McGill University
ECSE 425: COMPUTER ORGANIZATION AND ARCHITECTURE
Winter 2016
Final Examination

2:00 PM – 5:00 PM, April 25th, 2016

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 20 pages long. Please check that you have all 20 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.**
- **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: **_____SOLUTION_____**

Student Number: **_____**

Q1: _____/40 **Q2: _____/20** **Q3: _____/20**

Q4: _____/20 **Q5: _____/30** **Q6: _____/30**

Total:

Question 1: Get Shorty (40 pts total, 4 pts each)

Please provide precise, justified responses; no credit will be given for vague generalities.

a. *Virtual memory* requires *precise exceptions*.

i. What are *precise exceptions*?

When instruction *i* causes an exception, the exception is *precise* if the processor state can be saved such that it reflects the state of the processor after the sequential execution of instruction *i*-1. Out of order completion therefore makes guaranteeing precise exceptions difficult.

ii. Why are *precise exceptions* needed to support *virtual memory*?

When a page fault occurs, the load or store instruction that caused it must be re-executed after the virtual memory system makes the appropriate page available in main memory. Precise exceptions are needed to ensure that later instructions have not already executed, modifying internal processor state, as these instructions will be executed again.

b. Describe two different ways of measuring *cache performance*. Why does the choice of performance metric matter?

1. Miss rate.
2. Average memory access time.
3. CPI.
4. Execution time.

The choice of metric matters because simpler metrics (e.g., miss rate) don't fully capture the effect of a cache configuration on performance. The better the information used, the more robust the resulting evaluation. Execution time comparison is best!

c. What are the advantages and disadvantages of a virtually indexed, physically tagged L1 cache?

The advantage is that it can be indexed while the TLB is used to perform address translation. One disadvantage is that the size of the cache is limited to multiples of the virtual memory system page size (one page per way in the cache); at this limit, the cache size can only be increased by increasing associativity. Another is related to memory protection: the cache must be flushed on context switch, as the same virtual addresses in different programs may translate to different physical addresses.

d. Describe the advantages and disadvantages of *pipelining*.

The advantage of pipelining is that it increasing instruction throughput by partially overlapping the execution of instructions. The disadvantage is the increase in area, power, and design complexity related to the addition of hardware required to (a) separate different pipeline stages, (b) detect hazards between instructions in different pipeline stages, and (c) forward values between pipeline stages to mitigate hazards. Furthermore, ideal speedup (n for n pipeline stages) is difficult to achieve, due to clock skew and imbalanced effort per stage.

e. What is a *branch target buffer*? Why use one?

A branch target buffer, in addition to saving a branch prediction of a given instruction address, also saved the most recent branch target. This allows the processor to predict a branch taken, and fetch the appropriate next instruction, without an adder to calculate the target address; in fact, the instruction need not even be decoded, allowing for branch prediction in IF.

f. Describe two sources of overhead unique to VLIW processors.

1. *Increased code size*. When fewer independent instructions can be found than can executed simultaneously, the empty slots are filled with NOP, increasing code size.
2. *Wasted energy (power) and area (cost)*. Though VLIW generally saves power compared with RISC, when not all slots are filled with useful work, the unused functional unit wastes energy (power) and area (cost).
3. *Compile time*. The effort required to find independent instructions at compile-time, rather than runtime, translates to longer compilation.
4. *Hardware complexity*. The register file must service many more requests per cycle than a RISC processor, increasing its size dramatically.
5. *Control flow*. VLIW processors have many more branch delay slots to fill than a RISC processor, potentially reducing performance significantly for control-flow heavy code.
6. *Memory bandwidth*. More memory bandwidth is required as many more instructions are fetched in a single cycle.

g. Explain how each of the following can uniquely limit ILP (when all other resources are unlimited):

i. The size of the *re-order buffer*.

The size of the re-order buffer limits ILP by restricting the window of instructions from which independent instructions may be issued.

ii. The number of registers available for *register renaming*.

Renaming registers limit ILP by restricting the number of false dependencies that can be eliminated, e.g., between loop iterations.

h. Does single-threaded performance matter in the multi-core era? Justify your claim with support from the principles of design in computer architecture.

