ECE M116C - CS M151B Computer Architecture Systems - UCLA

Homework 3

Deadline: Friday, Nov. 4, 11:55 PM

Fall 2022

(Upload it to Gradescope.)

Q1. We have the following set of instructions (every instruction takes one cycle to complete except LW):

```
0x100: sub x5, x3, x2
0x104: xor x3, x3, x1
0x108: lw x5, x1, 4
0x10c: add x1, x3, x0
```

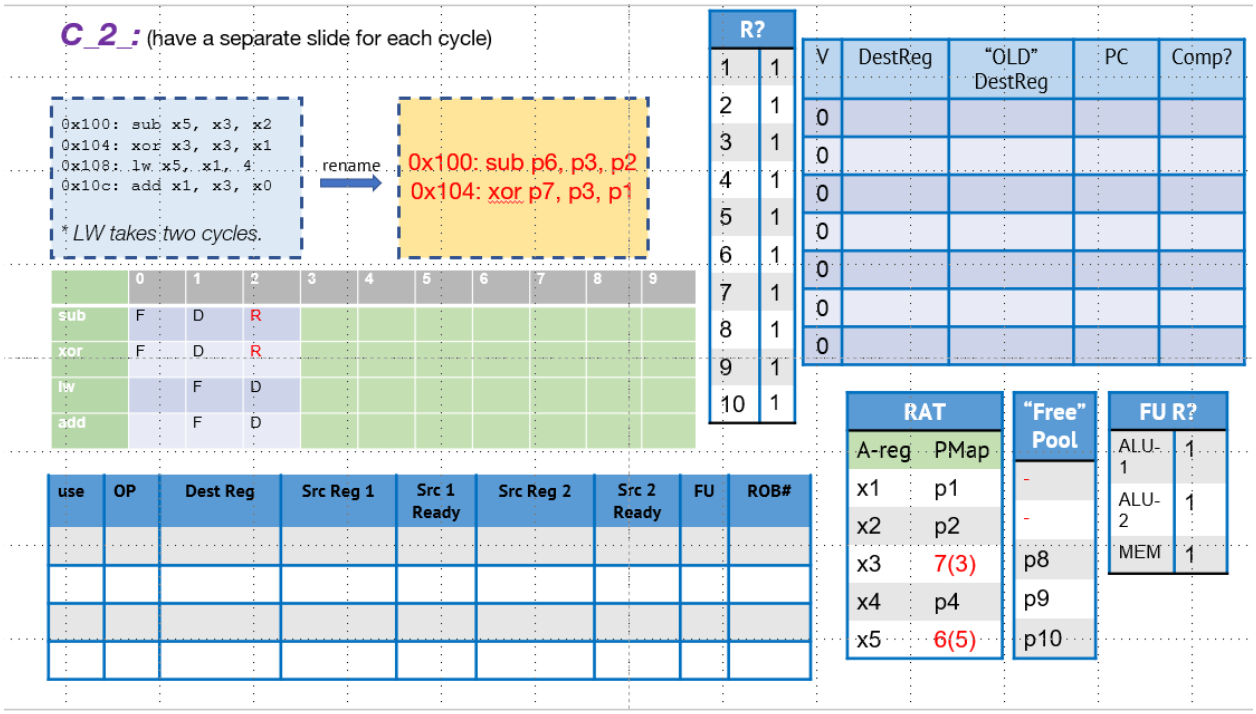
* LW takes two cycles.

We have a 2-issue out-of-order processor (you can access the template below).

