Computer System Architecture
6.823 Quiz #4
May 17th, 2017
Professors Daniel Sanchez and Joel Emer

Name: _____**Solutions**_____

This is a closed book, closed notes exam.
85 Minutes
18 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 19 and 20 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	_____	27 Points
Part B	_____	24 Points
Part C	_____	25 Points
Part D	_____	24 Points
TOTAL	_____	**100 Points**

Part A: VLIW Programming (27 points)

Consider the following C code sequence. Arrays A, B, C, and D contain N 32-bit integer elements.

```
for (int i = 0; i < N; i++) {
  if (A[i] >= B[i]) {
    C[i] = A[i] - B[i];
  } else {
    D[i] = B[i] – A[i];
  }
}
```

A corresponding MIPS implementation is shown below.

```
;; Initial values:
;;  r1 := &A[0], r2 := &B[0], r3 := &C[0], r4 := &D[0]
;;  r5 := &A[N] (first address after array A)

loop: LD  r6, 0(r1)
      LD  r7, 0(r2)
      SUB r8, r6, r7
      BLZ r8, else
      ST  r8, 0(r3)
      J   next
else: SUB r8, r7, r6
      ST  r8, 0(r4)
next: ADDI r1, r1, 4
      ADDI r2, r2, 4
      ADDI r3, r3, 4
      ADDI r4, r4, 4
      BNE r1, r5, loop
```

In the rest of the problem, assume a VLIW processor with the following characteristics:

- Functional units and load/store 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 4 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+4*).
- All other instructions take 1 cycle due to bypassing logic.
- Perfect branch prediction and 100% hit rate in the instruction and data caches.

Question 1 (8 points)

Consider a VLIW processor. Each instruction can contain up to two integer ALU operations (including branches) and two memory operations. In addition, in this machine, any operation can be predicated on inequality with any general-purpose register. For example:

`[r4>=0] ADDI r1, r1, 1` executes the add instruction only if `r4` is greater than or equal to zero; similarly,
`[r4<0]  ADDI r2, r2, 1` executes the add instruction only if `r4` is negative.

Write VLIW code for the original instruction sequence, assuming the VLIW architecture has the fixed latencies previously mentioned, and no stall logic except for cache misses. You may reorder and modify the code, and use registers `r1` to `r30`. For full credit, your implementation should use the minimum number of VLIW instructions (but no unrolling or software pipelining).

Label	Memory Unit	Memory Unit	ALU/Branch	ALU/Branch
loop	LD r6, 0(r1)	LD r7, 0(r2)		
			ADD r1, r1, 4	ADD r2, r2, 4
			ADD r3, r3, 4	ADD r4, r4, 4
			SUB r8, r6, r7	SUB r9, r7, r6
	[r8>=0] SW r8,-4(r3)	[r8<0] SW r9,-4(r4)	BNE r1, r5 loop	

Question 2 (13 points)

Fill in the following table by unrolling three loop iterations. For full credit, your implementation should use the minimum number of VLIW instructions, but no software pipelining. Assume you may use registers `r1` to `r30`. Assume `N` = 3*k*, for any integer *k* >= 1.

Label	Memory Unit	Memory Unit	ALU/Branch	ALU/Branch
loop	LD r6, 0(r1)	LD r7, 0(r2)		
	LD r8, 4(r1)	LD r9, 4(r2)	ADD r3, r3, 12	ADD r4, r4, 12
	LD r10, 8(r1)	LD r11, 8(r2)		
			ADD r1, r1, 12	ADD r2, r2, 12
			SUB r20, r6, r7	SUB r21, r7, r6
	[r20>=0] SW r20,-12(r3)	[r20<0] SW r21,-12(r4)	SUB r22, r8, r9	SUB r23, r9, r8
	[r22>=0] SW r22,-8(r3)	[r22<0] SW r23,-8(r4)	SUB r24, r10, r11	SUB r25, r11, r10
	[r24>=0] SW r24,-4(r3)	[r24<0] SW r25,-4(r4)	BNE r1, r5 loop	

Question 3 (6 points)

Let's see how well loop unrolling improved performance.

a) For your non-unrolled VLIW code in Question 1, what is the number of cycles per loop iteration in steady state?

6 cycles/iteration

b) For your unrolled VLIW code in Question 2, what is the number of cycles per unrolled loop iteration in steady state?

