EEL4713 Assignment #2
Spring 2012
Assigned: 9/13/12
In Lab Demo Due Date: 9/20/12
Due: 9/27/12 @ 11:55pm Via Sakai

Overview:
In this assignment you will be writing two programs and building more of the essential components for your MIPS processor. The programs may be written in the language of your choice though some guides for beginning programmers are provided for c, c++, python, and Matlab. The components designed will be essential to the processor's data path and are described in the Laboratory Section. In addition to the report, you will also be giving a demo of two different parts of your laboratory section on the in lab demo due date. You may show any two of the following deliverables: MIPS Assembler, MIPS Disassembler, add32, alu32, alu32control, or registerFile. In the demo, you will show the TA the deliverable for the lab demonstrating the functionality of your program or code. If you cannot get your program to perform properly or your simulation shows your component working incorrectly by the time you declare you are ready for submission, you may receive half-credit for the demo the following week. **IMPORTANT:** You have until the end the lab period to perform your demo for the TA, however, the TA will leave when there is no one else waiting in lab to demo. In short, ***do not show up late to the demo labs***.

In this assignment, you will perform the following tasks:
1.1: Go through a beginners programming tutorial
1.2: Create a table detailing the MIPS ISA
2.1: Write and run the MIPS assembler program on lab3.txt
2.2: Write and run the MIPS disassembler program on lab5.mif
2.3: Build and simulate a 32-bit adder
2.4: Build and simulate an ALU
2.5: Build and simulate an ALU controller
2.6: Build and simulate a register file
3: Review deliverables, prepare for demo, and write report

Section 1: Setup
1.1: Go through a beginners programming tutorial
If you already are comfortable with a particular programming language and have a compiler to program using that language, then you may skip using an online tutorial and jump to the next sub-section. If you have never programmed before or are looking to try out a new language you should go through the guides provided and reference them for code syntax support.

C/C++:
If you have never programmed, C is the recommended language to start. C is old and still used ubiquitously in academia and industry. C++ is similar to C syntactically. but uses an object oriented constructs.
http://www.cprogramming.com/tutorial.html

Matlab:
You should have access to computers with Matlab in the NEB computer room. Matlab is a very quick computational language good mostly for Matrix operations but it can also be used for text parsing as will be done in section 2.1. Though Matlab is incredibly easy to use determining the proper functions to use to properly parse through files may be difficult.
http://www.bumatek.boun.edu.tr/orgnizasyon/download/MATLAB_GUIDE_www.bumatek.boun.edu.tr.pdf (beginners tutorial)
http://www.mathworks.com/help/techdoc/ (Mathworks help documentation)

Python:
Python is a scripting language that is very good for writing code quickly. Python is object oriented and may be confusing if you have never experienced it. Python is open source and has a huge support community.
http://wiki.python.org/moin/BeginnersGuide

1.2: Create table detailing the MIPS ISA

In this part of the setup you will be creating a table that details the structure of the 29 MIPS instructions you will be implementing. Include the following columns: Type (R, J, I), Opcode, function bits (for R-type only), ALU function code (the one used by the ALU controller), ALU function performed, flags set (C, Z, S, V). Your ALU controller and Assembler/Disassembler should match the table you make. The portion of the table relating to ALU control signals should be similar to Figure 5.12 in your textbook that contains all 29 core instructions listed on the front of the green reference sheet (note: Figure 5.12 is based on a 2-bit encoding of ALUop, but you will need to extend the encoding of ALUop to three bits to accommodate the extra instructions)

Section 2: Laboratory

2.1: MIPS Assembler

You will now create an assembler in the language of your choice. Provided with this lab is a f.txt file (linked off of the course schedule), your program must parse through this text file and produce a .mif file. The mif file should look like the following

Sample MIF file: red is for comments, not for the format

```
WIDTH=32; size of your instructions
DEPTH=256; maximum length of your program

ADDRESS_RADIX=HEX; do not change
DATA_RADIX=HEX; do not change

CONTENT BEGIN
        000 : 3c050000;
        001 : 34a50100;
        002 : 34060908;
        003 : 3c1d0000;
        004 : afa50000;
        005 : a7a60004;
        006 : 34060b0a;
        007 : a7a60006;
        008 : 3c1e0000;
        009 : 37de0008;
        00A : 03c07820;
                END;
```

The text file provided will be in a human readable assembly format and should be translated to appear as the above in a hexadecimal format. Your program will not need to be case sensitive and will only need to work for text files provided by me. You can easily write your own text files as long as you follow the format for writing in assembly. This is an example of how you would breakdown an Or immediate instruction.

ori $6, $0, 0x0908

ori = opcode ‘d’ which is 0b001101
$6 is the target register, thus rt = 0b00110
$0 is the source register, thus rs = 0b00000
0x0908 is the immediate field

Your program would recognize this as an I-type instruction and thus would use the breakdown to create a bit-stream

Opcode	Rs	Rt	Immediate field
001101	00000	00110	0000100100001000

Concatenating the different parts will give a 32-bit binary representation of our instruction, which we will then convert to hexadecimal. In this case it turns into 34060908.

