Computer Architecture and Engineering
CS152 Quiz #4
April 11th, 2016
Professor George Michelogiannakis

Name:_____<ANSWER KEY>_____

This is a closed book, closed notes exam.
80 Minutes
21 pages

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.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

Writing name on each sheet	_____	1 Point
Question 1	_____	30 Points
Question 2	_____	30 Points
Question 3	_____	23 Points
Question 4	_____	16 Points
TOTAL	_____	100 Points

Question 1: VLIW Machines [30 points]

In this question, we will consider the execution of the following code segment on a VLIW processor.

```
loop:
  flw  f1, 0(x1)
  lw   x9, 0(x2)
  fmul f3, f1, f1
  add  x7, x5, x7
  sw   x7, 0(x1)
  fsw  f3, 0(x2)
  addi x1, x1, 4
  add  x2, x2, x9
  bne  x2, x5, loop
```

This code will run on a VLIW machine with the following instructions format:

Int Op	Branch	Mem Op	FP or Int Op	FP Add	FP Mul

Our machine has six execution units (in order from left to right in the instruction above). All execution units are fully pipelined and latch their operands in the first stage.

- One integer unit, latency two cycles.
- One branch unit, latency one cycle.
- One memory unit, latency three cycles, each unit can perform both loads and stores. You can assume a 100% cache hit rate (i.e., no variability).
- One functional unit that can deal with integer or floating point operations of any kind (additions and multiplications). Latency five cycles.
- One floating point add unit. Latency three cycles.
- One floating point multiply unit. Latency four cycles.

This machine has no interlocks. All register values are read at the start of the instruction before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction). Functional units write the register file at the end of their last pipeline stage, and there are no data forwarding or stalls. I.e., a functional unit that requires 4 cycles and starts an operation in cycle 1 will have its result be visible at the beginning of cycle 5 (writes at the end of cycle 4).

Q1.A Scheduling VLIW Code, Naïve scheduling [6 points]
Schedule operations into VLIW instructions in the following table. Show only one iteration of the loop. Schedule the code efficiently (try to use the least number of cycles), but do not use software pipelining or loop unrolling. You don't need to write in NOPs.

Inst	Int Op	Branch	Mem Op	FP/Int Op	FP Add	FP Mul
1			flw f1, 0(x1)			
2	add x7, x5, x7		lw x9, 0(x2)			
3	addi x1, x1, 4					
4			sw x7, 0(x1)			fmul f3, f1, f1
5						
6						
7	add x2, x2, x9					
8			fsw f3, 0(x2)			
9		bne x2, x5, loop				
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						

Also: What is the resulting throughput of the code in "floating-point operations per cycle"? Don't count flw and fsw as floating-point operations.

1/9

Q1.B Scheduling VLIW Code, Software pipelining [11 points]

Rewrite the assembly code to leverage software pipelining. Do not loop unroll. Schedule VLIW instructions in the following table. You should show the loop prologue and epilogue in addition to the body. Use a clear method to distinguish between instructions of different loops (iterations). You can use a different ink color, underlining, a number in parenthesis, or anything else that is clear.

The below only shows two loops but a more complete solution has more. In that case, the body contains 9 instructions.

Inst	Int Op	Branch	Mem Op	FP/Int Op	FP Add	FP Mul
1	add x7, x5, x7		flw f1, 0(x1)			
2			lw x9, 0(x2)			
3	add x7, x5, x7			addi x1, x1, 4		
4			sw x7, 0(x1)			fmul f3, f1, f1
5			flw f1, 0(x1)	addi x1, x1, 4		
6						
7			lw x9, 0(x2)			
8	add x2, x2, x9		fsw f3, 0(x2)			fmul f3, f1, f1
9		bne x2, x5, loop				
10						
11						
12	add x2, x2, x9		fsw f3, 0(x2)			
13		bne x2, x5, loop				
14						
15						
16						
17						
18						
19						
20						
21						

Also: What is the resulting throughput of the code in "floating-point operations per cycle"?

2/13

Q1.C More Aggressive Loop Unrolling [3 points]
If we were to unroll the loop to four times, could this be done efficiently in this processor? If you could add a resource such as a functional unit, what would it be?

We can't make the body smaller because the multiply unit is four cycles. The bottleneck remains the memory unit. Unrolling to four times would stress the one memory functional unit we have eve more. We should add at least one more.

Q1.D Cache Misses [3 points]
We made the assumption that loads and stores are predictable because we always hit in the cache. This is unrealistic. How can a VLIW processor such as the one we describe in this question respond to a cache miss?

It would have to stall until the cache miss is serviced and re-execute the same instruction. Stall and abort (with re-execute) are two ways of dealing with a miss.

Q1.E Variable Latency Functional Units [3 points]
In the processor of this question, assume that the floating point or integer functional unit has a variable latency depending on if it executes a functional point or integer operation. Ignoring structural hazards in the writeback stage, what problem does this create?

Since there is no hazard resolution in the hardware, the VLIW compiler needs to know the latency of each functional unit to know when each instruction will write to the register file, and thus what version of the register (old or new) subsequent instructions will see.

Q1.F Branches [4 points]
Assume that the branch resolution unit in this processor has a two-cycle latency, instead of one cycle as originally stated. Without prediction and without a branch delay slot, can we schedule an instruction the cycle after the branch (i.e., if the branch is in cycle 13, the question is for cycle 14)? With branch prediction, what is the challenge in case of mispeculation and what do we need to do about it in this processor?

No because we have not resolved the branch, therefore we do not know what the next instruction is. If we have branch prediction the challenge is not allowing instructions that were fetched as part of the mispeculation to change the state of the processor. In this processor, since all functional units have a latency of two cycles or more, it suffices to kill the mispredicted operations from the appropriate pipeline stages in the functional units. No operation would have changed the state of the processor by the time we resolve the branch.

