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`

   Answer: This register stores how much to add to the pc (program counter) register at the end of thr current instruction. Its value is determined by the kind of instruction: for jmp instructions, its value is taken from the ImmData field; or jz and jnz instructions, its value will be either 1 or taken from ImmData; and for all other instructions it's just 1. When the value is 1, it just means that the subsequently executed instruction is the next one in sequence.

   The value of the register is set in the execute phase. Normal instructions (like cmpi) set it to 1. jmp sets it to the value in the immediate field. jz sets it either to 1 or to the immediate, depending on the zero flag.

   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?

   Answer: One ALU is for updating the program counter, the other is for arithmetic operations on registers. Their names are dataALU, pcALU. We need two because we update the program counter concurrently with data calculations, since it's faster to do them at the same time.

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?

Answer: For all parts of this question, students must propose a change and explain its necessity, or why no change is needed to fulfill the stated requirement. Short answers with no justification are not to be given credit. Having said that, there are multiple possible ways to approach this challenge, and as long as the student proposes a reasonable solution and a justification, credit may be given.

For this part, an appropriate answer is: the program counter will need to be expanded to 10 bits, in order to accommodate values from 0 to 1023, since $2^{10} = 1024$. The program counter must be able to uniquely identify a position in the code store, and therefore needs as many values as there are instructions.

(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.

Answer: The instruction format will need to be expanded in order to accommodate bigger jumps. That is, the immediate field will need to be 10 bits, so that a jump instruction can target any location in the instruction store. So the total size will be 18-bits.

Alternatively, students may argue that the instruction format does not need to change, under the reasoning that jumps are relative: as long as we only jump to within 4 bits of the current location, we don't need a bigger immediate fied. This is an okay solution, as long as the students justify their answers.

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

Answer: Students may rightly raise the issue: if we change the size of the immediate field, what then of the registers? What will movi do if we add a 10-bit number (from the immediate) into a 4-bit register? They should discuss this issue, and there are various possible solutions. One solution would be to discard the excess bits when moving data from immediate field to a register (in which case no change is necessary to the register file). Another solution would be to expand the registers to 10 bits as well. Expanding the registers to 10 bits does not require further changes to the instruction format, as there will still only be four registers.

If students chose not to expand the immediate field, then the answer to this question should reflect that.

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

Answer: The instruction store must change in two dimensions: first, it must accomodate the larger size of each instruction (18 versus 12 bit) and it must accommodate more instructions (1024 versus 16). So in the end, we might have something like this:

reg [17:0] myROM [1023:0];

If students chose not to expand the immediate field, then the answer to this question should reflect that.

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

Answer: There is a dedicated pcALU just for updating the program counter. Since we expand the program counter, we will also need to expand the pcALU to 10 bits.

The data ALU does not need to change unless students decided to expand the size of the general purpose registers. You can deduct a point if the student incorrectly states that both ALUs need to be expanded.

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`.

Answer: One possible solution:

```
/*           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 };
myROM[3]  = {add,  Rg0, Rg0, 4'b0000 };
myROM[4]  = {add,  Rg0, Rg1, 4'b0000 };
myROM[5]  = {sub,  Rg2, Rg1, 4'b0000 };
myROM[6]  = {mov,  Rg1, Rg0, 4'b0000 };
myROM[7]  = {jmp,  RXX, RXX, 4'b0000 };
```

The goal is to produce a program capable of solving the general equation stated, not only with the specific values given. Obviously, student's solution must refer only to the registers Rg0, Rg1, Rg2, not the specific values 0011, 0110, 0001. The instruction `movi, RXX, Rg0, 4'b1011`, although it puts the correct answer in the correct register, is not a solution to this question.

Test the student's solution using the E15 source code, the E15 test bench, and the Verilog simulator.

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`.

