CS 237 Meeting 22 — 10/31/12

Announcements

1. Preregistration Pizza at 9 on Thursday.

A Mini-MIPS Microarchitecture

1. Last time, we began to consider how one could build a circuit to interpret programs written in a language like MIPS machine code.

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:

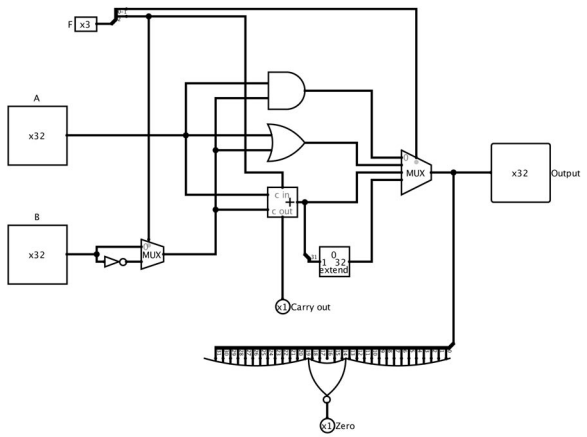
   A bank of 32 registers

   A single 32-bit program counter register

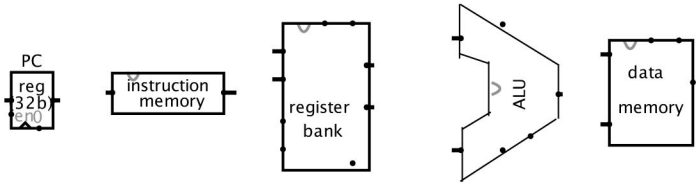
   A 32-bit data memory

   A separate 32-bit instruction memory

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.

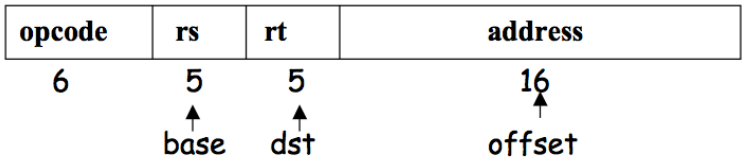


5. In our illustrations, these components will be represented by the following symbols:

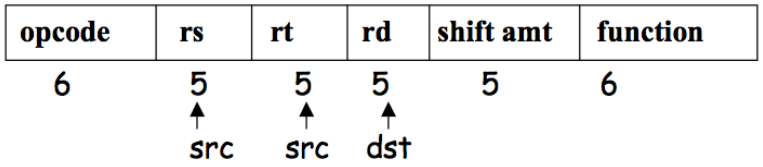


6. The actions our circuits perform are dictated by instructions encoded according to the conventions of the MIPS architecture. Given the reduced set of instructions we will consider, we only need to deal with two of the machine's formats.

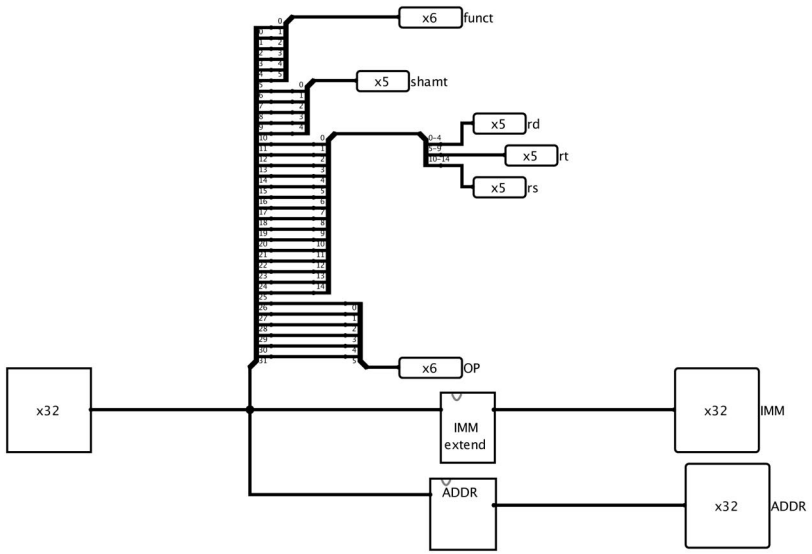
   I-Type Used for LW, SW, and BEQ (along with many MIPS instructions not in mini-MIPS):



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



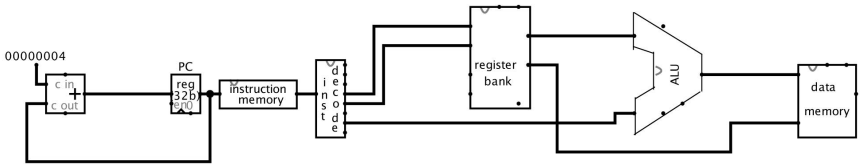
7. To avoid having the details of these instruction formats clutter up all of our circuits, we will use the following circuit as a sub-circuit in our Logsim implementation of this machine. It is a great illustration of Duane's claim that a wire can be a powerful computer! It pulls the instructions apart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



8. We can see how all these pieces fit together if we sketch out the data flow connections that would be required if all we wanted to do was implement the store word instruction. That is, we will focus on the pseudo-code:

```
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 ];

    }
```