8 cycles/iteration (8/3 was also accepted)

c) What is the speedup achieved by your unrolled VLIW code over your non-unrolled VLIW code?

6 / (8 / 3) = 9/4 = 2.25x speedup

Part B: Trace Scheduling (24 points)

In this part, you will analyze four trace schedules of the following MIPS code sequence, and write compensation code for them. Assume the following:

- `p` is a pointer to a 32-bit integer, held in `r1`.
- `a` is a 32-bit integer held in register `r2`, whose value is less than 100,000.
- `b` is a 32-bit integer held in register `r3`.
- The contents of `r3` and `r4` do not affect instructions at or after `next`.

```
        ;; if (p != NULL) a += *(p + b);
        ;; else a = (a + 1) / (b + 5);

  I0:  BNEZ r1, I5            ;; p != NULL ?
  I1:  ADDI r3, r3, 5         ;; r3 := b + 5
  I2:  ADDI r2, r2, 1         ;; r2 := a + 1
  I3:  DIV  r2, r2, r3        ;; a = (a + 1) / (b + 5)
  I4:  J    next
  I5:  ADD  r4, r1, r3        ;; r4 := p + b
  I6:  LD   r4, 0(r4)         ;; r4 := *(p + b)
  I7:  ADD  r2, r2, r4        ;; a += *(p + b)
next:  ...
```

To aid with trace scheduling, the ISA is augmented with two instructions discussed in lecture: *load-speculate*, which if used, is followed in the program by a *load-check*.

Instruction	Format and Description
Load-Speculate	`LD.S rt, offset(rs)` Load the contents of the effective address of `rs + offset` into register `rt`, but if the access faults, instead load zero into `rt` and set its poison bit, and do not cause an exception.
Load-Check	`CHK.S rt, target` Check if register `rt` was written by a `LD.S` that set its poison bit (e.g., due to a segmentation or page fault). If so, jump to `target` to service the exception and handle any necessary cleanup.

In the following questions, a trace scheduling compiler optimizes the above code sequence in different ways, assuming different properties of the branch `I0`. Write the compensation code for each given trace, *or explain why the trace schedule has no possible compensation code*. If `CHK.S` is necessary in your compensation code, do not worry about its target (e.g. use `exception` as the target).

Question 1 (6 points)

The optimized trace below assumes `I0` is frequently taken.

I5:	ADD r4, r1, r3	compensation:
I0':	BEZ r1, compensation	ADDI r3, r3, 5
I6:	LD r4, 0(r4)	ADDI r2, r2, 1
I7:	ADD r2, r2, r4	DIV r2, r2, r3
next:	...	
		J next
	Trace 1	**Compensation 1**

Question 2 (6 points)

The optimized trace below assumes `I0` is rarely taken.

I1:	ADDI r3, r3, 5	compensation:
I2:	ADDI r2, r2, 1	SUBI r2, r2, 1
I0:	BNEZ r1, compensation	ADD r4, r1, r3
I3:	DIV r2, r2, r3	LD r4, -5(r4)
next:	...	ADD r2, r2, r4
		J next
	Trace 2	**Compensation 2**

Question 3 (6 points)

The optimized trace below assumes `I0` is rarely taken.

I1:	ADDI r3, r3, 5	compensation:
I2:	ADDI r2, r2, 1	
I3:	DIV r2, r2, r3	No compensation is possible because in general we cannot recover the remainder of integer division (a + 1) / (b + 5)
I0:	BNEZ r1, compensation	
next:	...	
	Trace 3	J next **Compensation 3**

Question 4 (6 points)

The optimized trace below assumes `I0` is taken slightly less often than not taken, so also speculatively loads `*(p + b)`.

I5:	ADD r4, r1, r3	compensation:
I6':	LD.S r4, 0(r4)	CHK.S r4, exception
I1:	ADDI r3, r3, 5	SUBI r2, r2, 1
I2:	ADDI r2, r2, 1	ADD r2, r2, r4
I0:	BNEZ r1, compensation	
I3:	DIV r2, r2, r3	
next:	...	
	Trace 4	J next **Compensation 4**

Part C: Transactional Memory (25 points)

In this part you will analyze the operation of different hardware TM (HTM) designs, and the concurrency they achieve for different transaction schedules on a 2-core system. 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.

