CS-UY 2214 — Homework 4

Introduction

Submit your work on Gradescope. Submit your work in files named as specified in the questions.

Questions in this homework require you to read and understand Verilog source code. We haven't discussed all elements of Verilog syntax in class, but there are extensive resources available. Your first stop should be the Verilog cheat sheet posted on Classes. Other reference materials are available on online. The TAs can also help you read the code.

Problems

1. Write your answer to this question in a plain text file named `hw4.txt`. Label each answer with the appropriate question number.

   Download `e15_normal.zip` from Classes. Examine the source code for the E15 processor contained in the zip file. Base your answer to the following question on that source code.

   Consider the register `pcIncr`.

   (a) What is the `pcIncr` register used for? What is done with the value stored in it?

   (b) What is the value of the `pcIncr` register for each of the following instructions? `cmpi`, `jmp`, `jz`

   Examine the source code for the E15 processor. There are two ALUs.

   (c) What are the names given to the two ALUs?

   (d) What is the purpose of each of the ALUs? That is, what data are they operating on?

2. As a developer of software for the E15 processor, you are often frustrated by some of its design limitations. It has technical limits that make it infeasible for executing certain kinds of software. For example, try to imagine writing a web browser for the E15.

   In particular, consider this hardware limitation: the instruction store for E15 programs (i.e. the `myROM` array) is limited to a maximum of sixteen instructions. This means that it is impossible to write a program containing more than sixteen instructions. Obviously, this limitation means that many programs simply cannot be written for E15 hardware.

   Your task is to design a new processor (the "E15 Turbo"), based on the E15, that has a capacity for up to 1024 instructions. Starting with the E15 design provided to you, you will propose changes to accommodate the larger instruction store. This modification requires extensive modification to various parts of the processor. For each of the following areas of the processor, discuss what changes will be necessary. If no changes are needed to that part of the processor, say so, and explain why.

   There may be multiple possible answers for each part of the question, depending on what approach you take. You may consult the E15 processor's source code to help you. You may also cite the source code or use Verilog code to describe particular changes in your answer. For example, your answer may be in the form of a proposed "before and after" comparison of code. In all answers, credit will be given

for a thorough answer that demonstrates an understanding of the consequences of a given change, both its pros and its cons. Your answer must include a justification of your proposal.

This is a *design* question, not a *coding* question. Although you may use Verilog code to illustrate key facets of your design proposal, you are not required to do so, and you should *not* submit a complete source code implementation of your proposal. The only work required is English-language answers to the following questions.

(a) What changes will need to be made to the program counter (`pc`)? Why?

(b) Currently, the instruction format is 12 bits: four bits for the opcode, two bits for the source register, two bits for the destination register, and four bits for the immediate. What changes will be needed to the instruction format? Why? Consider all instructions in your answer.

(c) What changes will need to be made to the general-purpose registers (Rg0..Rg3)? Why?

(d) What changes will be needed to the instruction store (`myROM`)? Why?

(e) Currently, both ALUs use 4-bit inputs and a 4-bit output. What changes will be needed to the ALUs? Why?

3. Download `e15_normal.zip` from Classes. It contains the E15 processor source code, a test bench, and some simple E15 programs.

   Write a program in E15 assembly language in a file named `test1.v`. The first three instructions of your program must be these:

```
/*           OPCODE SRC  DST  IMMDATA */
myROM[0]  = {movi, RXX, Rg0, 4'b0011 };
myROM[1]  = {movi, RXX, Rg1, 4'b0110 };
myROM[2]  = {movi, RXX, Rg2, 4'b0001 };
```

   Complete the program so that it calculates the value of $2 * Rg0 + Rg1 - Rg2$. The result should be stored in Rg0. Once the final value is in Rg0, your program should execute the "jump-to-itself" instruction, which is used to signify the end of useful work.

   You'll need to modify the provided `E15Process.v` file in order to make it load your assembly program. Look for the line `` `include "program1.v" `` and change the filename to match that of your program (`test1.v`). Do not change the file `E15Process.v` in any other way. Test your code by compiling and running the `E15Process_tb.v` test bench and examining its output. Do not submit your modified `E15Process.v` or the test bench. Submit only your E15 assembly language file `test1.v`.

4. Although the E15 processor has an `add` opcode, it does not have the built-in ability to multiply numbers. One way to fix that is to provide a *software* solution: in this case, we want to build a program, written in E15 assembly language, that can multiply numbers, using only the limited instructions available in the hardware.

   Download `e15_normal.zip` from Classes. It contains the E15 processor source code, a test bench, and some simple E15 programs.

   Write a program in E15 assembly in a file named `multiplier.v`. Your program should be able to multiply any two 4-bit unsigned numbers in Rg0 and Rg1, storing the result in Rg2. Assume that the product fits in 4 bits. Assume that the inputs are positive.

   Let's initially assume that we want to multiply the decimal values 7 and 2. Use ROM location 0 to store 7 into Rg0 with the `movi` instruction; then use ROM location 1 to store the value 2 into Rg1.

   The remainder of the program, beginning at ROM location 2, should perform the multiplication of the value of Rg0 by the value of Rg1, and store the result into Rg2 before halting. Note that your program should work with an arbitrary multiplicand and multiplier, but for testing and grading purposes, make sure that your submitted code is for the specific values 7 and 2.

