Computer System Architecture
6.823 Quiz #4
May 15th, 2019

Name: _________SOLUTIONS__________

This is a closed book, closed notes exam.
80 Minutes
17 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 18 and 19 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	________	23 Points
Part B	________	26 Points
Part C	________	25 Points
Part D	________	26 Points
TOTAL	________	**100 Points**

Part A: VLIW (23 points)

Consider the following C code sequence and its corresponding MIPS assembly code below. A, B, X, and Y are `float` (32-bit floating point) arrays of size N.

```
for (i = 0; i < N; i++) {
   A[i] = X[i] + Y[i];
   B[i] = X[i] * Y[i];
}


// Initial values:
// r1 := &X[0]; r2 := &Y[0]; r3 := &A[0]; r4 := &B[0]
// r5 := &X[N]
```

```
I1:  loop: ld f1, 0(r1)
I2:        ld f2, 0(r2)
I3:        fadd f3, f1, f2
I4:        st f3, 0(r3)
I5:        fmul f4, f1, f2
I6:        st f4, 0(r4)
I7:        addi r1, r1, 4
I8:        addi r2, r2, 4
I9:        addi r3, r3, 4
I10:       addi r4, r4, 4
I11:       bne r1, r5, loop
```

Assume a VLIW processor with the following characteristics:

- Five functional units: 2 Memory Units for loads and stores, one INT ALU for integer operations (including branches), and 2 FP ALUs for floating point operations.
- All functional units are fully pipelined and latch their inputs.
- The data cache has two read/write ports and is fully pipelined (i.e., it can accept two new requests every cycle).
- All load instructions hit in the cache and take 3 cycles including writeback (i.e., if load instruction `I` starts execution at cycle *K*, then instructions that depend on the result of `I` can only start execution at or after cycle *K+3*).
- All integer ALU operations take a single cycle.
- All floating point multiplies take 2 cycles, and floating point adds take a single cycle
- Assume perfect branch prediction.

Question 1 (8 points)

Schedule one iteration of the loop on this processor on the following table, where each row corresponds to a VLIW instruction. For full credit, the loop should take the minimum number of VLIW instructions. You may use floating point registers `f0` to `f31` and integer registers `r1` to `r31` (`r0` is hardwired to 0 as usual).

Note: In case you need it, there is an extra table in the last page of the quiz.

Memory Unit	Memory Unit	INT ALU	FP ALU	FP ALU
ld f1, 0(r1)	ld f2, 0(r2)	addi r1, r1, 4		
		addi r2, r2, 4		
			fadd f3, f1, f2	fmul f4, f1, f2
st f3, 0(r3)		addi r3, r3, 4		
st f4, 0(r4)		addi r4, r4, 4		
		bne r1, r5, loop		

The above solution is possible. Note that if we put addi's of r3 and r4 before the stores and modify the constant offsets, there will be one fewer vliw instruction at the end for the branch (full credit in that case also).

Question 2 (10 points)

How many iterations of the loop do we have to unroll to cover all latencies among VLIW instructions (i.e., so that each VLIW instruction performs at least one operation)? Schedule the unrolled loop on the following table. Assume that the number of loop iterations N is a multiple of your chosen unrolling factor, so that you do not need prolog or epilog code. What is the resulting throughput in number of cycles per iteration of the original loop?

Memory Unit	Memory Unit	INT ALU	FP ALU	FP ALU
ld f1, 0(r1)	ld f2, 0(r2)			
ld f1, 4(r1)	ld f2, 4(r2)	addi r1, r1, 8		
		addi r2, r2, 8		
			fadd f3, f1, f2	fmul f4, f1, f2
st f3, 0(r3)			fadd f3, f1, f2	fmul f4, f1, f2
st f4, 0(r4)	st f3, 4(r3)	addi r3, r3, 8		
	st f4, 4(r4)	addi r4, r4, 8		
		bne r1, r5, loop		

If we use the solution with 1 fewer VLIW instruction, the answer is 0 since there are no empty cycles (full credit given for that solution). If you wrote down the solution with 1 extra VLIW cycle, you need to unroll the loop twice as shown above.

