McGill University
ECSE 425: COMPUTER ORGANIZATION AND ARCHITECTURE
Winter 2015

Final Examination

9:00 AM – 12:00 Noon, April 22th, 2015

Duration: 3 hours

Examiner: Prof. B. H. Meyer **Associate Examiner: Prof. W. J. Gross**

- **Write your name and student number in the space below. Do the same on the top of each page of this exam.**
- **The exam is 18 pages long. Please check that you have all 18 pages.**
- **There are six questions for a total of 160 points. Not all parts of all questions are worth the same number of points; read the whole exam first and spend your time wisely!**
- **This is a closed-book exam. You may use one double-sided sheet of notes; please turn this sheet in with your exam.**
- **Faculty standard calculators are permitted; no cell phones or laptops.**
- **Clearly state any assumptions you make.**
- **Write your answers in the space provided. Show your work to receive full credit, and clearly indicate your final answer.**

Name: ______________________________

Student Number: ______________________________

Name: ID:

Q1: ___________/40	**Q2: ___________/30**	**Q3: ___________/20**
Q4: ___________/20	**Q5: ___________/20**	**Q6: ___________/30**

Total:

Name: ID:

Question 1: Are These *Short Answer* Questions? (40 pts total)

Respond to each of the following; two to four sentences should be adequate in each case.

a. **(8 pts total)** Consider the following memory hierarchy. L1 is virtually indexed, physically tagged, and two-way set-associative. L2 is physically indexed and 16-way set-associative. L1 is write-back; L2 is write-through; data to main memory is buffered. Describe, in order of occurrence, all of the steps taken to service a read request when:

i. **(4 pts)** The access results in a TLB hit, L1 cache miss, L2 cache hit.

1. First, in parallel, both the TLB and L1 cache are accessed. The L1 cache is indexed using the page offset. The virtual page number (VPN) goes to the TLB.
2. Given a TLB hit occurs, the VPN is translated into a physical page number (PPN). The L1 cache miss means this the tag—the PPN—doesn't match either of the two tags in the set in the L1 cache.
3. If the accessed L1 block is dirty, it is written back to L2 (and main memory).
4. The PPN and page offset are then combined into a physical address that is presented to L2, where the access hits in one of 16 ways in the index set.
5. The requested portion of the cache block is then delivered to the processor, while the entire cache block placed in L1.

ii. **(4 pts)** The access results in a TLB miss, L1 cache hit.

1. First, in parallel, both the TLB and L1 cache are accessed. The L1 cache is indexed using the page offset. The virtual page number (VPN) goes to the TLB.
2. When the TLB miss occurs, the page table is walked until the appropriate translation is found, and the TLB is updated.
3. The VPN is then translated into a physical page number (PPN). Given that the access hits in L1, the PPN matches one of the tags of the accessed set in L1, and the desired portion of the block is delivered to the processor.

Name: ID:

b. **(4 pts)** List two reasons clock cycle time deviates from the ideal in a pipelined processor.

- Pipeline registers
- Unbalanced pipeline stages
- Clock skew

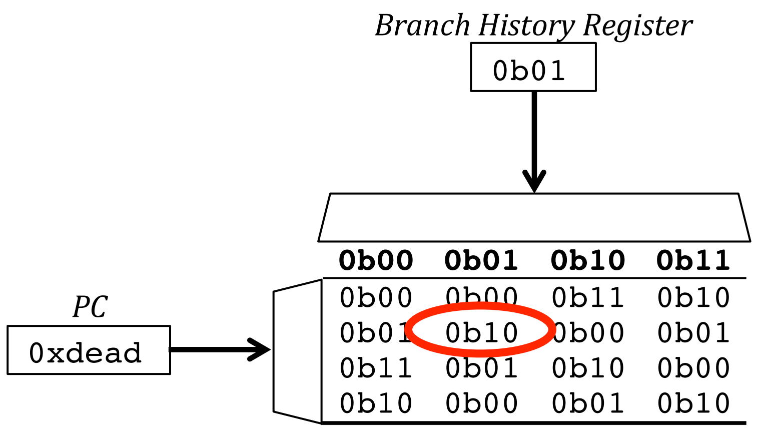
c. **(4 pts total)** Consider the following branch predictor state for a (2, 2) *correlating branch predictor*. All state is given in binary (e.g., `0bxy` where `x` and `y` are bits). Assume `0b1` indicates *taken* while `0b0` indicates *not taken*.

i. **(2 pts)** Given a branch instruction at address `0xdead`, *circle the branch predictor state* that is used to determine whether to take or not take the branch. *Hint*: not all state is used; don't circle everything.

ii. **(2 pts)** What prediction is made? Indicate how predictor state will be updated.

