CS-UY 2214 — Homework 3

Jeff Epstein

Introduction

Unless otherwise specified, put your answers in a plain text file named `hw3.txt`. Number each answer. Submit your work on Gradescope.

You may consult the E15 cheat sheet, which is available on Brightspace.

Problems

1. Read the following Verilog module. Do not enter it into your computer.

```
module lfsr(R, Reset, Clock, Y);
    input [2:0] R;
    input Reset;
    input Clock;
    output [2:0] Y;
    reg [2:0] Q;
    assign Y = Q;

    always@ (posedge Clock)
        if (Reset)
            Q <= R;
        else
            Q <= {Q[1], Q[0] ^ Q[2], Q[2]};
endmodule
```

   A comment on combinatorial vs sequential Verilog: in *combinatorial* (also known as *continuous*) Verilog, we use `assign` statements to connect wires to gates and other components; the connections are permanent, so any change in the elements in the right-hand side of an `assign` will be immediately reflected in the value of the left-hand side. In *sequential* Verilog, operations are synchronized to the clock, and changes to registers happen only at a positive clock edge. In the above code, we can clearly see the difference: the statement `assign Y = Q` tells us that the value of the register `Q` is continuously output on port `Y`. Conversely, operations within the `always` block are executed on each positive clock edge, that is, when a discrete new input is provided. Therefore, when we assign a new value to `Q`, that change is also reflected in `Y`.

   A comment on the syntax: the curly-brace notation `{Q[1], Q[0] ^ Q[2], Q[2]}` is used here to construct a 3-bit value from three constituent bits, from most to least significant: first `Q[1]`, then `Q[0] ^ Q[2]`, finally `Q[2]`.

   In the above module, if we initialize the module by asserting `Reset` and setting `R` to `100`, what sequence of values will be output on `Y` over the next five clock cycles? Give your answer as a sequence of six binary values, starting with `100`.

2. For each of the following E15 assembly instructions, represent the instruction as a 12-bit binary number. Write all 12 bits. State in prose what the instruction does, and specifically state what register or registers it modifies (including the four general-purpose registers Rg0 through Rg3, the program counter, and the zero flag). Recall that nearly all instructions modify the program counter, even if just to increment it. Also, keep in mind that the arithmetic instructions update the zero flag.

   (a) `{cmp, Rg2, Rg3, 4'b0000}`

   (b) `{add, Rg0, Rg3, 4'b0000}`

   (c) `{jz, RXX, RXX, 4'b0001}`

3. For each of the following 12-bit numbers, interpret the number as an E15 assembly instruction. Write the instruction, including its opcode and arguments, and state in prose what it does, and specifically state what register or registers it modifies, including the zero flag and program counter. If the assembly instruction ignores any of its register operands, use the symbol `RXX` to indicate so.

   (a) 1011 00 01 0011

   (b) 1010 01 01 0011

   (c) 0000 00 00 0000

   (d) 0000 00 00 0001

4. Consider each of the following fragments of E15 code. Each fragment is independent; do not assume a relationship between the fragments. Each fragment represents a part of a larger program, which is not given. Based only on the code provided, determine what is the address (i.e. ROM location) of the instruction that will execute after the last listed instruction. If the address of that instruction cannot be determined from the provided information, say so. Otherwise, give the numeric address in decimal.

   (a)
```
myROM[0] =  {mov, Rg0, Rg3, 4'b0000};
myROM[1] =  {add, Rg2, Rg1, 4'b0000};
myROM[2] =  {movi,RXX, Rg0, 4'b0110};
```

   (b)
```
myROM[6] =  {movi, RXX, Rg0, 4'b1010};
myROM[7] =  {movi, RXX, Rg3, 4'b1001};
myROM[8] =  {subi, RXX, Rg3, 4'b0001};
myROM[9] =  {cmp,  Rg0, Rg3, 4'b0000};
myROM[10] = {jnz,  RXX, RXX, 4'b1011};
```

   (c)
```
myROM[3] =  {movi, RXX, Rg1, 4'b0001};
myROM[4] =  {cmpi, RXX, Rg1, 4'b0010};
myROM[5] =  {jz,   RXX, RXX, 4'b1111};
```

