CS/ECE 552 : Introduction to Computer Architecture
Spring 2010
Prof. Wood
Problem Set #4 Solutions

Problem 4 – 15 Points

Consider the following code sequence and the datapath in figure 4.51 on page 362 of COD4e. Assuming the first instruction is fetched in cycle 1 and the branch is not taken, in which cycle does the 'add' instruction write its value to the register file? What if the branch IS taken? (Assume no branch prediction). Show pipeline diagrams.

```
          beq    $2, $1, loc
          xor    $1, $4, $3
          and    $3, $6, $7
          sub    $7, $5, $8
    loc:  add    $3, $6, $7
```

Not Taken: (assuming bypassing register file)

cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14
beq	IF	ID	EX	MEM	WB									
xor		*	*	*	IF	ID	EX	MEM	WB					
and						IF	ID	EX	MEM	WB				
sub							IF	ID	EX	MEM	WB			
add								IF	ID*	ID*	ID	EX	MEM	WB

Taken:

cycles	1	2	3	4	5	6	7	8	9
beq	IF	ID	EX	MEM	WB				
add		*	*	*	IF	ID	EX	MEM	WB

Problem 5 – 15 Points

Indicate all of the true, anti-, and output-dependencies in the following segment of MIPS assembly code:

```
    sub     $2, $7, $3
    add(1)  $4, $5, $6
    or      $1, $4, $5
    add(2)  $5, $2, $5
    sw      $4, 20($1)
    xor     $4, $1, $4
```

For the code above, which of the dependencies will manifest themselves as hazards in the pipeline in Figure 4.41 on page 355 of COD4e? How are these hazards resolved in this pipeline? Assuming the 'sub' instruction enters fetch (F) in cycle 1, in what cycle does the 'xor' instruction enter writeback (W)? Show your work in a pipeline diagram. (Assume that the register file cannot read and write the same register in the same cycle and get the new data.)

How does your answer change if you consider the pipeline in figure 4.60, on page 375 of COD4e? (Assume that the register file contains internal bypassing and can read and write the same register in the same cycle and get the new data.)

Answer:

There are 11 total dependencies, 6 of which are true.

True Dependencies (Read After Write):

1. sub$2 -> add$2
2. add(1)$4 -> or$4
3. add(1)$4 -> sw$4
4. add(1)$4 -> xor$4
5. or$1 -> sw$1
6. or$1 -> xor$1

Anti Dependencies (Write After Read):

7. or$5 -> add(2)$5
8. add(1)$5 -> add(2)$5
9. sw$4 ->xor$4
10. or$4 -> xor$4

Output Dependency (Write After Write):

11. and(1)$4 -> xor$4

Or and sw cause hazards.

cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sub	IF	ID	EX	M	WB										
add(1)		IF	ID	EX	M	WB									
or			IF	ID*	ID*	ID*	ID	EX	M	WB					
add(2)				IF*	IF*	IF*	IF	ID	EX	M	WB				
sw								IF	ID*	ID*	ID	EX	M	WB	
xor									IF*	IF*	IF	ID	EX	M	WB

All hazards resolved.

cycles	1	2	3	4	5	6	7	8	9	10
sub	IF	ID	EX	MEM	WB					
add(1)		IF	ID	EX	MEM	WB				
or			IF	ID	EX	MEM	WB			
add(2)				IF	ID	EX	MEM	WB		
sw					IF	ID	EX	MEM	WB	
xor						IF	ID	EX	MEM	WB

Problem 6 - 10 Points

Consider the pipeline in Figure 4.51 on page 362; assume predict-not-taken for branches and assume a "Hazard detection unit" in the ID stage as shown on page 379. Can an attempt to flush and an attempt to stall occur simultaneously? If so, do they result in conflicting actions and/or cooperating actions? If there are any cooperating actions, how do they work together? If there are an conflicting actions, which should take priority? What would you do in the design to make sure this works correctly? You may want to consider the following code sequence to help you answer this

question:

```
        beq $5, $2, loc  #assume that the branch is taken
        lw  $3, 40($4)
        add $2, $3, $4
        sw  $2, 40($4)
loc:    or  $5, $5, $2
```

Answer

A stall and flush can occur at the same time. This results in both cooperating and conflicting actions. Recall that a flush will turn all pipe stages into a nop and load the PC with a new value, while a stall will maintain the state of pipe stages prior to the stalling stage, turn the stage following into a nop, and keep the PC unchanged. The cooperating nop actions need no further attention; however an arbitration mechanism for dealing with conflicting actions must be implemented. The flush actions should take priority because its action removes the hazard that causes the stall. To make sure this works correctly, the hazard detection unit that decides on a stall should take the flush signal into account.