Using the least-significant two bits of 0xdead to index into the predictor, we want the second row (0xd = 1101). Given the state of the branch history register, we want the second column; consequently, the prediction is *taken.*

If the branch is taken, that entry updated to 0b11, and the BHR updated to 0b11. If the branch is not taken, that entry is updated to 0b00, and the BHR is updated to 0b10.



0b00	0b01	0b10	0b11
0b00	0b00	0b11	0b10
0b01	0b10	0b00	0b01
0b11	0b01	0b10	0b00
0b10	0b00	0b01	0b10

Name: ID:

d. (8 pts) Schedule the following code for execution on hardware capable of (i) *fine-grained multi-threading* (FGMT) and (ii) *simultaneous multi-threading* (SMT). Assume that two identical threads, A and B, are available; schedule a single iteration of the loop for each. Assume both the FGMT and SMT processors issue up two instructions of any type per cycle. Assume that integer instructions require one (1) cycle, memory accesses require two (2) cycles (addresses are calculated in the first cycle, and memory is accessed in the second), and floating point addition (fully pipelined) requires four (4) cycles. Complete the table, indicating the instruction (Inst) issued and the thread (Thrd, either A or B) from which it is issued, in each cycle.

```
L0: L.D     F0,0(R1)
    L.D     F6,-8(R1)
    ADD.D   F4,F0,F2
    ADD.D   F8,F6,F2
    S.D     F4,0(R1)
    S.D     F8,-8(R1)
    DADDUI  R1,R1,#-16
    BNE     R1,R2,L0
```

FGMT

Cycle	Instruction Slot 1		Instruction Slot 2	
	Thrd	Inst	Thrd	Inst
1	A	L.D F0	A	L.D F6
2	B	L.D F0	B	L.D F6
3	A	ADD.D F4	A	ADD.D F8
4	B	ADD.D F4	B	ADD.D F8
5				
6	A	S.D F4	A	S.D F8
7	B	S.D F4	B	S.D F8
8	A	DADDUI		
9	B	DADDUI		
10	A	BNE		
11	B	BNE		
12				

SMT

Cycle	Instruction Slot 1		Instruction Slot 2	
	Thrd	Inst	Thrd	Inst
1	A	L.D F0	B	L.D F0
2	A	L.D F6	B	L.D F6
3	A	ADD.D F4	B	ADD.D F4
4	A	ADD.D F8	B	ADD.D F8
5				
6	A	S.D F4	B	S.D F4
7	A	S.D F8	B	S.D F8
8	A	DADDUI	B	DADDUI
9	A	BNE	B	BNE
10				
11				
12				

Name: ID:

e. **(8 pts)** Schedule the following, unrolled, code for VLIW execution on a processor capable of issuing an integer (Int) instruction, floating-point instruction (FP), and memory access instruction (Mem) each cycle. Complete the table, indicating which instructions are issued each cycle, under the assumption that integer instructions require one (1) cycle, memory accesses require two (2) cycles (addresses are calculated in the first cycle, and memory is accessed in the second), and floating point addition (fully pipelined) requires four (4) cycles.

```
L0: L.D     F0,0(R1)
    L.D     F6,-8(R1)
    L.D     F10,-16(R1)
    L.D     F14,-24(R1)
    ADD.D   F4,F0,F2
    ADD.D   F8,F6,F2
    ADD.D   F12,F10,F2
    ADD.D   F16,F14,F2
    S.D     F4,0(R1)
    S.D     F8,-8(R1)
    S.D     F12,-16(R1)
    S.D     F16,-24(R1)
    DADDUI  R1,R1,#-32
    BNE     R1,R2,L0
```

Cycle	Int	FP	Mem
1			L.D F0
2			L.D F6
3		ADD.D F4	L.D F10
4		ADD.D F8	L.D F14
5		ADD.D F12	
6		ADD.D F16	S.D F4
7			S.D F8
8			S.D F12
9	DADDUI		S.D F16
10	BNE		
11			
12			

f. **(2 pts)** What is the performance advantage of multi-lane SIMD hardware over single-lane SIMD hardware?