Yes, single-threaded performance matters in the multi-core era. Software, generally speaking, is not perfectly and arbitrarily parallelizable. The execution time of the sequential portion of the application ultimately limits the achievable speedup (Amdahl's Law).

i. Briefly describe how standard vector architecture hides memory access latency. How does this differ from the approach taken in GPU architecture?

Vector architectures hide memory access latency by distributing accesses to consecutive addresses across multiple memory banks so that the requests may be serviced in parallel. GPU, however, do not have many memory banks. Instead, they hide memory access latency by switching between multiple threads.

j. Identify all dependencies in the following code. Which, if any, prevent the loop from being vectorized?

```
for (i=0; i<100; i=i+1) {
     A[i+1] = A[i] + C[i];
     B[i+1] = B[i] + A[i+1];
}
```

1. A[i+1] depends on A[i], which is calculated in the previous iteration.
2. B[i+1] depends on B[i], which is also calculated in the previous iteration.
3. B[i+1] also depends on A[i+1], which is calculated in the current iteration.

Dependencies (1) and (2), loop carried dependencies, prevent the loop from vectorized.

Question 2: Where Does This Pipeline End? (20 pts)

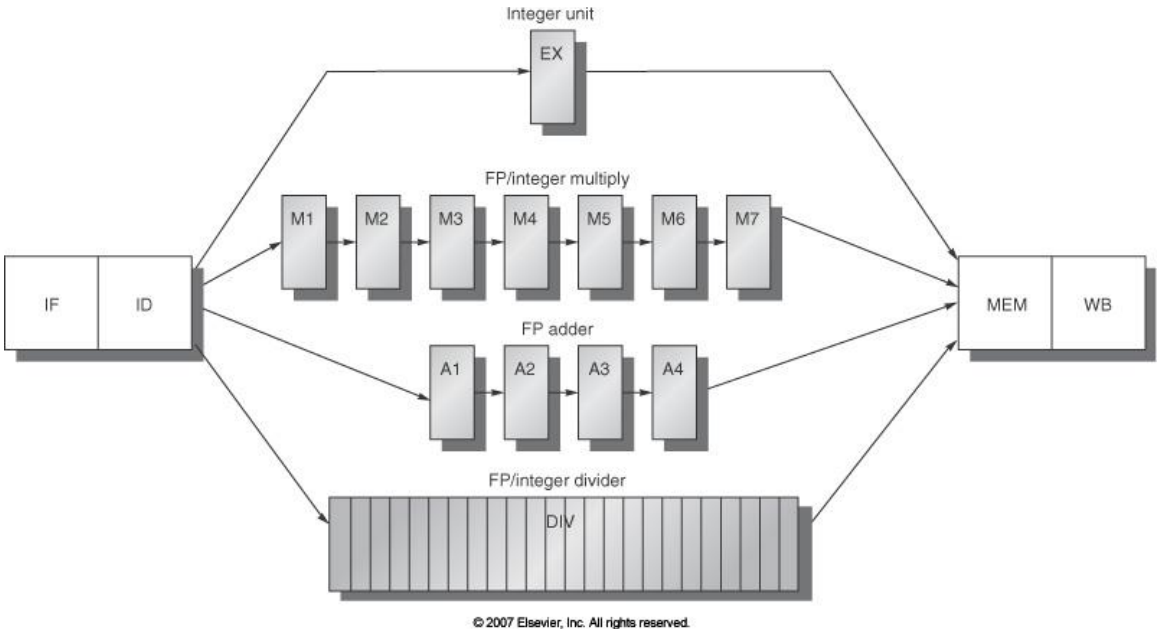
Consider the following code:

```
        LW      R6, 12(R1)
        ADDI    R7, R0, 2
EQ2:    BNEQ    R6, R7, EQ1
        ADD     R8, R2, R3
        MULT    R6, R5
        MFLO    R10
        J       END
EQ1:    ADDI    R7, R0, 1
        BNEQ    R6, R7, DEF
        SUB     R8, R2, R3
        LW      R9, 20(R1)
        MULT    R8, R9
        MFLO    R10
        J       END
DEF:    ADDI    R6, R0, 0
        ADDI    R8, R0, 0
        ADDI    R10, R0, 0
END:    SW      R6, 24(R1)
        SW      R8, 28(R1)
        SW      R10, 32(R1)
```

and corresponding initial memory state:

Address	Value
0x1000	5
0x1004	1
0x1008	3
0x100C	1
0x1010	2
0x1014	4
0x1018	8
0x101C	12
0x1020	3

This code will run on the standard MIPS floating point pipeline, illustrated below.