Problem 7 – 15 Points

Consider a pipeline where branches are predicted not-taken, and a taken branch introduces three-cycle penalty. Suppose you are considering adding a delayed branch slot to your instruction set architecture, so that taken branches would only have a two-cycle penalty.

Re-arrange or re-write each of the fragments so that it will work correctly with a branch delay slot and maximize performance. (The dots represent an unknown amount of other code that you can't change.) What is the average number of cycles that were saved or lost in each case if you used the delayed branch architecture? (Assume branches are taken 60% of the time.)

Consider the following three fragments of code:

```
Fragment 1:
        add $5, $5, $2
        beq $5, $6, Target
        lw $4, 0($2)
        .
        .
        .
Target: lw $1, 0($7)
        ...
```

```
Fragment 1: (with Delay Slot)
        add $5, $5, $2
        beqd $5, $6, Target
        nop
         lw $4, 0($2)
        .
        .
        .
Target: lw $1, 0($7)
        ...
```

Here we cannot place the first lw into the delay slot because $4 is not overwritten on the taken path. Likewise, the second lw cannot be placed in the delay slot because it is not known if $1 is overwritten on the not-taken path. Because the branch condition depends on $5, the add cannot be placed in the slot either. The correct answer is to insert a nop in the delay slot.

For the taken case, there is no performance difference from the original architecture (there is a three cycle delay in both). When the branch is not-taken, however, the performance is worse because not-taken prediction would have started the first lw a cycle earlier. Thus, the average cycles lost is: 60%*(0 cycle lost) + 40% * (1 cycles lost) = .4 cycles lost on average.

```
Fragment 2:

        add $5, $5, $2
        beq $5, $6, Target
        lw $4, 0($7)
        .
        .
        .
Target: sub $4, $8, $3
        ...
```

(Incorrect but Accepted Answer)

```
Fragment 2:

        add $5, $5, $2
        beqd $5, $6, Target
        lw $4, 0($7)
        .
        .
        .
Target: sub $4, $8, $3
        ...
```

One common answer here was to place the lw instruction into the delay slot. This seems like it should work, because $4 and $7 are independent of any instruction that comes before the branch. If the branch is not taken, the lw is executed normally. If the branch is taken, the value of $4 will be overwritten immediately by the sub instruction, allowing normal execution. The problem is that this solution does not address exceptions. By putting the lw into the delay slot, we could get an unexpected exception in the taken case.

For this reason, compilers do not usually schedule loads in delay slots. To appreciate this situation, consider the following code:

```
if (ptr != NULL)
   a = *ptr;
```

This would generate code that looks something like:

```
   lw $4, PTR
   beq $4, $0, null
   lw $1, PTR
   ...
null: ...
```

If you used delayed branches and moved the second lw into the delay slot, you will dereference *ptr even when it is null. In fragment #2, this case could arise if $7 is invalid when $5 == $6.

The correct answer involves duplicating the sub instruction. Since we know that the taken branch happens more often, we can optimize for this case. First, we put a copy of the sub instruction in the delay slot, and then jump to a new branch target called NewTarg, which sits one instruction after

Target. This way we've covered all of our bases: in the taken case, we have executed the sub correctly, in the non-taken case, the extra sub is overwritten immediately by the lw. Here the average cycles gained is: 60%*(1 cycle gained) + 40% * (1 cycles lost) = .2 cycles gained on average.

```
Fragment 2:

         add $5, $5, $2
         beq $5, $6, Target
         lw $4, 0($7)
         .
         .
         .
Target:  sub $4, $8, $3
         ...
```

```
Fragment 2:  (w/ Delay Slot)

         add $5, $5, $2
         beqd $5, $6, NewTarg
         sub $4, $8, $3
          lw $4, 0($7)
         .
         .
         .
Target:   sub $4, $8, $3
NewTarg: ...
```

```
Fragment 3:

         movei $2, 21
         .
         .
         .
         addi $4, $4, 1
         beq $4, $2, Target
         .
         .
         .
Target:  ...
```

```
Fragment 3: (with Delay Slot)

         movei $2, 21
         .
         .
         .
         addi $4, $4, 1
         beqd $4, $2, Target
          nop
         .
         .
         .
Target:  ...
```

In the above fragment, the only instruction that can be placed in the delay slot is a nop. One common answer was to decrement the immediate value in the movei instruction to 20 and then place the addi in the delay slot. However, because we don't know what happens to $2 between the movei and addi, this is not correct. (for example, imagine that the instruction right after movei was another movei that loaded 21 again)

Because we insert a nop into the delay slot, the effective penalty of the taken branch is 3 cycles. Thus, the average cycles lost is: 60%*(0 cycle lost) + 40% * (1 cycles lost) = .4 cycles lost on average.