Doubling the number of lanes approximately halves the cycles required to complete a vector operation, as the new lanes divide work with the existing lanes. Performance is less than doubled because start-up time is unaffected.

g. **(2 pts)** What is the challenge associated with exploiting the performance advantage of multi-lane SIMD hardware compared with single-lane SIMD hardware?

Adding lanes increases the potential throughput of the system, which in turn means higher memory bandwidth requirements. There is no advantage to doubling the number of lanes if memory bandwidth can't be likewise increased.

h. **(2 pts)** Why do vector architectures generally not utilize caches?

Caches are designed to exploit locality to reduce memory access delay. (1) Vector programs often don't have locality (e.g., gather-scatter), and (2) vector processors require massive bank-parallel memory architectures already designed to hide memory access latency through servicing multiple parallel requests simultaneously.

i. **(2 pts)** How do GPU architectures hide memory access latency?

Hardware thread contexts. When a GPU memory access causes a stall, one warp is exchanged for another. The Fermi architecture employs 48 hardware thread contexts, allowing any one warp to stall for a significant period of time without forcing the GPU to idle waiting for memory accesses to be serviced.

Name: ID:

Question 2: Cache with Cachet (30 pts)

Consider a hierarchical cache with the following contents. All state, tags and data are in hexadecimal format. Assume that L1 and L2 are *exclusive*. Assume *little-endianness*, i.e., that the least significant byte in a word is stored in the least significant address.

L1 Cache

Set	Valid	Dirty	Tag	Data
0	0	0	34	83 ab 57 88 31 e7 a0 5e 8f 4c e4 df 63 77 c6 5e
1	0	0	1f	d2 ce aa 7c 82 23 d7 d7 1f 64 76 c7 a3 dc 19 78
2	1	0	d3	e6 65 8d 3b e3 53 f8 d7 f8 40 3d f2 80 42 0a f0
3	1	1	2d	0c b8 00 e4 d7 60 58 b0 71 4e 10 f7 b9 5f 58 c7
4	0	0	67	5f 91 85 64 de 45 ca e8 84 b7 c0 41 6e 5c d4 64
5	1	1	d3	be 8a 35 b0 9b 75 70 48 08 06 a0 bd 13 c6 69 a0
6	0	0	8d	2a a4 7b bd 72 a1 f7 51 19 3b 55 fc d8 52 30 35
7	1	0	00	30 4b 12 c0 2f 51 8d 3a 8e 57 c7 8b be 44 93 a2
8	1	0	ab	df cd 5a f7 7f 4b cd b3 58 a5 4d 46 f5 30 91 e6
9	0	0	12	ee f9 86 7d 24 dd 98 7c 8c a7 de ec bc a5 06 cb
10	1	1	e3	2c 5b af c6 92 09 cf ca a9 e5 f2 87 d2 91 9b 1d
11	1	0	df	e8 af 5b 2e 4a 77 e0 0c 1d d0 ce 0b b8 ac a6 9c
12	0	0	ee	9d b7 11 e9 68 7c a3 4f 91 17 2f b3 3d 85 92 33
13	1	0	1d	cd 8b 2f 5d c8 de 16 90 d2 4e e3 20 12 83 f2 62
14	0	0	a0	d5 d0 a3 4b 9e 1a d4 55 a3 9b 1d 98 26 38 a5 e0
15	1	1	55	a5 65 2b 9f 55 20 72 e6 25 23 07 e1 bf 7f 72 1e