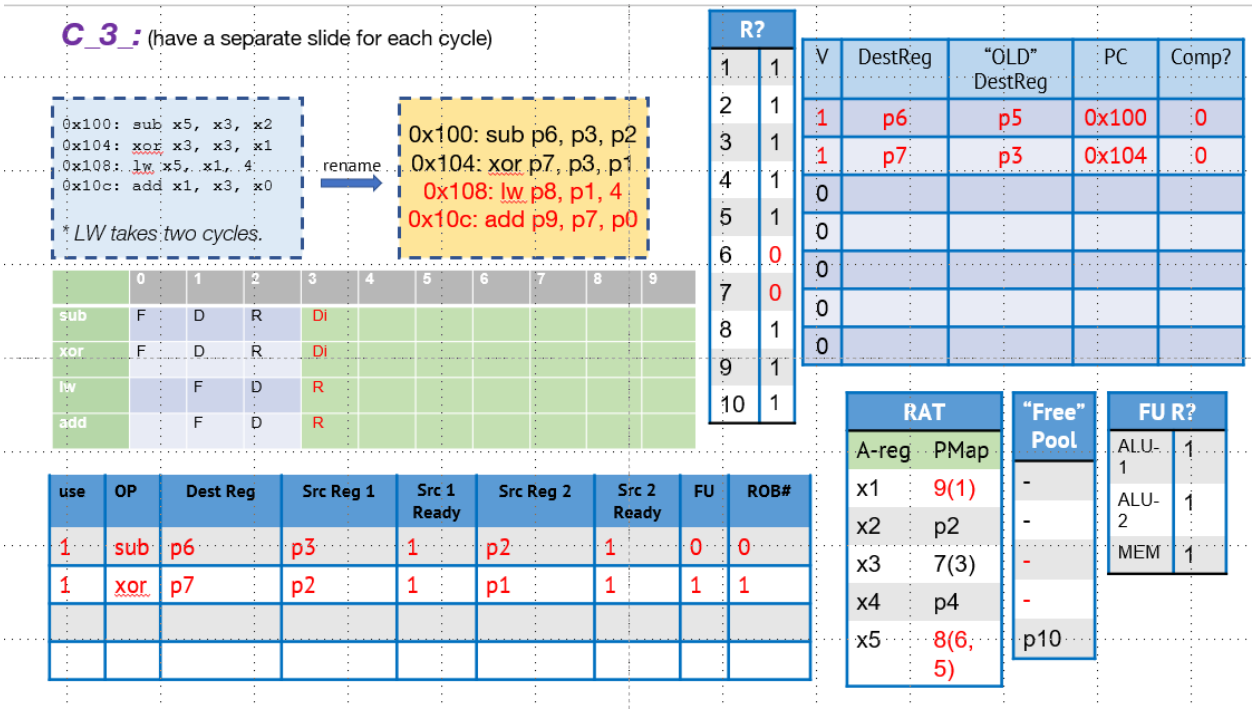
HW3-copy.pptx

Start from cycle 2, with Rename, and fill all the tables. In the green table (the one above the reservation station on the left), you should put R, Di (for dispatch), I (for issue), C, and Rt (for retire). If an instruction stays in one stage for multiple cycles, repeat the same character for it (e.g., F, D, R, Di, Di, I, C, C, Rt). `Comp` in ROB shows whether the instruction is completed or not. (Assume that you can retire a maximum of two instructions per cycle.)

