CS 2410 Mid term (fall 2018) **Name**:_____

Question 1 (6+6+3=15 points): Consider two machines, the first being a 5-stage pipeline operating at 1ns clock and the second is a 12-stage pipeline operating at 0.7ns clock. Due to data hazards, the 5-stage pipeline experiences a stall every 5 instructions, whereas the 12-stage pipeline experiences 4 stalls every 10 instructions. In addition, branches constitute 20% of the executed instructions with 60% of the branches taken.

a) Ignoring control hazards and taking into consideration only data hazards, which of the two pipelines is more efficient?

$CPI_5 = 1 + 0.2 = 1.2$
Average time per instruction (5-stage) = 1.2 * 1 = 1.2 ns

$CPI_{12} = 1 + 0.4 = 1.4$
Average time per instruction (12-stage) = 1.4 * 0.7 = 0.98 ns

The 12-stage pipeline is more efficient

b) Taking into account both control and data hazards, which of the two pipelines is more efficient assuming that branches are resolved in the third stage of the 5-stage pipeline and in the seventh stage of the 12-stage pipeline?. Branch are predicted "not taken" in both pipelines.

$CPI_5 = 1.2 + 0.2 * 0.6 * 2 = 1.44$
Average time per instruction (5-stage) = 1.44 * 1 = 1.44 ns

$CPI_{12} = 1.4 + 0.2 * 0.6 * 6 = 1.4 + 0.72 = 2.12$
Average time per instruction (12-stage) = 2.12 * 0.7 = 1.484 ns

The 5-stage pipeline is more efficient

c) Now assume that in the 12-stage pipeline, the branch target is determined in the third stage while the branch condition is determined in the seventh stage. Repeat part b if branches are predicted "taken" in the 12-stage pipeline (still predicted "not taken" in the 5-stage pipeline).

$CPI_{12} = 1.4 + 0.2 * (0.6 * 2 + 0.4 * 4) = 1.4 + 0.56 = 1.96$
Average time per instruction (12-stage) = 1.96 * 0.7 = 1.372 ns

The 12-stage pipeline is more efficient

Question 2 (15 points)

Assuming that Tomosulo's algorithm with speculation is used in a pipeline architecture with one FP multiply unit, one FP add unit, and two integer units. Assume a 2-issue pipeline with 2 CDBs and 10 reorder buffers. If during the execution of the following loop

```
L:   L.D     F1, 0(R1)
     MUL.D   F2, F1, F0
     L.D     F3, 0(R2)
     ADD.D   F4, F2, F3
     S.D     F4, 0(R2)
     DADDI   R1, R1, -8
     DADDI   R2, R2, -8
     BEQZ    R2, L
```

the ROB status at the beginning of a cycle, C, is as follows:

Name	Busy	Instruction	State	Destination	value
ROB0	No	L.D F1, 0(R1)	committed	F1	
ROB1	Yes	MUL.D F2, F1, F0	committed	F2	
ROB2	Yes	L.D F3, 0(R2)	Wrote back result	F3	
ROB3	Yes	ADD.D F4, F2, F3	executing	F4	
ROB4	Yes	S.D F4, 0(R2)	executing	20010	
ROB5	Yes	DADDI R1, R1, -8	Wrote back result	R1	
ROB6	Yes	DADDI R2, R2, -8	Wrote back result	R2	
ROB7	Yes	BEQZ R2, L	issued		
ROB8	Yes	L.D F1, 0(R1)	issued	F1	
ROB9	Yes	MUL.D F2, F1, F0	issued	F2	

(a) Show the content of the register status table at the beginning of cycle C assuming that no renaming registers are used. Note that an instruction marked "committed" will be replaced in cycle C by a new issued instruction.

The table should rename a register using the ROB reserved by the last instruction (issued but not committed yet) writing to that register.

	F0	F1	F2	F3	F4	…..	R1	R2	…..
ROB #		ROB8	ROB9	ROB2	ROB3		ROB5	ROB6	

(b) Show the content of the register status table at the beginning of the next cycle, C+1. Note that the ROB is implemented as a circular buffer.

