CS/ECE 552
Spring 2019
Homework 6
Due 11:59 PM Central Time on Wednesday, March 6th, 2019

See below for policy on working with others. The standard late assignment policy applies: you may submit up to 2 days late with a 20% penalty for each late day.

What to Hand In

To submit this assignment, zip or tar your Verilog files together and submit them as a **single file named <netID>-hw6.tgz or <netID>-hw6.zip** on Canvas. Inside this tarball/zip, all files for problem 1 should be a folder called hw6_1 and all files for problem 2 should be inside hw6_2 – you must keep this directory structure. *Also, please do not create a top-level folder above these.* For example, my Net ID is msinclair, so my submission would be called msinclair-hw6.tgz (or msinclair-hw6.zip) and in it would be hw6_1/, hw6_2/, and partners.txt. If you don't have experience with tar, I recommend consulting tutorials such as this one. In addition, before submitting you should **run the Verilog check on all the files** (just the new modules you are writing, you don't need to run it on your testbenches). *You may work both problems with your groupmates for the project and submit a single solution per group. If you choose to work separately, you should submit your own solution. Names must be included in the partners.txt file included in the supplied tar file (on Canvas) and Github Classroom.*

Instead of a schematic, for this assignment you should create a spreadsheet (e.g., in Excel or LibreOffice) detailing what values you give for each of the control signals for each instruction. The spreadsheet should be named **controlLogic.xlsx** and should be placed in the hw6_1 subdirectory.

For the second problem, you should submit all of your .asm files in the hw6_2 subdirectory.

Total Points: 20

As with previous homeworks, part of this homework requires using Verilog. A reminder of some important documents (all available on Canvas):

1. Follow the instructions on ModelSim Setup Tutorial to get your environment setup.
2. Read the Command-line Verilog Simulation Tutorial. Additional references are on Canvas under "Verilog Resources" on the main page.
3. Read the Verilog Cheat sheet and Verilog rules pages. Everything you need to know about Verilog are in these documents.
4. Read the Verilog file naming conventions page and adhere to those conventions. Additional information including a link to the script you can use to make sure your files adhere to these rules are on Canvas, directly under the file naming conventions page.
5. Read the Verilog rules checking page on Canvas and adhere to the conventions. This page also provides information on how you can check that your files conform to these rules.

You should simulate your solutions both to verify the correct function of your designs for yourself. You also have to hand in a copy of the Verilog files (or submit one per group and update partners.txt accordingly), as mentioned above.

Before starting to write any Verilog for the problem that uses Verilog, you should do the following:

1. Break down your design into sub-modules.
2. Define interfaces between these modules.
3. Draw schematics (either by hand, or on a computer) for these modules (these will be handed in as scanned schematic.pdf file).
4. Then start writing Verilog.

Problem 1 [10 points]

Using Verilog, design the control logic for your decoder. As we discussed in class, the job of the control logic is to determine what the appropriate values for each control signal is given an inputted instruction. You will only need combinational logic in this module. The inputs and outputs are:

Inputs:

- *Funct*[1:0] – the function this instruction is performing. From the WISC-SP19 ISA document, you should see that only certain ALU instructions care about the values of the function bits (e.g., ADD and SUB have the same upper 5 bits, and the 2 function bits are used to determine if we are doing an ADD or SUB.
- *OpCode*[4:0] – the opcode for the given instruction. This corresponds to the upper 5 bits (bits 15:11) for each instruction in the WISC-SP19 ISA.

(*Aside: you should think about why it is sufficient to pass in only these bits, instead of all 16 bits of the instruction*)

Outputs:

- *err* – this bit is set to 1 if the control logic encounters an error, 0 otherwise. For now, you can just set this to 1 if any inputs are undefined (this will be useful later in the project).
- *RegDst*[1:0] – these 2 bits are used to determine which register to send to the *writeRegSel* input of the register file. There are 4 possibilities for what to send to *writeRegSel*:
    1. 00 – select instruction bits 4:2
    2. 01 – select instruction bits 7:5
    3. 10 – select instruction bits 10:8
    4. 11 – select register 7 (used for JAL and JALR)
- *SESel*[2:0] – these 3 bits are used to determine how to sign or zero extend the immediate. There are 5 possibilities:
    1. 000 – zero extend the lower 5 bits of the instruction (bits [4:0]) to make it a 16-bit value.
        - This is used for instructions like XORI that have 5-bit, zero extended immediates.
    2. 001 – zero extend the lower 8 bits of the instruction (bits [7:0]) to make it a 16-bit value.
        - This is used for the SLBI instruction which has an 8-bit, zero extended immediate.
    3. 01X – sign extend the lower 5 bits of the instruction (bits [4:0]) to make it a 16-bit value.
        - This is used for instructions like ADDI that have 5-bit, sign extended immediates.
    4. 10X – sign extend the lower 8 bits of the instruction (bits [7:0]) to make it a 16-bit value.
        - This is used for instructions like BEQZ that have 8-bit, sign extended immediates.
    5. 11X – sign extend the lower 11 bits of the instruction (bits [10:0]) to make it a 16-bit value.
        - This is used for instructions like JAL that have 11-bit, sign extended immediates.
- *RegWrite* – this bit is 1 if the instruction writes to a register (e.g., ADDI) and 0 otherwise (e.g., HALT).