Assume:

- Full hazard detection and forwarding logic;
- The register file supports one write and two reads per clock cycle;
- Branch targets and conditions and jump targets are resolved in ID;
- Branches are handled by predicting not taken;
- J is treated as a branch instruction for branch prediction purposes;
- MFLO is treated as an ALU operation for hazard detection and forwarding purposes;
- Split L1 instruction and data caches service all requests in one clock cycle;
- Two or more instructions may simultaneously pass through MEM (WB) as long as only one makes use of the memory (register file);
- Structural hazards are resolved by giving priority to older instructions.

Complete the chart on the next page to show the execution of the loop if R1 = 0x1000 initially. Indicate pipeline stages with {**F**, **D**, **X**, **Mi**, **Ai**, **M**, **W**}, and stalls with **S**.

ID:

Name:

Correct execution path: 4 pts
Basics: 4 pts
Hazards: 6 pts
Branch behavior: 6 pts

DH: ADDI -> BNEQ
DH: ADDI -> BNEQ
SH: BNEQ -> SUB
DH: LW -> MULT

SH: MULT -> MFLO
DH: MULT -> MFLO
SH: MFLO -> J

Flush ADD
Don't flush SUB
Flush ADDI

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
LW R6, 12(R1)	F	D	X	M	W																																				
ADDI R7, R0, 2		F	D	X	M	W																																			
BNEQ R6, R7, EQ1			F	D	S	X	M	W																																	
ADD R8, R2, R3				F	S																																			
ADDI R7, R0, 1						F	D	X	M	W																															
BNEQ R6, R7, EQ2							F	D	S	X	M	W																													
SUB R8, R2, R3								F	S	D	X	M	W																												
LW R9, 20(R1)										F	D	X	M	W																											
MULT R8, R9											F	D	S	M1	M2	M3	M4	M5	M6	M7	M	W																		
MFLO R10												F	S	D	S	S	S	S	S	S	X	M	W																	
J END														F	S	S	S	S	S	S	D	X	M	W																
ADDI R6, R0, 0																					F																			
SW R6, 24(R1)																						F	D	X	M	W														
SW R8, 28(R1)																							F	D	X	M	W													
SW R10, 32(R1)																								F	D	X	M	W												

Question 3: You Should Have Guessed You'd See This One Again (20 pts)

a. (10 pts) Two computers A and B are identical except for their branch prediction schemes. Each computer uses an 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 ID1, and branch conditions are resolved in EX1.

If 25% 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.

A & B
1 pt: taken penalty
1 pt: untaken penalty
2 pts: equation

Solution
2 pts

Computer A – predicted not taken

Taken penalty: 3 – IF, ID1, ID2 (for the mispredicted branch successor)
Untaken penalty: 0 (target is known in IF)

CPI = $1 + 0.25 * (x * 3 + (1 - x) * 0) = 1 + 0.75 * x$

Computer B – predicted taken

Taken penalty: 1 – IF (target is known in ID1)
Untaken penalty: 3 – IF, ID1, ID2 (for the mispredicted branch successor)

CPI = $1 + 0.25 * (x * 1 + (1 - x) * 3) = 1.75 - 0.5 * x$

Solving for x

$1 + 0.75 * x = 1.75 - 0.5 * x$
$1.25 * x = 0.75 \Rightarrow 5/4 * x = 3/4 \Rightarrow x = 3/5 = 60\%$

b. (10 pts) Now consider the following C code:

```
for (int i=0; i<n; i++) {
     if (i % 2 == 0)
          … ;
     if (i % 3 == 0)
          … ;
}
```