Consider two HTM designs:

- **Eager & Pessimistic HTM** uses eager version management and pessimistic conflict detection. For every transactional load, the memory system checks whether this load reads an address in the write set of any other transaction, and declares a conflict if so. For every transactional store, the memory system checks whether this store writes an address in the read set or write set of any other transaction, and declares a conflict if so. Upon a conflict, the transaction receiving an invalidation or downgrade aborts, i.e. the *requester wins*.

- **Lazy & Optimistic HTM** 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. Analogous to pessimistic requester-wins, the *committer wins*.

The system runs a program consisting of the following two transactions.

Transaction X
`Begin`
`Read A`
`Write A`
`Read B`
`End`

Transaction Y
`Begin`
`Read A`
`Read B`
`Write B`
`End`

In the following questions, for timing, assume conflict detection and coherence happen in the same cycle a memory access executes.

Question 1 (6 points)

Suppose transaction X starts at cycle 0 and transaction Y starts at cycle 5, and they would produce the following schedule.

Cycle	0	5	10	15	20	25	30	35	40	45
Transaction X	`Begin`		`Rd A`		`Wr A`		`Rd B`		`End`	
Transaction Y		`Begin`		`Rd A`		`Rd B`		`Wr B`		`End`

a) In the absence of conflict detection (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) Given the two mentioned HTM designs, indicate in the following table at what cycle a conflict is detected, if any, and which transaction aborts (or neither).

	Conflict cycle	**Aborted Transaction (X, Y, or Neither)**
Eager & Pessimistic	**20**	**Y**
Lazy & Optimistic	**40**	**Y**

Question 2 (6 points)

Suppose transaction X starts at cycle 0 and transaction Y starts at cycle 5, and they would produce the following schedule.

Cycle	0	5	10	15	20	25	30	35	40	45	50	55
Transaction X	Begin		Rd A				Wr A			Rd B		End
Transaction Y		Begin		Rd A		Rd B		Wr B	End			

a) In the absence of conflict detection, 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) Given the two mentioned HTM designs, indicate in the following table at what cycle a conflict is detected, if any, and which transaction aborts (or neither).

	Conflict cycle	**Aborted Transaction (X, Y, or Neither)**
Eager & Pessimistic	**30**	**Y**
Lazy & Optimistic		**Neither**

Question 3 (6 points)

Suppose transaction X starts at cycle 0 and transaction Y starts at cycle 5, and they would produce the following schedule.

Cycle	0	5	10	15	20	25	30	35	40	45	50	55
Transaction X	Begin		Rd A	Wr A				Rd B			End	
Transaction Y		Begin			Rd A		Rd B			Wr B		End

a) In the absence of conflict detection, 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) Given the two mentioned HTM designs, indicate in the following table at what cycle a conflict is detected, if any, and which transaction aborts (or neither).

	Conflict cycle	**Aborted Transaction (X, Y, or Neither)**
Eager & Pessimistic	**20**	**X**
Lazy & Optimistic	**50**	**Y**

Question 4 (7 points)

Consider a different program consisting of the following transactions:

Transaction W
Begin
Write A
Read A
Read B
Write B
End

Transaction Z
Begin
Read A
Read B
Write B
Read B
End

Suppose transaction W starts at cycle 0 and transaction Z starts at cycle 5, and they would produce the following schedule.

Cycle	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70
Transaction W	Beg.			Wr A		Rd A				Rd B		Wr B		End	
Transaction Z		Beg.	Rd A				Rd B		Wr B		Rd B				End

a) In the absence of conflict detection, 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".

W before Z **Z before W** **Not serializable**

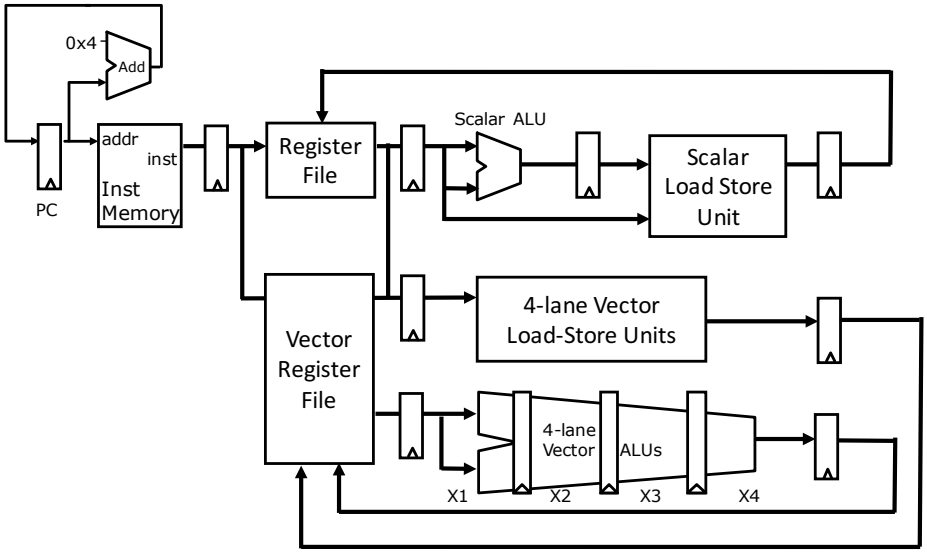
b) Given the two mentioned HTM designs, indicate in the following table at what cycle a conflict is detected, if any, and which transaction aborts (or neither).