L2 Cache

Set	Valid	Dirty	Tag	Data
0	0	0	22	d0 3c 82 d0 e4 04 93 50 9c 4f 9a 4a f1 6d 21 22
1	1	1	d2	54 9d 9b 59 ba aa a5 d0 66 ea 4d 7c b6 ef 9d 9d
2	1	0	17	05 ff 99 48 d9 a8 86 0b 07 60 74 60 97 42 c0 51
3	1	0	8d	ea b4 e8 01 cf 8c 54 96 3a 8f 5e b1 d1 58 53 07
4	0	0	88	20 03 80 6e 6b c1 71 3b 12 2f 3c 59 f0 6c f9 1c
5	1	0	2d	99 ec 4f d6 40 df 28 4d 0f 8f 64 66 ec 10 ff cc
6	1	1	22	ec de a2 6c 4b 93 ad f4 5b ed 48 2a cf b4 ca 45
7	1	0	8d	97 73 42 76 bb e0 4c 40 bd eb 5c b1 74 72 6a c4
8	0	0	a9	10 08 90 d3 59 48 d4 5d af 5c 97 b7 66 f2 fa 3c
9	0	0	0e	85 8f 34 e7 8a f9 2e d9 c5 1f 21 85 88 a0 03 15
10	1	1	31	5c 6a ec f3 e0 72 84 97 55 65 cc a1 7b 9d 61 24
11	1	0	b8	77 93 42 c7 a1 98 44 2b 15 90 1f d7 cf a5 94 40
12	1	1	ab	e5 05 bf ac 67 2d 7b f1 76 e9 cc 16 7d e8 1a c3
13	1	0	4d	c9 27 66 71 42 df 83 6d ca a9 59 e3 a9 fe 15 18
14	0	0	ab	5c ea ab 14 8a fa 11 b0 00 40 46 6c 72 89 0f f8
15	1	1	1c	a0 0b 47 8d b1 3e 6b 42 7d be 76 b8 1e 0c 45 88

a. **(4 pts)** What data is delivered when a read request is made for a 16-bit word at `0xabcd`? Indicate the changes made to any cache blocks in L1 or L2 as a consequence of servicing the request.

First, we look in L1.

16 bytes per block => the least-significant four bits are used to determine block offset.
16 sets => the next four least-significant bits are used to index the L1 cache.
Reading from 0xabcd => 0xc => set 12.

This block is invalid => L2 must be accessed.

L2 has the same properties as L1. Reading from 0xabcd => 0xc => set 12.

This block is valid, and the tag—0xab—matches. Hit! The requested data is 0x05bf.

L1 set 12 is updated to: valid, dirty, 0xab, 0xe5 05 bf ac 67 2d 7b f1 76 e9 cc 16 7d e8 1a c3.

L2 is unchanged.

b. **(6 pts)** Indicate the changes made to the cache (both L1 and L2) when `0x1234` is written to address `0x8d34`.

Writing to 0x8d34 => set 3.

This block is valid, but the tag—0x2d—does not match. The block is also dirty, and so much be written to L2. Looking in L2, in set 3, we find the block we're looking for (with tag 0x8d). As the caches are exclusive, the blocks in L1 and L2 are swapped.

L2 set 3 is updated to: valid, dirty, 0x2d, 0x0c b8 00 e4 d7 60 58 b0 71 4e 10 f7 b9 5f 58 c7.

L1 set 3 is updated to: valid, dirty, 0x8d, 0xea b4 e8 01 cf 8c 54 96 3a 8f **12 34** d1 58 53 07.

Name: ID:

Now consider a pipelined processor that runs at 2 GHz and has a base CPI of 0.9 when all cache accesses hit. The only instructions that read data from or write data to memory are loads (15% of all instructions) and stores (10% of all instructions).

The memory system for this computer is composed of a split L1 cache. Both the I-cache and D-cache are 2-way set-associative and hold 32 KB each. The I-cache has a 1% miss rate; the D-cache has a miss rate of 15%.

The 1 MB inclusive, unified L2 cache has an access time of 20 ns. Of all memory references sent to the L2 cache in this system, 80% are satisfied without going to main memory. Main memory has an access latency of 200 ns. Assume that L2 and main memory transfer times are accounted for in the access times above.

Assume that all caches use a write-back policy, and 40% of evicted data is dirty.

c. (4 pts) What is the average memory access time (in cycles) for instruction accesses?

CT = 1/2e9 = 0.5 ns

$AMAT_{L1I} = HT_{L1I} + MR_{L1I} * (HT_{L2} + (1 + WBR) * MR_{L2} * MP)$
$= 1 + 0.01*(20/0.5 + (1 + 0.4) * 0.20*200/0.5)$
= 2.5 cycles

d. (4 pts) What is the average memory access time (in cycles) for data accesses?