This loop can be implemented with three branches and a jump:

- *A*: a branch conditioned on the loop bound: branch to the end when the loop is over;
- *B* and *C*: two branches conditioned on the the value of `i`;
- a jump from the end of the loop to the beginning of it (where the loops bound is checked once again …)

Assume a (2, 2) correlating branch predictor sufficiently large enough that the above branches do not alias (conflict), and that all predictor state is initialized to taken (*T*). Complete the following table for the first four iterations of the loop. Indicate the order in which branches are predicted (A, B, and C), the predictor state at the time of the prediction, the branch history register (BHR) contents (*oldest*, …, *newest*), resulting prediction, and the outcome.

Branch	Predictor State ***TT TN NT NN***	BHR ***NO***	Prediction	Outcome
i =0; A	**TT** TT TT TT	TT	T	**N**
B	TT TT **TT** TT	NT	T	**N**
C	TT TT TT **TT**	NN	T	**N**
i =1; A	*NT* TT TT **TT**	NN	T	**N**
B	TT TT *NT* **TT**	NN	T	T
C	TT **TT** TT *NT*	TN	T	T
i =2; A	**NT** TT TT *NT*	TT	T	**N**
B	TT TT **NT** TT	NT	T	**N**
C	TT TT TT **NT**	NN	T	T
i =3; A	*NN* **TT** TT NT	TN	T	**N**
B	TT TT ***NN*** TT	NT	N	**T**
C	TT **TT** TT *TT*	TN	T	**N**

Predictor structure 2 pt
Outcome: 2 pt
BHR: 2 pt

Predictor selection: 2 pt
Predictor update: 2 pt

Incorrect structure reduces points available for selection or update, depending on what is simplified.

Likewise with incorrect outcome.

Question 4: This One is Double-Wide! (20 pts)

Consider the following C code.

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

This code can be implemented, in part, with the following MIPS assembly (refer to the attached MIPS Green Card as necessary). `$t0` is previously initialized to the base address of sum; `$t2` is previously initialized to the address of the last element of sum. `a` is already in `$f0`.

```
        addi   $t1, $t0, 4       # get address for i+1
top:    sltu   $t3, $t1, $t2     # compare i+1 and n (in $t2)
        beq    $t3, $zero, done  # branch if we are done
        lwc1   $f1, 0($t0)       # single precision load
        lwc1   $f2, 0($t1)       # single precision load
        mul.s  $f2, $f0, $f2     # single precision mult
        add.s  $f3, $f1, $f2     # single precision add
        addi   $t0, $t0, 4       # inc i
        addi   $t1, $t1, 4       # inc i+1
        j      top
done:
```

Execute the code through the completion of the first `j` instruction, 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	2
FP Adder (used by LI.D, too)	4	2
FP Multiplier	7	1
Load/Store	*2	2

* 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*; all functional units are fully pipelined; *perfect branch prediction*; and, *all instructions* write the common data bus (CDB). Complete the following table.

Inst.	Inst. Issue	Begin EX	Finish EX	Write CDB	Commit
addi $t1, $t0, 4	1	2	2	3	4
sltu $t3, $t1, $t2	1	($t1) **4**	4	5	6
beq $t3, $zero, done	2	($t3) **6**	6	7	8
lwc1 $f1, 0($t0)	2	3	4	5	8
lwc1 $f2, 0($t1)	3	4	5	6	9
mul.s $f2, $f0, $f2	3	($f2) **7**	13	14	15
add.s $f3, $f1, $f2	4	($f2) **15**	18	19	20
addi $t0, $t0, 4	4	5	5	6	20
addi $t1, $t1, 4	5	6	6	7	21
j top	5	(alu) **7**	7	8	21
sltu $t3, $t1, $t2	6	($t1) **8**	8	9	22
beq $t3, $zero, done	6	($t3) **10**	10	11	22
lwc1 $f1, 0($t0)	7	8	9	10	23
lwc1 $f2, 0($t1)	7	8	9	10	23
mul.s $f2, $f0, $f2	8	($f2) **11**	17	18	24
add.s $f3, $f1, $f2	8	($f2) **19**	22	23	24
addi $t0, $t0, 4	9	10	10	11	25
addi $t1, $t1, 4	9	(alu) **11**	11	12	25
j top	10	11	11	12	26