In addition your program should also recognize labels for jumping and branching in the code. Labels will be marked with a colon at the end and when called, will use the label name without the colon. For example:

```
ori $6, $0, 0x0908
ori $7, $0, 0x0607
loop:
add $5, $6, $7
beq $6, $6, loop
```

See chapter 2.10 of your textnook to understand how the location of labels in your code will be addressed in your code assembly. Your code's starting location for this assignment is 0x00400000.

Question 2.1.1: breakdown the following instructions into binary tables similar to the example:
(a) OR $4, $5, $2
(b) JR $3
(c) BNE $6, $6, loop (with the loop label two instructions before this instruction)

Question 2.1.2: If the loop label for part (c) of the previous question is 2 instructions after the branch, what is the immediate field value in hex?

Question 2.1.3: What is the furthest distance in number of instructions that can be branched to in the MIPS ISA? How could you branch to locations further away with this limitation?

2.2 MIPS Disassembler
You will now create a disassembler to convert the .mif file Lab5.mif (linked off of the course schedule) into human readable text. There will be two qualifications for your program to be considered to be fully working.

1) The human readable code provided is in the proper format
2) The text file produced can be used by your assembler program to recreate the mif file it was produced from

Your text file should appear as simple text. No header or footer is required. You should use the Lab3.txt file as an example of what the human readable code should look like.

To receive full credit for this lab, you will show the output text file from Lab5.mif and then use that same text file as an input to the assembler created in section 2.1. The output of the assembler should be identical to the input of the disassembler.

Question 2.2.1: If your mif file has 21 instructions with 3 branches that all go to the same location in your code and 1 jump that goes to its own location in a loop, how many lines should your human readable text file be (a label takes up one line)?

Question 2.2.2: Is your data memory byte addressable in your MIPS processor? How can you tell this from the ISA?

VHDL design
You will continue your design of a MIPS datapath. Below are the descriptions of several crucial components needed for datapath construction:

There will be situations in your datapath design where you will need to add two 32-bit data words. To accomplish this design, you will simulate a 32-bit adder called "add32". This component should have two 32-bit input signals called "in0" and "in1" and one 32-bit output word called "sum" that is the sum of the two input words. Compile, simulate, and test your design. Turn in the VHDL entity, architecture, and a

printout of a simulation trace showing the following combinations of input words and a few others of your choosing: (*do not forget to annotate your simulations!*)

- both inputs are 2's complement positive numbers
- both inputs are 2's complement negative numbers
- one input is a 2's complement negative number and the other is a 2's complement positive number
- both inputs are unsigned numbers
- both inputs are 0xFFFFFFFF
- both inputs are 0x00000000
- both inputs are 0x80000000

Next you will design the ALU and ALU control for the single cycle implementation of the MIPS datapath. Consult Figure 4.17 in your textbook to see how these components will fit into your datapath design. Read the ALU control section beginning on page 316 for insight on how your control should operate. Design and simulate your 32-bit ALU called "alu32". The inputs to this component are two 32-bit data words "ia" and "ib", one 4-bit control signal "control", one 5-bit signal containing the shift amount "shamt", and one 1-bit signal containing the shift direction "shdir". The outputs of this component are one 32-bit output word "o", one 1-bit carry flag "C" (only affected during add and subtract ALU operations), one 1-bit zero flag "Z" (asserted when the output word is equal to zero), one 1-bit signed flag "S" (asserted when the 31st bit of output word is equal to one), and one 1-bit overflow flag "V" (only affected during add and subtract ALU operations). Your ALU needs to perform the following operations:

- the sum of "ia" and "ib" when "control" = "0010"
- the difference of "ia" and "ib" when "control" = "0110"
- the logical AND of "ia" and "ib" when "control" = "0000"
- the logical OR of "ia" and "ib" when "control" = "0001"
- the logical NOR of "ia" and "ib" when "control" = "1100"
- the slt operation of signed "ia" and "ib" when "control" = "0111"
- the slt of unsigned "ia" and "ib" when "control" = "1111"
- the logical shift of "ib", in the direction indicated by "shdir" ('0' left and '1' right), and by the amount "shamt" when "control" = "0011"

The shift operation can be implemented in about 10 lines of code if you are familiar with the contents of the ieee.std_logic_arith and ieee.std_logic_unsigned libraries. Compile, simulate, and test your design. The alu32 section of the report should include all of the things discussed in the assignment 1 document including a simulation trace showing the ALU's output for each of the operations listed above.