5. Consider the following E15 assembly program.

```
/*          OPCODE SRC  DST  IMMDATA */
myROM[0] =  {jmp,  RXX, RXX, 4'b0010};
myROM[1] =  {jmp,  RXX, RXX, 4'b0000};
myROM[2] =  {movi, RXX, Rg0, 4'b1111};
myROM[3] =  {cmpi, RXX, Rg0, 4'b0000};
myROM[4] =  {jz,   RXX, RXX, 4'b1101};
```

```
myROM[5] =  {movi, RXX, Rg1, 4'b1111};
myROM[6] =  {cmpi, RXX, Rg1, 4'b0000};
myROM[7] =  {jz,   RXX, RXX, 4'b0011};
myROM[8] =  {subi, RXX, Rg1, 4'b0001};
myROM[9] =  {jmp,  RXX, RXX, 4'b1101};
myROM[10] = {subi, RXX, Rg0, 4'b0001};
myROM[11] = {jmp,  RXX, RXX, 4'b1000};
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};
```

(a) When this program is run, how many times will the instruction in ROM cell 8 be executed? Explain.

(b) How many times will the instruction in ROM cell 10 be executed? Explain.

(c) How many times will the instruction in ROM cell 12 be executed? Explain.

(d) How many times will the instruction in ROM cell 1 be executed? Explain.

(e) Write an English sentence explaining what this program does. Your description should be high-level, and should describe its overall effect, not its step-by-step behavior.

6. We've discussed the semantics of E15 assembly language. Now we need a machine that is capable of executing that language.

   Download and extract the file `e15_simple_incomplete.zip`, containing an incomplete implementation of an E15 processor in Verilog. The following files are included:

   - `full_adder_nodelay.v` — A 1-bit full adder, which you've seen before.
   - `4bit_adder.v` — A 4-bit adder, based on the 1-bit adder.
   - `E15Process.v` — The incomplete E15 processor, as well as its ALU.
   - `E15Process_tb.v` — The test bench for the E15 processor. Compile this file with `iverilog`, and run it in the simulator.

   Read `E15Process.v` completely. Note the lines marked with a "`TODO`" comment: your task is to complete these lines by providing an appropriate expression. Do not modify any other code. Do not add any additional components, wires, or registers. A correct implementation will allow your `E15Process.v` to execute any E15 assembly language program.

   In particular, you must provide expressions for the following wires:

   - `pcIncr` — Determines how much the program counter is incremented by. This wire is used as input to the `pcALU`. The value of this depends on the current opcode, as well as the value of the zero flag returned by the `dataALU`.
   - `storeVal` — Indicates the value to be stored into the destination register, for those opcodes that store a value. Depends on the current opcode, as well as the sum returned by the `dataALU`.
   - `operand1` — Indicates the value of the first of two operands to be passed to the `dataALU`. The value depends on the current opcode, the current instruction's `src` field, and may be one of the four register values or the immediate value. If a register, be careful to pass the *content* of the register, not its *name*.
   - `operand2` — Indicates the value of the second of two operands to be passed to the `dataALU`. The value depends on the current instruction's `dst` field, and may be one of the four register values. Be careful to pass the *content* of the register, not its *name*.

In your solution, you may use any wires defined in the E15 implementation, as well as any bitwise operators (`|`, `&`, `^`, ∼), the equality comparison operator (`==`), and the conditional operator (`?:`). Do not use Verilog arithmetic operators (`+`, `*`, `-`, etc).

Two small test E15 assembly language programs, `program1.v` and `program2.v`, are included. You can select which program to run by modifying the `` `include "program1.v" `` line in `E15Process.v`. Each test program also includes a comment indicating the expected result. You should verify the correct function of your E15 processor by comparing each program's actual output against its expected output.

Submit only your complete `E15Process.v`.

7. 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.

   Use your completed E15 processor from the previous exercise to develop and test your code for this exercise.

   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 non-negative.

   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.

   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: when you run the test bench, it will print out the final state of the registers.

   Briefly describe how your solution works in a few sentences.

   Submit only your E15 assembly language file `multiplier.v`. Do not submit your modified `E15Process.v` or the test bench.