Issue 2 pts
Begin ex 10 pts
- addi->sltu
- sltu->beq
- lwc1->mul.s
- mul.s->add.s
- j

Finish ex 2 pts
CDB 2 pts
Commit 4 pts

Question 5: High Thread Count (30 pts)

Consider again the MIPS assembly from Q4:

```
        addi   $t1, $t0, 4        # get address for i+1
top:    sltu   $t3, $t1, $t2      # compare i+1 and n (in $t2)
        beq    $t3, $zero, done   # branch if we are done
        lwc1   $f1, 0($t0)        # single precision load
        lwc1   $f2, 0($t1)        # single precision load
        mul.s  $f2, $f0, $f2      # single precision mult
        add.s  $f3, $f1, $f2      # single precision add
        addi   $t0, $t0, 4        # inc i
        addi   $t1, $t1, 4        # inc i+1
        j      top
done:
```

a. (10 pts) Schedule the code for a processor capable of *fine-grained multi-threading* (FGMT). Assume that two identical threads, A and B, are available; schedule a single iteration of the loop for each. Assume the processor issues up two instructions of any type per cycle. Assume the same execution delays and functional units as in Q4. Complete the table, indicating the instruction (Inst) issued and the thread (Thrd, either A or B) from which it is issued, in each cycle.

Cycle	Instruction Slot 1		Instruction Slot 2	
	Thrd	Inst	Thrd	Inst
1	A	addi		
2	B	addi		
3	A	sltu		
4	B	sltu		
5	A	beq		
6	B	beq		
7	A	lwc1	A	lwc1
8	B	lwc1	B	lwc1
9	A	mul.s		
10	B	mul.s		
11				
12				
13				
14				
15				
16	A	add.s	A	addi
17	B	add.s	B	addi
18	A	addi	A	j
19	B	addi	B	j

4 pts: basics
- One thread at a time
- Up to two instructions at a time
- Switch threads each cycle

