Computer System Architecture
6.823 Quiz #2
April 7th, 2017
Professors Daniel Sanchez and Joel Emer

Name: ________Solutions______________

This is a closed book, closed notes exam.
85 Minutes
13 Pages (+2 Scratch)

Notes:
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.
- Pages 14 and 15 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	________	25 Points
Part B	________	30 Points
Part C	________	20 Points
Part D	________	25 Points
TOTAL	________	**100 Points**

Part A: Complex Pipelining (25 points)

Consider the following MIPS instruction sequence. An equivalent sequence of C-like pseudocode is also provided. F1, F2, and F3 are floating point registers.

```
I1:  L.D       F2, 0(R2)         ; F2 = *r2;
I2:  L.D       F1, 0(R1)         ; F1 = *r1;
I3:  L.D       F2, 4(R1)         ; F2 = *(r1+4);
I4:  MUL.D     F3, F1, F2        ; F3 = F1 x F2;
I5:  ADD.D     F1, F2, F2        ; F1 = F2 + F2;
I6:  S.D       F3, 0(R2)         ; *r2 = F3;
I7:  S.D       F1, 4(R1)         ; *(r1+4) = F1;
```

Question 1 (4 points)

Fill out the table below to identify all Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) dependencies in the above sequence. Do not worry about memory dependencies for this question. The dependency between I3 and I4 is already filled in for you.

	Earlier (Older) Instructions						
Current Instruction	I1	I2	I3	I4	I5	I6	I7
I1	-						
I2	-	-					
I3	WAW	-	-				
I4	-	RAW	RAW	-			
I5	-	WAW	RAW	WAR	-		
I6	-	-	-	RAW	-	-	
I7	-	-	-	-	RAW	-	-

Question 2 (9 Points)

Calculate the number of cycles this code sequence would take to execute from issue of `I1` to the issue of `I7`, inclusive, on a single-issue in-order pipelined machine. The machine uses a scoreboard and has no bypassing (as in Lecture 8). The floating point multiplier, adder, and load/store units are fully pipelined, so issue is never stalled by a busy functional unit (FU). The FUs latch their inputs. Assume that functional units have latencies as shown in the table below. Register write-back takes one additional cycle. Ignore write-back conflicts. `I1` misses, but all other memory operations hit.

Operation	Load/Store that hits	Load/Store that misses	Multiplies	Adds
Latency	2 cycles	6 cycles	4 cycles	2 cycles

You may fill out the timing chart below to help you find the answer. Filling out the chart can give you partial credit. It is initialized for you below with the issue and completion/write-back cycles of I1.

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Instruction issue or <u>writeback</u>	I1	I2		<u>I2</u>			<u>I1</u>	I3		<u>I3</u>	I4	I5		<u>I5</u>	<u>I4</u>	I6

Cycle	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Instruction issue or <u>writeback</u>	I7															

An in-order core with scoreboard does not issue an instruction if its destination register appears in the writes-pending bit-vector. That is, it stalls on WAW hazards. Therefore I3 cannot issue until I1 completes. Because inputs are latched, WAR hazards do not cause stalls.

Partial credit was given if students recognize that I3 must complete after I1.

Number of cycles from issue of I1 to issue of I7, inclusive __17____

Question 3 (12 Points)

Manually rename registers in the code sequence to eliminate all WAR and WAW dependences, and reorder the instructions in the code sequence to minimize execution time. You may use register names from F1 to F7. Show the new instruction sequence and give the number of cycles this sequence takes to execute on the scoreboarded in-order pipeline. *Partial credit will be given for solutions with improved, but sub-optimal timing.*

Original instruction sequence	**Register-renamed and reordered sequence**	
`I1:   L.D      F2, 0(R2)`	I1':	`L.D      F4, 4(R1)`
`I2:   L.D      F1, 0(R1)`	I2':	`L.D      F1, 0(R1)`
`I3:   L.D      F2, 4(R1)`	I3':	`L.D      F2, 0(R2)`
`I4:   MUL.D    F3, F1, F2`	I4':	`ADD.D    F5, F4, F4`
`I5:   ADD.D    F1, F2, F2`	I5':	`MUL.D    F3, F1, F4`
`I6:   S.D      F3, 0(R1)`	I6':	`S.D      F5, 4(R1)`
`I7:   S.D      F1, 4(R1)`	I7':	`S.D      F3, 0(R1)`

You may fill out the timing chart below to help you find the answer, and for partial credit.

Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Instruction issue or <u>writeback</u>	I1'	I2'	I3' <u>I1'</u>	I4' <u>I2'</u>	I5'	<u>I4'</u>	I6'		<u>I5'</u> <u>I3'</u>	I7'						

Number of cycles from issue of I1 to issue of I7, inclusive ____10_______

Part B: Out-of-Order Processing (30 points)

This question uses the out-of-order Data-in-ROB machine described in the Quiz 2 Handout. We describe events that affect the initial state shown in the handout. Label each event with one of the actions listed in the handout. If you pick a label with a blank (_____), you also have to fill in the blank using the choices (i—v) listed below. If you pick "R. Illegal action", state why it is an illegal action. If in doubt, state your assumptions.

Example: Assume `T11` data becomes available. Instruction `I12` is issued and its effective address matches load buffer entry 4. Answer: (L, iv): Check the correctness of a speculation on memory address and find an incorrect speculation. (You can simply write L, iv)

a) Instruction `I8` finishes execution and replaces `T10`'s `src1` tag with data, and sets the `p1` bit.

(B, i): Satisfy a dependence on register value by bypassing a speculative value

b) Instruction `I8` finishes execution and writes back the new value of `R4` to `T8`'s dest `data` field, and sets its `pd` bit.

F: Write a speculative value using lazy data management

c) Instruction `I17` is dispatched to ROB entry `T17`. The instruction will write register `R6`, so tag `T17` is written into the `R6` entry of the rename table, and the valid bit is set.

G: Write a speculative value using greedy data management

d) Instruction `I17` is dispatched to ROB entry `T17`. The instruction's first operand is register `R3`, so value 3980 is copied from the register file into `T17`'s `src1` field, and the `p1` bit is set.

(C, i): Satisfy a dependence on register value by using a committed value

e) Instruction `I18` has no entry in the BTB, so PC `0xc0` is fetched for `I19`.

(E, ii): Satisfy a dependence on PC value by speculation using a dynamic prediction
OR
(D, ii): Satisfy a dependence on PC value by speculation using a static prediction

f) Assume `T11` data becomes available and the processor's divider unit is unpipelined (i.e. it can work on only one instruction at a time). Instruction `I13` is not issued until the divide finishes.

(A, vii): Satisfy a dependence on functional unit by stalling.

g) Assume all instructions through `I6` have committed. `I7` commits and writes 4000 into the `R2` entry of the register file.

P: Commit correctly speculated instruction, and replace old values with lazily updated values

h) Assume all instructions through `I7` have committed. `I8` commits and replaces `T10`'s `src1` tag with a data value and sets `T10`'s `p1` bit.

R: Illegal or broken action. There should be no tags to replace; tag replacement should have been handled in write-back.

i) Assume `T13` data becomes available, `I14` is issued, and the branch is found to be predicted correctly as not taken. The relevant branch prediction counter is decremented (unless it is already 0).

(I, iii): Speculatively update a prediction on branch direction using greedy value management OR because there isn't an undo log or buffer, accepted (H, iii) and (J, iii).

j) Assume instruction `I11` encodes an address offset of 4 (not shown in the figure). `I11` is issued, writes address 4004 into entry 3 of the load buffer, sets the corresponding valid bit, and loads data from the cache.

R: Illegal or broken action. The cache load will read stale data, as 4004 would have hit in the store buffer entry 2.