# of iterations unrolled: ________2_______ Iterations

Throughput: _____4______ Cycles / Iteration of original loop

Question 3 (5 points)

Suppose we applied software pipelining to improve throughput even further. What is the maximum throughput that can be achieved for this loop? Note that you do not need to write the software-pipelined loop to answer this question. Briefly explain what sets this maximum throughput.

Here, the only thing that matters is which functional unit will bottleneck our system given that we do full loop unrolling and software pipelining. Integer ops will be amortized away with enough loop unrolling, and there are less FP ops than memory ops. Since there are 4 memory ops and 2 memory units, that is at least 2 VLIW instructions per iteration.

Maximum Throughput: __2__ Cycles / Iteration of original loop

Part B: Vector Processors (26 points)

In this part, you will write code that targets the vector processor described in the quiz handout.

Consider the following C code sequence and its corresponding MIPS assembly code. Initially, registers r1 and r2 hold the addresses of A[0] and B[0], r3 holds the value x, and r10 holds the address of A[N].

```
     for (i = 0; i < N; i++) {
          if (A[i] > 0) {
               A[i] = x*B[i] + A[i];
          }
     }
```

```
I1:   loop: ld r4, 0(r1)
I2:         ~~bgez~~blez r4, skip
I3:         ld r5, 0(r2)
I4:         mul r5, r5, r3
I5:         add r5, r5, r4
I6:         st r5, 0(r1)
I7:    skip: addi r1, r1, 4
I8:         addi r2, r2, 4
I9:         bne r1, r10, loop
```

Question 1 (10 points)

Write an equivalent vector code for the above loop. Assume that arrays `A` and `B` do not overlap, and that `N` is a multiple of the maximum vector length. For full credit, your code should use the minimum possible number of instructions. You may use vector registers `v0` to `v31`, and scalar registers `r1` to `r31` (`r0` is hardwired to 0 as usual).

```
         addi r20, r0, 16
         setvlr r20  // Set vector length register to 16
         cvm         // Clear vector mask (enable all elements)


   loop: lv v4, r1
         sgt v4, r0
         lv v5, r2
         mul.vs v5, v5, v3
         add.vv v5, v5, v4
         sv v5, v1
         addi r1, r1, 64
         addi r2, r2, 64
         cvm
         bne r1, r10, loop
```

There are multiple ways you can put the sgt instruction, as long as it is before the sv instruction.

Question 2 (10 points)

Consider the case when vector A has a repeating pattern of [1 0 0 1 0 1 1 0]:

A = [1 0 0 1 0 1 1 0 1 0 0 1 ...]

a) Our processor uses a simple way of handling vector masks. For each vector instruction, each lane computes the results for all the elements regardless of the vector mask, and only writes back the elements for which the corresponding mask is set. What is the average throughput of the loop in steady state in terms of number of cycles per iteration?
55 cycles. The scalar instructions overlap with the last SV, so are not shown. The assumption is that CVM must wait for sv to end to reset the masks.

Note that since the final number is dependent on the assumptions you make for the sgt.vs and cvm instructions, we only took off points for correct latency between dependent vector instructions (9 cycles) and where your assumptions contradicted the description of the architecture we provided.

Instruction	Iteration i	Iteration (i+1)
`lv`	0	56
`sgt.vs`	9	
`lv`	18	
`mul.vs`	27	
`add.vv`	36	
`sv`	45	
`cvm`	54	

b) Suppose that our processor now has a density-time implementation. With this optimization, each lane **only processes the elements with a non-zero mask**. Because different lanes may have different numbers of active elements, the instruction completes execution when all lanes finish processing active elements. What is the average throughput of the loop in steady state in terms of number of cycles per iteration?

47 cycles. Now operations after SGT only have a 2-cycle instead of 4-cycle functional unit latency. If your solution had vector instructions take 2 less cycles compared to your latency in (a) we gave credit for that.

Instruction	Iteration i	Iteration (i+1)
`lv`	0	48
`sgt.vs`	9	

`lv`	18	
`mul.vs`	25	
`add.vv`	32	
`sv`	39	
`cvm`	46	

c) Consider when A has a repeating pattern of [1 0 0 0]:

A = [1 0 0 0 1 0 0 0 ...]