SOLUTION

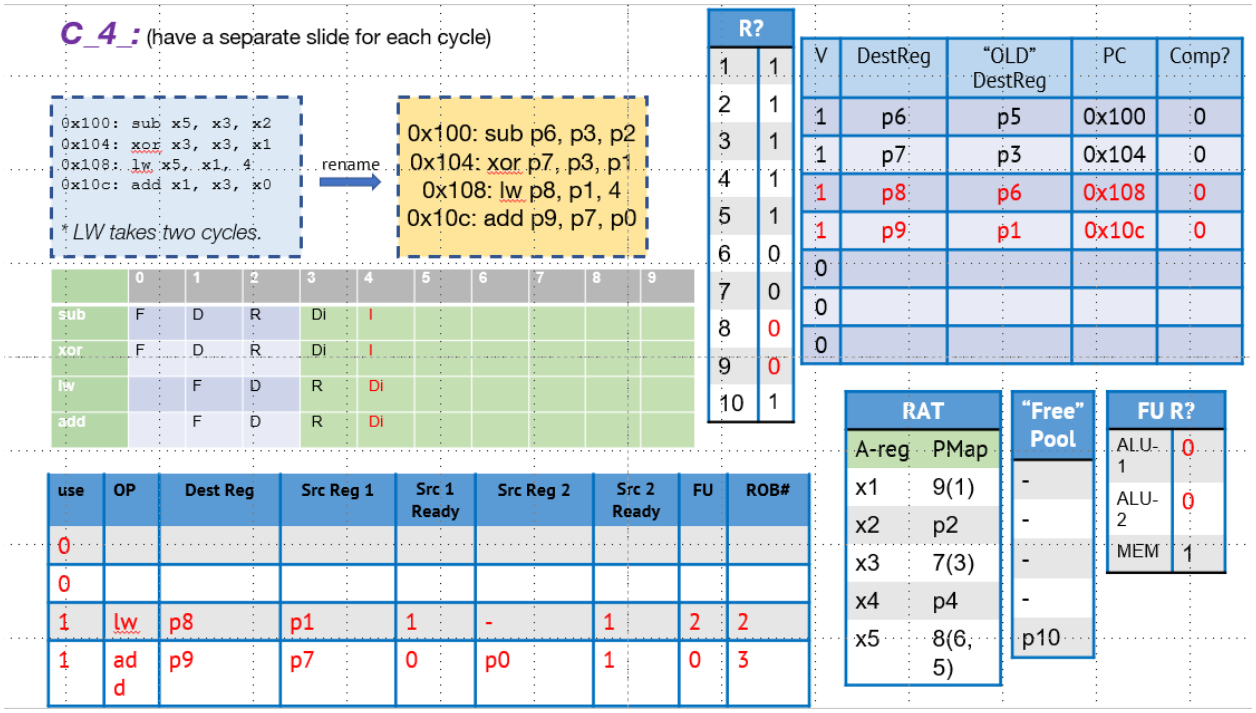


In cycle 2, we rename the destination registers of the "sub" and "xor" instructions (updating the Free Pool and the RAT to reflect the new assignments of architectural register to physical register)



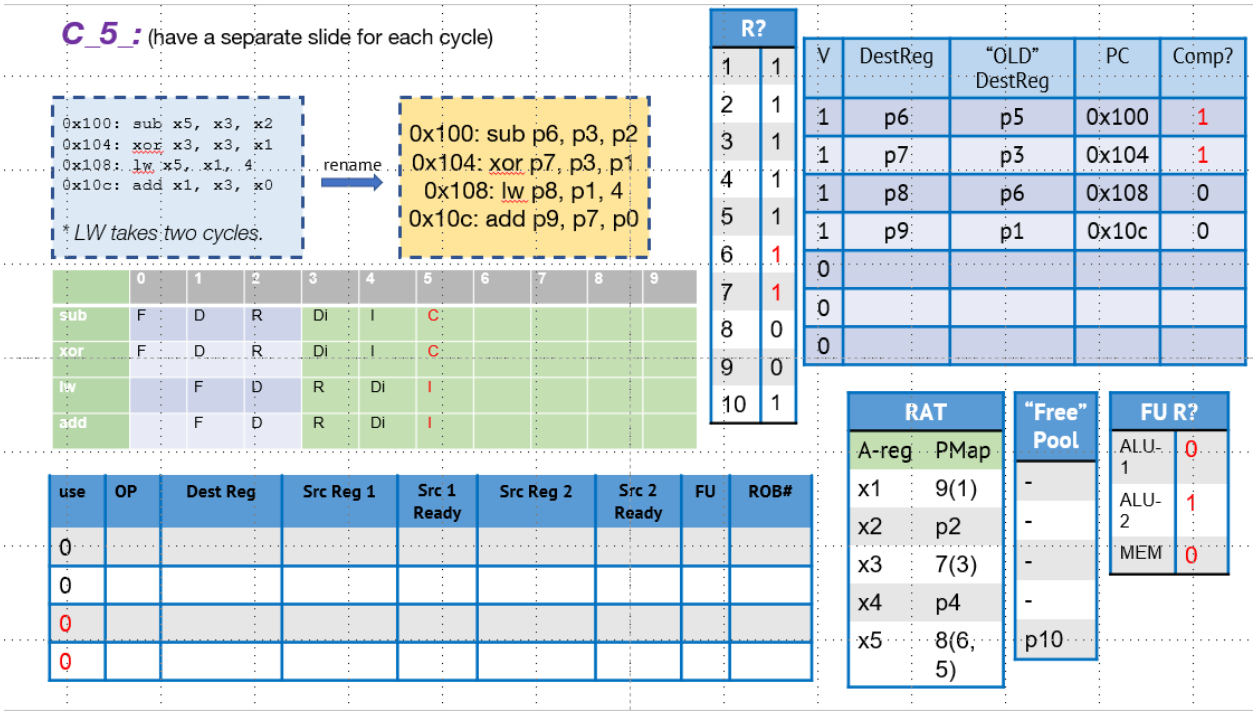
In cycle 3, we dispatch the "sub" and "xor" instructions into the Reservation Station and place them in the ROB as well. We also mark registers p6 and p7 as "not ready". In the reservation station, we note that all the source registers are ready for both instructions

