CS 237 Meeting 20 — 10/26/12

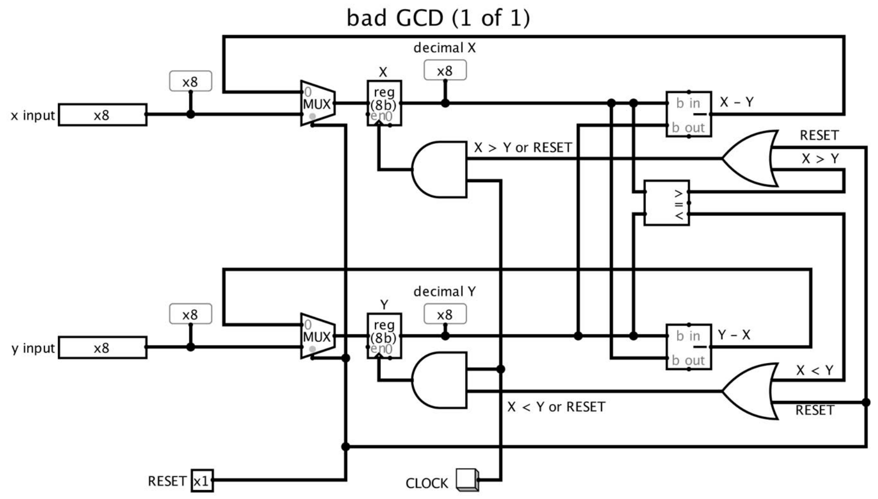
Announcements

1. Midterm: New date: Oct 29th. In class open book/notes.

2. Try to complete the linear feedback shift register lab in one sitting (and please put all the equipment away when you are done so it will not get in the way of others who need to use the Macs in that lab).

Synchronous Sequential Circuits

1. Consider the following attempt to solve the GCD circuit construction exercise included in last week's lab:



2. This circuit does not work as desired. It is an example of a sequential circuit with a bad synchronization problem.

   - The problems with the circuit are a result of the fact that the clock inputs to the two registers are not tied directly to the clock, but instead are connected to the output of gates whose inputs involve both the clock and circuit elements involving the outputs of the registers.
   - The intent of the circuits design is that the clock signal received by a register will only be 1 if the main clock has been set to 1 and the value in that register is currently greater than the value in the other register.
   - Assuming this is true when the clock first become 1, one register will see a rising clock pulse and load a new value.
   - This new value will quickly become the registers new output.
   - It will be smaller than the old output.
   - It will feed into the comparator used to determine which register should change.
   - If the decrease in value of the register that changed makes it less than the value in the other register, the output of the comparator will change.
   - The clock signal for the other register is the and of the output of the comparator that had been 0 and the main clock.
   - If the comparator changes it output while the main clock is still 1, the second register will see a rising clock pulse and load a new value.
   - TWO REGISTER UPDATES HAVE OCCURRED AT DIFFERENT TIMES AS A RESULT OF A SINGLE CLOCK PULSE.
   - This was probably not the intent of the designer of the circuit.

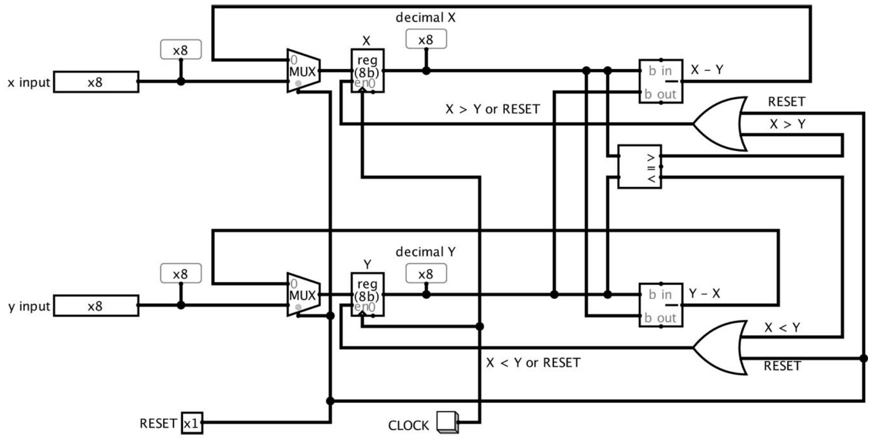
3. Things can get even worse. (In fact they were already). The updated comparator output may arrive before the updated output from the combination of the subtraction circuit and multiplexer that determines the input to the registers. In this case, the second register may actually load the wrong (a negative) value.

