RAIK 284H Final Project

Spring 2010

I. Purpose

You will be building an 8-bit RISC processor using the Altera Quartus II software. This will be a single-cycle processor consisting of a ROM component for storing instructions, a decoder for interpreting instructions, an ALU for performing operations, and a RAM unit for loading and storing register values. Then, you will be writing an assembler to build programs for your processor and writing a (fairly) simple assembly program to run on your processor. You will then download your finished program and processor to the Altera board to demonstrate its completeness. To accomplish all of this, you will be working in teams of two.

II. Processor Specifications

A. Overall Structure

Your processor must contain the basic sections required for a single-cycle processor, which include a clock, a program counter with jump/branch resolution, an instruction-fetching unit, a register file, an arithmetic logic unit, and a memory access unit. Your processor should have a high-level graphical design that shows all of these components at a minimum. For subcomponents, you may either use a graphical layout or a hardware description language such as VHDL (some restrictions apply, see table for details).

Component	**Implementation**
Overall design	Graphical layout
Clock unit	Provided
Program counter unit	Graphical layout or hardware language
Instruction fetching unit	Graphical layout or hardware language
Register file unit	Hardware language
ALU	Graphical layout or hardware language
Memory access unit	Graphical layout or hardware language

If you feel that other units are beneficial, you may implement them in the method of your choice. Keep in mind that graphically laying out components makes it easy to see how the component works, but makes it more difficult to make changes since all of the wires have to be remapped.

B. Implementation Details

Here are some required specifications for your processor and the types of instructions it will be able to execute. In certain places, you will be given a choice of how to implement your processor. These situations are indicated with ***. You are encouraged to be creative here, and consider what would be useful additional capabilities for your processor. As part of your assignment, provide a small, informal document that explains the decisions you made.

Instruction size	28 bits
Addressing space	8 bits
Immediate value size	8 bits
Number of registers	16

Each instruction will have the structure shown in the table below. RS, RT, and RD are all 4 bit register identifiers that map to one of the 16 registers. RS and RT are generally used as source registers (see the operation codes), while RD is a destination register.

Note: Register zero should ALWAYS contain the value zero, and should never be written to. You should ensure this by disallowing storing to register zero in the processor design. In addition, the assembler should warn that storing to register 0 will not maintain value if a program attempts it.

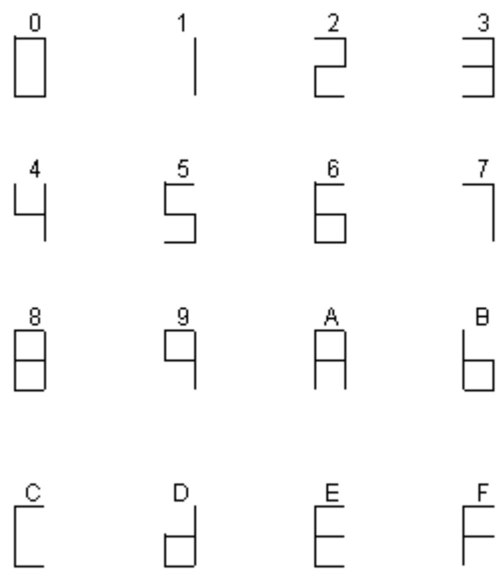
Op code	ALU code	RD	RS	RT	IMMEDIATE
4 bits	4 bits	4 bits	4 bits	4 bits	8 bits

The operation codes supported by the processor are as shown in the below table. You may notice that the rightmost bit can be used to determine if the operation requires the immediate value in the case of the alu/alui and jr/j instructions.

Op Code	**Name**	**Usage**	**Description**
0000	nop	Nop	No operation
0010	alu	alu RD, RS, RT	Performs an ALU operation with one of the ALU codes
0011	alui	alui RD, RS, IMMEDIATE	Performs an ALU operation with one of the ALU codes using an immediate value as one of the operands
0100	jr	jr RS	Jumps to line stored in register
0101	j	j IMMEDIATE	Jumps to the line stored in the immediate
1001	lw	lw RD, IMMEDIATE(RS)	Loads a value from RAM or dipswitch state
1011	sw	sw RT, IMMEDIATE(RS)	Stores a display digit
1101	beq	beq RS, RT, IMMEDIATE	Branches if two register values are equal
111* 0001 10*0	***	***	*Implement any instruction/instructions you would like, or none at all.*

Note that the lw and sw instructions are not just used for reading and writing to memory in the traditional sense. Instead, they are used for reading display digits, getting the state of dipswitches and toggle buttons, and storing values on the display. To accomplish this, special "addresses" are used which map to different inputs and outputs. The addresses you are required to use are listed below. Addresses 128-255 should correspond to valid storage locations, which must function like ordinary memory.