Question 2: Vector Machines [30 points]

In this question, we will consider the following code written in C:

```
for (int i=0; i<N; i++)
  A[i] = A[i] + B[i]*C[i]
```

You can assume VLMAX = 32 (vector registers contain 32 elements). Also, unless otherwise specified, N = 32. The base address of array A is contained in scalar register rA, B in rB, and C in rC. For the rest of the machine, assume the following:

- 16 lanes
- one ALU per lane: 2 cycle latency
- one LD/ST unit per lane: 2 cycle latency. Fully pipelined that latches operands at the first pipeline stage.
- all functional units have dedicated read/write ports into the vector register file
- no dead time
- no support for chaining
- scalar instructions execute separately on a control processor (5-stage, in-order)

Q2.A Vector Memory Memory and Register [6 points]

Write the vector assembly code for the above C code first in a vector memory-memory code, and then vector register code. You can use as many registers as you need, as well as temporary memory locations (e.g., rT). Remember to set VLR. You can use any pseudo-assembly language that makes sense (“add”, “mul”, etc).

Vector memory-memory

```
LI VLR 32
Mul rT, rB, rC
Add rA, rA, rT
```

Vector register

```
LI VLR 32
LV v1 0(rB)
LV v2 0(rC)
Mul v3, v2, v1
LV v4 0(rA)
Add v5, v3, v4
SV v5 0(rA)
```

Q2.B Compare [2 points]
Which of the two versions of the code from Q2.A is more stressful to memory (causes more memory accesses)? Also, which of the two versions provides more opportunities to exploit instruction-level parallelism?

Vector memory memory causes 6 memory accesses whereas register causes 4. Vector register provides more opportunities to exploit ILP because it has more independent instructions that can be re-ordered.

Q2.C Lane Tradeoffs [3 points]

Suppose we want to add two vector registers (add v1, v2, v3), followed by another addition to different registers (add v4, v5, v6). The next instruction after that uses a different functional unit. VLR=VLRMAX=32. What would you choose between an ALU with 8 lanes and 2 cycles dead time, and an ALU with 16 lanes and 8 cycles dead time?

The first case will need (32/8)*2 + 2 = 10 cycles. The second (32/16)*2 + 4 + 8 = 12 cycles. So the first option is faster.

Q2.D Vectorize [7 points]

How can the following code be vectorized? You can assume N=VLMAX. Clearly state any assumptions that you used for your answer for what the architecture provides, such as specialized instructions, registers, etc.

```
for (int i=0; i<N; i++)
     if (A[i+1])
          A[i] = A[i] + B[C[i]]
```

The indirection is not the issue here because we can load C to a register and use a load indirect instruction ("LVI") to load B, as long as the architecture supports that. For the if condition, we have to rely on a special instruction and predicate registers. We first load &(A[i]) into a register, and then &(A[i+1]) to a different register. We use the special instruction to set the predicate registers of A based on the values of A[i+i]. Then we perform the addition.

Q2.E Scheduling Vector Code, No Chaining [8 points]

Complete the pipeline diagram of the baseline vector processor running the following

code. Assume no chaining.

```
LI VLR 32
LV v1 0(rA)
LV v2 0(rB)
LV v3 0(rC)
Mul v3, v3, v1
Add v3, v3, v2
LV v4 0(rD)
Add v4, v4, v3
Add v4, v4, v4
saddi r1, r1, 8 // Scalar add
SV v4 0(rA)
```

The following supplementary information explains the diagram:

- Scalar instructions execute in 5 cycles: fetch (**F**), decode (**D**), execute (**X**), memory (**M**), and writeback (**W**).
- Vector instructions should write back in program order.
- A vector instruction is also fetched (**F**) and decoded (**D**). Then, it stalls (—) until its required vector functional unit is available.
- With no chaining, a dependent vector instruction stalls until the previous instruction finishes writing back all of its elements.
- A vector instruction is pipelined across all the lanes in parallel.
- For each element, the operands are read (**R**) from the vector register file, the operation executes on the load/store unit (**M**) or the ALU (**X**), and the result is written back (**W**) to the vector register file.
- A stalled vector instruction does not block a scalar instruction from executing.

Name

Inst	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	41
LI	F	D	X	M	W																																				
LV		F	D	R	M1	M2	W																																		
					R	M1	M2	W																																	
LV			F	D	-	R	M1	M2	W																																
							R	M1	M2	W																															
LV				F	D	-	-	R	M1	M2	W																														
									R	M1	M2	W																													
MUL					F	D	-	-	-	-	-	-	R	X1	X2	W																								
														R	X1	X2	W																							
ADD						F	D	-	-	-	-	-	-	-	-	-	-	R	X1	X	W																			
																			R	X1	X2	W																	
LV							F	D	-	-	-	-	-	-	-	-	-	-	-	R	M	M	W																	
																					R	M	M	W															
ADD								F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M	M	W											
																										R	M	M	W										
ADD									F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	X1	X2	W						
																															R	X1	X2	W						
saddi											F	D	X	M	W																								
SV												F	D	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	R	M	M	W			

Name _____

Q2.F With Chaining [4 points]
In the same code, how many fewer cycles would be required if we added chaining?

In a RAW hazard, chaining lets the instruction that writes forward to the instruction that reads after the functional unit produces the result but before it writes to the register file. The above code, there are 5 RAW hazards between vector instructions: LV (v3) -> mul -> add -> add -> add -> SV. For each of those, chaining would save 2 cycles (the first R of the second instruction lines up with the first W of the first instruction). So 10 cycles total.

Name _____

Question 3: Multithreading [23 points]

In this question, we will consider multithreading executing in a single-issue, in-order, multithreaded processor that supports a variable number of threads (as individual questions specify). The code we will be working with is:

```
loop:
  lw x1, 0(x3)
  lw x2, 0(x4)
  add x1, x2, x1
  sw x1, 0(x3)
  addi x3, x3, 4
  addi x4, x4, 4
  bne x1, x2, loop
```

The processor has the following functional units:

- Memory operation (load/store), 4 cycles latency (fully pipelined).
- integer add, 1 cycle latency.
- Floating point add and multiply unit, 3 cycles latency (fully pipelined).
- branch, 1 cycle latency.

The processor has a cache which has a 100% hit rate (4 cycle latency is with a cache hit). If an instruction cannot be issued due to a data dependency, the processor stalls. We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches. Throughout this question there is no communication between threads, and the loads and stores of each thread are to different addresses.