4. There is a simple design practice for avoiding such *race conditions*. In a *synchronous* sequential circuit, all registers have their clock inputs connected to the same signal. It won't arrive at all registers at precisely the same time because of differences in wire lengths, but it will be quite close.

5. This necessitates another way to arrange for some registers to change on a clock pulse when others stay the same.

- This could be accomplished by placing a multiplexer in front of every register that would choose between a new value and the register's current value.
- Better yet, real registers (and register in Logisim) usually include an enable input. The register only changes on a clock pulse if this input is 1.

6. A version of the GCD circuit that uses this approach is shown below:



A Mini-MIPS Microarchitecture

1. Now, we are ready to pull two of the strands running through this course together. We will explore the relationship between your introduction to machine language and digital logic by seeing how to build a digital circuit that interprets a significant (in functionality if not size) subset of the MIPS machine language.

2. The subset we will work with includes lw (load word), sw (store word), beq (branch if equal), five of the register to register R-type instructions (add, sub, and, or, sly (set less than)).

3. The circuit we build will involve several major components that are also essential parts of the abstract description of the MIPS machine language:

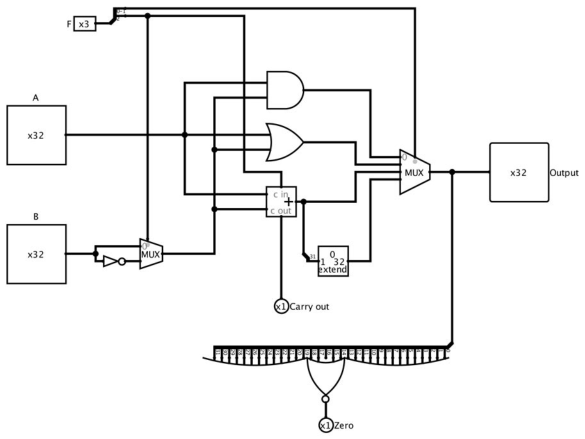
A bank of 32 registers A single instruction in the machine language can access the values of up to two registers and modify one register. Therefore, the sub circuit that implements the register bank must have three address inputs, two data outputs, and one data input. In addition, it will have a clock input and a write enable input that together determine when updates occur. Note that even though this memory is byte addressed, it only provides access at the word level. Good thing load byte is not in our list of things to implement.

A single 32-bit program counter register

A 32-bit data memory With one input bus, one output bus and clock and write enable inputs.

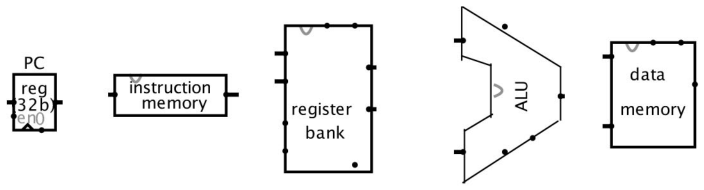
A separate 32-bit instruction memory This is a cheat that the book only weekly justifies. It is impractical to build a large memory that can output two locations simultaneously. Since the first, simple version of this machine implementation will update everything every clock cycle, separate instruction and data memories are essential.

4. One other significant component of our circuit will be an arithmetic and logical unit (ALU) that takes two inputs and performs one of a list of operations on these inputs (add, subtract, and, or, etc.) depending on several control inputs.



- Note that since the inputs and output of this sub-circuit represent 32-bit busses, the and, or and not gates represent collections of 32 bit, parallel gates.
- On the other hand, the NOR gate at the bottom of the diagram represents a nest of NOR gates combining the 32 output bits into a single bit output that indicates whether all 32-bits of its input are zero.
- The ALU sub-circuit contains a 32-bit adder, and 32-bit parallel and and or gates.
- A multiplexer controlled by two control inputs determines which of these devices output becomes the output of the ALU.
- There is also the option of running one of the inputs through a 32-bit parallel not gate.
- By negating one of the inputs and setting 1 instead of 0 on the carry in to the adder the adder can be used as a subtractor (recall that to negate a number in two's complement, we flip the bits and add 1).