$AMAT_{L1D} = HT_{L1D} + (1 + WBR) * MR_{L1D} * (HT_{L2} + (1 + WBR) * MR_{L2} * MP)$
$= 1 + (1 + 0.4)*0.15*(20/0.5 + (1 + 0.4)*0.20*200/0.5)$
= 33 cycles

e. **(4 pts)** What is the overall CPI of the system?

$CPI = CPI_{base}$ + instruction access stalls per instruction + data access stalls per instruction
$= 0.9 + (1) * (2.5 – 1) + 0.25 * (33 – 1) = 10.4$ cycles

f. **(4 pts)** What speedup is achieved if an L3 cache is added with *local hit rate* of 90% and access time of 40 ns?

$CPI_{base} = 10.4$

$AMAT_{L1I} = HT_{L1I} + MR_{L1I} * (HT_{L2} + MR_{L2} * (HT_{L3} + MR_{L3} * MP))$
$= 1 + 0.01*(20/0.5 + (1 + 0.4) * 0.20*(40/0.5 + (1 + 0.4) * 0.10*200)/0.5) = 2$ cycles

$AMAT_{L1D} = HT_{L1D} + (1 + WBR) * MR_{L1D} * (HT_{L2} + (1 + WBR) * MR_{L2} * MP)$
$= 1 + (1 + 0.4)*0.15*(20/0.5 + (1 + 0.4)*0.20*(40 + (1 + 0.4) * 0.1*200)/0.5) = 17.4$ cycles

$CPI_{L3} = 0.9 + (1) * (2 – 1) + 0.25 * (17.4 – 1) = 6$ cycles
Speedup = 10.4 / 6 = 1.73 x

g. **(4 pts)** Rather than improve the memory hierarchy as in part (f), would it be more advantageous to adjust compilation to reduce the number of memory accesses? What speedup is achieved if the fraction of loads and stores is reduced to 10 and 5% respectively? Assume the optimization removes the memory accesses and does not replace them with other instructions.

Speedup comes from two sources this time: change in memory access frequency, and reduction in total instruction count.

$CPI_{compiler} = 0.9 + (1) * (2.5 – 1) + 0.15 * (33 – 1) = 7.2$ cycles

Speedup $= CPI_{base} * IC_{base} / CPI_{compiler} * IC_{compiler} = 10.1 * 1 / (7.2 * 0.9) = 1.55$ x

It is slightly less advantageous to improve compilation in this case than add an L3.

Name: ID:

Question 3: You're So Predictable (20 pts)

Two computers A and B are identical except for their branch prediction schemes. Each computer uses a seven-stage pipeline:

IF ID1 ID2 EX1 EX2 MEM WB

Computer A uses a static predicted not-taken scheme. Computer B uses a static predicted taken scheme. In each case, branch targets are calculated in ID2, and branch conditions are resolved in EX2.

If 20% of instructions are conditional branches, what fraction of these branches must be taken for the two computers to have equivalent performance? Assume each computer achieves the ideal CPI for every other instruction other than conditional branches.

CPI = 1 + branch frequency * branch penalty
= 1 + branch frequency * (taken freq * taken penalty + untaken freq * untaken penalty)

Assume x is the fraction of branches taken.

Computer A – predicted not taken

Taken penalty: 4 – IF, ID1, ID2, EX1 (for the mispredicted branch successor)
Untaken penalty: 0

CPI = 1 + 0.2 * (x * 4) = 1 + 0.8 * x

Computer B – predicted taken

Taken penalty: 2 – IF, ID1
Untaken penalty: 4 – IF, ID1, ID2, EX1

CPI = 1 + 0.2 * (x * 2 + (1 – x) * 4) = 1 + (0.4 * x + 0.8 * (1 – x)) = 1.8 - 0.4 * x

Solving for x

1 + 0.8 * x = 1.8 – 0.4 * x
1.2 * x = 0.8
x = 2/3

If 2/3 of branches are taken, the performance of the two computers will be equivalent.

Name: ID:

Question 4: UnRFLMAO! (20 pts)

Consider the following C code.