	Conflict cycle	**Aborted Transaction (W, Z, or Neither)**
Eager & Pessimistic	**15**	**Z**
Lazy & Optimistic	**65**	**Z**

Part D: Vector vs. GPU Access Patterns (24 points)

In this part you will consider the performance effect of memory access patterns for vector processors and GPUs. The vector processor has the following features:

- Single-issue, in-order execution.
- Scalar instructions execute on a 5-stage, fully-bypassed pipeline.
- 32 4-byte floating-point vector registers, **16 elements per vector register**.
- **Four** vector lanes, with one ALU and one load-store unit per lane. Both units are fully-pipelined and can process vector elements from independent instructions.
- No support for vector chaining (a vector instruction stalls until all the elements of its source operands are available in the vector register file).
- The ALUs have a 4-cycle latency (3 for FP add/mul and 1 for writeback).
- The vector memory system has no cache and consists of 16 banks, with 4-byte word interleaving (`0x0` maps to bank 0, `0x4` to bank 1, etc.). Memory access latency is 16 cycles with a 2-cycle bank busy time (additional cycles between accesses to the same bank). A vector lane's load-store unit stalls if its required bank is busy.



The processor can issue a single (scalar or vector) instruction per cycle. Once it issues, a vector instruction uses the lanes' ALUs or load-store units for as many consecutive cycles as needed to produce all of its results. The vector register file has enough ports to keep the vector ALUs and load-store units fully utilized. The processor implements the MIPS ISA plus the following vector instructions:

Instruction	Meaning
`SETVLR Rs`	Set vector length register (VLR) to the value in Rs
`LV Vt, Rs, Stride`	Load vector register Vt starting at address in Rs, with stride immediate
`SV Vt, Rs, Stride`	Store vector register Vt starting at address in Rs, with stride immediate
`ADD.VV Vd, Vs, Vt`	Add elements in Vs, Vt, and store result in Vd

Question 1 (6 points)

The following C code sums all rows of `A`, a 16 x 16-element array of 4-byte floats, into array `Q`.

```
#define N (16)
float A[N * N], Q[N];
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    Q[j] += A[(i * N) + j];
```

Consider its vectorized code below that stores the `Q` array in a vector register, and loads `A` *with a memory access stride of 1*.

```
      ;; Initial values:
      ;;  r1 := &A[0], r2 := &A[N * N] (first entry after array A)
      ;;  r3 := &Q[0]
      ;;  v1 holds Q array, initially zero.

      ADDI    r20, r0, 16    ;; set r20 to 16
      SETVLR  r20            ;; use all 16 vector elements
loop: LV      v2, r1, 1      ;; access stride of 1
      ADD.VV  v1, v1, v2
      ADDI    r1, r1, 64     ;; point to the next row
      BNE     r1, r2, loop
      SV      v1, r3, 1      ;; access stride of 1
```

In steady state, if the `LV` instruction issues at cycle *X*, when does the `ADD.VV` instruction issue? Assume `&A[0]` maps to bank 0, `&A[1]` maps to bank 1, …, `&A[16]` maps to bank 0, `&A[17]` maps to bank 1, etc.

Instruction	Banks accessed	Cycle issued
`LV v2`	0-3	X
	4-7	X+1
	8-11	X+2
	12-15	X+3
`ADD.VV v1`	N/A	X + 19 = X + 3 + 16

There are no bank conflicts so bank busy time does not apply. The last set of four loads issues at X + 3. However the question underspecifies when writeback occurs relative to the 16-cycle load latency. Therefore X + 19, X + 20, and even X + 21 could be accepted.