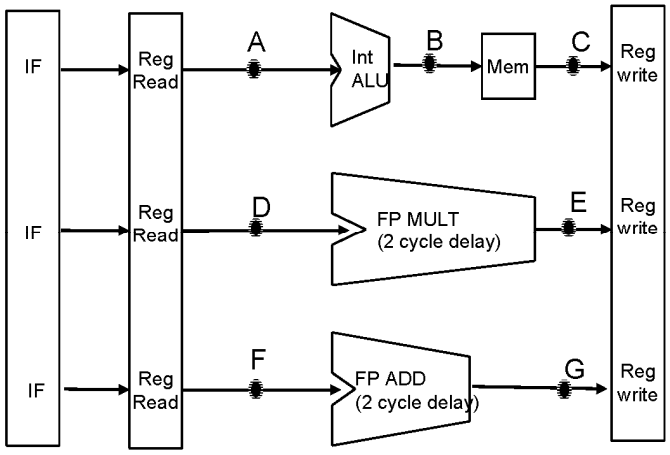
The instruction in ROB2 will be committed and the two next instructions "L.D F3, 0(R2)" and "ADD.D F4, F2, F3" will be issued and reserve ROB0 and ROB1, hence renaming F3 and F4 using ROB0 and ROB1.

	F0	F1	F2	F3	F4	…..	R1	R2	…..
ROB #		ROB8	ROB9	ROB0	ROB1		ROB5	ROB6	

Question 3 (2*7+1=15 points) Consider the VLIW architecture shown below which consists of three units; an ALU unit (1 cycle delay), a FP add unit (2 cycle delay) and a FP mult unit (2 cycle delay). Assume that the architecture executes the shown MIPS instructions (L.D, S.D, ADD.D, MULT.D, L, S, ADD), with each VLIW instruction composed of up to three MIPS instructions, one for each unit. The points marked A, B, … , G in the figure are used to identify forwarding paths. For example, C → D denotes a forwarding path from point C to point D. The usefulness of the path C → D can be demonstrated by the example shown in the following table where it is used to forward F0 from the "LD F0, 100(R1)" instruction to the "MULT.D F1, F0, F2" instruction:

Example for the use of the forwarding path from **C → D**			
	Int ALU	FP MULT	FP ADD
Cycle 1	L.D F0, 100(R1)		
Cycle 2			
Cycle 3		MULT.D F1, F0, F2	

```
L.D      F1, I(R1)     // load to FP reg.
S.D      F1, I(R1)     // store to FP reg.
ADD.D    F1, F2, F3    // FP add
MULT.D   F1, F2, F3    // FP multiply
L        R1, I(R2)     // load to int reg.
S        R1, I(R2)     // store to int reg.
ADD      R1, R2, R3    // Integer add
```

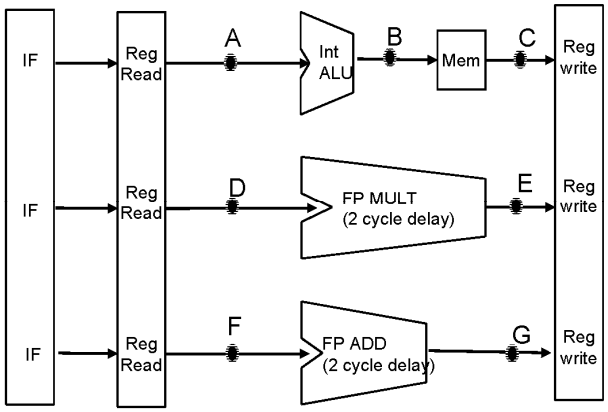


Complete the following tables to demonstrate an example of a use case for the forwarding path indicated in each table. You may use any of the above instructions in any order and with any registers. If a path does not have any use given the above instructions (should not be implemented) indicate so by writing "not useful" next to the table.