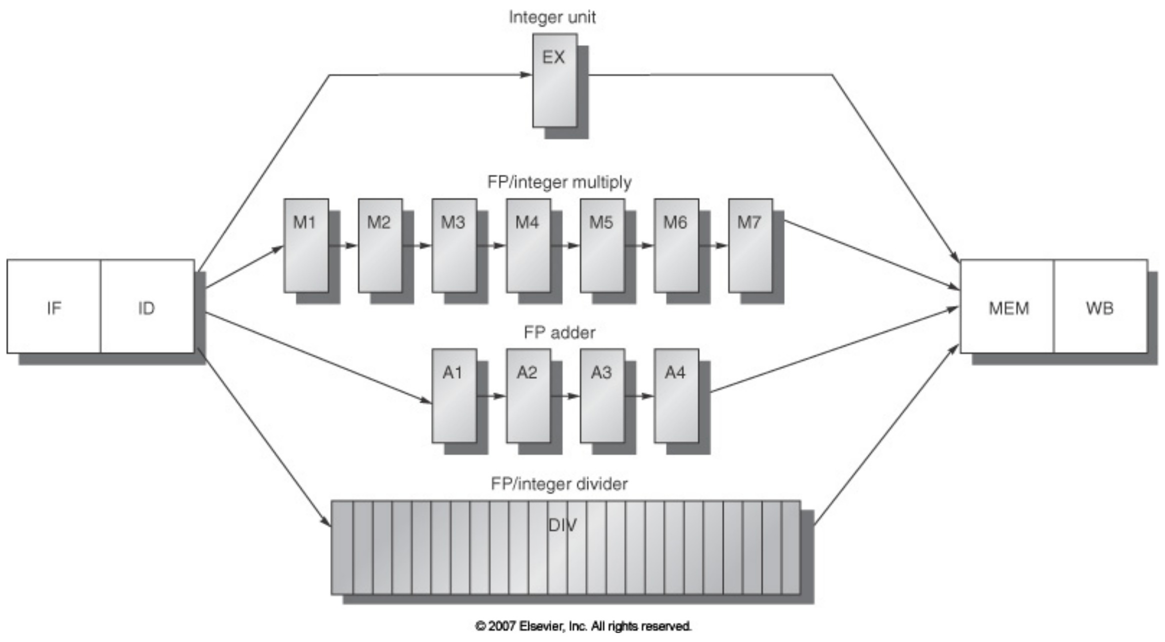
```
for (int i=n; i>0; i--) {
     sum[i-1] = sum[i] + a;
}
```

This code can be implemented with the following assembly.

```
       LI      R2, n
       LI.D    F2, a
Loop:  L.D     F4, 0(R1)
       ADD.D   F6, F2, F4
       S.D     F6, -8(R1)
       DADDUI  R1, R1, #-8
       DADDUI  R2, R2, #-1
       BNEZ    R2, Loop
```

`LI` loads an immediate value in an integer register; `LI.D` loads a double-precision floating-point value in a floating-point register by using the FP adder.

Unroll the loop once and reschedule the instructions to minimize delay within the loop body. Assume the standard MIPS floating point pipeline (forwarding, split L1 caches, etc):



Further assume that control hazards are managed using a *branch delay slot*. On the following page, write out the re-scheduled instructions, including their operands. Insert placeholders for any stalls not eliminated by rescheduling.

Name: ID:

Additional Page

First, we unroll, rename, and insert stalls.

```
        LI          R2, n
        LI.D        F2, A
Loop:   L.D         F4, 0(R1)       # no second load; value is dependent upon first store
        <stall, LD.D to ADD.D>      # additional stall on 1st iteration only (LI.D to ADD.D)
        ADD.D       F6, F2, F4
        <stall, ADD.D to ADD.D>
        <stall, ADD.D to ADD.D>
        <stall, ADD.D to ADD.D>
        ADD.D       F10, F2, F6     # second value to store generated here
        S.D         F6, -8(R1)
        <stall from ADD.D>
        S.D         F10, -16(R1)
        DADDUI      R1, R1, #-16
        DADDUI      R2, R2, #-2
        BNEZ        R2, Loop
        <stall, branch flush>       # empty branch delay slot
```

Next, we reschedule.

```
        LI.D        F2, A           # swapped to remove 1st iteration stall
        L.D         F6, 0(R1)       # load moved out of the loop
        LI          R2, n
Loop:   ADD.D       F6, F2, F6
        <stall, ADD.D to S.D, ADD.D>
        DADDUI      R2, R2, #-2
        S.D         F6, -8(R1)
        ADD.D       F6, F2, F6      # second value to store generated here
        DADDUI      R1, R1, #-16
        BNEZ        R2, Loop
        S.D         F6, 0(R1)       # full branch delay slot
```

8 cycles per iteration!