Does this have higher throughput (i.e., lower cycles per iteration) with our density-time optimization compared to the previous pattern of array A? Explain why or why not.

This has lower throughput since one lane has all of its elements masked, so it will slow down the rest of the lanes' execution.

Question 3 (6 points)

Suppose now that the vector processor **supports chaining**, does **not** have the density-time implementation from Question 2, and array A has the previous repeating pattern of [1 0 0 1 0 1 1 0]:

A = [1 0 0 1 0 1 1 0 1 0 0 1 ...]

With chaining, a vector instruction that depends on a previous instruction can start execution if the first set of elements it processes is either already written to the vector register file or is available in the writeback stage (we add the requisite bypass paths).

What is the throughput of the loop in steady state? Assume that the vector mask register is updated at the end of the cycle when an entire `s--.vs` instruction is finished (i.e., vector instructions cannot be chained after `s--.vs` instructions).

If it is assumed that chaining allows SGT to set masks on elements already operated on by previous LV:

Instruction	Iteration i	Iteration (i+1)
`lv`	0	39
`sgt.vs`	4	
`lv`	13	
`mul.vs`	18	
`add.vv`	23	
`sv`	28	
`cvm`	37	

Else, we add 5 cycles to the above solution, which comes to 44 cycles.

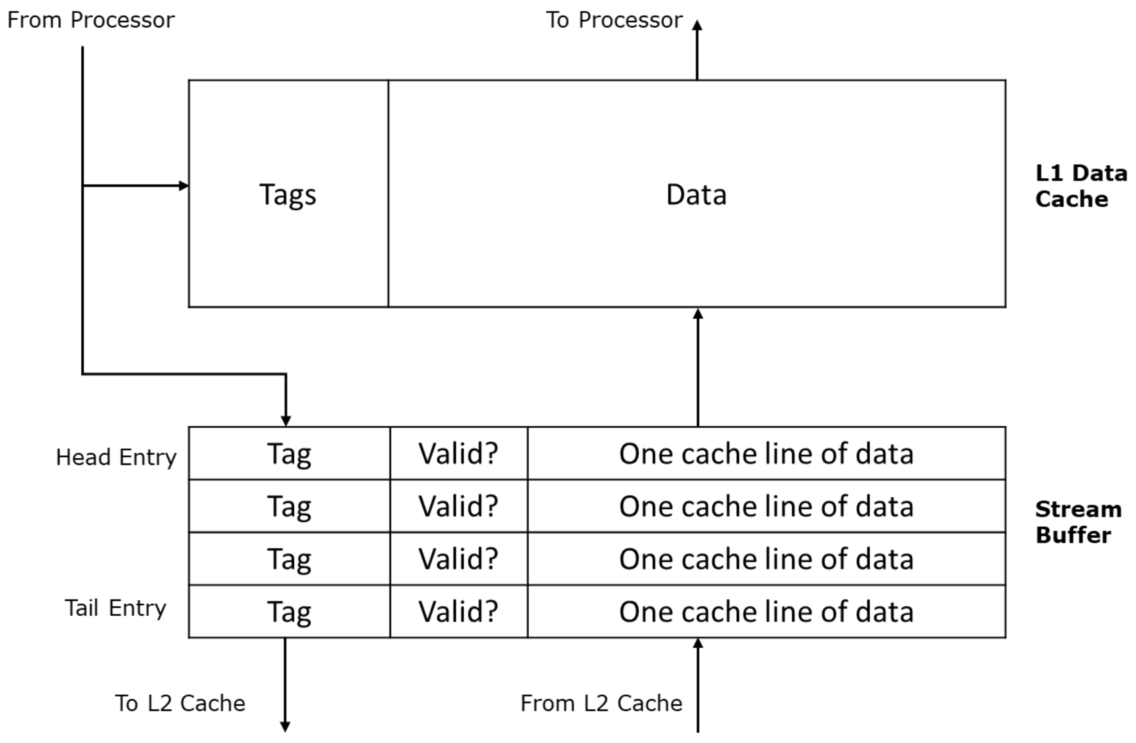
Again, we mostly looked at whether you had correct latency between dependent instructions due to chaining, which should be 5 cycles, As shown below:

Cycle	0	1	2	3	4	5
LV	D	M1	M2	M3	M4	W
		D	M1	M2	M3	M4
			D	M1	M2	M3
				D	M1	M2
ADDV						D

Notice that ADDV can be issued at cycle 5 since "a vector instruction that depends on a previous instruction can start execution if the first set of elements it processes is either already written to the vector register file or is available in the writeback stage"

Part C: Reliability (25 points)

Ben Bitdiddle wants to add a stream prefetcher between the L1 data cache and the L2 cache of his processor. This stream prefetcher predicts L1 cache misses and fetches the predicted cache lines speculatively. To avoid polluting the L1 cache, the stream prefetcher buffers prefetched lines into a **stream buffer**, a 4-entry tagged FIFO queue shown below. Each stream buffer entry contains the prefetched data, the corresponding tag, and a valid bit.



When a cache miss occurs in the L1 data cache, the stream prefetcher requests the next 4 consecutive cache lines from the L2, enters their tags in the buffer, and sets the valid bits to zero. Each valid bit is set once the corresponding entry is prefetched from the L2 cache. For instance, an L1 miss to a cache line with *line address* L will cause lines L+1, L+2, L+3, and L+4 to be prefetched.

Subsequent accesses to the L1 data cache that miss compare their address against the head of the buffer to see if it contains a valid entry with a matching tag. If the access hits in the head entry of the buffer, the buffer serves the data to the L1 cache, and the prefetcher initiates a fetch for the line that follows the tail entry of the buffer. If the access misses in the head entry of the buffer, the request is forwarded to the L2, the stream buffer is flushed, and the next 4 consecutive lines are prefetched from the L2.

Question 1 (10 points)

The L1 data cache consists of **16B cache lines** and is initially empty. The following sequence of events occur in the system:

Cycle	Event
`0`	Load to address `0x00` misses in L1 cache and the stream buffer
`50`	Four following cache lines arrive at the stream buffer
`100`	Load to address `0x10` misses in L1 cache, and hits in the head of the stream buffer
`110`	Load to address `0x20` misses in L1 cache, and hits in the head of the stream buffer
`120`	Load to address `0xC0` misses in L1 cache and the stream buffer
`170`	Four following cache lines arrive at the stream buffer
`200`	Load to address `0xD0` misses in L1 cache, and hits in the head of the stream buffer

Indicate whether the different fields **of the head entry of the buffer** are ACE, unACE, or unknown for each of the following cycle intervals. Explain any assumptions you make.

Cycle Interval	Tag	Valid bit	Data
`0-50`	unACE	unACE	unACE
`50-100`	unACE	unACE	ACE
`100-110`	unACE	unACE	ACE
`110-120`	ACE	unACE	unACE
`120-170`	unACE	unACE	unACE
`170-200`	unACE	unACE	ACE

The Tag is only ACE if the head is going to be accessed by a load that misses in the buffer, since a bit flip may cause a hit that serves the wrong data.
The valid bit is always unACE since it is never accessed when it is zero (the only ACE-ness we care about it is if valid bit flips from 0->1, serving invalid data).
The Data is unACE if it is overwritten before being read by a load, and ACE otherwise (we assume all loads are from ACE instructions).

Question 2 (5 points)

For the given sequence of events in Question 1, what is the Architectural Vulnerability Factor (AVF) of the **data field** in the **head entry**?

AVF = # Average number of ACE bits in a cycle / total # bits in the structure = 90/200 = 0.45

Question 3 (5 points)

We now have a different program that traverses a linked list. Each node object in the linked list is stored across 2 consecutive cache lines, and different nodes are stored in non-consecutive cache lines. Qualitatively describe how the AVF of the data field would differ between the head and tail entry for this program.

The AVF would be higher in the head entry than the tail entry. Almost all of the head entries will be read, whereas almost none of the tail entries will be read.

Question 4 (5 points)

Ben wants to add protection from random bit flips for the three fields in his stream buffer. For each field, indicate the most appropriate protection mechanism among the following:

- **No protection**
- **Parity bit**: Ability to detect single bit flips
- **ECC**: Ability to correct single bit flips, and detect two bit flips.

Justify your answer for each field with one or two sentences.