Question 2 (6 points)

Now consider a different program that sums elements of `A` with stride 16 into array `T`, as shown below. You can think of this as summing the "columns" of `A` into `T`.

```
#define N (16)
float A[N * N], T[N];
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    T[j] += A[i + (j * N)];
```

The vectorized code below now loads `A` *with a memory access stride of 16*.

```
      ;; Initial values:
      ;;  r1 := &A[0], r2 := &A[N] (first entry after first row of A)
      ;;  r3 := &T[0]
      ;;  v1 holds T array, initially zero.

      ADDI    r20, r0, 16   ;; set r20 to 16
      SETVLR  r20           ;; use all 16 vector elements
loop: LV      v2, r1, 16    ;; A[i], A[i+16], A[i+32], ...
      ADD.VV  v1, v1, v2
      ADDI    r1, r1, 4     ;; point to the next column
      BNE     r1, r2, loop
      SV      v1, r3, 1     ;; access stride of 1
```

In steady state, if the `LV` instruction issues at cycle *X*, when does the `ADD.VV` instruction issue? Assume `&A[0]` maps to bank 0, `&A[1]` maps to bank 1, …, `&A[16]` maps to bank 0, `&A[17]` maps to bank 1, etc.

Instruction	Banks accessed	Cycle issued
`LV v2`	0	X
	0	X + 1 * 2 (for bank busy time)
	0	X + 2 * 2
	…	
	0	X + 15 * 2
`ADD.VV v1`	N/A	X + 46 = X + 15 * 2 + 16

All bank accesses conflict so each of the 16 loads (following the first) suffers a busy bank penalty of 2 cycles. The last load issues at X + 15 * 2. However the question underspecifies when writeback occurs relative to the 16-cycle load latency, so X + 46 and X + 47 were accepted. X + 48 was accepted as long as it was not derived from assuming the first load suffers a bank busy penalty.

You now code these sequential and strided sums to run on a GPU with the following features:

- 16 threads per warp that share the same PC and thus execute the same instruction in lockstep.
- Each of the 16 lanes has one ALU and one load-store unit.
- A 64 KB cache is shared among all threads in a warp. Cache lines are 64 bytes. *If all threads in a warp hit in the cache, the memory access latency is low for all threads*. If any thread in the warp misses in the cache, all threads in the warp stall for the long access to memory, even in some hit (this is because all threads in the warp execute in lockstep). The cache can process up to 16 misses in parallel.

For the following questions, *assume the cache is initially empty.* Assume the base address of the 16x16 array `A` is `0x1000`, i.e. `&A[0] = 0x1000`.

Question 3 (6 points)

You translate the code in question 1, which performs sequential vector accesses, to run on a GPU. Each of the 16 threads of a warp executes a single iteration of the inner loop. Together, the threads of a warp load 16 contiguous elements of `A` at a time.

Qualitatively compare the memory access latencies of the loads in the first and second iterations of the outer loop. Are the latencies equal or is one larger than the other? Why?

The latencies are equal, because they both miss.

All threads of a warp (inner loop iteration) load an address from the same cache line (16 threads * 4 bytes / thread). Because the cache is empty, the requests for the first iteration miss together in lockstep. In the next outer loop iteration, all threads of a warp access a different cache line. They miss together, and so on. There is no reuse when all threads of a warp access the same line.

Question 4 (6 points)

You translate the code in question 2, which performs 16-element strided vector accesses, to run on a GPU. Each of the 16 threads of a warp executes a single iteration of the inner loop. Together, the threads of a warp load 16 elements of `A` that are 16 locations away at a time.

Qualitatively compare the memory access latencies of the loads in the first and second iterations of the outer loop. Are the latencies equal or is one larger than the other? Why?

The latencies of the second iteration loads are less than the first iteration loads, because the former all hit together in lockstep.

All threads of a warp (inner loop iteration) load an address from a different cache line (stride of 16 elements is 64 bytes). Because the cache is empty, the requests for the first iteration miss together in lockstep. In the next outer loop iteration, each thread loads the next element contiguous to its first load. The first group load brought all 16 cache lines into the cache, so the second group load all hit together in lockstep. Each thread experiences 1 miss and 15 hits due to spatial locality.

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.

Extra VLIW Instruction Table

Use this as scratch space or if you need a new one to answer one of the questions in Part A.

Label	Memory Unit	Memory Unit	ALU/Branch	ALU/Branch