   Answer: One possible solution:

```
/*           OPCODE SRC  DST  IMMDATA */
myROM[0]  = {movi, RXX, Rg0, 4'b0111 };
myROM[1]  = {movi, RXX, Rg1, 4'b0010 };
myROM[2]  = {movi, RXX, Rg2, 4'b0000 };
myROM[3]  = {cmpi, RXX, Rg0, 4'b0000 };
myROM[4]  = {jz,   RXX, RXX, 4'b0100 };
myROM[5]  = {add,  Rg1, Rg2, 4'b0000 };
myROM[6]  = {subi, RXX, Rg0, 4'b0001 };
myROM[7]  = {jnz,  RXX, RXX, 4'b1110 };
myROM[8]  = {jmp,  RXX, RXX, 4'b0000};
myROM[9]  = {jmp,  RXX, RXX, 4'b0000};
myROM[10] = {jmp,  RXX, RXX, 4'b0000};
myROM[11] = {jmp,  RXX, RXX, 4'b0000};
myROM[12] = {jmp,  RXX, RXX, 4'b0000};
myROM[13] = {jmp,  RXX, RXX, 4'b0000};
myROM[14] = {jmp,  RXX, RXX, 4'b0000};
myROM[15] = {jmp,  RXX, RXX, 4'b0000};
```

   Run the student's solution in the Verilog simulator in conjunction with the provided E15Process.v and E15Proces_tb.v files. You can modify the ImmData field of the first two instructions to test it with various inputs. Make sure the student included at least one "halt" instruction (`{jmp, RXX, RXX, 4'b0000}`) so their program ends.

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?

   Answer: Before, we only had two operations, so 1-bit was enough. Now we have three operations (+, -, *) so we need at least two bits to represent all possibilities. The legal values are similar to before: 1 means add, 0 means subtract, and any other values (2 or 3) means multiply.

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

   Answer: If left in its original state, the store stage wouldn't store the ALU result into the destination register. The multiplication would be performed, but the product would be discarded.

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.

Answer: My complete solution is below. Student code should be very close to this, since I walk them through each change. Please do some basic sanity checking on their code: make sure that the numeric values they assign to the new opcodes don't overlap with old opcodes, and run the test bench to make sure it gets the right answer.