All three fields need some form of protection since they are sometimes ACE (even the valid bit, since we can generally have cases where a load accesses the buffer while it is prefetching lines from another miss). However, ECC is overkill since the stream buffer contains a copy of the data -- if we detect an error, the processor can simply go to the L2 cache to fetch the correct data. Thus, the correct answer here is parity bit for all fields.

If you answered No protection for Valid bit because it is unACE for the given trace, we gave full points also.

Part D: Transactional Memory (26 points)

We are given a 2-core system that supports hardware transactional memory (HTM). For any HTM design, the memory system dynamically tracks the set of addresses read or written by each transaction (i.e., its read set and write set) as accesses are performed.

Our HTM implements a **lazy & optimistic HTM**, which uses lazy version management and optimistic conflict detection. Conflicts are detected when a transaction attempts to commit. The finished transaction validates its write-set with coherence actions. If any of its writes appear in the read- or write-set of other transactions in the system, a conflict is declared. On a conflict, the committing transaction is given priority (i.e., committer wins). A transaction that aborts due to a conflict re-executes after waiting for 30 cycles.

Each core in the system executes the transactional code shown below:

```
int txns;
...
while (true) {
  // Code wrapped by atomic is a single transaction
  atomic {
    txns++;  // update shared counter, takes about 10 cycles
    work();  // takes about 1000 cycles
  }
}
```

Each thread runs the same transaction code in a loop. This transaction first increments the `txns` shared counter, then performs around 1000 cycles worth of work. The `work()` function causes negligible conflicts, so we will focus on the conflicts caused by reads and writes to the `txns` shared counter. For the following questions, assume that `txns` is stored in address T.

Question 1 (8 points)

Suppose transaction X starts at cycle 0 in core 0, and transaction Y starts at cycle 30 in core 1, and they would produce the following schedule of memory operations:

Cycle	0	10	20	30	40	…	1030	1040	1050	1060	1070
Txn X	`Begin`	`Rd T`	`Wr T`	`work() accs`				`End`			
Txn Y				`Begin`	`Rd T`	`Wr T`	`work() accs`				`End`

Assume that `work()` accesses (reads and writes), shown greyed out, never conflict.

a) In the absence of conflict detection and version management (i.e., no HTM), if the memory operations interleaved in the given order, would the transactions be serializable? If so, circle what would be the apparent commit order of the transactions, or circle "Not serializable".

[✓ **X before Y**] **Y before X** **Not serializable**

b) Does our lazy & optimistic HTM cause any of these transactions to abort? If so, indicate at what cycle the abort happens.

Transaction(s) aborted ……. X ……. [✓ Y] ……. X and Y ……. none

Cycle at which abort happens, if any _____1040____

c) Remember that both cores run transactions in a loop. In the best case, roughly what portion of the transactional work done in our 2-core system will be discarded due to aborts? (*Hint:* consider different skews between transactions X and Y.) Roughly, what speedup do you expect from this code compared to a single-thread implementation? Briefly explain your answers.

The answer is around 50% work aborted, and thus no speedup. Notice that unless the transactions line up exactly such that both X end and Y begin at cycle 1040, one will always abort the other. Thus, only one transaction will do committed work almost all the time.

Question 2 (7 points)

We rewrite our transactions so that they update the `txns` shared counter at the end of the transaction instead of at the beginning, as shown below:

```
atomic {
  work();  // takes about 1000 cycles
  txns++;  // update shared counter, takes about 10 cycles
}
```

Suppose transaction X starts at cycle 0 in core 0, and transaction Y starts at cycle 30 in core 1, and they would produce the following schedule of memory operations:

Cycle	0	10	20	30	40	…	1030	1040	1050	1060	1070
Txn X	`Begin`		`work() accs`			`Rd T`	`Wr T`	`End`			
Txn Y				`Begin`		`work() accs`			`Rd T`	`Wr T`	`End`

Assume that `work()` accesses (reads and writes), shown greyed out, never conflict.

a) In the absence of conflict detection and version management (i.e., no HTM), if the memory operations interleaved in the given order, would the transactions be serializable? If so, circle what would be the apparent commit order of the transactions, or circle "Not serializable".