Part C: Multithreading (20 points)

In this problem you will evaluate the throughput improvement of multithreading on the following code, which computes the per-element product of two arrays:

```
float A[1024], B[1024], P[1024];
…
for (int i = 0; i < 1024; i++)
  P[i] = A[i] * B[i];
```

Here is the corresponding MIPS assembly code:

```
      ;; Assume:
      ;; R1 holds address of A[i]; initialized to base address of A
      ;; 4096(R1) holds address of B[i], based on offset from A[i]
      ;; 8192(R1) holds address of P[i], based on offset from A[i]
      ;; R2 holds number of iterations remaining; initialized to N

I1:   loop: lw.s  F1, 0(R1)
I2:         lw.s  F2, 4096(R1)
I3:         mul.s F3, F1, F2
I4:         sw.s  F3, 8192(R1)
I5:         addi  R1, R1, 4
I6:         addi  R2, R2, -1
I7:         bnez  R2, loop
```

You run this code on a single-issue in-order processor. Assume the following:

- The processor can fetch and issue one instruction per cycle.
- If an instruction cannot be issued due to a data dependency, the processor stalls.
- Loads/stores take **4 cycles** (i.e., if instruction I1 starts execution at cycle N, then instructions that depend on the result of I1 can only start execution at or after cycle N+4); multiplies take **3 cycles**; and all other instructions execute in **1 cycle**.
- The load/store unit and multiplier are fully pipelined (i.e., can start a new request each cycle).
- The end-of-loop branch is always predicted correctly.

Question 1 (5 Points)

Suppose the code runs on a multithreaded processor that performs **fixed switching**: the processor switches to the next thread every cycle (round-robin), and if the instruction of the next thread is not ready, it inserts a bubble into the pipeline. What is the minimum number of threads required to fully utilize the processor every cycle while running this code? Explain.

4 threads. The longest stall is 4 cycles between the `lw.s` and `mul.s`. To hide that latency, we require an additional 3 threads.

Question 2 (5 Points)

Now suppose the multithreaded processor performs **data-dependent switching:** the processor only switches to another thread when an instruction cannot execute due to a data dependence. If no threads have a ready instruction, the processor inserts a bubble into the pipeline. What is the minimum number of threads required to fully utilize the processor every cycle while running this code? Explain.

3 threads.

Necessity: the processor must always stall on the 3-cycle dependence between I3 and I4, therefore a thread switch occurs here. Three threads are needed to hide the latency.

Sufficiency: in steady state, the processor can issue instruction sequence I4, I5, I6, I7, I1, and I2 without stall. With three threads, these six contiguous stall-free cycles are more than enough to hide the 4-cycle latency between I2 and I3.

Question 3 (10 Points)

Assume the **fixed-switching** policy of Question 1. **Reorder and edit** the sequence of instructions to minimize the number of threads that fully utilize the multithreaded pipeline. How many threads do you need? Explain.

Partial credit will be given for solutions with a reduced, but sub-optimal number of threads.

Original instruction sequence	**Write a reordered and edited sequence**

`loop: lw.s  F1, 0(R1)`
`      lw.s  F2, 4096(R1)`
`      mul.s F3, F1, F2`
`      sw.s  F3, 8192(R1)`
`      addi  R1, R1, 4`
`      addi  R2, R2, -1`
`      bnez  R2, loop`
	`loop: lw.s  F1, 0(R1)`
	`      lw.s  F2, 4096(R1)`
	`      addi  R2, R2, -1`
	`      mul.s F3, F1, F2`
	`      addi  R1, R1, 4`
	`      sw.s  F3, 8188(R1)`
	`      bnez  R2, loop`

