CS224 – Spring 2020 - Lab #4 v6

MIPS Single-Cycle Datapath and Controller

Dates: **All Sections** Submission Deadline 15/04/2020, Wednesday at 1:30 PM

Online Submission Only

No Prelab!

No late submission!

Part 1 is Mandatory, Part 2 is Optional!

Purpose: In this lab you will use the digital design engineering tools (System Verilog HDL, Xilinx Vivado) to modify the single-cycle MIPS processor. You will expand the instruction set of the "MIPS-lite" processor by adding new instructions to it. To do this, you must first determine the RTL expressions of the new instructions then modify the datapath and control unit of the MIPS. To implement the new instructions will require you to modify some System Verilog modules in the HDL model of the processor. To test and prove correctness, you will simulate the microarchitecture

Summary

Part 1 (100 points): System Verilog model for Original10 + New instructions. You will submit a PDF report plus written code pieces for this part. There will be separate assignments for each Section in Unilica to submit your PDF and your code.

Part 2 (Optional): Simulation of the MIPS-lite processor and Implementation and Testing of new instructions. You will NOT submit your code for this part.

If we suspect that there is cheating, we will send the work with the names of the students to the university disciplinary committee.

[REMINDER!] In this lab and the next one, we assume SystemVerilog knowledge, as well as the ability to use tools such as Xilinx Vivado, since you all took CS223 – Digital Design. If you are not familiar with these, please get used to them as soon as possible.

Part 1. (100 points)

As part of the Part 1 you will prepare a report that contains the following 7 items, one per page. Also you will submit your codes along with your PDF.

a) Cover page, with university name, department name, and course name and number at the top, "Design Report", Lab # (e.g. 4), Section #, and your name and ID# in the middle, and the date of your lab at the bottom.

b) **[15 points]** Determine the assembly language equivalent of the machine codes given in the imem module in the "Complete MIPS model.txt" file posted on Unilica for this lab. In the given System Verilog module for imem, the hex values are the MIPS machine language instructions for a small test program. Dis-assemble these codes into the equivalent assembly language instructions and give a 3-column table for the program, with one line per instruction, containing its location, machine instruction (in hex) and its assembly language equivalent. [Note: you may dis-assembly by hand or use a program tool like the one in Unilica.]

c) **[15 points]** Register Transfer Level (RTL) expressions for each of the new instructions that you are adding (see list below for your section), including the fetch and the updating of the PC.

d) **[20 points]** Make any additions or changes to the datapath which are needed in order to make the RTLs for the instructions possible. The base datapath should be in black, with changes marked in red and other colors (one color per new instruction).

e) **[25 points]** Make a new row in the main control table for each new instruction being added, and if necessary, add new columns for any new control signals that are needed (input or output). Be sure to completely fill in the table—all values must be specified. If any changes are needed in the ALU decoder table, give this table in its new form (with new rows, columns, etc). The base table should be in black, with changes marked in red and other colors. {Note: if you need new ALUOp bits to encode new values, you should also give a new version of Table 7.1, showing the new encodings}

f) **[25 points]** Write a test program in MIPS assembly language, that will show whether the new instructions are working or not, and that will confirm that all existing old instructions still continue to work. Don't use any pseudo-instructions; use only real MIPS instructions that will be recognized by the new control unit.

Table of instructions to implement-- by section

Section	MIPS instructions
Sec 1	Base: Original10 New: “ble”, “subi”
Sec 2	Base: Original10 New: “nop”, “sw+”
Sec 3	Base: Original10 New: “bge”, “swapRM”
Sec 4	Base: Original10 New: “jm”, ll/sc OR “bgt”
Sec 5	Base: Original10 New: “jalm”, lui
Sec 6	Base: Original10 New: jalr, “push”

The Original10 instructions in “MIPS-lite” are add, sub, and, or, slt, lw, sw, beq, addi and j.

Instructions in quotes (e.g. ”push”) are not defined in the MIPS instruction set. They don’t exist in any MIPS documentation; they are completely new to MIPS. You will create them, according to the definitions below, then implement them.

