CS150 Fall 2012 — Solutions to Homework 6

October 6, 2012

Problem 1

a.) Answer: 0.09 ns
This delay is given in Table 65 as T_{ILO}, specifically 'An–Dn LUT address to A'.

b.) Answer: 0.41 ns
In Table 65, this delay is labeled as T_{DICK}, or 'AX–DX input to CLK on A–D Flip Flops.'

c.) Answer: 0.50 ns
The setup time must be found as a combination of the propagation delay from A1 to A from (a) and the setup time from (b). This follows from the Virtex5 slice diagram, which shows that the flip-flop input mux may take either AX or output A.

d.) Answer: 0.40 ns
Again, this comes from Table 65. The delay in question is T_{CKO} 'Clock to AQ–DQ outputs.'

e.) Answer: 2 ns
This entire circuit's max clock rate is given in Table 52 as 500 MHz. The calculation for the clock period is as follows:

$$T_{CLK,min} = \frac{1}{f_{min}} = \frac{1}{500\,\text{MHz}} = 2\,\text{ns}$$

f.) Answer: 1.1 ns
Register-to-register delay is defined by the sum $t_{clk\to q} + t_{pd} + t_{setup}$. From (e), we know that this must total less than 2 ns. We can find the combinational logic delay t_{pd} by subtracting.

$$t_{pd} = 2\,\text{ns} - t_{clk\to q} - t_{setup}$$

From (d), we know that $t_{clk\to q} = T_{CKO} = 0.40\,\text{ns}$. The setup time must be found as a combination of the propagation delay from A1 to A and the setup time from (b). This follows from the Virtex5 slice diagram, which shows that the flip-flop input mux may take either AX or output A. Therefore, we have:

$$\begin{aligned} t_{pd} &= 2\,\text{ns} - t_{clk\to q} - t_{setup} \\ &= 2\,\text{ns} - T_{CKO} - (T_{DICK} + T_{ILO}) \\ &= 2\,\text{ns} - 0.40\,\text{ns} - (0.41\,\text{ns} + 0.09\,\text{ns}) \\ &= 1.1\,\text{ns} \end{aligned}$$

Problem 2

a.) For the single-cycle CPU, the minimum clock period is simply the sum of the delays through all five sub-components (not stages, as there is only one stage).

$$\begin{aligned} t_{clk,1s} &\geq 250\,\text{ps} + 150\,\text{ps} + 200\,\text{ps} + 250\,\text{ps} + 100\,\text{ps} \\ &\geq 950\,\text{ps} \end{aligned}$$

The assumption that the pipelining overhead is zero means that the minimum clock period of the pipeline CPU is simply the longest individual stage delay. In this case, that is a tie between $FetchInstruction$ and $MemoryRead/Write$, so the minimum clock period is 250 ps.

$$\begin{aligned} t_{clk,5s} &\geq \max(250\,\text{ps}, 150\,\text{ps}, 200\,\text{ps}, 250\,\text{ps}, 100\,\text{ps}) \\ &\geq 250\,\text{ps} \end{aligned}$$

Since the single-cycle CPU takes exactly one clock cycle per instruction, the total amount of time taken (for the fastest clock rate) is then given below:

$$\begin{aligned} t_{clk,1s} * 2000 &= 950\,\text{ps} * 2000 \\ &= 1900\,\text{ns} \end{aligned}$$

The program completes in 1900 ns on the single-cycle CPU.

For the five-stage pipelined CPU, the first instruction takes five clock cycles to finish, as the CPU has a longer latency than with a single-cycle implementation. However, one new instruction then completes every cycle after that. This CPU then takes 2004 cycles to complete the program starting from an idle state. The total time taken is then:

$$\begin{aligned} t_{clk,5s} * 2004 &= 250\,\text{ps} * 2004 \\ &= 501\,\text{ns} \end{aligned}$$

b.) For the single-cycle CPU, shortening the latency of any sub-stage influences the maximum clock rate. If $Execute/ALU$ took only 150 ps instead of 200 ps, the overall clock period would drop.

$$\begin{aligned} t_{clk,1s} &\geq 250\,\text{ps} + 150\,\text{ps} + 150\,\text{ps} + 250\,\text{ps} + 100\,\text{ps} \\ &\geq 900\,\text{ps} \end{aligned}$$

This would shorten the execution time of the above program to $2000 * 900$ ps or 1800 ns on the single-cycle CPU.

However, for the five-stage pipeline CPU, the minimum clock period is the maximum delay of any stage, neglecting the effects of pipelining registers.

$$\begin{aligned} t_{clk,5s} &\geq \max(250\,\text{ps}, 150\,\text{ps}, 150\,\text{ps}, 250\,\text{ps}, 100\,\text{ps}) \\ &\geq 250\,\text{ps} \end{aligned}$$

Since the minimum clock period does not change when the $Execute/ALU$ stage becomes faster (as it was not on the critical path), the time required for the pipelined CPU to complete the program is still 501 ns.