Also please make sure that the assignment to aluOperation is correct: it should be 1 when the opcode is add or addi, 2 or 3 when the opcode is mul or muli, and 0 when the opcode is sub, subi, cmp, or cmpi. For any other opcode, we don't care about the value aluOperation. My implementation below shows possible approach.

```
module E15Process(input clk,
                  output reg [3:0] pc,             // Program Counter
                  output reg [3:0] opCode,         // op code
                  output reg [1:0] src, dst,       // src and dst register
                  output reg [3:0] immData,        // "Immediate" data
                  output reg [3:0] r0, r1, r2, r3, // Registers
                  output tri [3:0] mBus, dstBus    // Buses
                  );

   parameter fetch=2'd0, decode=2'd1, exec=2'd2, store=2'd3;

   parameter Rg0 = 2'b00, Rg1 = 2'b01, Rg2 = 2'b10, Rg3 = 2'b11,
             RXX = 2'b00;

   parameter
     jmp  = 4'b0000, jz   = 4'b0010,  jnz = 4'b0011,
```

```
  movi = 4'b1001, mov  = 4'b1000,
  addi = 4'b1011, add  = 4'b1010,
  subi = 4'b1101, sub  = 4'b1100,
  cmpi = 4'b1111, cmp  = 4'b1110,

  mul = 4'b0001, muli = 4'b0101
  ;

parameter   hiZ = 4'bz;

parameter   bEn_R0 = 3'd0, bEn_R1 = 3'd1,
  bEn_R2 = 3'd2, bEn_R3 = 3'd3,
  bEn_ALU = 3'd4, bEn_Imm = 3'd5;

reg  [3:0] pcIncr; // Program Counter Increment
wire [3:0] pcRes;  // Output of pc ALU
wire       pcz;    // Unused - zero flag for pc ALU

reg [1:0]  myState;      // State (phase of excution)
reg [11:0] myROM [15:0]; // ROM (holds program)
reg [3:0]  mbEn, dbEn;   // Used to determine which tri-state buffers are
                         // enabled for master bus and destination bus.

reg [1:0]  aluOperation; // Determines if ALU adds or subtracts or multiplies
reg [3:0]  aluOut;    // Register to hold output of main ALU
reg        zFlag;     // Zero flag
wire [3:0] resVal;    // Output (combinational) of ALU
wire       zVal;      // Output (combinational) of ALU zero flag

initial
  begin

     `include "program1.v"

     pc = 4'b0000;
     myState = fetch;

  end

always @(posedge clk)
  case(myState)
    fetch:
      begin
         {opCode, src, dst, immData} = myROM[pc];
         myState <= decode;
      end

    decode:
      begin
         case(opCode)
           movi, addi, subi, cmpi, muli:
```

```
          begin
             mbEn <= bEn_Imm;
          end
        mov, add, sub, cmp, mul:
          case(src)
            Rg0: mbEn <= bEn_R0;
            Rg1: mbEn <= bEn_R1;
            Rg2: mbEn <= bEn_R2;
            Rg3: mbEn <= bEn_R3;
          endcase
      endcase

      case(dst)
        Rg0: dbEn <= bEn_R0;
        Rg1: dbEn <= bEn_R1;
        Rg2: dbEn <= bEn_R2;
        Rg3: dbEn <= bEn_R3;
      endcase

      aluOperation <= ((opCode == add) | (opCode == addi)) ? 1 :
                      ((opCode == mul) | (opCode == muli)) ? 2 : 0;
      myState <= exec;
  end

exec:
  begin
      case(opCode)
        addi, add, subi, sub, cmpi, cmp, muli, mul:
          begin
             pcIncr <= 4'd1;
             mbEn <= bEn_ALU;
             aluOut <= resVal;
             zFlag <= zVal;
          end
        jmp: pcIncr <= immData;
        jz:  pcIncr <= zFlag ? immData : 4'd1;
        jnz: pcIncr <= zFlag ? 4'd1 : immData;
        default: pcIncr <= 4'd1;
      endcase
      myState <= store;
  end

store:
  begin
      case(opCode)
        movi, mov, add, addi, sub, subi, mul, muli:
          case(dst)
            Rg0: r0 <= mBus;
            Rg1: r1 <= mBus;
            Rg2: r2 <= mBus;
            Rg3: r3 <= mBus;
```

```
                endcase
            endcase
            pc <= pcRes;
            myState <= fetch;
         end

     endcase

   assign mBus = (mbEn==bEn_R0)  ? r0      : hiZ;
   assign mBus = (mbEn==bEn_R1)  ? r1      : hiZ;
   assign mBus = (mbEn==bEn_R2)  ? r2      : hiZ;
   assign mBus = (mbEn==bEn_R3)  ? r3      : hiZ;
   assign mBus = (mbEn==bEn_Imm) ? immData : hiZ;
   assign mBus = (mbEn==bEn_ALU) ? aluOut  : hiZ;

   assign dstBus = (dbEn==bEn_R0)  ? r0 : hiZ;
   assign dstBus = (dbEn==bEn_R1)  ? r1 : hiZ;
   assign dstBus = (dbEn==bEn_R2)  ? r2 : hiZ;
   assign dstBus = (dbEn==bEn_R3)  ? r3 : hiZ;

   fancyALU dataALU(aluOperation, mBus, dstBus, zVal, resVal);
   fancyALU pcALU(2'b01, pc, pcIncr, pcz, pcRes);

endmodule


module fancyALU(
    input [1:0] operation,
    input [3:0] src, dst,
    output zFlag,
    output [3:0] res);
    assign res = (operation==1) ? (dst + src) : (operation==0) ? (dst - src) : (dst*sr
    assign zFlag = !(res);

endmodule
```