[✓ X before Y] **Y before X** **Not serializable**

b) Does our lazy & optimistic HTM cause any of these transactions to abort? If so, indicate at what cycle the abort happens.

Transaction(s) aborted ……. X ……. Y ……. X and Y ……. [✓ none]

Cycle at which abort happens, if any ______________

We modify our lazy & optimistic HTM to perform **early commits**. In this new HTM, instead of waiting for a transaction to finish before trying to commit, we allow a single transaction in the system to start committing **before it finishes execution**.

In this HTM, a core can commit a transaction only if it has a **commit token**. There is only one commit token in the system, which cores send to each other over time. As soon as a core receives the commit token, it starts committing its currently running transaction: all its writes **are immediately made visible to other transactions**, and later reads or writes by the committing transaction abort any other transaction they conflict with (i.e., a non-committing transaction will abort if it sees a read form the committing transaction to a line in its write set, or a write to a line in its read or write sets). Once the committing transaction finishes, the core sends the commit token to another core.

When a core does not have the commit token, it runs a transaction like the lazy & optimistic HTM: the core tracks the transaction's read and write sets and buffers its writes until it receives the commit token. If the core receives the commit token while the transaction is running, it starts the commit process immediately; if the transaction finishes before its core has received the commit token, the core stalls until it receives the commit token, and commits the transaction at that point.

Early commits preserve transactional semantics even though non-committing transactions see the writes of the committing transaction before it finishes. This is because the accesses of the committing transaction appear ordered before those of non-committing transactions.

Question 3 (6 points)

Consider the original code, with `txns++` at the **beginning** of the transaction, and the transaction schedule from Question 1, but using the **early-commit HTM**. Like before, transaction X starts at cycle 0 in core 0, and transaction Y starts at cycle 30 in core 1, and they would produce the following schedule of memory operations:

Cycle	0	10	20	30	40	…	1030	1040	1050	1060	1070
Txn X	`Begin`	`Rd T`	`Wr T`	`work() accs`				`End`			
Txn Y				`Begin`	`Rd T`	`Wr T`	`work() accs`				`End`

Assume that `work()` accesses (reads and writes), shown greyed out, never conflict. Also assume that **core 0 initially has the commit token**.

a) Does our early-commit HTM cause any of these transactions to abort? If so, indicate at what cycle the abort happens.

Transaction(s) aborted .... X .... Y .... X and Y .... [✓ **none**]

Cycle at which abort happens, if any ____________

b) Remember that both cores run transactions in a loop. In the best case, roughly what portion of the transactional work done in our 2-core system will be discarded due to aborts? Roughly, what speedup do you expect from this code compared to a single-thread implementation? Briefly explain your answers.

This time, the two transactions will almost always successfully commit no matter the skew unless they line up exactly such that Y begins at most 10 cycles after X. Thus, the expected speedup is 2X.

Question 4 (5 points)

Now assume the `work()` routine takes around **100 cycles** for transactions running in core 0, and **1000 cycles** for transactions running in core 1. Consider the three implementations we have explored so far:

1. Lazy & optimistic HTM running transactions with `txns++` at transaction begin.
2. Lazy & optimistic HTM running transactions with `txns++` near transaction end.
3. Early-commit HTM running transactions with `txns++` at transaction begin.

Which of these three implementations will achieve the highest throughput in terms of **transactions per second**? Which will achieve the lowest throughput? Briefly explain why.

Note: Remember that, in our 2-core early-commit HTM, a core always sends the commit token to the other core after it commits one transaction.

The key point here is to look at the commit pattern of the 1000-cycle and 100-cycle transactions for these three systems.

For 1, the often-committing 100-cycle transactions will almost always conflict with the 1000-cycle transaction, and thus only 100-cycle transactions will mostly commit. For 3, it's the other way around since the 1000-cycle transaction will delay the 100-cycle transaction's commit while it has the commit token, and thus it will have lower txns/sec even compared to 1. 2 is the best performing system by the given metric since both transaction types will almost always commit due to the conflicting memory operations happening just before commit.

Scratch Space

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

Extra VLIW Instruction Table

Use this as scratch space or if you need a new one to answer a question from Part A.

Memory Unit	Memory Unit	INT ALU	FP ALU	FP ALU