EEL4713 Assignment #3

Spring 2012
Assigned: 1/26/12
Demo: 2/9/12 in Lab
Due: 2/11/12 @ 11:55pm Via Sakai

Section 1: Setup

There is no setup in this assignment.

Section 2: Textbook Questions

Chapter 4 questions: 4.2.1-3, 4.8.1-3

Section 3: Laboratory

This laboratory will consist of two parts: one on MIPS simulation (3.1) and one on VHDL design (3.2).

(3.1): MIPS simulation

In this part of the assignment, you will use the application "bugspim" in much the same way that you used "spim" from assignment 1. "bugspim" simulates a MIPS processor like "spim" does except that the microprocessor it simulates has a number of bugs that make the execution of certain instructions incorrect. Your task is to identify these bugs by running MIPS assembly code in "bugspim" and verifying whether the behavior of instructions follow the instruction set specifications.

There are a total of 5 instructions with bugs: one ALU operation, two control flow instructions, an instruction that uses immediate values, and a memory access instruction. In your answer, specify: a) which instructions have bugs, b) what is the incorrect behavior you observed, and c) what assembly code you used to determine the incorrect behavior.

Download the BugSPIM simulator with the following command:

```
sudo wget http://www.acis.ufl.edu/~ipop/edu-docs/bugspim
```

If you get a "permission denied" message when attempting to execute bugspim try the command:

```
sudo chmod 755 bugspim
```

Then execute. If you have any problems just post a query on a Sakai discussion group and we will attempt to answer it in a timely fashion.

(Hint: be systematic while searching for the instructions with bugs and use short instruction sequences with step-by-step simulation).

(3.2): 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 and simulate a 32-bit adder called "add32". This component should have two 32-bit input signals called "in0" and "in1", 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: (don't forget to annotate your simulations)

- both inputs are 2's complement positive numbers
- both inputs are 2's complement negative numbers
- one input is 2's complement negative number and the other is 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 device 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 to this device 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 2 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 uses as an input to the ALU. Begin your design by creating a table 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). Once you have created this table 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 2 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 and simulate a 32-register, 32-bit register file 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” is asserted. 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 signal of type “registerarray” with name “ra”. The defined signal can be used in following manner “ra(0) <= 0xDEADBEE7”.

Note: in addition to submitting your reports on Sakai, you must demonstrate your components functionality in Lab on the DEMO date listed on the first page of this Assignment. You also need to submit your design files via Sakai. Please submit a zip file containing all your design files. Failure to submit your design files will cause you to receive a zero on the design sections of the Assignment. Please make sure that you only submit your VHDL code, BDF files, mif files, and/or symbol files and don’t send any other files generated by Quartus (i.e. do not send the entire project).

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.

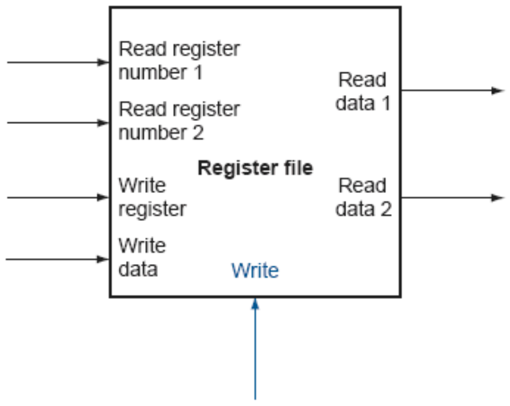

FIGURE B.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

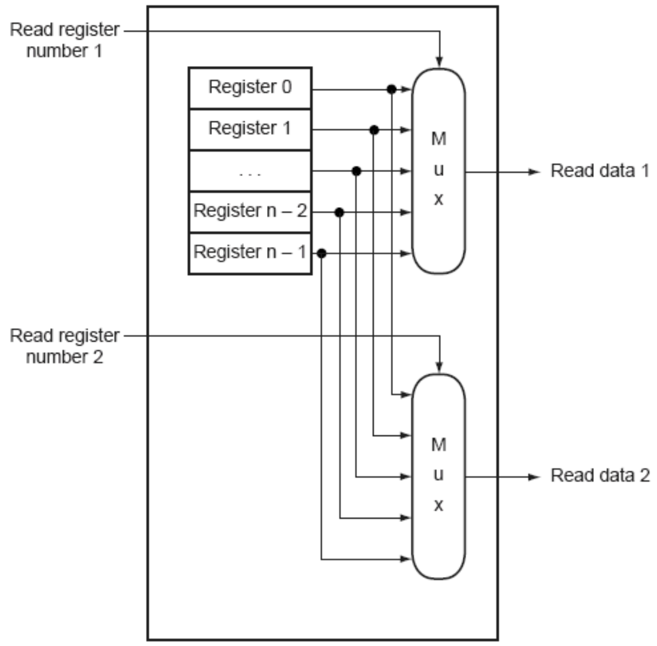

FIGURE B.8.8 The implementation of two read ports for a register file with *n* registers can be done with a pair of *n*-to-1 multiplexors each 32 bits wide. The register read number signal is used as the multiplexor selector signal. Figure B.8.9 shows how the write port is implemented.

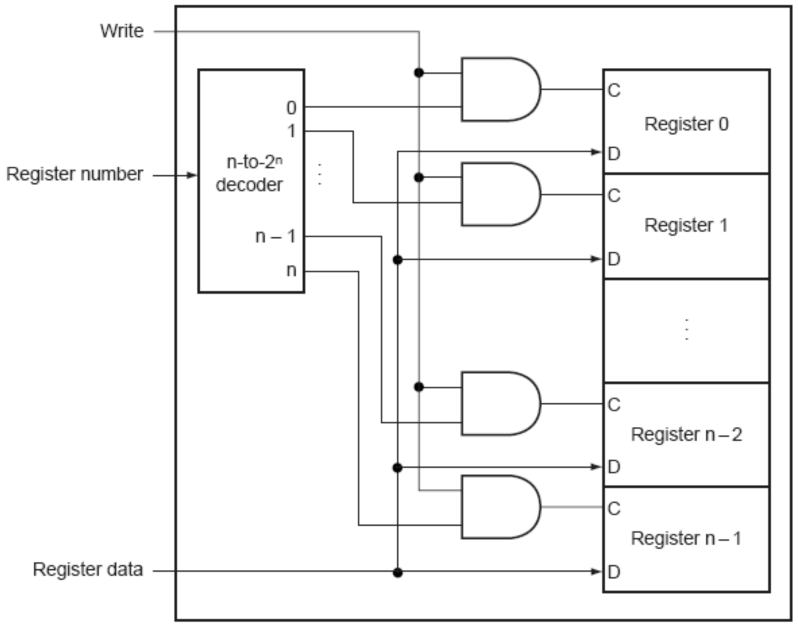

FIGURE B.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the *C* input to the registers. All three inputs (the register number, the data, and the write signal) will have set-up and hold-time constraints that ensure that the correct data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