Computer Architecture — Due on Wednesday, April 8, 2015

Assignment 4

CS3350B — *University of Western Ontario*

Submission instructions. With each assignment, you are required to hand in an Assignment Submission Form, on which you certify that the material you've handed in is exclusively your own work. Put this form inside a 9-by-12-inch envelope along with your assignment, bearing your name and the course number CS3350b, which will be used to return your assignment. Hand in your envelope (contained with your completed assignment and the Assignment Submission Form) in class, or drop it in the CS3350b locker (locker #308, on the third floor of the Middlesex College building) by the midnight on the due date.

PROBLEM 1. [20 points] Consider the following `MIPS` instructions to be executed on a pipelined processor:

```
and $s0, $s1, $s2
xor $s2, $s0, $s1
```

where `$s1 and $s2` are assumed to hold values *before* executing these two instructions. The processor uses a 5-stage pipeline, as defined in class. The successive five stages of this pipeline are denoted by IF, ID, EXE, MEM, WB.

1.1 Indicate dependences and their type (read after write or write after read) among the above two MIPS instructions.

$s0: read after write; $s2: write after read.

1.2 Assume that there is no forwarding mechanisms in this pipelined processor. Then, indicate hazards and add the appropriate `nop` (or `stall`) instructions to eliminate those hazards in the following pipeline execution diagram:

	Clock cycles								
Instructions	1	2	3	4	5	6	7	8	9	…
`and`	IF	ID	EX	MEM	WB					
`xor`		IF	ID	EX	MEM	WB				

A bubble is inserted in clock cycle 4, by changing the `xor` instruction to a `nop`. However, the value of $s2 is still not ready because of no forwarding policy. Another bubble is inserted in clock cycle 5, by changing the `xor` instruction to a `nop`.

	Clock							
Instructions	1	2	3	4	5	6	7	8
`and`	IF	ID	EX	MEM	WB			
`nop`		*IF*	*ID*	*bubble*	*bubble*	*bubble*		
`nop`			*IF*	*ID*	*bubble*	*bubble*	*bubble*	
`xor`				IF	ID	EX	MEM	WB

1.3 Assume that there is an ALU-ALU forwarding but no other forwarding mechanisms, like a forwarding from the MEM to the EX stage. Indicate hazards and add the appropriate `nop` instructions to eliminate those hazards, if any.

This hazard can be solved by ALU-ALU forwarding.

PROBLEM 2. [20 points] This exercise is intended to help you understand the relationship between delay slots, control hazards and branch execution in a pipelined processor. We assume that the following `MIPS` code

```
loop:  lw $t2, 0($s2)
       addi $s2, $s2, 4
       bne $t2, $0, loop
       sll $t2, $t0, 2      # assuming $t0 holds some value
       sw $t2, 0($s2)
```

is executed on a pipelined processor with a 5-stage pipeline and full forwarding (that is all the forwarding mechanisms defined in class). We assume that `$s2` initially stores the base address of a 32-bit integer array `A` in `C` code. Assume that `A[] = {5, 73, 0}` and that there is no special branch comparator inserted in the processor.

Use a table similar in format as below to draw the pipeline execution diagram of the above `MIPS` code applied to the above array `A`.

	Clock									
Instructions	1	2	3	4	5	6	7	8	9	...
`lw`	IF	ID	EX	MEM	WB					
⋮										

2.1 Assume that the processor uses none of the following techniques with branch instruction:

– delayed branch (see Slide 25-26 of the PDF version of Lecture 6.2)

Also, assume that branches execute in the EX stage. Draw the pipeline execution diagram until the second iteration finishes. For each forwarding, indicate its type.