bge, ble, bgt: these I-type instructions do what you would expect—branch to the target address, if the condition is met. Otherwise, the branch is not taken. Example: bge $t2, $t7, TopLoop

jalm: : this I-type instruction is a jump, to the address stored in the memory location indicated in the standard way. But it also puts the return address into the register specified. Example: jalm $t5, 40($s3)

jm: this I-type instruction is a jump, to the address stored in the memory location indicated in the standard way. Example: jm 40($s3)

nop: this I-type instruction does nothing, changes no values, takes one clock cycle. Except for the opcode (which you need to assign a code to), the I-type instruction field values are irrelevant. Example: nop
{Note: an R-type nop exists in real MIPS, it is sll $0, $0, 0. But the nop here is different, so it is “nop” !}

push: this I-type instruction does what you would expect—push a register value onto stack. Example: push $a3 {Note: the assembler automatically puts 29 in the rs field, and 0 into the immediate field, of the machine code instruction.}

subi: this I-type instruction subtracts, using a sign-extended immediate value. subi $t2, $t7, 4

swapRM: this I-type instruction exchanges the data in two locations: the exchange is between a register value and a memory value (addressed in the standard way). Example: swapRM $v0, 1004($sp)

sw+: this I-type instruction does the normal store, as expected, plus an increment (by 4, since it is a word transfer) of the base address in RF[rs]. {Note: these kind of auto-increment instructions are useful when moving through an array of data words.}

Part 2: Simulation and Implementation (Optional)

a) Complete the System Verilog model of single-cycle MIPS by designing a 32-bit ALU module (one is partly specified already in "Complete MIPS model.txt") and save this ALU module by itself in a new file with a meaningful name. Make this file the basis of a new Xilinx Vivado project. In simulation, check its syntax, and then simulate this ALU, using a testbench that you will write. When you are sure that the 32-bit ALU is working correctly in simulation, you can now use it in the MIPS-lite datapath. When you have integrated your working 32-bit ALU into the "Complete MIPS model.txt" file, you are ready to simulate the MIPS-lite single-cycle processor in Xilinx Vivado.

b) Make a New Project, giving it a meaningful name, for your single-cycle MIPS-lite. Do Add Source for the System Verilog modules given in "Complete MIPS model.txt" (modified with your working ALU), and Save everything (you don't have to use Complete MIPS model.txt, you can write your code if you find it more convenient). Make the necessary changes in order to support newly added instructions.

c) Study the small test program loaded into instruction memory (in the imem module) that you dis-assembled in part b) of your Preliminary Design Report. What is the program attempting to do? Extend the imem module so that it will also include newly added instructions as well.

d) Now make a System Verilog testbench file and using Xilinx Vivado, simulate your MIPS-lite processor executing the test program. Study the results given in the simulation window. Find each instruction, and understand its values. Why is writedata undefined for some of the early instructions in the program?

e) Now modify the simulation, in order to show more information. Make changes to the System Verilog modules as needed so that the 32-bit values of PC and the Instruction are made to be outputs of the top-level module. Then modify the testbench file, so that they are displayed in the simulation.

Part 3. Submit your code for MOSS similarity testing

Combine all the new and modified System Verilog codes into a file called **StudentID_StudentFirstName_StudentLastName_SecNo_LabNo_LAB.txt in Part 1.** Add the System Verilog code of the testbench module used for your simulation in Part 1), plus the top-level module file that you used for Part 1), in their final form. Also, add all the Verilog modules whose code is new or modified from "Complete MIPS model.txt". DO NOT include any unmodified System Verilog modules, only those whose code you changed or created. For this part make sure that you submit a txt file, other formats get 0.

Submit your MIPS codes for similarity testing to the Unilica > Assignment specific for your section. You will upload one file, named **StudentID_StudentFirstName_StudentLastName_SecNo_LabNo_LAB.txt,** containing the programs described in the above paragraph. Be sure that the file contains exactly and only the codes which are specifically detailed in Part 1. Check the specifications! *Even if you didn't finish, or didn't get the System Verilog codes working correctly, you must submit your code to the Unilica*

Assignment for similarity checking. Failure to submit your codes will result in a lab score of 0. Your codes will be compared against all the other codes in all sections of the course, by the MOSS program, to determine how similar it is, as an indication of plagiarism. **So be sure that the code you submit is code that you actually wrote yourself!** [Warning: DON'T use code provided by another student, or found somewhere on the internet, etc, since MOSS will determine that yours is similar to theirs!] All students must upload their Part 1 code to Unilica > Assignment

Also, for the report for Part 1 you will upload one file:
StudentID_StudentFirstName_StudentLastName_SecNo_LabNo_Report.pdf
We use MOSS testing only for code submissions.

LAB POLICIES

1. Students will earn their own individual lab grade. The TAs may ask you questions about your submission.
2. Lab score will be reduced to 0 if the code is not submitted for similarity testing, or if it is plagiarized. MOSS-testing will be done, to determine similarity rates. Trivial changes to code will not hide plagiarism from MOSS—the algorithm is quite sophisticated and powerful. Please also note that obviously you should not use any program available on the web, or in a book, etc. since MOSS will find it. The use of the ideas we discussed in the classroom is not a problem.