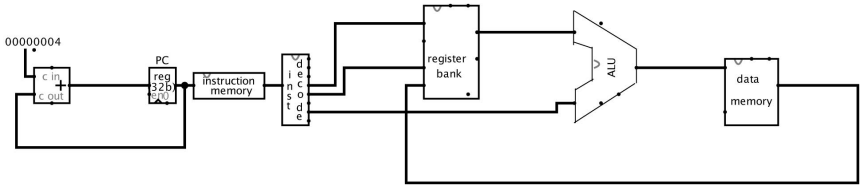
9. The components and connections need for this instruction are shown below:



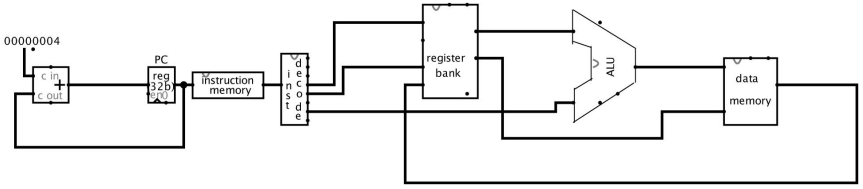
- The loop on the left includes the program counter register. It implements the first two statements of our pseudo-code by sending the value of the PC to the instruction memory to fetch the current instruction and incrementing the register by 4 to move on to the next instruction.
- We use three outputs of the decode instruction circuit to access the source register, the base register and the offset of the memory location.
- The two register numbers from the instruction are sent to the register file/bank. The base register's value is sent to the ALU so that it can be added to the offset. The value of the register to store is sent to the memory.

- Finally, the sign-extended offset is sent as the second operand to the ALU.

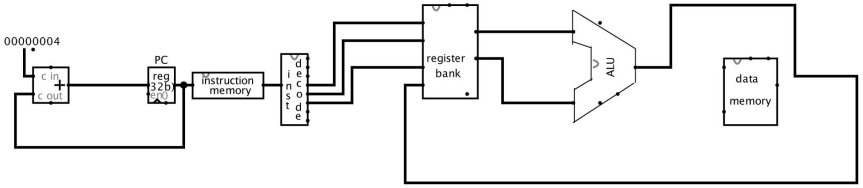
10. We can continue in this way considering the data path connections needed for each instruction or type of instruction in isolation. For the load word instruction, the following connections would be used:



- There are only two differences between this circuit and the one for SW:
  - The target register number goes to the register bank's write address input instead of its second read input.
  - The output of the memory is fed back to the register bank.
- The actions of the devices at the ends of the connections that differ between these two circuits are controlled by additional logic that depends on the bits in each instructions encoding. Therefore, we can combine them to build one circuit that could execute either instruction:



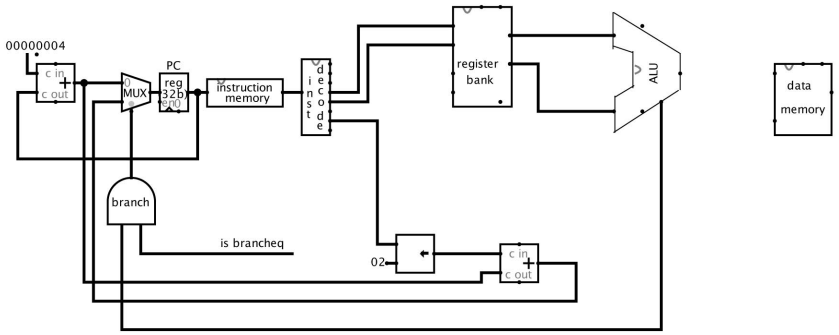
11. The connections required for the R-Type instructions (add, sub, etc.) are also similar to the load and store circuits.



- This time, the data memory is not connected at all. Instead, the output of the ALU goes back to the register bank since R-Type instructions work strictly with registers.
- There are also new connections to the register address inputs since three inputs are needed.
- Importantly, the input to the register file's write address comes from a different output of the instruction decoder. This means we cannot just merge this data flow diagram with the load or store diagram as we were able to merge load with store. If we did, we would end up with one input connected to two different sources.
- A similar conflict exists on the write data input to the register bank. For R-type instructions, this comes from the ALU. For lw, it comes from memory.
- Finally, the lower input to the ALU for R-type instructions comes from the register bank rather than from the sign extended immediate field.
- Shortly, we will see how to resolve all these conflicting input needs using multiplexers.

12. To complete the process of looking at how to build dataflow paths for individual instruction types, we need to consider the beq instruction.
    - This instruction depends on "Z" control output of the ALU that tells whether the current ALU output is equal to 0.
    - The instruction is executed by telling the ALU to subtract the values of the two registers being compared and then testing the Z control output.
    - Since the next value of the PC can now either be determined by the IMM field of the beq instruction or by adding 4 to the

preceding PC value, a multiplexer is inserted before the input to the PC, making it possible to choose the next value by sending a control signal to the multiplexer.



13. The multiplexer used in the data flow for beq is a nice simple example of how we can build a circuit in which control signals send to a multiplexer select between several possible inputs to a sub-circuit. We can use this technique to merge the circuits we have considered for all of the instruction types we want to execute. We will just insert multiplexers and control inputs anywhere merging circuits require two inputs to a sub-circuit.