	Clock									
Instructions	1	2	3	4	5	6	7	8	9	10
`lw`	IF	ID	EX	MEM(1)	WB					
`addi`		IF	ID	EX	MEM	WB				
`bne`			IF	ID	EX(2)	MEM	WB			
`nop`				*bubble*	*bubble*	*bubble*	*bubble*	*bubble*		
`nop`					*bubble*	*bubble*	*bubble*	*bubble*	*bubble*	
`lw`						IF	ID	EX	MEM(1)	WB

	Clock									
Instructions	7	8	9	10	11	12	13	14	15	16
`addi`	IF	ID	EX	MEM	WB					
`bne`		IF	ID	EX$^{(2)}$	MEM	WB				
`nop`			*bubble*	*bubble*	*bubble*	*bubble*	*bubble*			
`nop`				*bubble*	*bubble*	*bubble*	*bubble*	*bubble*		
`lw`						IF	ID	EX	MEM	WB

(1) MEM-ALU forwarding
(2) ALU-IF forwarding

2.2 Assume that delay slots are used and the pipeline for the branches are not taken, that is, we continue execution down the sequential instruction stream. Draw the pipeline execution diagram until the above code ends.

	Clock										
Instructions	1	2	3	4	5	6	7	8	9	10	11
`lw`	IF	ID	EX	MEM	WB						
`addi`		IF	ID	EX	MEM	WB					
`bne`			IF	ID	EX	MEM	WB				
`sll`				IF	*discarded*						
`lw`					IF	ID	EX	MEM	WB		
`addi`						IF	ID	EX	MEM	WB	
`bne`							IF	ID	EX	MEM	WB

	Clock									
Instructions	8	9	10	11	12	13	14	15	16	17
`sll`	IF	*discarded*								
`lw`		IF	ID	EX	MEM	WB				
`addi`			IF	ID	EX	MEM	WB			
`bne`				IF	ID	EX	MEM	WB		
`sll`					IF	ID	EX	MEM	WB	
`sw`						IF	ID	EX	MEM	WB

PROBLEM 3. [20 points] Consider the following `C` code:

```
for (i = 0; i < n; ++i)
   a[i] = b[i] + i;
```

where `a` and `b` are 32-bit integer arrays of size `n`. We give a corresponding `MIPS` instruction sequence:

```
       add $t0, $0, $0     # $t0 = 0, which corresponds to i in C code
loop:  lw $s1, 0($s4)      # assume $s4 stores the base address of array b
       add $s0, $s1, $t0   # $s0 gets b[i] + i
       sw $s0, 0($s2)      # assume $s2 stores the base address of array a
       addi $t0, $t0, 1    # ++i
       addi $s2, $s2, 4    # get address of a[i+1]
       addi $s4, $s4, 4    # get address of b[i+1]
       slt $t2, $t0, $s5   # assume that $s5 holds n
       bne $t2, $0, loop   # if $t2 == 1, go to loop
```

Assume that the above MIPS instructions will be executed on a 5-stage pipelined processor (as defined in class). Also, one can ignore control hazards (but not data hazards, of course) and assume that full-forwarding (as defined in class) is implemented.

3.1 Draw the pipeline execution diagram without unrolling (one iteration of the loop would be enough) and compute the average CPI (clock cycle per instruction) of the loop. You may consider the two cases: using or not using instruction re-ordering

	Clock									
Instructions	1	2	3	4	5	6	7	8	9	10
`lw`	IF	ID	EX	MEM	WB					
`addi $s4`		IF	ID	EX	MEM	WB				
`add`			IF	ID	EX	MEM	WB			
`sw`				IF	ID	EX	MEM	WB		
`addi`					IF	ID	EX	MEM	WB	
`addi`						IF	ID	EX	MEM	WB

	Clock					
Instructions	7	8	9	10	11	12
`slt`	IF	ID	EX	MEM	WB	
`bne`		IF	ID	EX	MEM	WB

Thus, the CPI of the loop is 12 / 8 = 1.5.