Name: ID:

Question 5: This One is Double-Wide! (20 pts)

Consider once more the code from Q4 (note that n has been set to 2, and a to 8).

```
      LI     R2, #2
      LI.D   F2, #8
Loop: L.D    F4, 0(R1)
      ADD.D  F6, F2, F4
      S.D    F6, 8(R1)
      DADDUI R1, R1, #-8
      DADDUI R2, R2, #-1
      BNEZ   R2, Loop
```

This time, execute the code to completion assuming a *super-scalar* out-of-order processor with support for hardware speculation, and the following functional units:

Functional Unit	Cycles to Execute	No. of Functional Units
Integer ALU	1	4
FP Adder (used by LI.D, too)	4	2
FP Multiplier	7	1
Load/Store	*2	4

* The first cycle of Load/Store execution is used to calculate the effective address, and can proceed even if other operands are not yet ready.

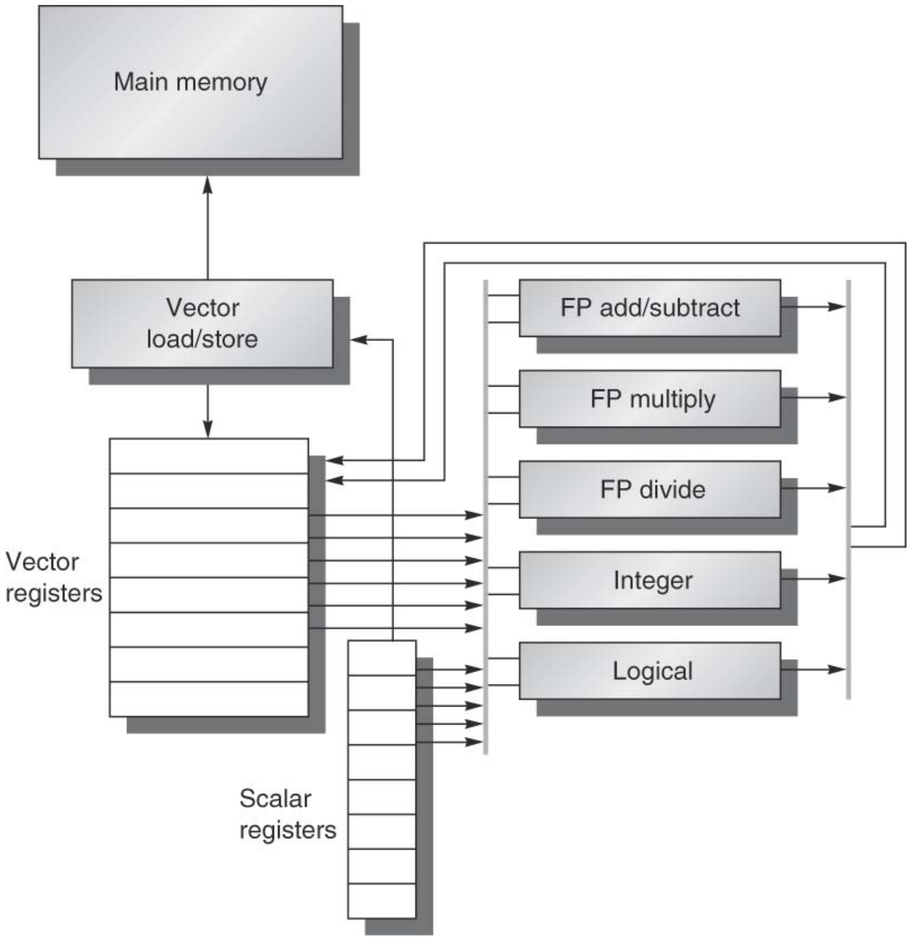
Assume: the processor can issue and commit *two instructions per cycle*, a single reservation station per functional unit, and *perfect branch prediction*. Complete the following table.