Example for the use of the forwarding path from **E → F**			
	Int ALU	FP MULT	FP ADD
Cycle 1		MULT.D F0, F1, F2	
Cycle 2			
Cycle 3			ADD.D F3, F3, F0

Example for the use of the forwarding path from **E → A**			
	Int ALU	FP MULT	FP ADD
Cycle 1		MULT.D F0, F1, F2	
Cycle 2			
Cycle 3	S.D F0, 100(R1)		

```
L.D      F1, I(R1)     // load to FP reg.
S.D      F1, I(R1)     // store to FP reg.
ADD.D    F1, F2, F3    // FP add
MULT.D   F1, F2, F3    // FP multiply
L        R1, I(R2)     // load to int reg.
S        R1, I(R2)     // store to int reg.
ADD      R1, R2, R3    // Integer add
```



Example for the use of the forwarding path from **E → B**			
	Int ALU	FP MULT	FP ADD
Cycle 1		MULT.D F0, F1, F2	
Cycle 2	S.D F0, 100(R1)		
Cycle 3			

Example for the use of the forwarding path from **C → A**			
	Int ALU	FP MULT	FP ADD
Cycle 1	ADD R0, R1, R2		
Cycle 2			
Cycle 3	ADD R4, R3, R0		

Example for the use of the forwarding path from **C → B**			
	Int ALU	FP MULT	FP ADD
Cycle 1	L R1, 100(R2)		
Cycle 2	S R1, 200(R2)		
Cycle 3			

Example for the use of the forwarding path from **B → A**			
	Int ALU	FP MULT	FP ADD
Cycle 1	ADD R0, R1, R2		
Cycle 2	ADD R4, R3, R0		
Cycle 3			

Example for the use of the forwarding path from **B → D**			
	Int ALU	FP MULT	FP ADD
Cycle 1	NOT USEFUL		
Cycle 2			
Cycle 3			

Question 4 (4+6=10 points): Consider a superscalar architecture with two pipelined units, one for load/store and one for ALU instructions. The following two tables indicate the order of execution of two threads, A and B, and the latencies mandated by dependences between instructions. For example, A2 and A3 should execute after A1 and there should be at least four cycles between the execution of A3 and A5 and at least three cycles between A4 and A5. In other words, the tables indicate the schedule if the instructions of each of the threads are executed with no multithreading.

Note that an instruction that executes on one pipeline (for example A1 on the load/store pipeline) cannot execute on the other pipeline. Also you cannot issue instructions out of order.

time	Load/store pipeline	ALU pipeline
t	A1	
t+1	A3	A2
t+2		A4
t+3		
t+4		
t+5		
t+6	A5	
t+7	A6	A7

time	Load/store pipeline	ALU pipeline
t	B1	B2
t+1		B3
t+2	B4	B5
t+3		
t+4	B6	
t+5		B7
t+6	B8	
t+7		

Show the execution schedule for the two threads on the two pipelines assuming:

(a) Fine grain multithreading

time	Load/store pipeline	ALU pipeline
t	A1	
t+1	B1	B2
t+2	A3	A2
t+3		B3
t+4		A4
t+5	B4	B5
t+6		
t+7	B6	
t+8	A5	
t+9		B7
t+10	A6	A7
t+11	B8	
t+12		
t+13		
t+14		

(b) Simultaneous multithreading (with priority always given to thread A)

time	Load/store pipeline	ALU pipeline
t	A1	
t+1	A3	A2
t+2	B1	A4
t+3		B2
t+4		B3
t+5	B4	B5
t+6	A5	
t+7	A6	A7
t+8	B6	
t+9		B7
t+10	B8	
t+11		
t+12		
t+13		
t+14		

Question 5 (4+4+3+4=15 points): Consider a system with a two level cache where 20% of the memory references are “stores” and 80% are “loads”. Out of all the memory references made by the CPU, 94% are found in the L1 cache and 50% of the L1 cache misses are found in L2. Assume that it takes one cycle to access L1, 10 cycles to access L2 (either to write a word or a block or to read a word or a block) and 100 cycles to access memory (either to write or read a block). Assume also that “write allocate” is used and that there is a very large and fast "write buffer" between L2 and memory while there is no “write buffers” between L1 and L2 (the CPU stalls until the writing is complete onto L2).

(a) Compute the average memory access time if L2 uses "write back" while L1 uses "write through”.