Address	**Input/Output map**
0	Input the status of the dip switches as an 8-bit integer.
1	Input the status of the left push button (0 if up, 1 if down)
2	Input the status of the right push button (0 if up, 1 if down)
3-127	*** *(Implement this however you would like)*
0	Output a number to the digits (display in hex, see below)
1	Set the left decimal point
2	Set the right decimal point
3-127	*** *(Implement this however you would like)*



For example, you could use lw $2, 1($0) to read the value from the left push button and store the result in register 2. Also, you could use sw $5, 0($0) to store the number in register 5 into the display. For example, the following simple program would continually display the state of the push buttons in the decimal points.

```
loop: lw $1,1($0)
sw $1,1($0)
lw $1,2($0)
sw $1,2($0)
jmp loop
```

For the alu and alui operation codes, the following arithmetic operations should be allowed in the alu code slot. If you feel that allowing more alu operations would be beneficial, you may do so with the slots that are currently reserved for no operation.

Alu code	Operation	Description
0000	add	Adds the two operands
0001	***	*Implement this however you would like.*
0010	sub	Subtracts the second operand from the first
0011	***	*Implement this however you would like.*
0100	and	Performs the binary AND operation
0101	nand	Performs the binary NAND operation
0110	or	Performs the binary OR operation
0111	nor	Performs the binary NOR operation
1000	xor	Performs the binary XOR operation
1001	xnor	Performs the binary XNOR operation
1010	slt	Returns 255 if the first operand is less than the second operand, otherwise returns 0.
1011	sge	Returns 255 if the first operand is greater than or equal to the second operand, otherwise returns 0.
11**	***	*Implement this however you would like.*

C. Simulation

In order to demonstrate that your design works, you should create simulation files which use boundary input conditions to test the functionality of components in your processor. This will help you both by ensuring you that your processor is robust and by giving reason for partial credit in case your processor implementation does not run the on the Altera board. You are required to simulate your ALU component to demonstrate correctness, but you are encouraged to simulate other components as well. The ALU simulation should cover all of the operations and several (at least 3) representative operands for that operation.

III. Assembler Specifications

In order to create programs for you processor, you will write an assembler in a language of your choice (Java, C++, or C# recommended). This assembler must be able to run in Windows. It should be a simple command-line program with the syntax **myassembler infile outfile**. The input file should be the source assembly program. The output file should be a file that conforms to Altera's MIF format for read-only memory. A sample assembly program and MIF output file are included in the project handout.

Your assembler should make 2 passes over the code. On the first pass, the assembler should parse all symbols and check for syntax errors. On the second pass, the assembler should resolve all final addresses (such as mapping labels to real addresses for jump targets).

Please include specific instructions on how to build your program. If you use C++/C# with Visual Studio, provide a solution or project file. If you use Java, provide a makefile for building your program. If you wish to use another language, please double-check with the TA first (the following languages are OK without asking: Java, C++, C, C#, VB.NET, Perl, and PHP).

IV. Assembly Program

To test your processor, you will implement a batch calculator. This calculator will run in two phases. In the first phase, the user will be able to enter a series of numbers (no more than 100 numbers will be entered), which the program will store in the data memory of the processor. In the second phase of the program, the user will be able to scroll through the stored values using the pushbuttons, and then select an ALU operation (add, and, or, nand, nor, xor, xnor) to perform on them.

Specifically, the following behaviors will be observed in Phase 1. (The left dot will be on while in this phase)

Left Button	Right Button	State/Action
Unpressed	Unpressed	Display shows the number of stored values.
Pressed	Unpressed	Display shows the value in the dips switches. When the left button becomes unpressed, the value in the dip switches will be saved to memory.
Unpressed	Pressed	Display shows the last stored value (0 if no previous value).
Pressed	Pressed	Display shows the number of stored values, with right dot on. When either button is released, move to Phase 2.

The following behaviors will be observed in Phase 2. (The right dot will be on while in this phase)

Left Button	Right Button	State/Action
Unpressed	Unpressed	Display shows value at current memory position.
Pressed	Unpressed	Decrement memory location once (wrap around if at first stored element) and display the new address.
Unpressed	Pressed	Increment memory location once (wrap around if at last stored element) and display the new address.
Pressed	Pressed	Perform ALU operation on all stored values together. Display shows the result, and memory becomes reset. When either button is released, move back to Phase 1. The ALU operation performed depends on the state of the dip switches according to the following table.