Forwarding and other pipeline complications would make it harder to say that one single stage is the critical register-register path.

Problem 3

This question contains no branching or jumping; therefore, we are concerned with read-after-write and load-use hazards. Because there is no negative-edge triggered writeback and no forwarding, an instruction cannot leave the decode stage before all the instructions that write to registers it reads have *left* the writeback stage.

The pipeline diagram below shows the progress of the program through the pipeline. Instructions highlighted in red are not yet ready to leave their current stage; this means that that pipeline stage is stalled while a bubble (or no-op) is inserted ahead of it.

Cyc	IF	ID	EX	M	WB
0	sub $t3, $s0, $s2				
1	lw $t4, 20($s2)	sub $t3, $s0, $s2			
2	add $t2, $t3, $s3	lw $t4, 20($s2)	sub $t3, $s0, $s2		
3	or $t3, $t4, $t2	add $t2, $t3, $s3	lw $t4, 20($s2)	sub $t3, $s0, $s2	
4	or $t3, $t4, $t2	add $t2, $t3, $s3	*bubble*	lw $t4, 20($s2)	sub $t3, $s0, $s2
5	or $t3, $t4, $t2	add $t2, $t3, $s3	*bubble*	*bubble*	lw $t4, 20($s2)
6		or $t3, $t4, $t2	add $t2, $t3, $s3	*bubble*	*bubble*
7		or $t3, $t4, $t2	*bubble*	add $t2, $t3, $s3	*bubble*
8		or $t3, $t4, $t2	*bubble*	*bubble*	add $t2, $t3, $s3
9		or $t3, $t4, $t2	*bubble*	*bubble*	*bubble*
10			or $t3, $t4, $t2	*bubble*	*bubble*
11				or $t3, $t4, $t2	*bubble*
12					or $t3, $t4, $t2
13					

Numbering from cycle zero (when the 'sub' is in the IF stage), the CPU stalls on cycles 3, 4, 6, 7, and 8. This corresponds with stalling with the 'add' in the ID stage for two cycles and stalling with the 'or' in the ID stage for three cycles.

Problem 4

a.) This code segment has a read-after-write hazard. The instruction at program counter address 0x7C ('add $t5, $t4, $t1') depends on the value of $t1 generated by the previous instruction ('add $t1, $t2, $t3').

Since the processor has no negative-edge triggered writeback, the instruction in the decode stage would have to be stalled until after all previous instructions that it depends on have *left* the write-back stage. The pipeline chart below shows the progress of the two instructions, with red text indicating that an instruction will stall in a particular stage.

Cyc	IF	ID	EX	M	WB
0	add $t1, $t2, $t3				
1	add $t5, $t4, $t1	add $t1, $t2, $t3			
2		add $t5, $t4, $t1	add $t1, $t2, $t3		
3		add $t5, $t4, $t1	*bubble*	add $t1, $t2, $t3	
4		add $t5, $t4, $t1	*bubble*	*bubble*	add $t1, $t2, $t3
5		add $t5, $t4, $t1	*bubble*	*bubble*	*bubble*
6			add $t5, $t4, $t1	*bubble*	*bubble*
7				add $t5, $t4, $t1	*bubble*
8					add $t5, $t4, $t1
9					

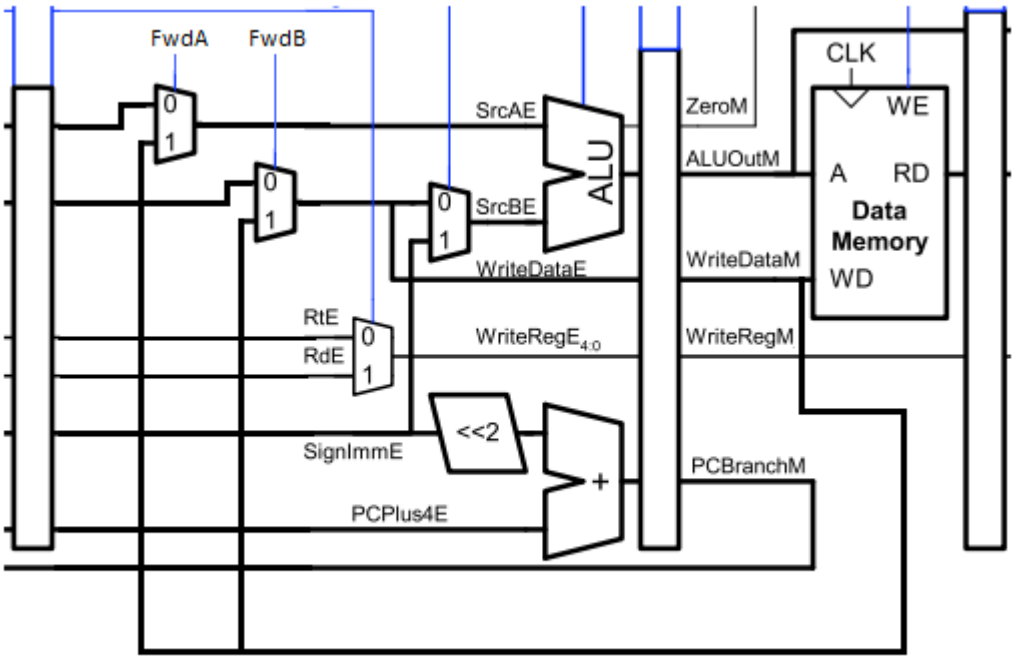
Three extra cycles are introduced without forwarding.