We rename the "lw" and "add" instructions, updating the Free Pool and the RAT like previously.



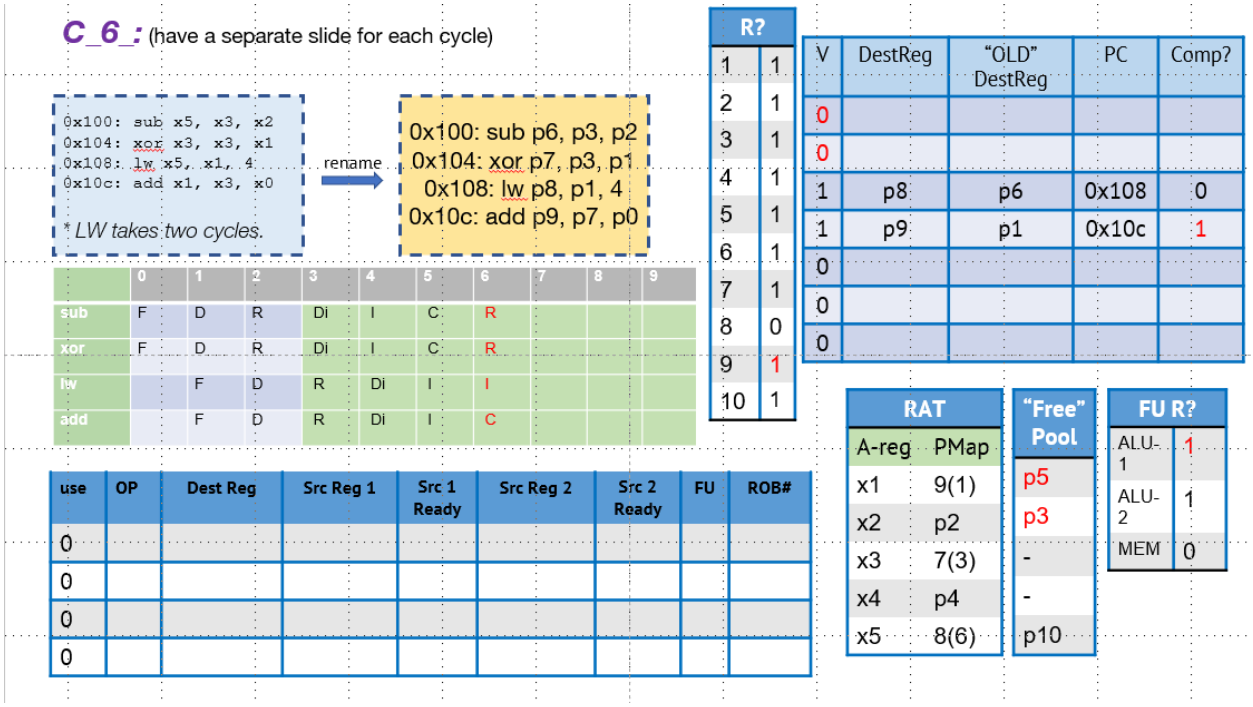
In cycle 4, we issue the "sub" and the "xor", as the source registers for both instructions are ready and the functional units assigned are ready. We clear "sub" and "xor" from the Reservation Station. We mark that ALU-1 and ALU-2 (used by "sub" and "xor") are not ready.