Dip switch value	Operation
0	and
1	nand
2	or
3	nor
4	xor
5	xnor
6-127	*** (implement this however you would like)

V. Presentation

To demonstrate that your processor works, you will give a short presentation to the instructors in which you show your program and assembler. These presentations will be during dead week, at the same time the processor is due. A signup for times will be available closer to the due date.

VI. Grading

You will be graded using the following metrics:

1. Completeness and functionality of the processor implementation (90 points)
    a. Design compiles and simulates properly (30 points)
    b. Downloads to the board and runs properly (30 points)
    c. Conforms to all specifications (30 points)
2. Completeness and functionality of the assembler (40 points)
    a. Assembles programs to the MIF format (30 points)
    b. Handles errors in the code gracefully (10 points)
3. Presentation (20 points)

VII. Timeline

The following deadlines for project components must be met in order to stay on schedule (and get a good grade!).

Date	**Items Due**
April 11	Assembler and first draft of assembly program
April 18	Program registers, ALU, and test cases
April 25	Fetch, decode, memory-mapped I/o, debouncer, and test cases
April 28	Complete Processor, assembler, and assembly program

VIII. Included Files

Included with this description are a number of files, which should aid you in your quest to complete the processor. Here is a brief description of what each file is for.

display.mif – This MIF file contains a memory layout that converts a 4-bit index to the pattern to display it as a hex digit.

clock1hz.vhd – This VHDL file steps the clock down to slower speed so your processor will work. You can change the speed if you want by modifying the constant, however, the speed it is set to run at is 2 kHz and should be sufficiently slow such that your processor will not fail due to unresolved circuits.

SampleAssemblyProgram.s – This is a sample MIPS assembly program (which should run on your processor, if you are daring enough to try it). This program runs a vending machine program that accepts and returns change. You may use this as a template for what format your registers and comments should look like. Any questions about MIPS assembly can be directed to the TA.

SampleMIF.mif – This MIF file is the compiled version of SampleAssemblyProgram.s. Use it as an example of the output your assembler should generate.

IX. Useful Information

Here are some things you will need to set up for the project.

1. Assign the correct device

   Go to Assignments→Device…
   Choose FLEX10K from the device family
   Choose EPF10K70RC240-4 from the devices list
2. Map pins

   Go to Assign → Pin/Location/Chip
   Add the following pin mappings to your inputs and outputs.

   <u>Inputs</u>
   a. on-board oscillator (@25.175MHz) is connected to pin 91.
   b. push-buttons (active-low):
   FLEX_PB1 28
   FLEX_PB2 29
   c. Dip Switches ('1' when switch is open)
   FLEX_SWITCH-1 41
   FLEX_SWITCH-2 40
   FLEX_SWITCH-3 39
   FLEX_SWITCH-4 38
   FLEX_SWITCH-5 36
   FLEX_SWITCH-6 35
   FLEX_SWITCH-7 34
   FLEX_SWITCH-8 33

   <u>Outputs</u>
   d. Dual-digit seven-segment display (active-low)

Display Segment	pin for digit 1	pin for Digit 2
a	06	17
b	07	18
c	08	19
d	09	20
e	11	21
f	12	23
g	13	24
Decimal Point	14	25

X. Hints

Here are some hints as you work on this:

- Pay attention to useful patterns in the opcodes. They are designed to minimize the complexity of logic inside the processor.
- For the display component, it might be a good idea to include two small LPM_ROM components, one for each digit, containing the display.mif file. Alternately, you could implement the 7-segment display separately as a component.
- Quartus II compiles slowly. Very slowly. Take this in to account while doing your development, and have a plan for downtime. Watching Quartus during its compilation will eat your productivity very quickly.
- It might be helpful to develop top-down, but you should definitely test (using simulation or otherwise) bottom-up.
- One feature of Quartus is its ability to use conduits to manage the connections between blocks. To learn more about this, it may be a good idea to run through some of the tutorials for Quartus II that are inside its help system.
- Start early. Even if you can't put in many hours before two weeks before the processor is due, at least getting a start on it at an earlier time will help you be able to have some time to understand what is going on.
- The implementation-specific features are an optional portion of this project. However, even if you do not implement anything for them intentionally, you should be able to explain what will happen if those op codes are used or that memory is accessed.
- The pins may function in the reverse of how you would expect them to. For instance, the display pins will display a segment if a 0 is output on them, but turn it off if a 1 is output on them. Similar behaviors may exist for the push buttons and dip switches. You may need to use NOT gates or LPM_INV (for arrays) to flip the inputs/outputs to achieve expected output.
- Altera's Quick Reference to LPM (the Library of Parameterized Modules) can be useful. It is available at http://www.altera.com/literature/catalogs/lpm.pdf