Name _____

Q3.A Scheduling [5 points]

First consider that the processor can handle two threads at the same time. You have a choice between round-robin scheduling where if the thread that was going to be selected in not ready a bubble is inserted instead, and dynamic hardware scheduling where the processor at every cycle picks a thread that is ready (in a round-robin manner if both threads are ready).

With the code and assumptions given, will this make a different in performance (i.e., how quickly each thread completes a pre-defined large number of iterations)?

Does your answer change if we have an ideal cache and with a 10% chance a load/store misses at the cache at imposes a 50-cycle penalty?

With the assumptions given, each thread has the same hazards and goes through the same instructions with the same latencies, so when one stalls (e.g., due to RAW) the other stalls too. So the two scheduling policies will not make a difference. However, if we now risk having a cache miss, one thread may miss in the cache and the other may not. In that case, the ideal scheduler will provide more cycles to the thread that is ready. With round-robin, the time for one thread to complete does not depend on other threads, except for cache misses and communication.

Name _____

Q3.C Number of Threads [10 points]

Now lets consider the same single-issue in-order processor as originally described, with round-robin scheduling and a 100% cache hit rate, but a variable number of threads that it can support. What is the minimum number of threads needed to fully utilize the processor, assuming you reschedule the assembly as necessary to minimize the number of threads? Also, assume an infinite number of registers.
With the same instruction sequence that you used, what is the number of threads if all adds are floating point adds.

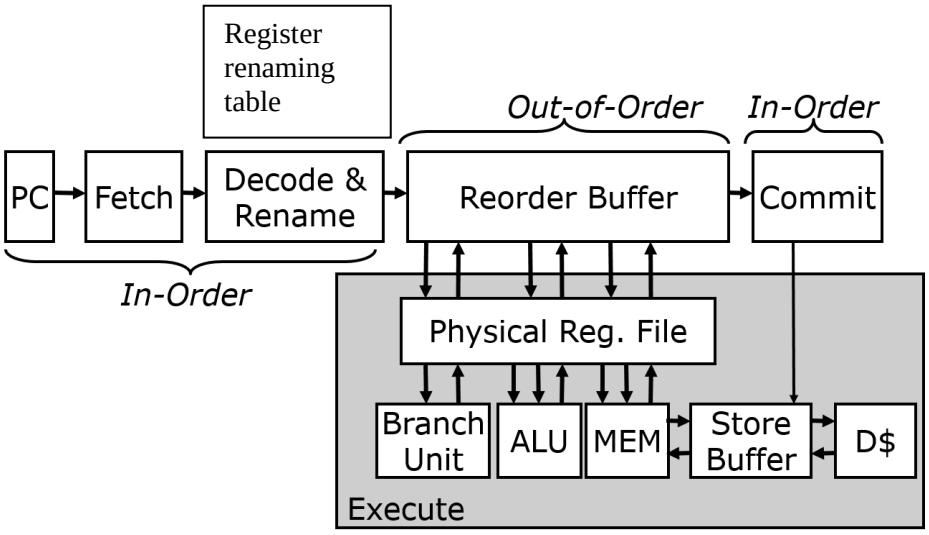
```
loop:
   lw x1, 0(x3)
   lw x2, 0(x4)
   addi x4, x4, 4  // This instruction was reordered
   add x1, x2, x1
   sw x1, 0(x3)
   addi x3, x3, 4
   bne x1, x2, loop
```

The first load is pushed to the memory in cycle 1. It writes back the result at the end of cycle 4. In between, instructions 2 and 3 issued and are able to execute, but in cycle 4 there was a bubble because instruction 4 issued but can't execute. With the latencies and instructions provided, there are no more bubbles. Therefore, two threads are needed to keep this processor busy.

If adds are floating point adds (only one add will be affected), their latency is now 3 cycles instead of 1. Now there are two bubbles between instructions 4 and 5 (RAW). In the first bubble, instruction 5 issues and in the next bubble instruction 6 issues. Neither can execute. Instruction 6 can't execute because of the WAR hazard. To cover these two bubbles we need three threads.

Name _____

Q3.C Pipeline [7 points]



Above you can see a diagram of an in-order issue, out-of-order execute processor with a physical register file that performs register renaming. This diagram is for a single-threaded processor. If we wanted to make this processor two-way multithreaded with round-robin scheduling, which block(s) do we *have to* duplicate, and which block(s) we *should* probably make larger in order to avoid having them be a performance bottleneck? Do not make the processor superscalar. Explain why for each block.

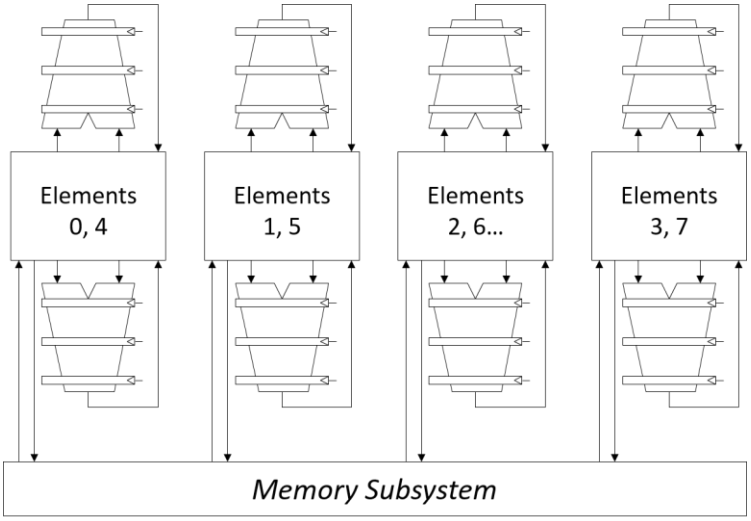
We have to duplicate the PC register to two PC registers, one for each thread. We also have to duplicate the register renaming table because each thread has its own set of architectural registers. Those are the only blocks we have to duplicate to ensure functional correctness in this processor. We should make the physical register file larger because now we have two threads looking to store data in registers. We should also make the reorder buffer (ROB) larger because now we have instructions from two threads that are independent. Therefore, we need a larger ROB to find instruction-level parallelism. Finally, we can also use a deeper store buffer because there will be more pending (completed but not committed) stores, and more functional units.