3.2 Apply loop unrolling (as well as instruction re-ordering, if you like) on the above MIPS code for two iterations. Write the corresponding MIPS instruction code. You may consider the two cases: using or not using 2-issue MIPS instructions, like on Slides 12 and 13 of the PDF version of the set of slides 6.2.

```
        add $t0, $0, $0
loop:   lw $s1, 0($s4)
        addi $s4, $s4, 4     # get address of b[i+1]
        add $s0, $s1, $t0
        sw $s0, 0($s2)
        addi $t0, $t0, 1
        lw $s1, 4($s4)       # load b[i+1]
        addi $s4, $s4, 4     # get address of b[i+2]
        add $s0, $s1, $t0    # b[i+1] + i+1
        sw $s0, 4($s2)       # write to a[i+1]
        addi $t0, $t0, 1     # ++i
        addi $s2, $s2, 8     # get address of a[i+2]
        slt $t2, $t0, $s5
        bne $t2, $0, loop
```

3.3 Draw the pipeline execution diagram of your `MIPS` instructions (one iteration of the new loop would be enough) and compute the average CPI of the loop.

	Clock										
Instructions	1	2	3	4	5	6	7	8	9	10	11
`lw`	IF	ID	EX	MEM	WB						
`addi`		IF	ID	EX	MEM	WB					
`add`			IF	ID	EX	MEM	WB				
`sw`				IF	ID	EX	MEM	WB			
`addi`					IF	ID	EX	MEM	WB		
`lw`						IF	ID	EX	MEM	WB	
`addi`							IF	ID	EX	MEM	WB

	Clock									
Instructions	8	9	10	11	12	13	14	15	16	17
`add`	IF	ID	EX	MEM	WB					
`sw`		IF	ID	EX	MEM	WB				
`addi`			IF	ID	EX	MEM	WB			
`addi`				IF	ID	EX	MEM	WB		
`slt`					IF	ID	EX	MEM	WB	
`bne`						IF	ID	EX	MEM	WB

Thus, the CPI of the loop is 17 / 12 ≈ 1.4.

PROBLEM 4. [10 points]

A 4-processor shared-memory multiprocessor configuration implements write-back cache using the MESI (**M**odified, **E**xclusive, **S**hared, **I**nvalid) algorithm for cache coherency. Assume that location 0x0010 is not in any cache at the start of the following sequence.

Consider the following read/write operations:

(a) Processor 0 reads from location 0x0010
(b) Processor 0 writes to location 0x0010
(c) Processor 2 reads from location 0x0010
(d) Processor 3 reads from location 0x0010
(e) Processor 2 writes to location 0x0010
(f) Processor 1 reads from location 0x0010
(g) Processor 3 writes to location 0x0010

Show the state (M, E, S or I) for the cache line containing location 0x0010 in each processor cache after each operation. Also note any transfers to/from memory if any occurs.

Solutions to operations (a) and (b) are given in the following table. Complete the table for operations (c) - (g).

	State				
Action	P0	P1	P2	P3	Memory transfers
(a) P0 read miss	E	I	I	I	P0 reads a cache line from memory
(b) P0 write hit	M	I	I	I	-
(c) P2 read miss	*S*	*I*	*S*	*I*	*P0 writes a cache line to memory;* *P2 reads a cache line from memory*
(d) P3 read miss	*S*	*I*	*S*	*S*	*P3 reads a cache line from memory*
(e) P2 write hit	*I*	*I*	*M*	*I*	-
(f) P1 read miss	*I*	*S*	*S*	*I*	*P2 writes a cache line to memory;* *P1 reads a cache line from memory*
(g) P3 write miss	*I*	*I*	*I*	*M*	*P3 reads a cache line from bus or memory*

PROBLEM 5. [15 points] Consider a shared-memory multiprocessor that consists of three processor/cache units where cache coherence is maintained by a MESI protocol. The private caches are direct mapped. Assume that words X1, X2 and X3 are in the same cache line. Given the following sequence of events, identify each miss as a cold miss (CM), a true sharing miss (TM), a false sharing miss (FM), or a hit (H). Explain briefly the reasons.

Clock	Processor 1	Processor 2	Processor 3	CM, TM, FM or H
1	Read X1			*CM*
2		Read X2		*CM*
3			Read X3	*CM*
4	Write X1			*H*
5			Write X3	*FM*
6		Read X1		*TM*
7	Write X2			*FM*
8			Read X1	*FM*
9			Read X2	*H*

PROBLEM 6. [15 points] Consider a pipe-lined process (like the laundry example given in class) with s stages. Assuming n tasks are being processed by this pipe line. In each of the following scenario, compute

1. the speedup w.r.t a serial execution,
2. the percentage of time during which the pipeline runs at full occupancy

6.1 Each stage runs within the same amount of time (as we did in Quiz 3)

Speedup $= (n \times s)/(n + s - 1)$, *the percentage of time* $= (n - s + 1)/(n + s - 1)$.

