Project 2: Little RiSC-16 (10%)

ENEE 350: Computer Organization, Fall 2009
Assigned: Tuesday, Sep 22; Due: Tuesday, Oct 6

Purpose

This project is intended to familiarize you with the gate-level components that make up the *control* and *datapath* elements of a processor. You are to do the following problems and hand in hardcopy.

1. Design a 4-bit ALU

Design a 4-bit ALU that takes two 4-bit signed 2's complement numbers as input (A and B) and produces one 4-bit result and a status bit. The ALU should perform the following functions:

1. **ADD**. Add **A** and **B**, output 4-bit value on **D_DATA**. Use a ripple carry scheme.
2. **SUB**. Subtract **B** from **A**, output 4-bit value on **D_DATA**. Use a ripple carry scheme.
3. **NAND**. Produce on **D_DATA** the 4-bit complement of the bitwise **AND** of **A** with **B**.
4. **LT**. Produce the 4-bit value 1 if **A** < **B**; otherwise, the 4-bit value 0. Output on **D_DATA**.
5. **EQ**. Output on 1-bit status signal **ZERO**. Produce 1 if **A** = **B**, otherwise, produce 0.

The inputs of the ALU should be the following:

1. **A** — a 4-bit value
2. **B** — a 4-bit value
3. **SUB** — carry-in signal to be used for subtraction
4. **ADD/NAND/LT**— 3 bits, only one of which will be asserted at a time (note that you do not need a dedicated line for the **SUB** function)

The outputs of the ALU should be the following:

1. **D_DATA** — the 4-bit result value
2. **ZERO** — A 1-bit signal that is 1 if **A**=**B**, otherwise it is 0

Draw the ALU showing each individual bit-path, using only 1-bit full adders and any additional basic logic gates that you require (e.g. NOT, AND, OR, XOR, and their complements).

2. Design a D-type flip-flop and 4x4-bit register file

Design a D-type flip-flop and show its clock timing for changing inputs. Use the flip-flop to build a 4 x 4-bit register file (i.e. four registers, each 4 bits wide). The register file should have two read ports and one write port; this means that in a given cycle the register file should be able to read out 4-bit values from two (possibly different) registers and write a 4-bit value into a (possibly different) third register. The register file should have the following inputs:

1. **CLK** — a 1-bit clock signal
2. **W_ENABLE** — a 1-bit signal indicating that the register file should allow writes
3. **D_ADDR** — a 2-bit register number indicating the register to be written
4. **D_DATA** — a 4-bit value to be written into register **D_ADDR**
5. **A_ADDR** — a 2-bit register number indicating a register to be read

6. **B_ADDR** — a 2-bit register number indicating a register to be read

The register file should have the following outputs:

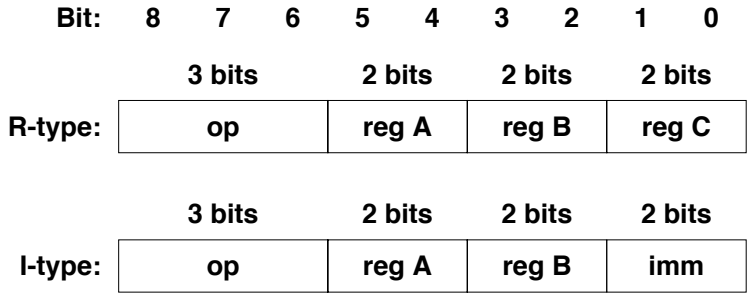
1. **A_DATA** — the 4-bit data value at register **A_ADDR**
2. **B_DATA** — the 4-bit data value at register **B_ADDR**

You should use basic logic gates to implement the decoders and 8-way output multiplexor.

3. Design a simple instruction decoder

Design a decoder that accepts as input a 9-bit signal that encodes the following information:

The 9-bit signal contains 5 pieces of information:



1. **OP** — contains one of five 3-bit values (assume there will be no invalid opcodes):

```
000 — ADD  (R-type)  —  rA <- rB + rC
100 — SUB  (R-type)  —  rA <- rB - rC
001 — NAND (R-type)  —  rA <- rB NAND rC
010 — SLT  (R-type)  —  rA <- 1 if A<B, 0 otherwise
011 — BEQ  (I-type)  —  instruct the ALU to perform the EQ test
```

2. **RegA** — 2-bit register identifier
3. **RegB** — 2-bit register identifier
4. **RegC** — 2-bit register identifier
5. **IMM** — a 2-bit 2's complement *signed* immediate value (this is an important point)

Note that **RegC** and **IMM** overlap each other.

The finite state machine should decode this 9-bit instruction and output the following signals:

1. **W_ENABLE** — enables the updating of the register file
2. **A_ADDR** — 2-bit register number to be read
3. **B_ADDR** — 2-bit register number to be read
4. **D_ADDR** — 2-bit register number to be written if **W_ENABLE** is asserted
5. **SUB** — 1-bit carry-in subtraction signal to the ALU
6. **ADD/NAND/LT**— 3-bit signal indicating the desired ALU operation

4. Design a simple processor

At this point, it should be fairly clear that you have just built the components of a rudimentary processor. Design a processor that implements ADD, SUB, NAND, SLT, and BNE where BNE updates the contents of the program counter (semantics: if **A**!=**B** then **PC** gets the value **PC**+**IMM**,

otherwise **PC** gets the value **PC**+1; ignore any overflow). Note that the semantics of this BNE are a bit simpler than the BNE in Project 1 in which the PC gets the value PC+1+IMM.

You need to add two new standalone registers (i.e. not part of the register file): **PC** (4-bit) and **INST** (9-bit), where **PC** contains the value of the program counter and **INST** contains the current instruction; you can assume that **INST** is updated automagically as soon as **PC** changes. You may use boxes to represent the modules that you have already built. You may add muxes, adders, and simple logic gates. The only new logic that you need to add is the updating of **PC**, for which you may use dedicated adders.

Overflow in the ALU

Notice that you cannot subtract -8 (0x1000) from any positive number; similarly there are few positive numbers that you can legally subtract -7 from, etc. Therefore there are many cases where you would like to perform the **LT** test between two numbers, but for which you cannot perform subtraction to do the test. These can be handled by doing an **XOR** of the sign bits; if the bits are opposite (1+0 or 0+1, **XOR**=1), then it is obvious which is less than, which is greater than. Otherwise, you will have to subtract, in which case there can be an overflow problem. For overflow, you can test the carry-in versus the carry-out of the last full adder (the one for the most significant bit). If they are the same, you can ignore the overflow bit; if they are different, you have an overflow. On overflow, the hardware would normally signal an exceptional condition and immediately set the **PC** to a hardware-defined value—this in effect is a jump to a software handler that knows it is supposed to clean up after a weird situation in which we caused overflow.

For the project, doing less-than on large numbers (like -7 < 7) and detecting overflow will be extra credit; it is **not required**. You should at least do the subtract test to determine less-than, even though this is technically wrong in some cases. You will get additional credit if you handle the odd cases like (-7 < 7) where subtraction does not work—for this you can use the **XOR** scheme. You will also get extra credit if your circuit detects overflow; you can either detect overflow and send a signal to a box labeled “overflow handler”, or (even more credit), you can detect overflow and set the **PC** to the binary value ‘1111’.