(i) the average number of cycles for a load is

All loads go to L1 + 0.06 go to L2 and cause data to move from L2 to L1 + 0.03 miss in L2

$1 + 0.06 * 10 + 0.03 * 100 = 4.6$ cycles

(ii) the average number of cycles for a store is

All stores go to L2 + 0.06 misses in L1 will move data from L2 to L1 + 0.03 misses in L2

$10 + 0.06 * 10 + 0.3 * 100 = 13.6$ cycles

(iii) the average memory access time is

$0.2 * 13.6 + 0.8 * 4.6 = 2.72 + 3.68 = 6.4$ cycles

(b) Computer the average memory access time if both L1 and L2 use "write back" assuming that, on average, 40% of the evicted blocks are dirty.

All go to L1 + 0.06 go to L2 and cause data to move from L2 to L1 + 0.03 miss in L2

$1 + 0.06 * (0.4 * 10 + 10) + 0.03 * 100 = 4.84$ cycles

Note that writing a dirty block back from L1 to L2 costs 10 cycles (no write buffer) while writing a dirty block back from L2 to memory does not cost anything (a very large write buffer).

Question 6 (5*3=15 points)

To access data from a typical DRAM, we first have to activate the appropriate row. Assume that this brings an entire page of size 4KB to the row buffer. Then we select a particular column from the row buffer. If subsequent accesses to the DRAM are to the same page, then we can skip the activation step; otherwise, we have to close the current row and precharge the bitlines for the next activation. Another DRAM popular policy is to proactively close a row and precharge the bitlines as soon as an access is over. For this question, assume that it takes 12 cycles to close a row and precharge, 8 cycles to activate a row and 2 cycles to read a column.

(a) Complete the following assuming that the DRAM is lightly accessed (accesses are separated by tens of cycles):

a. Access time for closed row policy is

open new row + column access = 10

b. Access time for open row policy in case of a row buffer hit is

column access = 2

c. Access time for open row policy in case of a row buffer miss is

close old row + open new row + column access = 22

(b) For what values of the row buffer hit rate, r, would you chose the open row policy if accesses are separated by at least 25 cycles?

If $2r + 22 (1-r) < 10$ that is if $r > 12/20 = 60\%$

(c) How will your answer to part b change if the memory is accessed very heavily (an access to a bank is initiated as soon as the memory bank is not busy)?

In this case, the access time for the closed row policy will be 22 since each access will have to wait until the row for the previous access is closed. Hence, the open row policy is always better.

Question 7 (6+6+3=15 points): Consider the multiplication of two *100x100* matrices on a system with a 64KB fully associative cache and 32B blocks (4 words, each 8 bytes). Answer the following assuming row-wise allocation and noting that 100*100*8 ≈ 80KB is needed to store a *100* x *100* matrix.

(a) When executing the following code

```
for (i = 0 ; i < 100 ; i++)
        for (j = 0 ; j < 100 ; j ++)
        {   r = 0;
            for (k = 0; k < 100; k++)
                  r = r + A[i][k]*B[k][j];
            C[i][j] = r;
        };
```

the average miss rate when accessing the elements of A is __0.25%_____

the average miss rate when accessing the elements of B is ___25%___

(b) When executing the following code, which computes two rows of C in each iteration of the outer loop

```
for (i = 0 ; i < 100 ; i=i+2)
        for (j = 0 ; j < 100 ; j ++)
        {   r1 = 0 ;
            r2 = 0 ;
            for (k = 0; k < 100; k++)
                  { r1 = r1 + A[i][k]*B[k][j];
                    r2 = r2 + A[i+1][k]*B[k][j];
                  }
           C[i][j] = r1;
           C[i+1][j] = r2;
        };
```

the average miss rate when accessing the elements of A is ___0.25%____

the average miss rate when accessing the elements of B is ___12.5%___

(c) It is possible to generalize the code in (b) so that *m* rows of C are computed in each iteration of the outer loop, where *m* is one of 4, 5, 10, 20, 25 or 50. What value of *m* would you use to obtain the lowest average cache miss rate of A and B (you do not have to write the code for that generalization – also you may ignore the miss rate for C)? – You need to give a good reason for your answer.

800 bytes are needed to store one row of A (or column of B). Hence, the 64KB cache can hold 80 rows/columns of A/B. Hence, I would use *m* = 25 since it is the largest value such that the cache can hold *m* rows of A and *m* column of B.

This will keep the average miss rate of A at 0.25% and reduce the miss rate for B by a factor of 25 (to 25% / 25 = 1%).