4 pts: dependencies
- addi->sltu
- sltu->beq
- lwc1->mul.s
- mul.s->add.s

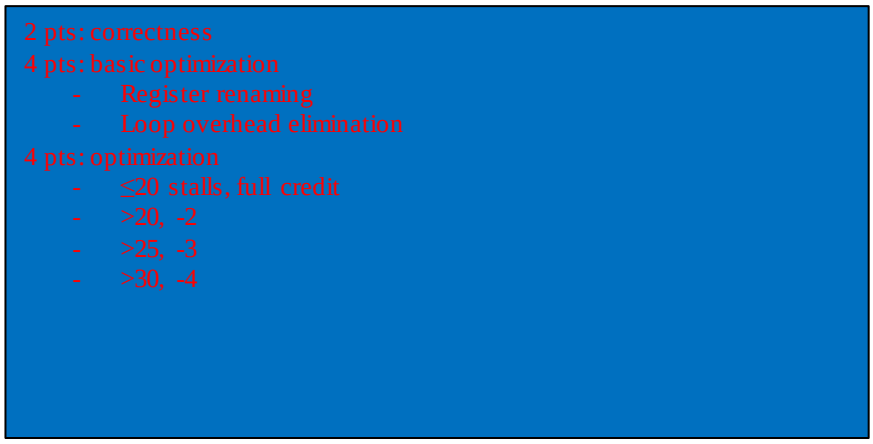
2 pts: delay
- mul.s

b. (10 pts) Now unroll the loop twice (so three copies of the loop body execute per iteration) and re-schedule the code to minimize stalls.

```
        addi   $t1, $t0, 4
top:    sltu   $t3, $t1, $t2
        beq    $t3, $zero, done
        lwc1   $f1, 0($t0)
        lwc1   $f2, 0($t1)
        1 s
        mul.s  $f2, $f0, $f2
        6 s
        add.s  $f3, $f1, $f2
        lwc1   $f4, 4($t0)
        lwc1   $f5, 4($t1)
        1 s
        mul.s  $f5, $f0, $f5
        6 s
        add.s  $f6, $f4, $f5
        lwc1   $f7, 8($t0)
        lwc1   $f8, 8($t1)
        1 s
        mul.s  $f8, $f0, $f8
        6 s
        add.s  $f9, $f7, $f8
        addi   $t0, $t0, 12
        addi   $t1, $t1, 12
        j      Top
done:
```

```
        addi   $t1, $t0, 4
top:    sltu   $t3, $t1, $t2
        beq    $t3, $zero, done
        lwc1   $f2, 0($t1)
        lwc1   $f1, 0($t0)
        mul.s  $f2, $f0, $f2
        lwc1   $f5, 4($t1)
        lwc1   $f4, 4($t0)
        mul.s  $f5, $f0, $f5
        lwc1   $f8, 8($t1)
        lwc1   $f7, 8($t0)
        mul.s  $f8, $f0, $f8
        add.s  $f3, $f1, $f2
        addi   $t0, $t0, 12
        addi   $t1, $t1, 12
        add.s  $f6, $f4, $f5
        2 s
        add.s  $f9, $f7, $f8
        j      top
done:
```

Left: 18+3*(1+6) = 39 cycles. Right: 18+2 = 20 cycles. Another cycle can be removed (but an additional stall added) by observing that the second load each iteration after the first is redundant, or by grouping loads for mul.s (early) and add.s (later).



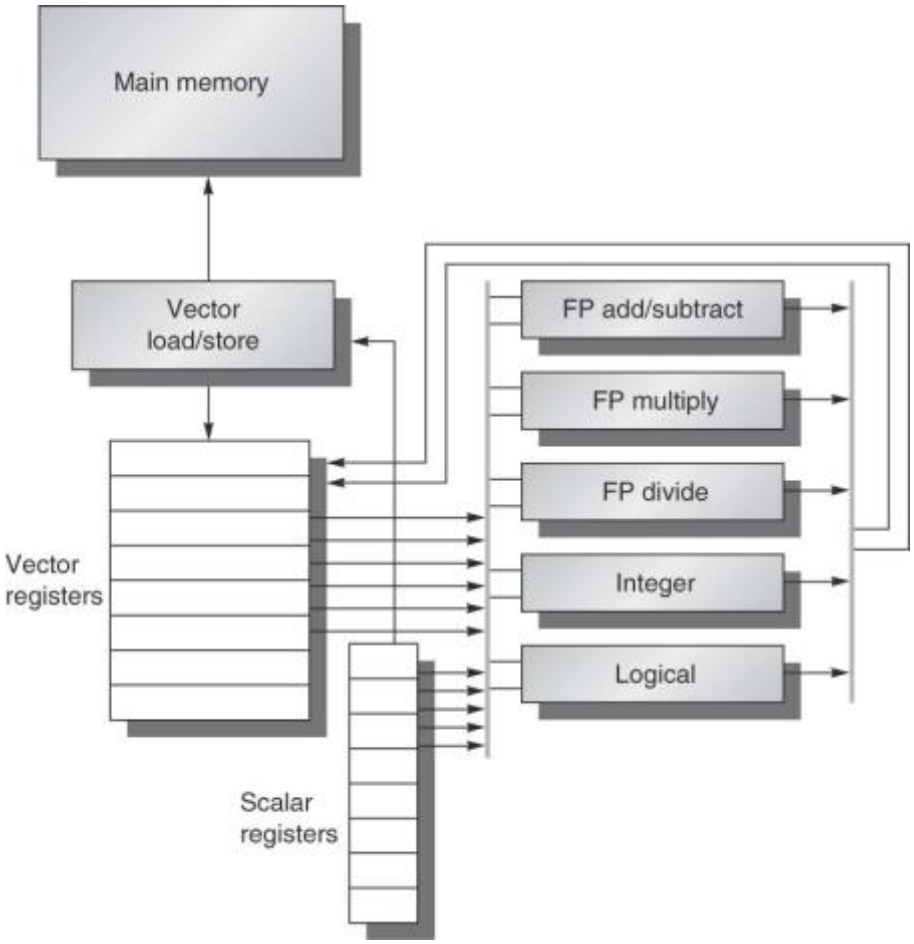
c. (10 pts) Schedule your unrolled code for execution on a VLIW processor capable of issuing an integer (Int) instruction, floating-point instruction (FP), and memory access instruction (Mem) each cycle. Assume the same execution delays and functional units as in Q4. Complete the table, indicating which instructions are issued each cycle.

