ENGR 3410: HW#2 Solutions

November 5, 2005

2.1 Joe Optimal

Joe is a smart guy. Saving control signals is a good idea, and fun to boot! All we need to do is take a look at the control signal table we generated in class (Fig. 1). As we can see, aside from the Don't Care situations (X's), the $ALUSrc$ and $MemToReg$ signals are the same. Where they are not, $MemToReg$ has an X, or Don't Care value. Therefore, we can use $ALUSrc$ to drive the (originally) $MemToReg$ multiplexor.

Func	10000	100010	XXX	XXX	XXX	XXX
Op	000000	000000	100011	101011	000100	000010
	add	sub	lw	sw	beq	j
RegDst	1	1	0	X	X	X
ALUSrc	0	0	1	1	0	X
Mem2Reg	0	0	1	X	X	X
RegWr	1	1	1	0	0	0
MemWr	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ALUCtrl	Add	Sub	Add	Add	Sub	X

Figure 1: Control signals for a single-cycle processor

2.2 Three Little Instructions

In this problem, we are taking the original in-class single-cycle processor and adding three instructions, BNE, LW_R, and WAI. The Branch Not Equal compares two registers and if they are not equal, branches to the target instruction. The LW_R was specified a little weird in the problem. The RTL can be interpreted as either:

```
Reg[$rt] = Mem[$rs + $rt]  or  Reg[$rd] = Mem[$rs + $rt]
```

simply because we're making up the instruction, so the output can go where we want it ($rd or $rt). Either is acceptable since the $rd case is what we have been using in class.

2.2.1 Branch not equal

The branch not equal case adds the control signal *BNE* decoded from the instruction, as well as some circuit changes in the instruction fetch shown in Fig. 2. The BNE condition is when the zero flag is *not* set and when the *BranchNotEqual* control line is set. This case is taken care of via the inverter, AND, and OR gates in the instruction fetch.

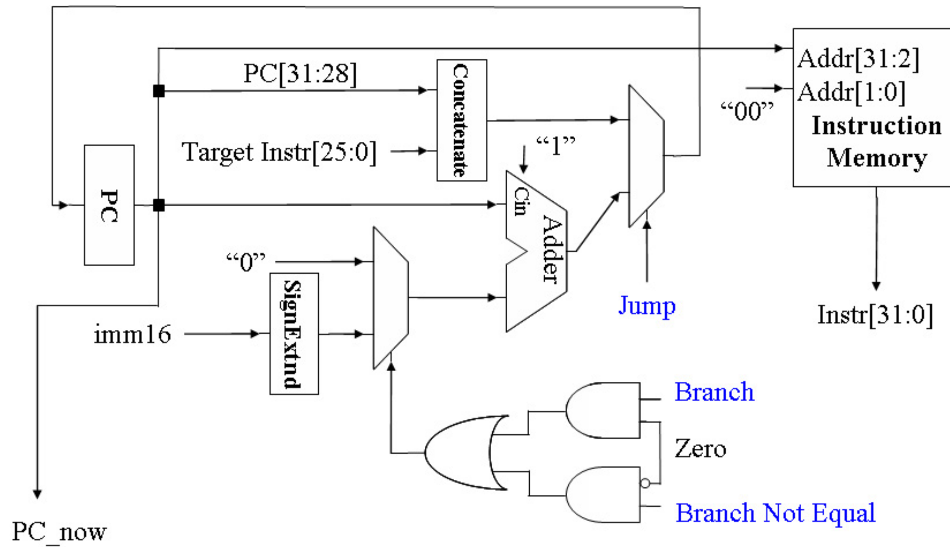

Figure 2: Modified instruction fetch supporing BNE and WAI

2.2.2 LW_R

LW_R is very similar to our basic load-word instruction, but instead of using an offset in an immediate, we store it in a register. This means we need to add two

registers together and use the result to address memory for a read. We have all the necessary datapath elements from the original single-cycle processor, it is just simply a matter of control to get this to work. The controls are shown in Fig. 4 along with the rest of the new instructions

2.2.3 WAI

WAI instructs us to load the current program counter value into register $rt. This requires both a datapath change as well as a control change. The datapath change includes taking the PC value from the instruction fetch (shown in Fig. 2 with an implicit pad of two zeros on the LSB side, not shown) and feeding it into a slightly modified datapath shown in Fig. 3. The change is just a new control signal, *PCtoReg*, controlling a new multiplexor that switches the input to the register data write port.

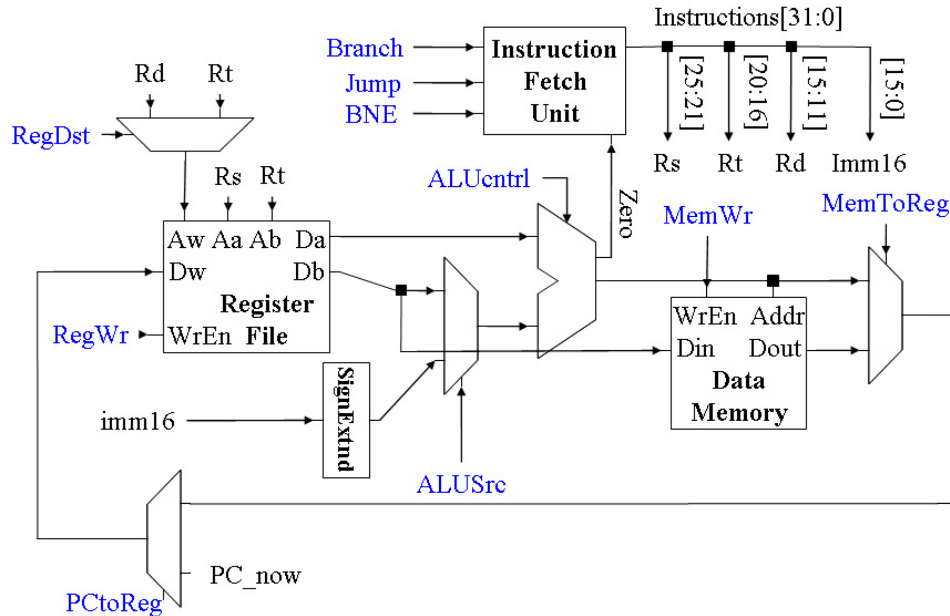

Figure 3: Data path supporting all three new instructions

2.2.4 Control signals

The control signals for all supported instructions (up to this point) are shown in Fig. 4.

	add	sub	lw	sw	beq	J	Wai	LW_R	BNE
RegDst	1	1	0	X	X	X	0	1	X
ALUSrc	0	0	1	1	0	X	X	0	0
Mem2Reg	0	0	1	X	X	X	X	1	X
RegWr	1	1	1	0	0	0	1	1	0
MemWr	0	0	0	1	0	0	0	0	0
Branch	0	0	0	0	1	0	0	0	0
Jump	0	0	0	0	0	1	0	0	0
ALUctrl	Add	Sub	Add	Add	Sub	X	X	Add	Sub
BNE	0	0	0	0	0	0	0	0	1
PCtoReg	0	0	0	X	X	X	1	0	X

Figure 4: New control signals supporting the three new instructions

2.3 Swappin'

Unfortunately, we cannot do the swap function in a single cycle of our current processor. The shortest sequence of MIPS instructions that can implement the swap instruction is:

```
xor $s0, $s0, $s1
xor $s1, $s0, $s1
xor $s0, $s0, $s1
```

This requires more than one write to the register file to two different register destinations (as well as an xor instruction which we do not have). We can't do that.