Hint: you will need a loop to repeatedly add one register to another.

You'll need to modify the provided `E15Process.v` file in order to make it load your assembly program. Look for the line `` `include "program1.v" `` and change the filename to match that of your program (`multiplier.v`). Do not change the file `E15Process.v` in any other way. Test your code by compiling and running the `E15Process_tb.v` test bench and examining its output. Do not submit your modified `E15Process.v` or the test bench. Submit only your E15 assembly language file `multiplier.v`.

5. In the previous question, we "taught" the E15 processor to do multiplication with a software solution. Another approach is a *hardware* solution: that is, extend the E15's instruction set to support a new multiplication instruction. We'd like to be able to write code like this:

```
/*           OPCODE SRC  DST  IMMDATA */
myROM[0]  = {movi , RXX , Rg0 , 4'b0111};
myROM[1]  = {movi , RXX , Rg1 , 4'b0010};
myROM[2]  = {mul  , Rg1 , Rg0 , 4'b0000};
myROM[3]  = {muli , RXX , Rg1 , 4'b0011};
myROM[4]  = {jmp  , RXX , RXX , 4'b0000};
```

This code uses the new instructions `mul` (multiply) and `muli` (multiply immediate). If this program works correctly, the final value of Rg0 should be 14 (1110_2) and the final value of Rg1 should be 6 (0110_2). Our goal now is to implement these new instructions in the E15 processor.

In the following procedure, I will walk you through the steps necessary to implement the new opcodes. Follow these steps closely, and make the indicated changes to the code.

(a) Download `hw4.zip` and extract the provided files:

- `E15Process_mul.v` – starter code for a modified version of the E15 that supports the new opcodes; right now, this file is incomplete.
- `E15Process_mul_tb.v` – a test bench for the modified E15 processor; this is actually completely identical to the usual test bench, except for the name of the processor source file that it `` `include ``s.
- `program1.v` – the E15 program shown above: a simple test of the new instructions.

Read `program1.v`. Then, try to compile the test bench. Compilation will fail: the program (`program1.v`) uses new opcodes (`mul` and `muli`) that haven't been defined yet in the processor. In the following steps, we'll fully define the new opcodes, so that we will be able to compile and run the test bench.

(b) Look at `E15Process_mul.v`. This is the same as the "usual" E15 processor, with the difference that I've introduced a more advanced ALU that can do multiplication, in addition to addition and subtraction. Read the definition of the new `fancyALU` module, starting on line 141. Notice also that the old 1-bit `addNotSub` input has been replaced with a 2-bit `operation` input. Make sure you understand the new ALU module.

(c) Let's add the new opcodes. Around line 15 of `E15Process_mul.v`, you'll see a `parameter` declaration, assigning 4-bit numeric values to each opcode. Add entries for the two new opcodes, `mul` and `muli`. You can give them any 4-bit numeric value, as long as that values isn't already used by another opcode. (Try to add the new declarations on an existing line: if you insert a new line, it will mess up the line numbers used in the rest of this procedure.)

(d) Look at the `fetch` stage, starting around line 55. We don't need to change anything here.

(e) Look at the `decode` stage, starting around line 61. We added a new immediate opcode, so we need to make sure that the immediate field is put on the mBus when we decode `muli`. Modify the `case` clause on line 64 to include `muli`, in addition to the other immediate opcodes.

In the next `case` clause, on line 68, we handle non-immediate opcodes. Add `mul` to this clause, so that the source register will be put on the mBus when we decode `mul`.

The `decode` stage also needs to set the function selector for the ALU. This happens on line 84, where we assign to the register `aluOperation`. Right now, this statement assumes that there are only two possibilities: add and subtract. Modify this statement so that it correctly sets the ALU function selector according to the desired operation: addition, subtraction (including comparison), or multiplication. You may want to use use the `?:` operator. Note that if the current instruction isn't an arithmetic operation, it doesn't matter what the value of the ALU function selector is.

(f) Look at the `exec` stage, starting around line 88. Here we collect the output of the ALU and decide how much to advance the program counter. In the first clause, on line 91, we handle all the ALU opcodes. Our two new opcodes will be handled similarly, so add them to this clause.

(g) Look at the `store` stage, staring around line 106. Just like in the previous stage, we need to add the two new opcodes to the list of ALU opcodes (excluding `cmpi` and `cmpi`), so that the ALU result will be stored into the proper register. Add them to the clause on line 109.

That's it! Compile and run the test bench `E15Process_mul_tb.v` with these commands:

```
iverilog -o E15Process_mul_tb.vvp E15Process_mul_tb.v
vvp E15Process_mul_tb.vvp
```

If you've made the changes correctly, you should see the expected results from the program (i.e. `r0=14` and `r1= 6`). Check the final results of the registers to make sure they match what you expect.

Write your answers to the following questions in a plain text file named `hw4.txt`. Label each answer with the appropriate question number. Base your answers on the modified E15 processor we developed in this problem.

i) Why did we need to replace the 1-bit operation selector of the ALU with a 2-bit operation selector? What specific values can this input have and what is their significance?

ii) What would happen if we didn't modify the `store` stage for our new opcodes? Would the multiplication operation still be be performed?

In addition to answers to the above questions, submit your complete, modified `E15Process_mul.v`. Do not submit the test bench or the test program.