Name _____

Question 4: Potpourri [16 points]

Q4.A Vector Processors [5 points]

Assume a register vector processor (no vector memory-memory operations) with vector registers that can hold 8 elements and has 4 lanes in its integer functional unit (shown below). The alternative processor is a similar vector processor that has vector registers that can hold 16 elements but only 2 lanes in its integer functional unit. Both architectures have an infinite number of registers. If we want to add 64 array elements (code shown below), which of the two will complete faster? Assume loads and stores take one cycle, and load instructions have to be before adds, and adds have to be before stores.




```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

The first processor needs 64/8 = 8 loads and 8 stores, so 16 cycles. The addition takes 64/4 =16 cycles. So 32 cycles overall. The second processor needs 64/16 = 8 (4 loads and 4 stores). The addition takes 64/2 = 32 cycles. So the first processor is faster.

Name _____

Q4.B Multithreaded Processors [4 points]

Assume a multithreaded processor with a cache, 64 integer registers, and in-order issue and commit. We have four threads, and we want to compare statically interleaving them cycle-by-cycle by having thread 1 issue an instruction in cycle 1, 5, 9, etc (option 1), against option 2 where we statically execute 512 cycles of thread 1, then 512 of thread 2, etc. Name one advantage and one disadvantage of option 2 compared to option 1.

One disadvantage is that if thread 1 blocks, the rest of its cycles before the next thread executes are wasted. One advantage is that a thread running for a number of cycles consecutively warms up the cache and brings its own data in, before other threads have the change to evict them due to capacity or conflict. Other answers are also possible.

Q4.C VLIW Processors [3 points]

Both VLIW and out-of-order superscalar processors exploit instruction-level parallelism. What is the motivation to choose VLIW processors instead of out-of-order superscalar? How does this affect hardware and compiler complexity?

VLIW can issue multiple operations per cycle with simple hardware. Out-of-order superscalar need complex control logic. Independent operations are scheduled statically by the compiler into the same VLIW instruction. The compiler's job becomes more complex.

Name _____

Q4.D Multithreaded with VLIW and Vector [4 points]
Suppose we have two threads (which may have different instructions), and we want to merge their instructions into one (mega-)thread. We have the option of running this (mega-)thread on a VLIW and a vector processor. Give one reason why we would choose VLIW (and what assumption or condition that depends on), versus a reason we would choose the vector processor (and what assumption or condition that depends on).

This has to do with how different the instruction streams are. The advantage of VLIW is that it provides more flexibility such that if threads have different operations each cycle, VLIW can mix different kinds of operations in the same cycle. On the other hand, if threads are perfectly synchronized (unlikely but possible), the vector multithreaded processor will increase throughput because it can do more operations of the same kind in the same cycle.

Appendix

This is the code and other information for question 1. You may detach the appendix.

```
loop:
  flw  f1, 0(x1)
  lw   x9, 0(x2)
  fmul f3, f1, f1
  add  x7, x5, x7
  sw   x7, 0(x1)
  fsw  f3, 0(x2)
  addi x1, x1, 4
  add  x2, x2, x9
  bne  x2, x5, loop
```

This code will run on a VLIW machine with the following instructions format:

Int Op	Branch	Mem Op	FP or Int Op	FP Add	FP Mul

Our machine has six execution units (in order from left to right in the instruction above). All execution units are fully pipelined and latch their operands in the first stage.

- One integer unit, latency two cycles.
- One branch unit, latency one cycle.
- One memory unit, latency three cycles, each unit can perform both loads and stores. You can assume a 100% cache hit rate (i.e., no variability).
- One functional unit that can deal with integer or floating point operations of any kind (additions and multiplications). Latency five cycles.
- One floating point add unit. Latency three cycles.
- One floating point multiply unit. Latency four cycles.

This machine has no interlocks. All register values are read at the start of the instruction before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction). Functional units write the register file at the end of their last pipeline stage, and there are no data forwarding or stalls. I.e., a functional unit that requires 4 cycles and starts an operation in cycle 1 will have its result be visible at the beginning of cycle 6 (writes at the end of cycle 5).

Name _____

Computer Architecture and Engineering

CS152 Quiz #5

April 27th, 2016

Professor George Michelogiannakis

Name: _____<ANSWER KEY>_____

This is a closed book, closed notes exam. 80 Minutes

19 pages

Notes:

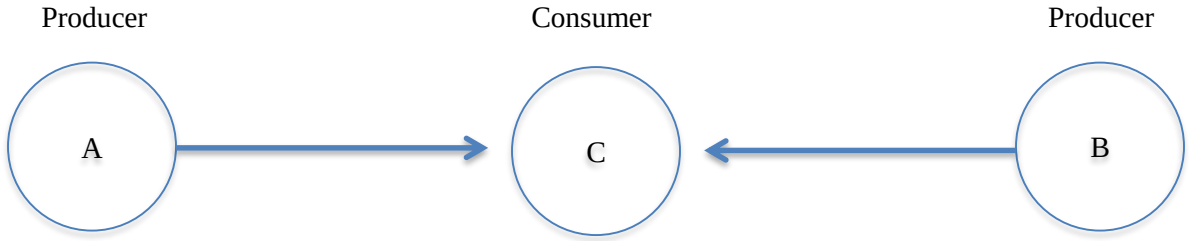
- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not taken the quiz. If you have inadvertently been exposed to a quiz prior to taking it, you must tell the instructor or TA.
- You will get no credit for selecting multiple-choice answers without giving explanations if the instruction ask you to explain your choice.

Writing name on each sheet	_____	1 Point
Question 1	_____	30 Points
Question 2	_____	32 Points
Question 3	_____	21 Points
Question 4	_____	16 Points
Total	_____	100 Points

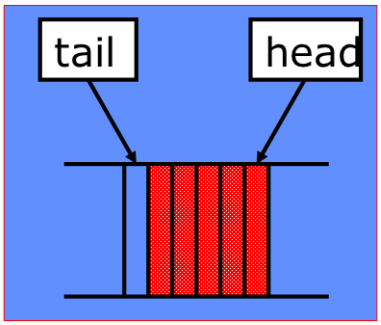
Name _____

Question 1: Memory Models [30 points]

In this question, we will consumer the scenario of **a variable number** producer threads (as questions specify) with one consumer. Producer threads run the same code to generate **two** elements of 8 each bytes for the consumer. The consumer is thread C. An example with two producers:



Communication for each thread is handled by a FIFO. There is **only one FIFO** for all. Producers put the new elements at the location pointed by the tail (in adjacent locations). The consumer reads one element at the location pointed by the head and then increments it.



The code is as follows (you can't change or reorder intructions):

Producer code:

```
(1) Load Rtail, 0(tail)

(2) Store 0(Rtail), x

(3) Rtail  = Rtail  + 1

(4) Store 0(Rtail), y
(5) Rtail  = Rtail  + 1

(6) Store 0(tail), Rtail
```

Consumer code:

```
(1) Load Rhead, 0(head)

spin: (2) Load Rtail, 0(tail)
      (3) if Rhead  == Rtail  goto spin

      (4) Load R, 0(Rhead)

      (5) Rhead  = Rhead  + 1

      (6) Store 0(head), Rhead
      (7) process(R)
```

Name _____

Q1.A Adversary Effects and Fences [6 points]

Lets first assume a system with quiescent consistency, where there are no guarantees of the order in which loads and stores are made visible, even those from the same processor. The only guarantee is provided by adding fences. Also, we have only one producer and one consumer thread.
First explain what can go wrong in the above code without fences using quiescent consistency (i.e., what is the correctness problem). Be specific by providing an example erroneous outcome. Then, describe where you would **add fence(s)** to ensure correctness.

With no guarantees instruction 6 in the producer can happen before 4 or 2 in the producer. There are similar adverse scenarios with the consumer with Rhead and the value that is read. To fix this, we have to add a fence after instruction 5 in the producer, and after 3 in the consumer. Two fences total.

Q1.B Optimization With Fences [5 points]

Still with one producer and one consumer thread and quiescent consistency, re-write the consumer's code such that it consumes **two** elements and uses one fence. Include the fence in your code. Don't write or modify the code for the producer.

```
Load Rhead, 0(head)
Load Rtail, 0(tail)
if Rhead  == Rtail || Rhead == Rtail - 1  goto spin
FENCE
Load R1, 0(Rhead)
Load R2, 1(Rhead)
Rhead  = Rhead  + 2
Store 0(head), Rhead
process(R)
```

Name _____

Q1.C Thread Safety [9 points]

Now we will revert to the original code (ignore your modification in Q1.B). We maintain quiescent consistency, but now we'll have **two** producer threads and one consumer. Further assume that the architecture provides an atomic instruction:

```
Swap (m), R:
        Rt <= M[m];
        M[m] <= R;
        R <= Rt;
```

Rewrite the **producer's** code to ensure correctness.

```
Spin: Load Rtail, 0(tail)
      Addi Rtail2, Rtail, 2
      Swap(tail), Rtail2
       if Rtail2 != Rtail goto spin
      Store 0(Rtail), x
      Store 1(Rtail), y
```

Name _____

Q1.D Optimization [6 points]

In Q1.C, how many memory (or cache) accesses does each failed attempt to enter the critical section produce? Pick **one** of the following alternative atomic operations and describe how you would use it to reduce the number of memory or cache accesses. You do not have to write the code as long as your explanation is clear (but you can if you wish).

```
Fetch&Add (m), R_V, R:
        R <= M[m];
        M[m] <= R + R_V;
```

```
Test&Set (m), R:
        R <= M[m];
        if  R==0 then
        M[m] <= 1;
```