Cycle	Int	FP	Mem
1	addi		
2	sltu		
3	beq		
4			lwc1
5			lwc1
6		mul.s	lwc1
7			lwc1
8		mul.s	lwc1
9			lwc1
10		mul.s	lwc1
11			lwc1
12		mul.s	
13	addi	add.s	
14	addi		
15		add.s	
16			
17			
18			
19	j	add.s	
20			
21			
22			
23			
24			

2 pts: instruction to slot mapping
8 pts: dependencies and delays
- sltu->beq
- lwc1->mul.s
- lwc1, mul.s->add.s

Question 6: Architecture Unchained (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.

Now consider the following VMIPS assembly sequence for DAXPY.

```
L.D      F0,a      ;load scalar a
LV       V1,Rx     ;load vector X
MULVS.D  V2,V1,F0  ;vector-scalar multiply
LV       V3,Ry     ;load vector Y
ADDVV.D  V4,V2,V3  ;add
SV       V4,Ry     ;store the result
```

a. (6 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.

The L.D is handled by the scalar data path. In this case, there are four convoys.:

1. LV
2. MULVS.D / LV
3. ADDVV.D
4. SV

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

4 pts: dependencies
- LV->MULVS.D
- (but not for) MULVS.D/LV
- MULVS.D, LV->ADDVV.D
- ADDVV.D->SV

2 pts: cycles calculation

b. (8 pts) Now assume *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 three convoys:

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

A chime requires 64 cycles, approximately. 64 * 3 = 192 cycles in total. 256/192 = 1.33x speedup.

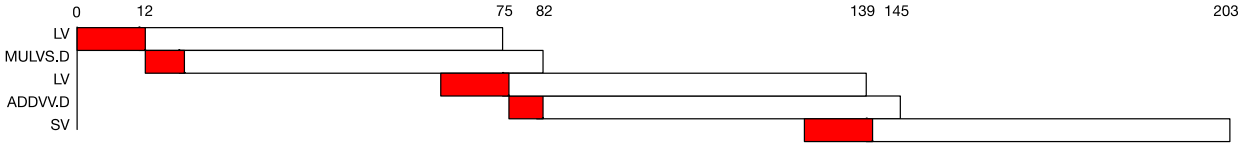
4 pts: chaining
- LV->MULVS.D
- LV->ADDVV.D

2 pts: strucutural hazards
2 pts: cycles estimation

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 the the memory access instructions.

1. LV starts at time 0, producing its first element in cycle 12, its last at 12 + 63 = 75.
2. MULVS.D works concurrently, starting in cycle 12, its first result appearing in 19, and its last in cycle 19 + 63 = 82.
3. A structural hazard prevents the second LV from starting before the first one finishes. The second LV therefore starts 12 cycles before the first one finishes (the first request enters the pipeline just after last request from the previous LV); the first element is ready at cycle 76, the last at 139.
4. The ADDVV.D starts as soon as the first element from the second LV is ready, at cycle 76, producing its first value at 82; its last appears at 145.
5. The SV can't start until after the previous LV finishes; as before, its first request enters the pipeline after the last request for the LV. The first SV finishes at 140; the last at 203.



The error introduced by counting chimes, in this case, is (203 – 192) / 203 = 5%.

d. (6 pts) Now consider a multi-lane architecture. Given *two lanes*, use chimes to estimate speedup relative to part (a) for a system (i) *without chaining*, and (ii) *with chaining*.

Using two lanes, chimes take half as many cycles; in this case, 32 cycles instead of 64 cycles.

i. 4 * 32 = 128 cycles. Speedup = 256 / 128 = 2x.
ii. 3 * 32 = 96 cycles. Speedup = 256 / 96 = 2.67x.

Additional Page