- *DMemWrite* – this bit is 1 if the instruction writes to data memory (e.g., ST) and 0 otherwise (e.g., HALT).
- *DMemEn* – this bit is set to 1 if the instruction will read or write from data memory (e.g., ST), and 0 otherwise (e.g., HALT).
- *ALUSrc2* – this bit is set to 1 if the ALU should use the second register read from the register file (e.g., ADD), and 0 if the ALU should use immediate (e.g., ADDI)
- *PCSrc* – this bit is set to 1 if the next PC should use the PC given by the branch/jump (e.g., BEQZ), and 0 if the PC should use PC + 2 (e.g., ADD).
- *MemToReg* – this bit is set to 1 if the WB stage should write back the output of the data memory (e.g., LD), and 0 if the WB stage should write back the output of the ALU (e.g., ADD).
- *DMemDump* – this bit is set to 1 if we want to dump the contents of memory (only useful later in the project), and 0 otherwise. The only case where we want this to happen is if we get a HALT instruction.
- *PCImm* – this bit is set to 1 when we want to the next PC to get the result of a J/JAL, 0 otherwise.
- *Jump* – this bit is set to 1 when we want the next PC to get the result of a JALR/JR, 0 otherwise.

Do not make any changes to the provided control_hier.v file.

Testing instructions

We have provided you a skeleton for the testbench. You should write your own tests for your control logic in this testbench file. You will not be graded on your testbench, but the better coverage you provide in your testbench, more likely it is your design works properly. You can run the testbench in your hw5_1 directory using the command: wsrun.pl control_hier_bench *.v

Problem 2 [10 points]

Develop instruction level tests for your processor. In this problem each group will develop a set of small programs that are meant to test whether your processor implements these instructions correctly. You will write these programs in assembly, run them on an instruction emulator to make sure what you wrote is indeed testing the right thing. The eventual goal is to run these programs on your processor's Verilog implementation and use them to test your implementation.

Info about how to write assembly code and also about how to use the assembler can be found in the Using the assembler page. Details about what each instruction means is available in the ISA specification page.

Each team will be responsible for one randomly assigned instruction (along with common instructions JAL and JALR) and must develop a set of simple programs for that instructions. Each team will also have to write programs for JAL and JALR instructions along their assigned instruction. The table below gives the assignment of instructions to each team (NOTE: there are more instructions than teams, so some instructions are not covered):

Group	Instruction
1	SCO
2	SEQ
3	SLE
4	BGEZ
5	BEQZ
6	ROL

7	SEQ
8	XORI
9	ADD
10	STU
11	XOR
12	BTR
13	SUBI
14	ANDNI
15	SRLI
16	RORI
17	SLT
18	SLL
19	ROR
20	ANDN
21	ST
22	BNEZ
23	SUB
24	SRL
25	LBI

Note: I got the list of teams from Canvas. If you didn't sign up you likely aren't on the list. Please contact me and you'll be added.

To get you started below are two example tests for the add instruction.

add_0.asm

```
lbi r1, 255
lbi r2, 255
add r3, r1, r2
halt
```

add_1.asm

```
lbi r1, 255
lbi r2, 0
add r3, r1, r2
halt
```

You will notice one thing. The add test uses the lbi instruction also! Your goal while writing these tests is to isolate your instruction as much as possible and minimize the use of the other instructions. Identify different corner cases and the common case for your instruction and develop a set of simple test programs.

The work flow we will follow is:

1. Write test in WISC-SP19 assembly language.
2. Assemble using assembler assemble.sh

3. Simulate the test in the simulator and make sure your test is doing what you thought it was doing. Use the simulator: wisccalculator

Read the following documents on how to use to assembler, simulator, and more on how to write tests:

- Using the assembler
- WISC-SP13 Simulator-debugger
- Other example test programs to understand syntax etc.
- Test Programs FAQ

Below is a short demo:

```
prompt% assemble.sh add_0.asm
Created the following files
loadfile_0.img  loadfile_1.img  loadfile_2.img  loadfile_3.img
loadfile_all.img  loadfile.lst

prompt% wiscalculator loadfile_all.img

WISCalculator v1.0
Author Derek Hower (drh5@cs.wisc.edu)
Type "help" for more information

Loading program...
Executing...
lbi r1, -1
INUM:        0 PC: 0x0000 REG: 1 VALUE: 0xffff
lbi r2, -1
INUM:        1 PC: 0x0002 REG: 2 VALUE: 0xffff
add r3, r1, r2
INUM:        2 PC: 0x0004 REG: 3 VALUE: 0xfffe
halt
program halted
INUM:        3 PC: 0x0006
Program Finished

prompt%
```

The simulator will print a trace of each instruction along with the state of the relevant registers. You should examine these to make sure that your test is indeed doing what is expected.

What you need to do:

- Write a set of tests for your instruction. Name them <instruction>_[0,1,2,3,...].asm
- Use your discretion to decide how many tests you need
- Identify corner cases. Think about possible bugs in the hardware.
- Tests should be short and target specific cases, NOT all cases at once.
- Limit the number of other instructions used besides what you're testing. If the test fails it should ideally be from the instruction you're testing and not another that was used.

- In addition to your assigned instruction, everyone must write tests for the **JAL** and **JALR** instruction
- Write comments in your assembly code explain what the test is doing (comments use "//" for our assembler)
- Make sure that all of your assembly files will assemble. **You won't get credit for malformed assembly.**
- The goal of this problem is to make sure you understand the ISA and develop targeted tests for the hardware. Understanding the ISA is required (and extremely helpful!) before building hardware for it!