Design and simulate the control logic for your 32-bit ALU called "alu32control". The control unit takes as inputs the 6-bit function field of a MIPS opcode "func" and 3 bits from the main control unit "ALUop" as discussed in class. The output of the ALU control is the 4-bit "control" signal that is used as an input to the ALU. Once you have created the table in section 1.2 it should be easy to implement a device that outputs the correct ALU control signal based on the "ALUop" and "func" values listed in your table. The alu32control section of the report should include all of the things discussed in the assignment 1 document including a simulation trace showing the controller's output for each entry shown in your table.

A very crucial component of your MIPS datapath will be the register file. At the end of this assignment, I have attached an excerpt from Appendix B in your textbook that describes the register file in detail. After reading the excerpt design, you will simulate a 32-bit register file containing 32 registers that is called "registerFile". The inputs to the register file are as follows. For register file reads, the signals "rr0(0:4)" and "rr1(0:4)" specify two operand registers (i.e., those for which contents should be made available in the outputs "q0(0:31)" and "q1(0:31)"). For register writes, the input "rw(0:4)" specifies which register should be written with the input word "d(0:31)" on the rising edge of the signal clock "clk" and when the write-enable signal "wr" isasserted. Make up your own test cases for simulation. Your design could reuse the "reg32" component from assignment 2 or alternatively the following construct in VHDL:

1. type registerarray is array(X downto 0) of std_logic_vector(X downto 0);
2. signal ra: registerarray;

The first line defines new a type that is an array of std_logic_vector's; the second line initializes a signal of type "registerarray" with name "ra". The defined signal can be used in following manner "ra(0) <= 0xDEADBEE7".

Section 3: Report
Each of the remaining assignments should follow the following format:

1. The report should contain the following sections
   - Cover Page – with your name, date, and assignment number
   - Introduction – briefly comment on the assignment and what it accomplishes
   - Setup – briefly comment on the tools you used in this lab and the tutorials used to prepare you for the work
   - Design and Testing Assembler/Disassembler (programming language of choice):
     - Description of the program and its function
     - Well commented program code (if longer than 2 pages please attach in the appendix)
     - Output of program given the lab3.txt or Lab5.mif file
     - Label answers to questions asked throughout the lab document. (e.g., Question 2.2.1 in the lab document should be Answer 2.2.1 in your report)
   - Design and Testing Components (VHDL):
     - Description of the components and their function. If relevant, provide boolean descriptions.
     - Well commented VHDL code (if longer than 2 pages please attach in the appendix)
     - Tests that were performed to verify that the components work correctly. Provide functional simulation waveforms and make sure to describe the test they represent. In other words: **ANOTATE!**
   - Appendix
2. The report needs to be typed. Annotations should not be scans of hand-drawn annotations and illegible annotations are almost as bad as no annotations.
3. Any waveform that is not labeled and annotated will not be considered as valid.
4. Make sure you use the names for inputs, outputs, and entities as outlined in the assignment. This will be essential for connecting these components together in later assignments.

Grade Breakdown

Section	**Deliverables**	**Percentage of Total grade**
Demo	Demo 2 parts of the lab to the TA by the demo date (5 points each)	10%
1.1	Programming language you are using in this lab and the reason why.	3%
1.2	ISA table	4%
2.1	Program code (3%), output of program (6%), questions (6%)	15%
2.2	Program code (3%), output of program to text (6%), text to mif using output from disassembler (2%), questions (4%)	15%
2.3	VHDL (3%), annotated simulations (7%)	10%
2.4	VHDL (3%), annotated simulations (12%)	15%
2.5	VHDL (3%), annotated simulations (12%)	15%
2.6	VHDL (3%), annotated simulations (7%)	10%
Report	Report follows the format given in section 3 (2%), report is neat and professional (1%)	3%

Register Files

One structure that is central to our datapath is a *register file*. A register file consists of a set of registers that can be read and written by supplying a register number to be accessed. A register file can be implemented with a decoder for each read or write port and an array of registers built from D flip-flops. Because reading a register does not change any state, we need only supply a register number as an input, and the only output will be the data contained in that register. For writing a register we will need three inputs: a register number, the data to write, and a clock that controls the writing into the register. In Chapters 5 and 6, we used a register file that has two read ports and one write port. This register file is drawn as shown in Figure B.8.7. The read ports can be implemented with a pair of multiplexors, each of which is as wide as the number of bits in each register of the register file. Figure B.8.8 shows the implementation of two register read ports for a 32-bit-wide register file.

Implementing the write port is slightly more complex since we can only change the contents of the designated register. We can do this by using a decoder to generate a signal that can be used to determine which register to write. Figure B.8.9 shows how to implement the write port for a register file. It is important to remember that the flip-flop changes state only on the clock edge. In Chapters 5 and 6, we hooked up write signals for the register file explicitly and assumed the clock shown in Figure B.8.9 is attached implicitly.