```
Compare&Swap(m), R_t, R_s:
   if (R_t==M[m])
      then  M[m]=R_s;
            R_s=R_t ;
            status <= success;
      else  status <= fail;
```

Each failed attempt causes one memory access for the swap and one for the load, so two total. Fetch and add is the answer here because we can use it to replace the code's original load of Rtail and the two next instructions (instruction 1, 2, and 3). So the code has no extra loads for synchronization with fetch and add.

Name _____

Q1.E Sequential Consistency [4 points]

Now lets assume an architecture that provides sequential consistency. Describe what you need to do to ensure correctness with the original code as described in the beginning of this question and:

(a) One producer and one consumer.

(b) Two producers and one consumer.

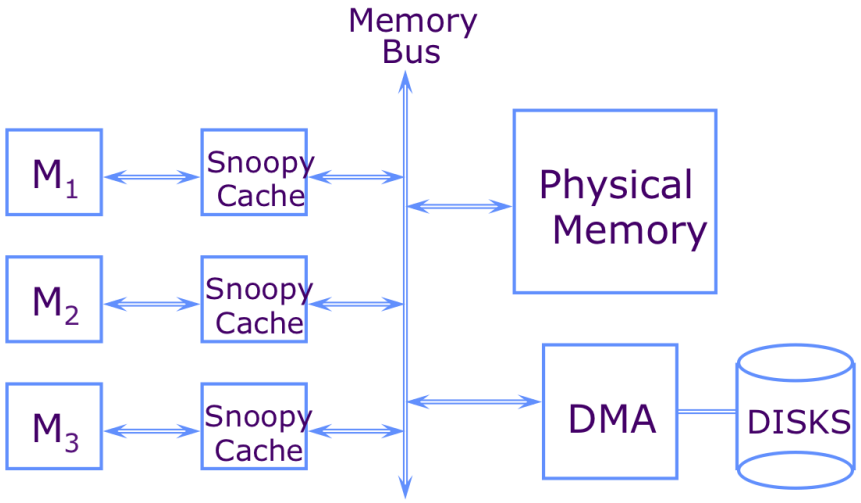
Does the sequential consistency machine need atomic operations?

(a) The sequential consistency machine does not need fenches nor their latency.
(b) The sequential consistency machine does not operate correctly with the original code. It needs locks. Or it can provide correctness without atomic operations, but algorithms for doing so are complex.

Name _____

Question 2: Snoopy Cache Coherence [32 points]

In class we discussed MSI and MESI cache coherence protocols on a bus-based processor. We will assume 3 cores in a processor. Each core has one snoopy write-back cache and is connected to the bus. There is also a memory controller and a DMA engine connected to an array of hard disk drives. The bus has a latency of one cycle.



For this question, we will consider an extended version of MESI, the simplified MOESI protocol. The MOESI protocol has five states for each cache line as follows (states are explained from the viewpoint of a single cache):

- Invalid (I): Cache line is not valid (does not exist).
- Shared (S): A shared cache line may exist in other caches. A line in shared state can be read, but not written to without causing a transition to a new state. The value in the cache is the same as the value in main memory.
- Exclusive (E): The value is the same as the value in main memory. This line is not cached in any other cache (other caches have it as invalid).
- Modified (M): The value in the cache line has been modified. Therefore, it is different than the value in main memory. This line is not in any other cache.
- Owned (O): Cache line is dirty (has been modified). Thus, it is different than the value in main memory. In this state, the cache line exists in other caches. Only one cache can have the line in the owned state, and that contains the most up to date data. Other caches may have the line in the shared state. If the cache line in the owned state is updated (written), the cache must *broadcast* the new value to all other caches.

As a default, if a line is written, the MOESI protocol jumps to the modified (M) state if that line does not exist in other caches. Otherwise, it jumps to the owned (O) state.

The state transition diagram for MESI is provided in the appendix.

Name _____

Q2.A Transitions to the Owned State [7 points]

To make use of the new state, we need to define transitions to and from it. For each of the possible transitions, define if they are allowed. If they are, explain what is the trigger (i.e., a local cache update, remote read, etc), the condition (e.g., other sharers exist), and if there is an action (e.g., writeback). Do not answer for (O) -> (M) in this question.

(I) -> (O): Trigger: local cache writes the cache line. Condition: Other sharers exist. Action: Broadcast up to date value

(O) -> (I): Not allowed. If another sharer writes, it will go to owned and the local cache would transition to shared.

(S) -> (O): Trigger: local cache writes. Condition: Other sharers exist (which is most likely). Action: broadcast up to date value.

(O) -> (S): Trigger: another cache writes. No condition or action.

(E) -> (O): Not allowed. While in E no other cache has a copy so just transition to M instead.

(O) -> (E): Not allowed.

(M) -> (O): Trigger: Another cache reads. Action: Broadcast modified value.

(O) -> (M): Do not answer this.

Name _____

Q2.B Why MOESI [3 points]

Why would we choose MOESI over MESI? Do many loads favor MESI or MOESI? Do many stores many MESI or MOESI?

MOESI has the advantage that one cache can write a line and still allow other caches to be in the shared state. With MESI, each write would invalidate the line in all other caches that would cause future cache misses. Many loads favor MOESI. Many stores favor MESI the owned state causes a broadcast for each store.

Q2.C Owned to Modified [5 points]

Lets focus on the transition from owned (O) to modified (M) in MOESI. If a cache writes a cache line that is in the owned state (O), should the transition be taken or should the cache line remain owned? If the transition from owned (O) to modified (M) is taken, an invalidate is sent to all other caches. To answer, use the example of (a) code with a small number of writes (stores) and many reads and (b) code with a large number of writes and few reads. The same code runs on all cores. Is your answer based on correctness, or is it a matter of improving performance such as by increasing the cache's hit rate or decreasing the number of messages?

Correctness is not violated in either case. The key is that while in owned (O) state, each write to the cache triggers a broadcast to all other caches. The benefit is that other caches can be in shared. In case (a) with few writes, we prefer to stay in owned because the expensive operation (writes) is infrequent, and we have many reads so we prefer to have them stay in shared. In case (b), we prefer to transition to modified (M).

Name _____

Q2.D Sequential Consistency [3 points]

Does cache coherency guarantee sequential consistency? Explain your answer.

It does not. A system with an out-of-order cache and coherency still does not guarantee sequential consistency.

Q2.E Snoopy Bus Avoids Transients [3 points]

The finite state machines for MOESI and MESI have no transient states with the snoopy bus, but they do with directories. What is the property of the bus that lets us avoid transient states?

The bus has to provide atomicity. This means that once a message enters, it reaches all other caches and any replies to that message enter the bus and complete, before any other request can enter the bus.

Name _____

Q2.F Coherency Misses [11 points]

Consider the following code running in **two** cores. For this question we will use the original MESI protocol instead of MOESI. The state diagram for MESI can be found in the appendix.