The decrement of `R2` is independent of all instructions but the branch, including the second `lw.s`, so it can execute without waiting for the result of that load. To break up the multiply and store, we must carefully hoist the increment of `R1` (by reducing the offset of the store by 4), as there was an anti-dependence between the `sw.s` and increment. With N total threads, each instruction is executed at the following cycles for the first thread:

```
loop: lw.s  F1, 0(R1)        1
      lw.s  F2, 4096(R1)     1 + N
      addi  R2, R2, -1       1 + 2N
      mul.s F3, F1, F2       1 + 3N
      addi  R1, R1, 4        1 + 4N
      sw.s  F3, 8188(R1)     1 + 5N
      bnez  R2, loop         1 + 6N
```

To avoid stalling on the 4- and 3-cycle latencies, we require both

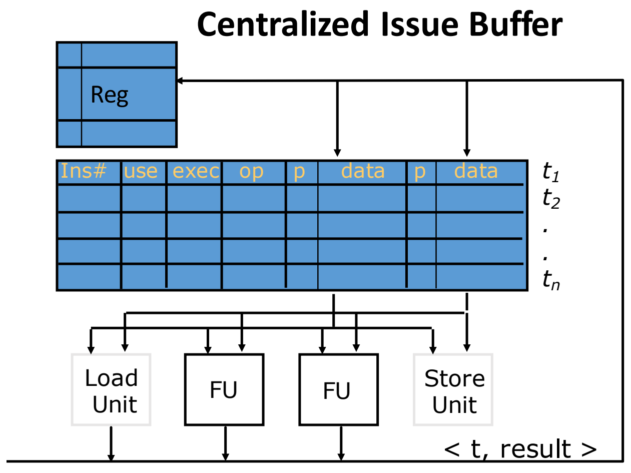
$$(1 + 3N - 1 + N) \geq 4 \Rightarrow N \geq 2$$

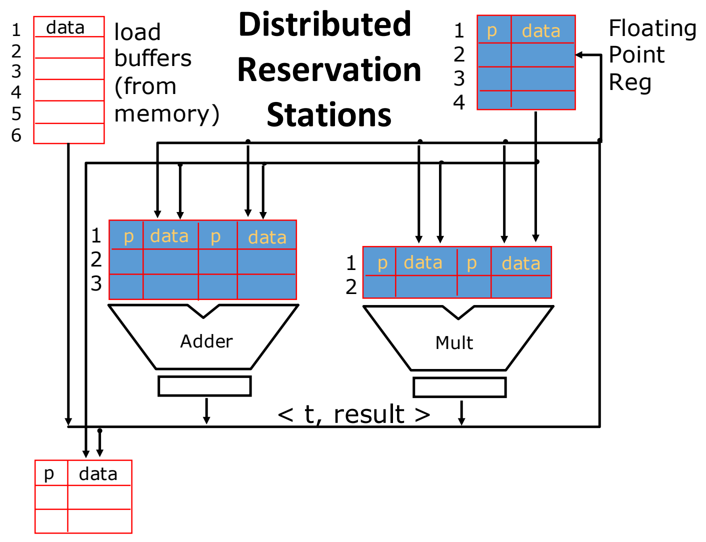
and

$$(1 + 5N - 1 + 3N) \geq 3 \Rightarrow N \geq 1.5$$

Number of threads needed to fill the pipeline with reordered code: ___2_____

Part D: Centralized vs. Decentralized Issue (25 points)





This problem focuses on the issue logic of a superscalar out-of-order machine. You will explore the tradeoffs between a centralized issue buffer (left) and decentralized reservation stations (right). In both designs, an issue buffer entry *is allocated when each instruction is decoded and is freed when the instruction is dispatched to a functional unit*. The decentralized design (right), introduced by Tomasulo in the IBM 360/91, distributes the issue buffer entries around the processor, with one set of entries per functional unit. In such a design, the distributed entries are called "reservation stations". Do not worry about instruction commit or speculative buffering; you will focus on stages from entry allocation to instruction dispatch and completion.