What happens if the same register is read and written during a clock cycle? Because the write of the register file occurs on the clock edge, the register will be valid during the time it is read, as we saw earlier in Figure B.7.2. The value returned will be the value written in an earlier clock cycle. If we want a read to return the value currently being written, additional logic in the register file or outside of it is needed. Chapter 6 makes extensive use of such logic.

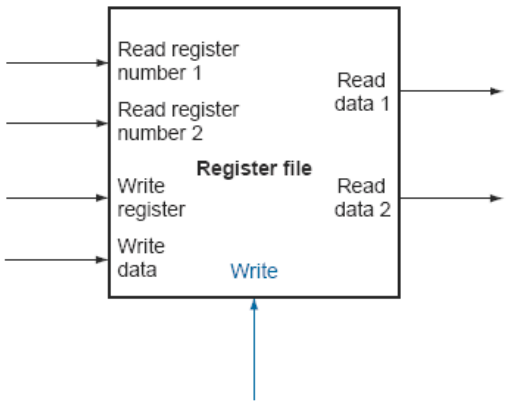

FIGURE B.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

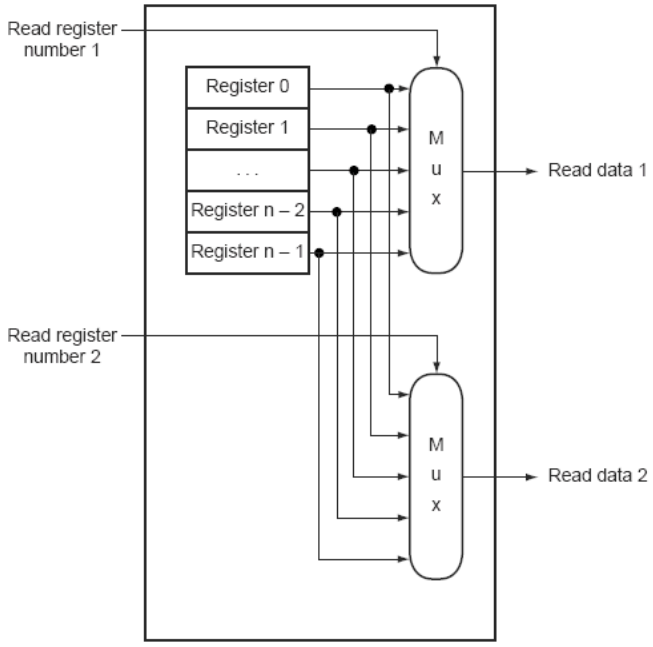

FIGURE B.8.8 The implementation of two read ports for a register file with *n* registers can be done with a pair of *n*-to-1 multiplexors each 32 bits wide. The register read number signal is used as the multiplexor selector signal. Figure B.8.9 shows how the write port is implemented.

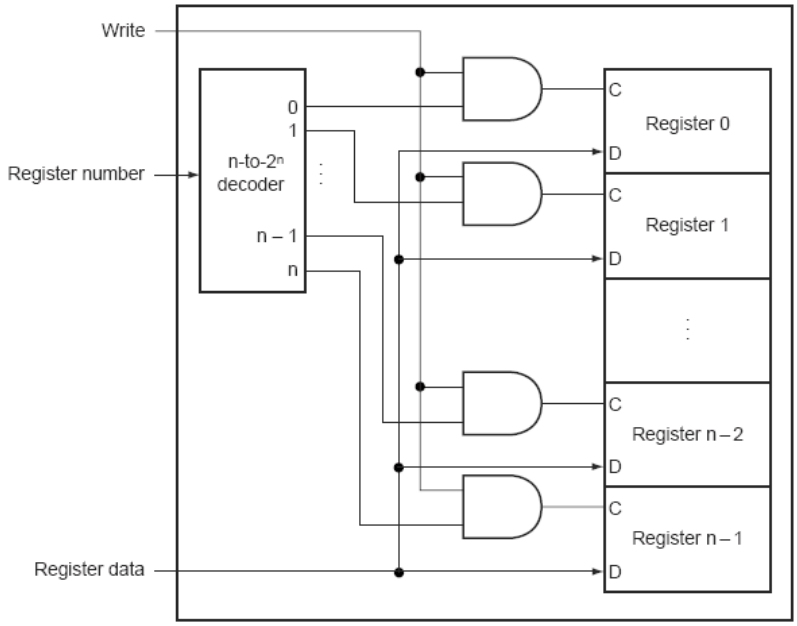

FIGURE B.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the *C* input to the registers. All three inputs (the register number, the data, and the write signal) will have set-up and hold-time constraints that ensure that the correct data is written into the register file.