CS 152 Computer Architecture and Engineering

Final Exam

May 8th, 2018

Professor Krste Asanovic

Name:_________________________

This is a closed book, closed notes exam.

170 Minutes. 26 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 exam.
- You must not discuss an exam's contents with other students who have not taken the exam. If you have inadvertently been exposed to an exam prior to taking it, you must tell the instructor or TA.
- You will receive no credit for selecting multiple-choice answers without giving explanations if the instructions ask you to explain your choice.

	CS152	Your Points
Question 1	25 points	
Question 2	20 points	
Question 3	20 points	
Question 4	15 points	
Question 5	20 points	
Question 6	20 points	
Question 7	22 points	
Question 8	28 points	
Total	170 points	

Name: ______________________________

Question 1: Out-of-Order Execution [25 points]

Question 1.A (12 points)

We execute the following program on an out-of-order core with a unified physical register file. There are 6 entries in the ROB. Show the state snapshot of the pipeline when the cache miss is resolved and Instruction 1 has just completed (but not committed). Assume the cache miss penalty is 100 times longer than any functional unit latency and the free list is in FIFO order. We have already completed the first instruction for you.

```
1.  lw x4, 0(x1)          # cache miss
2.  mul x4, x4, x4
3.  lw x5, 0(x2)          # cache hit
4.  add x3, x5, x6
5.  sw x4, 0(x1)
6.  sub x4, x3, x1
```

Name: ______________________________

Rename Table		Physical Register File			Free List
x0		p0	<x3>	p	~~p5~~
x1	p3	p1	<x5>	p	p8~~p8~~
x2	p4	p2	<x6>	p	p9~~p9~~
x3	p6→p0	p3	<x1>	p	p0~~p0~~
x4	p7→ p5→p8 →p10	p4	<x2>	p	p10~~p10~~
x5	p1→p9	p5	<x4>	p	p11
x6	p2	p6	<x3>	p	p12
x7		p7	<x4>	p	
x8		p8		p~~p~~	
x9		p9	<x5>	p	
x10		p10	<x4>	p	
x11		p11			
x12		p12			
...		...			

ROB										
valid	complete	exception	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
v	c		lw	p	p3			x4	p7	p5
v			mul	p	p5	p	p5	x4	p5	p8
v	c		lw	p	p4			x5	p1	p9
v	c		add	p	p9	p	p2	x3	p6	p0
v			sw		p8	p	p3	-	-	-
v	c		sub	p	p0	p	p3	x4	p8	p10

Question 1.B (6 points)

We execute the following program (different from Question 1.A) on the same machine as in Question 1.A. All six instructions complete, but Instruction 3 and Instruction 6 report exceptions. Show the state snapshot of the pipeline after all misspeculated instructions have been rolled back. That is, after precise architectural state has been restored and the correct exception handler is about to be fetched.

```
1.   lw x2, 0(x1)
2.   add x2, x2, x3
3.   sw x2, 0(x4)          # exception
4.   addi x4, x4, 4
5.   add x3, x3, x4
6.   lw x4, 0(x3)          # exception
```

Rename Table		Physical Register File			Free List
x0		p0	<x4>	p~~p~~	p11
x1	p3	p1	<x5>	p	p12
x2	p8	p2	<x6>	p	p4
x3	p10→ p6	p3	<x1>	p	p5
x4	p0→ p9→ p7	p4	<x2>	p~~p~~	p0
x5	p1	p5	<x2>	p~~p~~	p10
x6	p2	p6	<x3>	p	p9
x7		p7	<x4>	p	
x8		p8	<x2>	p	
x9		p9	<x4>	p~~p~~	
x10		p10	<x3>	p~~p~~	
x11		p11			
x12		p12			
...		...			

ROB										
valid	complete	exception	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
v~~v~~	c		lw	p	p3			x2	p4	p5
v~~v~~	c		add	p	p5	p	p6	x2	p5	p8
v~~v~~	c	x	sw	p	p7	p	p8	-	-	-
v~~v~~	c		addi	p	p7			x4	p7	p9
v~~v~~	c		add	p	p6	p	p9	x3	p6	p10
v~~v~~	c	x	lw	p	p10			x4	p9	p0

Question 1.C (2 points)

How many physical registers does a data-in-ROB design have?