The following applies to both designs. Your desired average throughput is 1.5 instructions per cycle. Consider a stream of floating point instructions that consists of 2/3 adds and 1/3 multiplies. For this stream, you observe that the average latency of an instruction *from allocation in issue buffer to functional unit completion* is 12 cycles. The processor's adder and multiplier are each fully pipelined. *Once an instruction is dispatched* to the functional unit, both the adder and multiplier take 3 cycles.

Type of operation	Stream instruction ratio	FU latency
Add	2/3	3 cycles
Multiply	1/3	3 cycles

Average throughput	Average total latency
1.5 instructions per cycle	12 cycles

Question 1 (8 points)

Consider the centralized issue buffer. In steady state, how many issue buffer entries are in use on average?

Average throughput:
T = 1.5 IPC

We need the average latency of an instruction from allocation to dispatch, but are given allocation to completion, as well as functional unit latencies. First compute the average latency in the functional units:

Lfu = 3 cycles (average of 3 and 3)

The average occupancy latency of an entry is therefore

Locc = Ltotal – Lfu = 12 – 3 = 9 cycles

Little's law:
Nocc = T * Locc = 1.5 * 9 = **13.5 entries**.

Average issue buffer entries used ____13.5_____

Question 2 (9 points)

Now consider the decentralized design. You observe that the average latency of add instructions *from allocation to completion* is 14 cycles for the stream of interest. How many reservation station entries are in use, on average, at each functional unit (adder, multiplier) in steady state?

Since the average throughput of all instruction types is 1.5 IPC, then the throughput of the instruction types individually scales with their ratio.

Tadd = 2/3 * 1.5 = 1 IPC
Tmul = 1/3 * 1.5 = 0.5 IPC

The next step is to determine the average latency that an instruction spends occupying its reservation station entry, i.e. from allocation to dispatch. This is straightforward for adds.

Laddocc = Laddtotal – Laddfu = 14 – 3 = 11 cycles

We must find the total latency for multiplies. The average total latency of all instruction types is 12 cycles. We know that adds contribute 2/3 to that average.
Ltotal = 2/3 Laddtotal + 1/3 Lmultotal
- ⇨ Lmultotal = 3 * (12 – 14 * 2 / 3) = 36 - 28 = 8 cycles

Since the average latency of multiply instructions from allocation to completion is 8 cycles, we can determine the average occupancy latency in a reservation station from the functional unit latency.
Lmulocc = Lmultotal – Lmulfu = 8 – 3 = 5 cycles

Now apply Little's Law:

Naddocc = Tadd * Laddocc = 1 * 11 = 11 entries
Nmulocc = Tmul * Lmulocc = 0.5 * 5 = 2.5 entries

Average Adder reservation station entries	**11**
Average Multiplier reservation station entries	**2.5**

Question 3 (4 points)

Again consider the decentralized design. Suppose we run an instruction stream symmetric to the original. It is comprised of 1/3 adds and 2/3 multiplies, and the average latency of *multiply* instructions from allocation to functional unit completion is 14 cycles. How many reservation station entries are in use, on average, at each functional unit (adder, multiplier) in steady state? *For full and/or partial credit, explain your reasoning.*

In the symmetric stream, we will swap all instances of an add-type variable with a multiply-type variable (ratios, throughput, latencies, etc). Therefore the final solution is just a swap of Question 3.

Average Adder reservation station entries	**2.5**
Average Multiplier reservation station entries	**11**

Question 4 (4 points)

Qualitatively, name one advantage and one disadvantage of the distributed reservation station design over the centralized issue buffer. Explain.

Advantage: At instruction completion, an associative lookup in performed on issue buffers to find tags to replace with new data values. With distributed reservation stations, this associative lookup is performed on more, smaller structures, saving energy.

Disadvantage: We can use the full capacity of the centralized issue buffer to react to different instruction stream compositions, whereas we would need more aggregate decentralized entries to handle a variety of instruction streams.

Scratch Space

Use these extra pages if you run out of space or for your own personal notes. We will not grade this unless you tell us explicitly in the earlier pages.