Inst.	Inst. Issue	Begin EX	Finish EX	Write CDB	Commit
`LI R2, #2`	1	2	2	3	4
`LI.D F2, #8`	1	2	5	6	7
`L.D F4`	2	3	4	5	7
`ADD.D F6`	2	**7**	10	11	12
`S.D F6`	3	**11**	12	13	14
`DADDUI R1`	3	4	4	5	14
`DADDUI R2`	4	5	5	6	15
`BNEZ R2`	4	**7**	7	8	15
`L.D F4`	5	6	7	8	16
`ADD.D F6`	5	**9**	12	13	16
`S.D F6`	6	**13**	14	15	17
`DADDUI R1`	6	7	7	**9**	17
`DADDUI R2`	7	8	8	9	18
`BNEZ R2`	7	**10**	10	11	18

Name: ID:

Questions 6: *Vector* Sounds Like an 70s Superhero (30 pts total)

Assume the following vector architecture with 64-element vector registers, and one functional unit of each type.



Further assume the following *start-up times* for each functional unit above:

Functional Unit	Start-up Time
FP add/subtract	6
FP multiply	7
FP divide	20
Vector load	12

Each functional unit may begin a new operation the cycle after the last element of the previous operation enters the pipeline. *I.e.*, when two vector store instructions execute one after the other, the start-up time penalty is paid only once.

Name: ID:

Now consider the following VMIPS assembly sequence.

```
LV       V1,Ra     # V1[i] <- Mem(Ra+i)
MULV.D   V2,V1,V3  # V2[i] <- V1[i] * V3[i]
ADDV.D   V4,V1,V3  # V4[i] <- V1[i] + V3[i]
SV       Rb,V2     # Mem(Rb+i) <- V2[i]
SV       Rc,V4     # Mem(Rc+i) <- V4[i]
```

a. **(5 pts)** Assuming *no chaining* is available, identify the *convoys* in the above assembly and use *chimes* to estimate the number of cycles required to execute the above VMIPS assembly.

In this case, there are 4 convoys:

1. LV
2. MULV.D / ADDV.D
3. SV
4. SV

A chime requires 64 cycles, approximately. 64 * 4 = 256 cycles in total.

b. **(5 pts)** Now assuming *chaining* is available. Identify the *convoys* in the above assembly and use *chimes* to estimate the speedup that is possible.

In this case there are 3 convoys:

1. LV / MULV.D / ADDV.D – *chaining*
2. SV – chaining not possible, just one L/S unit
3. SV – chaining not possible, just one L/S unit

This requires 64 * 3 = 192 cycles, or achieves 256/192 = 1.33 x speedup.

c. **(10 pts)** Assuming *chaining* is available, now use the start-up latencies for each functional unit to determine the number of cycles required to execute the above assembly. How much error is introduced when using *chimes* to approximate latency?

The critical path is through the sequence of LV and SV instructions. LV starts at time 0, producing its first element in cycle 13, and its last in cycle 12 + 64 = 76. The first elements for the SV instructions are produced in cycles 12 + 7 = 19 and 12 + 6 = 18 respectively; the SV need only wait for the L/S unit.

As the start-up time for the first SV can be overlapped with the production of the final LV elements (and likewise for the second SV w.r.t. the first SV), the complete sequences finishes after 12 + 64 + 64 + 64 = 204 cycles.

The error introduced by counting chimes, in this case, is (204 – 192) / 204 = 6%.

Name: ID:

Now consider the following C code.

```
for (int i=0; i<n; i++) {
     sum = sum + Y[i]*b;
     W[i+1] = W[i] * Y[i+1];
}
```

d. **(4 pts)** Identify any dependencies in the above code, and indicate their type.

There is recursion dependence with *sum*.

There is loop-carried dependence between W[i] and W[i+1].

e. **(6 pts)** Re-write the above loop to increase loop-level parallelism.

Loop-level parallelism can be improved by dividing the above loop into two.

```
for (int i=0; i<n; i++) {
        temp[i] = Y[i]*b;
}

for (int i=0; i<n; i++) {
        sum = sum + temp[i];
        W[i+1] = W[i] * Y[i+1];
}
```

Now the first loop is fully parallel; the second loop could be further parallelized by dividing the final reduction (of sum) into several more intermediate steps.