not have access to it). This means you must summarize the purpose of this project, as well as the tasks you have been asked to complete. This also means you cannot refer to any portion of this document in your write-up.
- Include some sort of introduction.
- Use section headings to organize the document.
- Provide enough detail so we can reproduce your lab *exactly* as you designed it without looking at your submission.
- Include a separate section on the distribution of work for the jointly-done parts of the lab, including the report.

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

- Discuss your implementation for *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.**
- Include an image of the final datapath.
- Describe, on a high level, how the datapath works, and how a processor is an extension of a state machine.
- Provide a description of how the heart rate monitor code works (i.e., on a high level, describe what the instructions themselves are trying to accomplish).
- Discuss one hardware change to your processor that could reduce the number of instructions your program needs.

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