```
(1) LW  x1, 0(x5)
(2) LW  x2, 0(x6)
(3) SW  x3, 0(x6)
(4) SW  x2, 0(x5)
(5) LW  x1, 0(x6)
```

Do not optimize or re-order the code. Assume the processor guarantees sequential consistency. Also, the addresses in x5 and x6 map to different cache lines.

Assume the following execution sequence:
A.1, A.2, B.1, B.2, A.3, B.3, B.4, A.4, A.5, B.5 (A and B are the two cores)

Fill in the table below with the states of the cache lines at every step. Also, count the number of communication events. A communication event is a message sent from the cache as part of the transition, and is part of certain transitions. For example, shared to owned causes communication to broadcast the new value. Assume invalid to exclusive does not cause communication (i.e., we assume the cache knows there is no other sharer), but invalid to shared does only if another cache has the line as exclusive. The initial state of both cache lines in both caches is invalid.

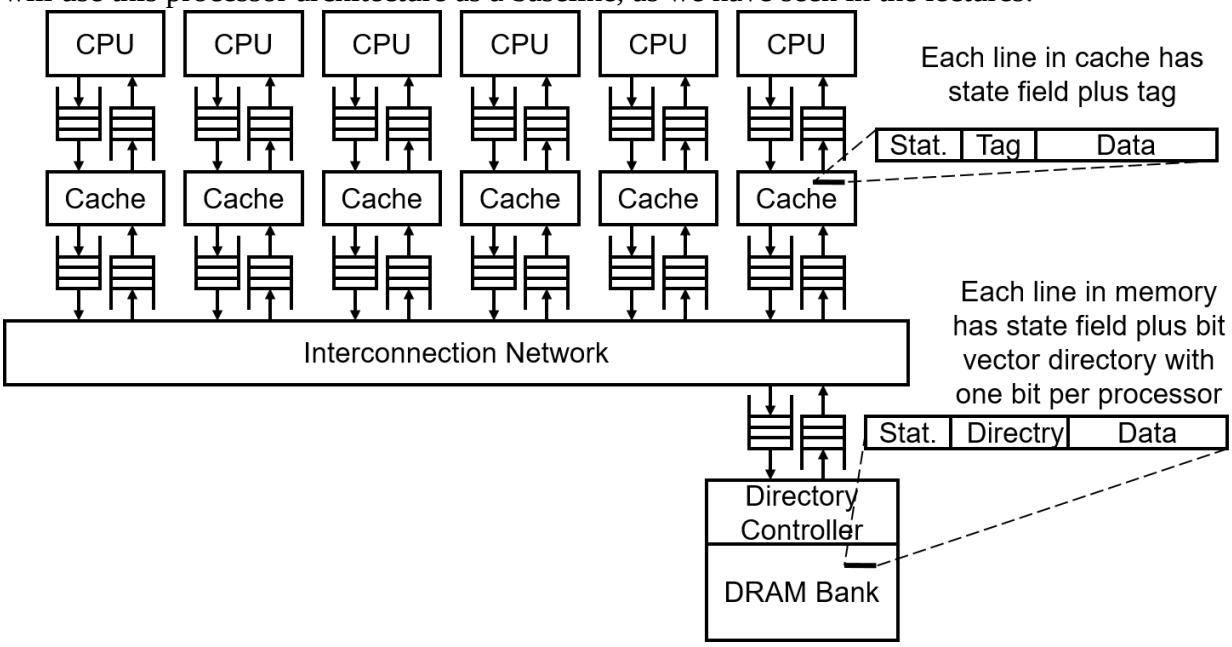
Core: Instruction	State x5 cache line in core A	State x6 cache line in core A	State x5 cache line in core B	State x6 cache line in core B
A: LW x1, 0(x5)	E	I	I	I
A: LW x2, 0(x6)	E	E	I	I
B: LW x1, 0(x5)	S	E	S	I
B: LW x2, 0(x6)	S	S	S	S
A: SW x3, 0(x6)	S	M	S	I
B: SW x3, 0(x6)	S	I	S	M
B: SW x2, 0(x5)	I	I	M	M
A: SW x2, 0(x5)	M	I	I	M
A: LW x1, 0(x6)	M	S	I	S
B: LW x1, 0(x6)	M	S	I	S

7 communication events.

Name _____

Question 3: Directory Protocols [21 points]

Even directory-based coherence is not easy to scale to large core counts. For this question, we will use this processor architecture as a baseline, as we have seen in the lectures:



For simplicity, we assume that there is only one directory. In the above figure, we show 6 cores. However, for this problem we will discuss how things change as we scale up to 2048 cores. Assume the MSI protocol where cache lines can be in 6 states including transients in the caches, and 13 states in the directory.

Q3.A Size of Entries [4 points]

For each cache line, the directory maintains state and a bit vector to record which of the cores have that cache line. For example with 6 cores, a bit vector of "100011" means that cores 1, 5, and 6 have that cache line.
If a cache line has 64 bytes of data, how many bits of the coherence protocol in the directory, and how many bits for coherency in a cache? Answer first for 6 cores and then for 2048 cores. Ignore the tag.

With 6 cores we have $\log_2(6)$ bits in every cache line for every 64 bytes of data. So $\log_2(6)$ bits. The answer is the same for 2048 cores.
For the directory the answer changes. For every 64 bytes of data, we have $\log_2(13) + 6$ bits for 6 cores, and $\log_2(13) + 2048$ bits for 2048 cores.

Name _____

Q3.B Alternatives to Bit Vectors [7 points]

From your answer above, it should be apparent that the bit vector each directory maintains in each cache line quickly becomes a problem as we scale the number of cores. One alternative is to replace the bit vector with a linked list of core IDs. In that case, if cores 2 and 10 share a cache line, the directory contains the numbers "2" and "10" for that cache line. With 2048 cores, each of those numbers needs 11 bits. Assume no other overhead for the linked list.

First determine how many core IDs we can store if we limit the space for the list to 1024 bits. Then, propose a mechanism to deal with the case where we have more cores than your answer sharing the line. For example, if you answer that we can keep up to 10 sharers, how would we deal with 12 sharers? In your answer, state how and when a cache line can revert back to "normal" operation, when the number of sharers drop.

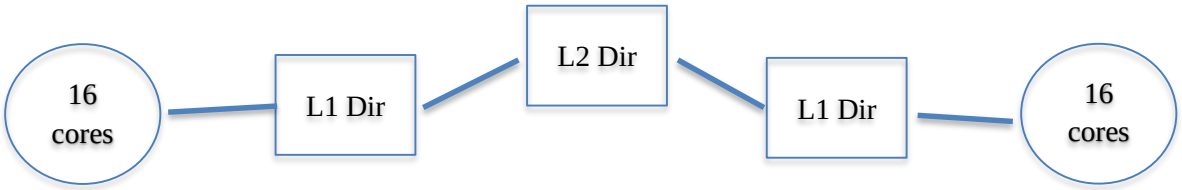
Do not worry about performance in your solution, just correctness. Also, do not add extra directories. Maintain one directory for the entire processor.

We can store 1024 / 11 rounded down = 93 core IDs. If we get more than 93 sharers, we would have to force the directory to broadcast any messages to sharers of that line to all cores, whether they share it or not. That means we no longer have to keep track of what cores share this line, but the expense is that now we have to broadcast until that line is evicted from the directory or a core modifies the line (which forces all other sharers to invalidate so we know there is only one core that has the line at that point).

Name _____

Q3.C More Directories [7 points]

Based on the possible performance losses from solutions that still keep one directory, now we would like to add a *hierarchy* of directories. Hierarchical directories work much like a hierarchy of caches. We will assume only two levels of directories. In the first level, we have one directory every 16 cores (128 directories for 2048 cores). In the second level, we still have a single directory. A core's cache sends a message to its first-level directory. If that cache line is only shared within the group of processors that this first-level directory serves, the request is satisfied and an answer is returned. Otherwise, the request is further propagated to the second-level directory, which then forwards messages to any first-level directories that serve cores which have that cache line. The second-level directory does not send messages to cores directly, and treats groups as a single domain. I.e., is a core in a group shares a line, the second-level directory does not need to know which caches(s) is that group the line is in.



What is the advantage of this design that helps scalability?
Then, at a high level, describe the sequence of events and messages if core A has a line in the shared state and wants to write to it. Core B is in the same 16-core group and has the same line in the shared state. Also, core C has it in shared, but is in a different 16-core group.

The advantage is that first-level directories only maintain sharer state (e.g., the bit vector) for 16 cores. The second-level directory only maintains a bit vector for 128 first-level directories.

In that scenario, core A sends a message to its first-level directory. That directory then sends a message to invalidate core B's copy, and also propagates the message to the second-level directory. That then sends it to the first-level directory that serves core C, which forwards it to core C. The acknowledgment from core C is sent via the same directories to core A's first-level directory which also waits for the acknowledgment from core B. Then, core A's first-level directory sends the final acknowledgment to core A which then transitions that line to the modified state.

Name _____

Q3.D More Levels [3 points]

Name one advantage and one disadvantage of increasing the levels of our directory hierarchy to three from two. Does having a hierarchy of directories affect where we want to place cores that tend to share data (e.g., producer – consumer)?

One advantage: less state required per directory. This makes directories smaller and thus can be clocked faster. Disadvantage: more messages between directories. Having groups motivates placing cores that share data in the same group to reduce communication outside the group.

Name _____

Question 4: Potpourri [16 points]

Q4.A Measure of Performance [5 points]

Suppose we write a parallel program with a critical section. We provide mutual exclusion with a store conditional, as the example below for the consumer, as we have seen in the lectures:

