CS 152 Fall 2001 Homework #4 Solutions
P&H: 5.3, 5.9, 5.15, 5.17, 5.20, 5.21, 5.23, 5.25, 5.27, 5.28, 5.29

5.3
Instructions supported by this multicycle datapath – lw, sw, beq, add, sub, and, or, slt, and j.
RegDst = 0
The R-type instructions need RegDst = 1 in order to write to $rd.
The lw, sw, beq, and j instructions would still work.

MemtoReg = 0
MemtoReg = 1 is needed to write the memory value to a register in a lw instruction.
The sw, beq, add, sub, and, or, slt, and j instructions would still work.

IorD = 0
In other to read or write to an address other than the PC (as in lw or sw), IorD = 1 is needed.
The beq, add, sub, and, or, slt, and j instructions would still work.

ALUSrcA = 0
ALUSrcA = 1 is needed to use the value of $rs, which all instructions except for j do.
Only the j instruction would still work.

5.9
A swap instruction requires two writes, and in a single-cycle implementation, both of these writes would have to be on the same cycle. So the register file would have to support two write operations on one clock cycle.
5.15
addi instruction

No modifications required for the multicycle datapath of Figure 5.33 on page 383.

Modifications to the finite state machine of Figure 5.42 on page 396:
5.17
Modifications to the multicycle datapath of Figure 5.33 on page 583:

Modifications to the finite state machine of Figure 5.42 on page 396:
5.20
The jump memory instruction behaves much like a load word until the memory is read. The data coming out of memory needs to be deposited into the PC. This will require a new input to the multiplexor controlled by PCSource. We add a new state coming off of state 3 that checks for Op = “jump memory” (and modify the transition to state 4 to ensure Op = “lw”). The new state has PCWrite, PCSource = 11. The transitions to states 2 and 3 need to also include Op = “jump memory.”

5.21
add3 instruction

Modifications to the multicycle datapath of Figure 5.33 on page 583:
Modifications to the finite state machine of Figure 5.42 on page 396:

You may use bits [9-5] instead of bits [4-0] for the additional register if you choose.

5.23
There are many possible solutions here, but all of them include a mux and a new RegRead control signal to select which register to read. One possible solution is to add a write signal to A and break up state 1 into two states, in which A then B are read. It is possible to avoid adding the write signal to A if B is read first. Then A is read and RegRead is held stable (because A always writes. Another possibility is to read A first because it may be needed to calculate an address. You could then postpone reading B until state2 and avoid adding an extra cycle for the load and store instructions. An extra cycle would be need for branch and R-type instructions.
5.25

Execution time = Instructions * CPI * Clock rate
The number of instructions doesn’t affect our calculation, so we leave it out.

For 500 MHz version:
Average time per instruction = [.23(5)+.13(4)+.43(4)+.21(3)] * 2ns = 8.04ns
[loads + stores + R-types + branches]

For 750 MHz version:
Average time per instruction = [.23(7)+.13(6)+.43(5)+.21(5)] * 1.33ns = 7.45ns

Speedup = (Old time – New time)/Old time = (8.04 – 7.45)/8.04 = 7.3% faster

5.27
Block Copy - Multicycle & Microcode
Using the multicycle datapath from the textbook (Fig 5.33) changes are made below in aqua dashed lines. The basic RTL for the block copy instruction using this modified datapath is also referenced below. When instruction is fetched you have register $t1, $t2, and $t3. Make sure that the instruction register has a write enable so that the register file addresses are kept around. First you want to read R($t1) and R($t2) and get these values into the register for Reg1Data (register A) and Reg2Data (register B). Make sure these registers have write enables also. Next you want to use the value in register A to be used as address for memory to get the first value to be copied. At the same time you can use increment the register A value using the ALU. In the next cycle you can write the value back to memory using the memory address in register B. At the same you can write the newly incremented source address back into the register file, and use the ALU to increment the destination address in register B, and to get the length of the array into register A. In this final step you can write the value of incremented destination address back into the regfile. In the same cycle you can also decrement the length by one and check to see if it’s negative. If so then block copy is done, if not it needs to write the length back into the register file and continue the loop. Another way is to store the length in it’s own separate register (RegC, not shown in diagram), instead of ALUOut, so that you won’t need to write it back into the regfile. This saves you a cycle on every loop.

In hardware it would take 4cycles for the last loop + (5cycles)*(length -1) + 2cycles for instruction fetch and decode. The five cycles for the loop comes from having to store the length back into R($3). [If use extra register (RegC) to store length, instead of storing it to ALUOut then to R($3); it’s just (4cycles)*(length) + 2cycles.] In the software version it would take 3cycles for the beq + (length-1)*(24cycles). The 24 cycles are based on the multicycle state diagram (Fig 5.42) where lw is 5cycles, sw is 4cycles, each addi is 4 cycles, and branch is 3cycles. (i.e. 5+4+(4*3)+3)

Basic RTL for Block Copy:
regA <- R($t1), regB <- R($t2)
MemDataReg <- M(regA), ALUout <- regA + 4
Mem ← MemDataReg at M(regB), R($t1) ← ALUout, ALUout ← regB + 4, regA ← R($t3)
R($t2) ← ALUout, ALUout ← regA - 1, decide if should loop again

5.28
For bcp, we need to add a field for the mux going into the Din of the ideal memory, MemSrc, with the values B or MemDataReg. We need to add a new field RegSrcA with values Rs or Rd and a new field for the counter, counter with possible values of load and dec, adding two bits (because we need a nop). Also, we need to add a value to the SRC1 field, ALU_2d, adding onebit of control to this field. For the sequence field, we need to add a branch, which will fetch if the counter is zero, but jump back to the loop if not.

5.29

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU</th>
<th>SRC1</th>
<th>SRC2</th>
<th>ALUOut</th>
<th>Mem</th>
<th>MemReg</th>
<th>PCWr</th>
<th>MemSrc</th>
<th>rsord</th>
<th>counter</th>
<th>Seq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Mem</td>
<td>IR</td>
<td>ALU</td>
<td></td>
<td>Seq</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>Dispatch</td>
<td>Add</td>
<td>PC</td>
<td>ExtSel</td>
<td>ReadPC</td>
<td>IR</td>
<td>ALU</td>
<td>rs</td>
<td>Dispatch</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Add</td>
<td>RegA</td>
<td>ExtSel</td>
<td>Rs-PC</td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>Bcp</td>
<td>Add</td>
<td>RegA</td>
<td>0</td>
<td>Rs-ALU</td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Add</td>
<td>RegA</td>
<td>0</td>
<td>Rd</td>
<td>Load</td>
<td>Seq</td>
<td></td>
<td>Seq</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td>Add</td>
<td>ALU_2d</td>
<td>4</td>
<td>WALU</td>
<td></td>
<td>feedback</td>
<td>Dec</td>
<td>Seq</td>
<td></td>
<td>Branch</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALU_2d</td>
<td>4</td>
<td></td>
<td>ReadALU</td>
<td></td>
<td></td>
<td></td>
<td>Seq</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The value Rs in the field SRC1 has been changed to RegA to avoid confusion when we try to read the Rd register instead. The idea in this solution is to only use an extra register so that we can keep the next address we want to read from and the next address we want to write to in two different registers. In addition, the timing works out so that you read
from an offset off of Rs in one cycle, store the value in the MemDataRegister, then feed
that value back into the memory and write from an offset off of Rd. Of course, we need a
counter that is loaded to the value Reg[Rt] that decrements only when we want it to.
When the counter reaches 0, we know we have finished our looping and we can fetch the
next instruction.