# RAIK 284H Final Project

### 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 unit for storing instructions, a decoder for interpreting instructions, an ALU for performing operations, and a RAM unit for loading and storing register values. As part of the project, 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 provided Altera board to demonstrate its completeness. To accomplish all of this, you will be working with a partner.

### 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 decoder, 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 HDL |
| Instruction fetching unit | Graphical layout or HDL |
| Register file unit | Graphical layout or HDL |
| 8-bit ALU and shifters | Graphical layout or HDL |
| Memory units (ROM and RAM) | Graphical layout or HDL |
| Decoder | HDL |

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 | 16 bits |
| Addressing space | 8 bits |
| Immediate value size | 6 bits |
| Number of registers | 8 (one is used as the return address register, RA) |

There are three types of instructions. The structure of each type is shown in the table below. RS, RT, and RD are all 3-bit register identifiers that map to one of the 8 registers. RS and RT are generally used as source registers (see the operation codes), while RD is a destination register. Each register is 8-bit wide.

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 would not maintain value if a program attempts it.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Op code | RD | RS | RT | ALU code | Instruction type |
| 4 bits | 3 bits | 3 bits | 3 bits | 3 bits | R-Type (involves 3 registers) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op code | RD | RS | Immediate | Instruction type |
| 4 bits | 3 bits | 3 bits | 6 bits | I-Type (immediate value) |

|  |  |  |  |
| --- | --- | --- | --- |
| Op code | Unused | Immediate | Instruction type |
| 4 bits | 6 bits | 6 bits | J-Type (immediate value) |

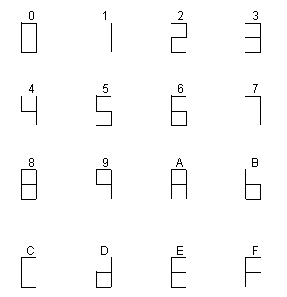
The operation codes supported by the processor are as shown in the below table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **Instruction** | **Description** | **Format** | **# of Registers** |
| Logical and Arithmetic | and | And | R | 3 |
| or | Or | R | 3 |
| xor | Xor | R | 3 |
| sll | Shift left logical | R | 3 |
| srl | Shift right logical | R | 3 |
| add | Add with sign | R | 3 |
| sub | Subtract with sign | R | 3 |
| addiu | Add immediate unsigned | I | 2 |
| subiu | Subtract immediate unsigned | I | 2 |
| addi | Add immediate with sign | I | 2 |
| subi | Subtract immediate with sign | I | 2 |
| Unconditional branch | j | Jump | J | 0 |
| jr | Jump register | R | 1 |
| jal\* | Jump and link | J | 1 (RA) |
| Conditional branch | beq | Branch on equal | I | 2 |
| bne | Branch on not equal | I | 2 |
| slt | Set on less than | R | 3 |
| Data Transfer | lw | Load word | I | 2 |
|  | sw | Store word | I | 2 |

Note that the number of instructions is greater than the values that can be represented using 4 bits (0 to 15). One option is to set the Op code of all R-Type logical and arithmetic instructions to 0 and then use the ALU code to specify the operation. **Implementation of *jal* is also optional**. You can get up to 10 points of extra credit for completing it.

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 to 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)

j loop

For the ALU code, the following 8 arithmetic operations should be allowed in the ALU code slot.

|  |  |  |
| --- | --- | --- |
| **Alu code** | **Operation** | **Description** |
| 000 | and | Performs the binary AND operation |
| 001 | or | Performs the binary OR operation |
| 010 | add | Adds the two operands |
| 011 | sub | Subtracts the second operand from the first |
| 100 | xor | Performs the binary XOR operation |
| 101 | slt | Performs a set on less than |
| 110 | shift-left-logical | Shifts left by up to the size of immediate field |
| 111 | shift-right-logical | Shifts right by up to the size of immediate field |

### C. Simulation

In order to demonstrate your implementation, you should create simulation files which use boundary input conditions to test the functionality of components in your processor. This will help you 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.

### Assembler Specifications

In order to create programs for you processor, you will write an assembler in a language of your choice. This assembler can run off a web-browser or be a stand-alone application. It should take an **input file (infile) and produce an output file (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).

If your assembler is a stand-alone program, 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, Python, and PHP).

### Assembly Program

To test your processor, you will write a program that multiplies two 8-bit values. The two values will be entered through the dip switches. The process is as follows. To enter the first value, you set the dip switch bank (FLEX\_SWITCH) to the value that you like. Switch number 8 is the LSB and switch numer 1 is the MSB. You then push FLEX\_PB2 to store this value to a memory location. You then repeat the process to enter the second number but this time store the value to a different memory location. Your program then multiplies these two values using shift-add approach as shown in class. The final result should appear on the two seven-segment displays as a 8-bit HEX value. If overflow occurs, a decimal point should light up.

### 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.

### Grading

You will be graded using the following metrics:

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

### 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 3 | Assembler and first draft of assembly program, ROM and RAM modules |
| April 12 | Register files (from lab), PC, 8-bit ALU, and test cases |
| April 19 | Fetch, decode, memory-mapped I/O, debouncer, and test cases |
| April 26 | Complete Processor, assembler, and assembly program |

### 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 is a sample MIF file to illustrate the MIF format. It also 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.

### 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

1. Map pins

Go to Assign 🡪 Pin/Location/Chip

Add the following pin mappings to your inputs and outputs.

Inputs

1. on-board oscillator (@25.175MHz) is connected to pin 91.
2. push-buttons (active-low):

FLEX\_PB1 28

FLEX\_PB2 29

1. 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

Outputs

1. 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

### 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>