5. In our illustrations, these components will be represented by the following symbols:

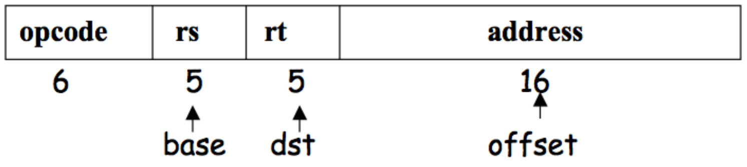


6. For most of the other digital circuits we have designed, we have started with an algorithm in pseudo-code (or C!). We can take the same approach here. The algorithm the mini-MIPS processor must execute is basically:

```
while ( 1 ) {
    IR = MEM[PC];

    PC = PC + 4;

    OPCODE = IR{31:26};

    if ( OPCODE == SW ) {
        RT = IR{20:16};     // base
        RS = IR{25:21}      // source

        OFFSET = sign-extend( IR{15:0} );
        ADDR = OFFSET + REGS[ RT ];

        MEM[ ADDR ] = REGS[ RS ];

    } else if ( OPCODE == LW ) {

        RT = IR{20:16};     // base
        RS = IR{25:21}      // destination

        OFFSET = sign-extend( IR{15:0} );
        ADDR = OFFSET + REGS[ RT ];

        REGS[ RS ] = MEM[ ADDR ];

    }  else if ( OPCODE == 0 ) {  // R-type

        FUNCT = IR{ 5:0 };

        RS = IR{25:21}      // operand 1
        RT = IR{20:16};     // operand 2
        RD = IR{15:11};     // destination

        if ( FUNCT == ADD ) {
            REGS[ RD ] = REGS[ RS ] + REGS[ RT ];
        } else if ( FUNCT == SUM  ) {
            REGS[ RD ] = REGS[ RS ] - REGS[ RT ];
        }  else
        }
    } else if ( OPCODE = BEQ ) {

        RS = IR{25:21}      // operand 1
        RT = IR{20:16};     // operand 2

        if ( REGS[ RS ] == REGS[ RT ] ) {
            OFFSET = 4 * sign-extend( IR{15:0} );
            PC = OFFSET + PC
        }

    } else if ...
```

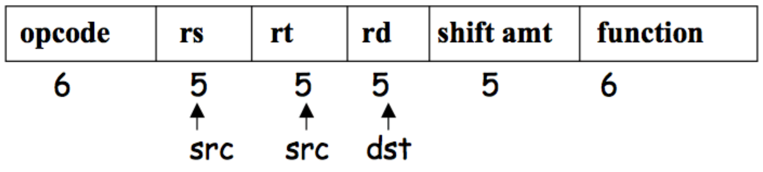
7. A lot of the "pseudoness" of this code involves the fact that the machine's operation is determined by the various sub-fields of the various MIPS instruction formats. The two formats we need to remember are:

   I-Type Used for LW, SW, and BEQ (along with many MIPS instructions not in mini-MIPS):



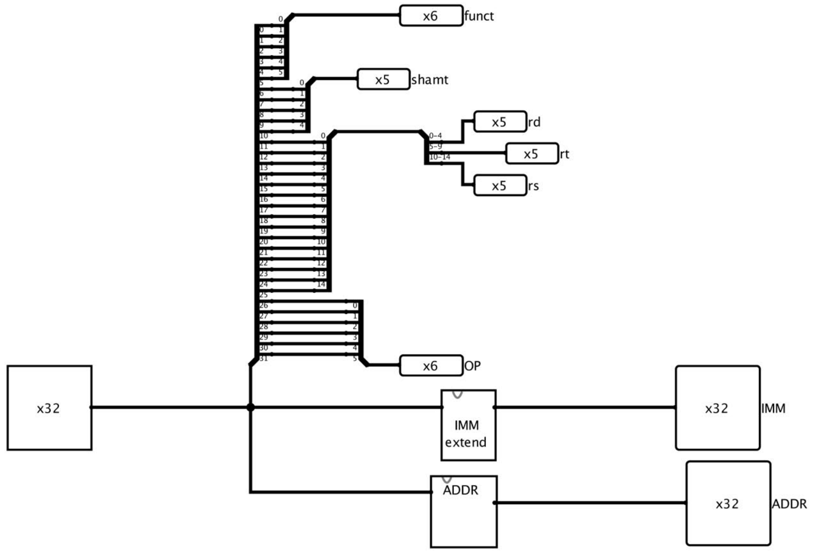
opcode	rs	rt	address
6	5	5	16
	base	dst	offset

   R-Type Used for ADD, SUB, SLT, AND, and OR:



opcode	rs	rt	rd	shift amt	function
6	5	5	5	5	6
	src	src	dst		

8. To avoid having the details of these instruction formats clutter up all of our circuits, we will use the following circuit as a sub-circuit in our Logsim implementation of this machine. It is a great illustration of Duane's claim that a wire can be a powerful computer! It pulls the instructions apart into various sub-fields doing sign-extension in the cases where that is needed:



x6
funct
x5
shamt
x5
rd
x5
rt
x5
rs
x6
OP
x32
IMM extend
x32
IMM
ADDR
x32
ADDR