b.) If a negative-edge triggered writeback is added to this CPU, an instruction can leave the decode stage when the last preceding instruction whose writeback results it depends on is *in* the writeback stage. This is because the older instruction's writeaback will happen one half-clock period before the instruction has the leave the decode stage.

As a result of this change, the CPU will only stall the second add in the decode stage for two cycles (not three). Therefore, two instructions will take one fewer cycle cycle to complete. In (a), they took 9 cycles to complete, so they will now take 8 cycles. This is depicted in the pipeline stage chart below.

Cyc	IF	ID	EX	M	WB
0	add $t1, $t2, $t3				
1	add $t5, $t4, $t1	add $t1, $t2, $t3			
2		add $t5, $t4, $t1	add $t1, $t2, $t3		
3		add $t5, $t4, $t1	*bubble*	add $t1, $t2, $t3	
4		add $t5, $t4, $t1	*bubble*	*bubble*	add $t1, $t2, $t3
5			add $t5, $t4, $t1	*bubble*	*bubble*
6				add $t5, $t4, $t1	*bubble*
7					add $t5, $t4, $t1
8					

c.) The modified pipeline is shown below, with added control signals 'FwdA' and 'FwdB'.



(Undotted 4-way intersections do not connect, all tee intersections connect)

d.) The control unit will set the values of the 'FwdA' and 'FwdB' signals based on:

- whether a register is written by the instruction in the MEM stage (the value of $RegWriteM$)
- the destination register for the instruction in the MEM stage (the value of $WriteRegM[4:0]$)
- the value of \$rs from the instruction in the EX stage (called RsE in P&H)
- the value of \$rt from the instruction in the EX stage (called RtE in P&H)

Because the forwarding mux on the path to the 'B' source for the ALU appears before the immediate mux, the control unit actually does not care what type of instruction is in the execute stage. The Verilog to implement this module is below:

```
module forward(
   input       RegWriteM,
   input [4:0] WriteRegM,
   input [4:0] RsE,
   input [4:0] RtE,
   output      FwdA,
   output      FwdB
);

   wire MatchA, MatchB;
   assign MatchA = (RsE == WriteRegM);
   assign MatchB = (RtE == WriteRegM);

   assign FwdA = (RegWriteM) ? MatchA : 1'b0;
   assign FwdB = (RegWriteM) ? MatchB : 1'b0;

endmodule
```

Problem 5

One issue is performing the increment of $s0 after the branch instruction This must be corrected in a way that retains correctness in the presence of a branch delay slot.

Corrected version with branch delay slot:

```
     addi  $s1, $0, 0
     add   $s0, $0, $0
for: add   $t1, $s3, $s0
     add   $t3, $s4, $s0
     lw    $t2, 0($t1)
     add   $s1, $t2, $s1
     lw    $t5, 0($t3)
     addi  $s0, $s0, 1
     bne   $s0, $s2, for
     sub   $s1, $s1, $t5
```

If the branch delay slot is not taken into account, the following correction will suffice:

```
     addi  $s1, $0, 0
     add   $s0, $0, $0
for: add   $t1, $s3, $s0
     add   $t3, $s4, $s0
     lw    $t2, 0($t1)
     add   $s1, $t2, $s1
     lw    $t5, 0($t3)
     sub   $s1, $s1, $t5
     addi  $s0, $s0, 1
     bne   $s0, $s2, for
```

A further optimization can be realized by incrementing the pointers directly and calculating the final pointer.

Optimized, corrected version with branch delay slot (not necessary for credit):

```
     addi  $t1, $s3, 0    # $t1 holds start pointer for in1
     addi  $t3, $s4, 0    # $t3 holds start pointer for in2
     add   $t5, $t1, $s2  # $t5 holds final pointer within in1
for: lw    $t2, 0($t1)    # load element from in1 to $t2
     lw    $t4, 0($t3)    # load element from in2 to $t4
     addi  $t1, $t1, 1    # increment pointer to next element of in1
     addi  $t3, $t3, 1    # increment pointer to next element of in2
     add   $s1, $t2, $s1  # add element of in1 to sum
     bne   $t1, $s5, for  # branch backwards if not at end pointer
     sub   $s1, $s1, $t4  # subtract element of in2 from sum in delay slot
```

This cuts the number of instructions in the main loop from 8 to 7, asymptotically increasing performance by 14%.