6.2 Each stage, but the first one, runs within t units of time (say pico-seconds) meanwhile the first stage runs within $r\,t$ units of time where r is a constant greater than one, thus, the first stage is slower than the other ones.

Full occupancy happens whenever s tasks utilize s stages. Consider as origin of time, the time at which the first task enters the first stage. From that moment on, every $r \times t$ units of times, a new task enters the first stage. Since one task runs within $r \times t + (s-1)t$ units of times, full occupancy occurs provided that the following holds:

$$(s-1)r \times t \ \leq \ r \times t + (s-1)t,$$

that is,

$$(s-2)r \times t \ \leq \ (s-1)t,$$

hence,

$$r \ \leq \ \frac{s-1}{s-2}. \tag{1}$$

Hence Equation (1) is a necessary and sufficient condition for the first task to participate to a full occupancy for a period of time given by

$$\tau \ = \ r \times t + (s-1)t - (s-1)r \times t = (2r + s - r \times s - 1)t.$$

More generally, Equation (1) is a necessary and sufficient condition for the tasks $i, i+1, \ldots, i+s-1$ to experience full occupancy together for a period of time given by τ. Therefore, full occupancy happens (from time to time) if and only if Equation (1) holds. Consequently, if r is sufficiently large, full occupancy never happens, see Figure 1.

Consider the i-th task, for $s \leq i \leq n - s$. It follows from the previous reasoning that this task will be involved in full occupancy for a total period of time of $s \times \tau$ during its life time. Therefore, the percentage of time during which full occupancy happens is:

$$\frac{s(2r + s - rs + -1)}{r + s - 1}$$

For instance, $s = 10$ and $r = 17/16$ satisfy Equation (1) and lead to a percentage of $80/161$. Consequently, having a first stage which is slightly slower than the others can cause full occupancy to drop to about 50 %.

Finally, the speedup is independent of the fact that full occupancy is happening or not; it is given by:

$$\frac{n \times (r + (s-1))}{n \times r + (s-1)},$$

which is asymptotically equivalent to $\frac{r+(s-1)}{r}$. With $s = 10$ and $r = 17/16$, this gives $161/17$, while $s = 10$ and $r = 1$ give 10, which is slightly better.

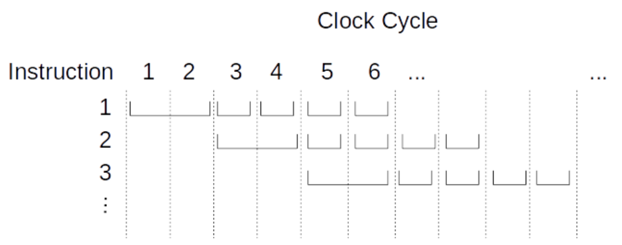

Figure 1: The pipeline execution diagram with the first stage running $r\,t$ units.

6.3 Each stage, but the last one, runs within t units of time (say pico-seconds) meanwhile the last stage runs within $r\,t$ units of time where r is a constant greater than one, thus, the last stage is slower than the other ones.

One should reason as in 6.2.

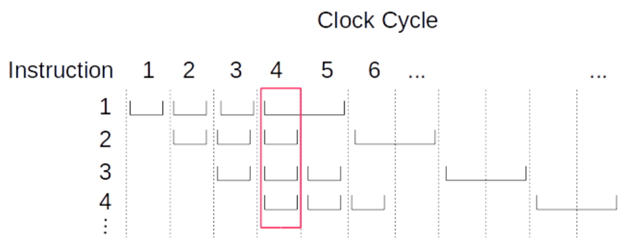

Figure 2: The pipeline execution diagram with the last stage running $r\,t$ units.