We dispatch the "lw" and "add" instructions, placing them in the Reservation Station and the ROB. We mark p8 and p9 as "not ready". We note that register p7 is not ready (is the destination register for "xor"). Note: we arbitrarily assign functional unit ALU-1 to the "add". We could have also picked ALU-2. In this case, it doesn't matter which one you pick, but in reality, you'd pick the one that would be ready first.



In cycle 5, both the "sub" and the "xor" complete. We mark both instructions as "complete" in the ROB. We mark registers p6 and p7 as "ready". We free the functional units they used (ALU-1 and ALU-2).

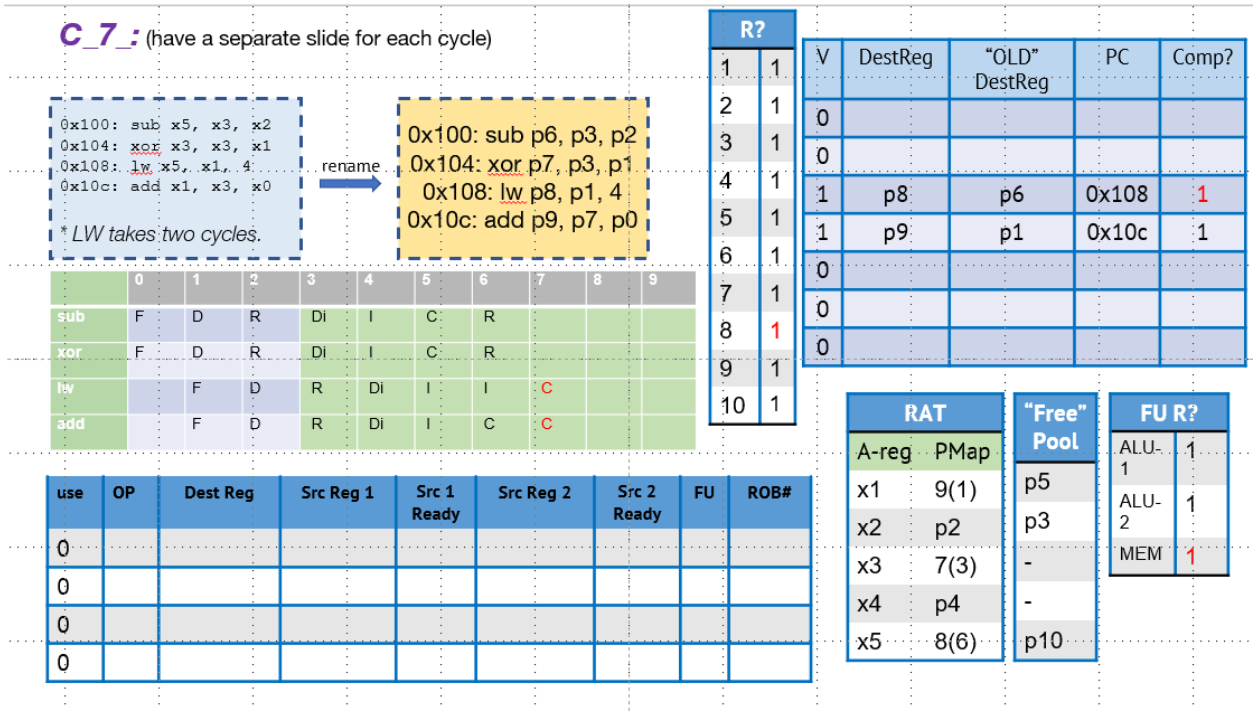
We issue the "lw" and the "add". We see that register p7 is now ready (as the "xor" completes in this cycle), so all the "add"'s source registers are ready. We also note that the functional units needed are ready. We clear the "lw" and "add" from the reservation station and mark the respective functional units as "not ready".



In cycle 6, we retire the "sub" and the "xor", with the "sub" retiring before the "xor", as we retire in-order. We remove the "sub" and the "xor" from the ROB and place p5 and p3 back into the Free Pool (these were the old register mappings for x5 and x3, respectively).