```
try:    Load-reserve R_head, (head)
spin:   Load R_tail, (tail)
        if R_head==R_tail goto spin
        Load R, (R_head)
        R_head = R_head + 1
        Store-conditional (head), R_head
        if (status==fail) goto try
        process(R)
```

Is CPI (clocks per instruction) an accurate measure of performance? Can you propose a more accurate measure of performance for parallel applications?

The problem with CPI in this example (same as with locks) is that a core can be executing instructions but not be making forward progress. In the above example, trying again after failing the store conditional increases the instruction count, but is not helping the code make progress. A more accurate measurement will be completion time. That is, the time is takes for the slowest thread of the same application to complete.

Name _____

Q4.B Sequential Consistency with Out-of-Order Cores [3 points]

Can an out-of-order execute and out-of-order commit core be used in a sequential consistent system? How about an out-of-order execute but in-order commit?

No to the first and yes to the second. The key is what order loads and stores are made visible outside of the core and to the cache. Out-of-order commit cores violate sequential consistency in the order loads and stores arrive to the cache.

Q4.C Misses with Coherence [4 points]

In a uniprocessor system, we previously discussed that keeping the number of lines constant in a cache and increasing the cache line size tends to decrease the number of cache misses. In a cache coherent system executing a parallel program with locks and critical sections, is increasing the cache line size always a win?

No. Cache coherence applies in a cache line granularity. Especially with locks, small 4 or 8 byte values will frequently be accessed by different threads. If those values are different but map to the same cache line, they will cause false sharing coherence misses, caused by the coherence protocol invalidating entire lines.

Name _____

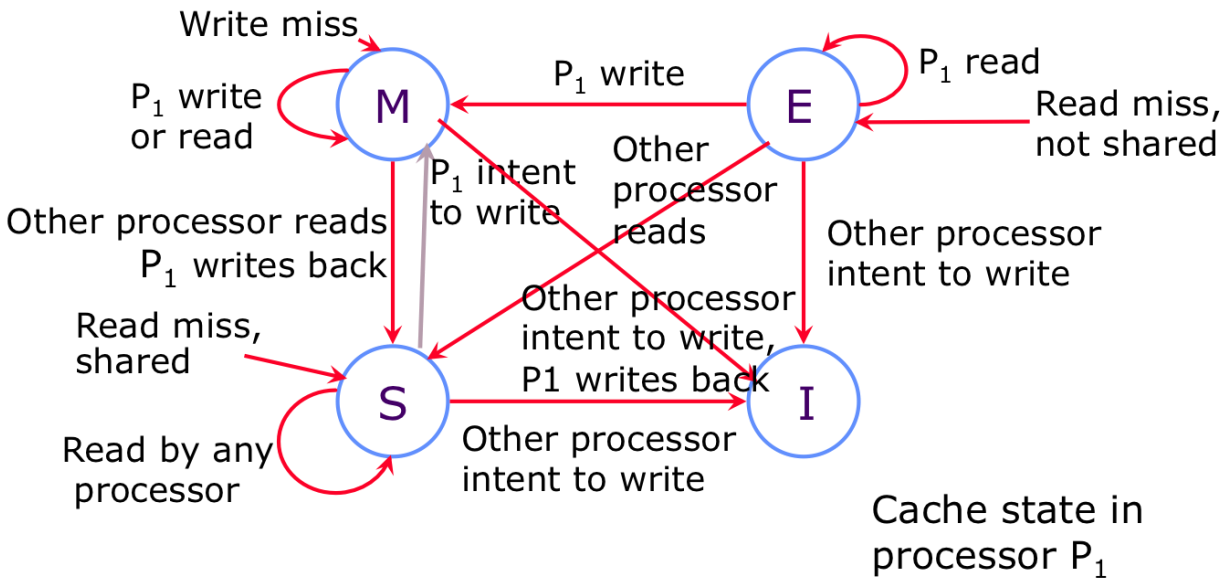
Q4.D Multiple Synchronization Primitives [4 points]

In class and in problem 1 of this quiz, we mentioned multiple synchronization primitives such as test&set, fetch&add, compare&swap, etc. Why do modern ISAs provide these many primitives even though only one is enough to guarantee correctness, such as by implementing locks?

Multiple reasons. ISAs need to provide different alternatives because the underlying hardware or architectural decisions can make one of these primitives more efficient. Also, the same is true for characteristics of program behavior such as the number of writes and contention. In addition, different primitives can be used for different needs. For example, non-blocking functions need non-blocking promitives (such as compare&swap) for example.

Appendix

The state machine for the unmodified MESI protocol:



YOU MAY DETACH THIS PAGE