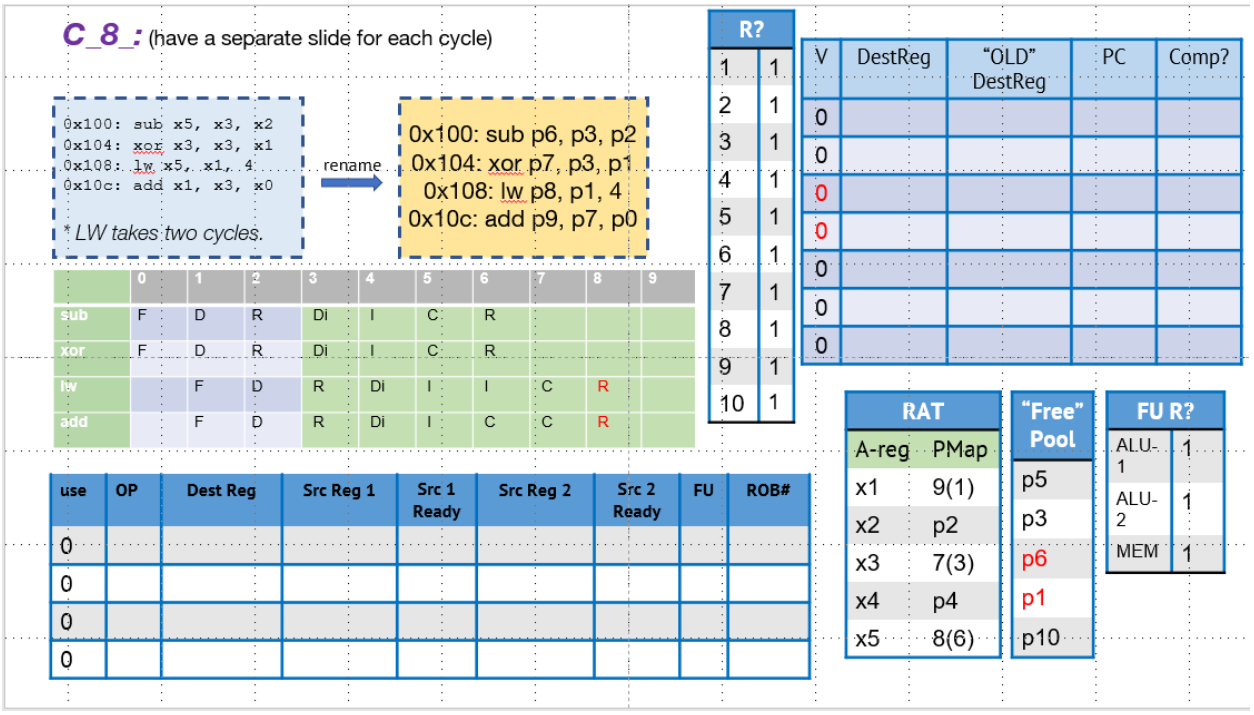
The "lw" is still in the issue stage as loads take two cycles in this problem. The MEM functional unit is still not ready

We complete the "add". We mark the "add" as "complete" in the ROB. We mark p9 as "ready". We mark ALU-1 as "ready".



In cycle 7, "lw" completes, as two cycles have passed in the issue stage. We mark "lw" as "complete" in the ROB. We mark p8 as "ready". We mark MEM as "ready".

The "add" is still in the complete stage as the "add" can't retire before the "lw", due to the in-order retiring requirement.



In cycle 8, we retire the "lw", then the "add". We clear both instructions from the ROB. We place p6 and p1 back into the Free Pool (these were the old register mappings for x5 and x1, respectively).

Q2. Consider the instruction sequence below. Assume we have forwarding with a 5-stage pipeline processor (in-order).

```
      addi x1, x0, 1
begin:
      beq x1, x0, end
      and x2, x0, x0
      addi x2, x2, 1
      add x3, x3, x2
      addi x1, x1, -1
      jal x0, begin
      xor x0, x0, x0
      xor x0, x0, x0
      xor x0, x0, x0
```

```
end:
      add x0, x0, x0
```

a. If we have an always <u>not taken</u> branch predictor with branches being resolved at MEM stage, draw the timeline (by putting f,d,e,m,w) for each instruction. You don't need to show the forwarding. If the instructions needed to be flushed and replaced by bubbles show it with b (e.g., eb, mb, and wb for bubbles in each stage). What is the total number of cycles?

SOLUTION

Before we start this problem, we need to assess the assembly execution, assuming perfect execution, to determine the flow of executed instructions. When we execute the "beq" for the first time, we do not branch as x1 != x0, and we fall through to the "and". When we reach the "jal", this is an unconditional branch, and we go back to "beq". The second time through the "beq", we see that x1 == x0, so we branch to the "add" instruction pointed to by the "end" label".

Inst	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
addi	f	d	e	m	w														
beq		f	d	e	m	w													
and			f	d	e	m	w												
addi				f	d	e	m	w											
add					f	d	e	m	w										
addi						f	d	e	m	w									
jal							f	d	e	m	w								
xor								f	d	eb	mb	wb							
xor									f	db	eb	mb	wb						

xor										fb	db	eb	mb	wb				
beq											f	d	e	m	w			
and												f	d	eb	mb	wb		
addi													f	db	eb	mb	wb	
add														fb	db	eb	mb	wb
add															f	d	e	m	w

Answer: 19 cycles

Since branches are always not taken, we correctly fall through the first "beq", with no penalty. When we reach the "jal", we assume not taken, so we execute the next three instructions (three "xor"), as branches aren't resolved till the MEM stage. At this point, we've realized we made a mistake, so we need to flush the three "xor" instructions. When we jump back to the "beq", we fall through again to the next instructions. The issue is we need to jump to the instruction that the "end" label points to this time, so we need to flush the "and", "addi", and "add" instructions. We then execute the final "add" at the "end" label.

b. Now assume that branches are resolved in the <u>DE stage</u> (Still always not taken). What is the new total number of cycles (draw the table)?

Ints	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
addi	f	d	e	m	w										
beq		f	d	e	m	w									
and			f	d	e	m	w								
addi				f	d	e	m	w							
add					f	d	e	m	w						

addi						f	d	e	m	w					
jal							f	d	e	m	w				
xor								fb	db	eb	mb	wb			
beq									f	d	e	m	w		
and										fb	db	eb	mb	wb	
add											f	d	e	m	w

Answer: 15 cycles. The only difference between part b and a is that instead of needing to flush three instructions in case of a misprediction of always not taken, we only need to flush one instruction, as we resolve branches in the decode stage instead of MEM. Thus, we need to flush one "xor" after the "jal", and the "and" after the second iteration through the "beq".

c. Now assume we add a (perfect) BTB with <u>always taken</u> predictions and the branches are still resolved in the DE stage. What is the new total number of cycles (draw the table)?

Ints	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
addi	f	d	e	m	w									
beq		f	d	e	m	w								
add			fb	db	eb	db	wb							
and				f	d	e	m	w						
addi					f	d	e	m	w					
add						f	d	e	m	w				
addi							f	d	e	m	w			
jal								f	d	e	m	w		
beq									f	d	e	m	w	
add										f	d	e	m	w

Answer: 14 cycles. We assume a perfect branch target buffer (BTB), which means the PC we need to jump to is always there. We also now assume that branches are always taken. In the first pass through the "beq", we realize we don't want to jump, in which we need to flush the "add" (the "beq" points to the "add" instruction at "end" label). We pass through the "jal" normally as we want to jump, as well as the second iteration through the "beq" as we jump to the "add" at "end" label.

d. Finally, assume we have a BHT added to our perfect BTB with 100% accuracy. What is the new total number of cycles (draw the table)?

Ints	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
addi	f	d	e	m	w								
beq		f	d	e	m	w							
and			f	d	e	m	w						
addi				f	d	e	m	w					
add					f	d	e	m	w				
addi						f	d	e	m	w			
jal							f	d	e	m	w		
beq								f	d	e	m	w	
add									f	d	e	m	w

Answer: 13 cycles. By adding a branch history table (BHT), with 100% accuracy, we always predict the right outcome for a branch. The first iteration through the "beq" is a success and we don't branch. The "jal" is a success as we branch. The second iteration through the "beq" is also a success and we branch (assuming 100% accuracy).