(arch registers) + # (ROB entries)

Question 1.D (5 points)

Most contemporary out-of-order processors have a separate issue window from the ROB, which holds instructions decoded and renamed but not issued into execution. This issue window also contains register tags, presence bits, and the pointer to the ROB for each instruction.

In this design, the ROB is usually several times larger than the issue window. Explain why.

Decoded and renamed instructions stay longer in the ROB (until commit) than in the issue window (until execution).

Name: ______________________________

Question 2: Virtual Memory [20 points]

Question 2.A (5 points)

Suppose we have a virtual memory system with 1024-byte pages. Assume that physical addresses are 32 bits and each page table entry takes up 32 bits. If we use a two-level page table and each level takes up a single page, how many bits will be in the virtual page number and how many in the page offset? How many bytes of virtual memory can we address?

VPN: $2 * \log_2(1024 / 4) = 2 * (10 - 2) = 2 * 8 = 16$

Offset: $\log_2(1024) = 10$

Memory size: $2^{(10 + 16)} = 2^{26} = 64$ MB

Question 2.B (5 points)

Suppose we have a virtual memory system with 2048-byte pages. Assume that physical addresses are 64 bits, each page table entry takes up 64 bits, and each level of the page table takes up a single page total. How many levels of paging would we need in order to cover 32 GB (2^{35} bytes) of virtual memory?

$2048 / 8 = 2^{11} / 2^3 = 2^{(11 - 3)} = 2^8$

$2^{8x} * 2^{11} = 2^{(8x+11)} = 2^{35}$
$8x + 11 = 35$
$x = (35 - 11) / 8 = 24 / 8 = 3$ levels

Name: ______________________________

Question 2.C (5 points)

You are asked to design a virtually indexed, physically tagged cache. A page is 4096 bytes. The cache must have 256 lines of 64 bytes each. What associativity must the cache have to not worry about aliases?

$2^8 * 2^6 = 2^{14} = 2^{12} * 2^2$

4-way set-associative

Question 2.D (5 points)

Now assume we have 4096-byte pages and a direct-mapped, virtually-indexed, physically-tagged cache with 256 lines of 64 bytes each. An inclusive L2 is used to detect aliasing. If the L2 detects an alias for the physical address 0x80007243, which sets in the L1 could contain the aliased entry? The sets are indexed starting from zero. Give your answers in decimal.

(0x80007243 % 4096) / 64 = 0x243 >> 6 = 9

256 / 4 = 64

9 + 64 = 73

9 + 2 * 64 = 137

9 + 3 * 64 = 201

Sets 9, 73, 137, and 201 must be checked.

Question 3: Load/Store Units and Memory Consistency [20 points]

Table 3.1 shows the current state of the store queue in an out-of-order core of a chip multiprocessor (CMP). Stores are kept in the store queue until they commit. The instruction number indicates the order of instructions in the program, with lower numbers being earlier in program order.

Table 3.2 shows the values stored in the non-blocking data cache. Loads following cache misses can read the data from the data cache if the memory model is not violated. Caches are coherent across processors.

Table 3.3, Table 3.4, Table 3.5, and Table 3.6 show the current state of the load queue. Assume that all stores and loads are for the full 32-bit word and aligned to 32 bits. The processor uses conservative out-of-order load/store execution.

Instruction Number	Address	Value
3	0x1000	0xF00D3ABC
6	0x2000	Unknown
11	Unknown	Unknown
15	0x3000	0xDEADBEEF
17	Unknown	Unknown

Table 3.1 Store Queue

Valid?	Address	Value
Y	0x1000	0xAACCBDAF
Y	0x2000	0xBADE2140
Y	0x3000	0x1234ABCD
N	0x4000	Unknown

Table 3.2 Data Cache

Question 3.A (5 points)

Under ***sequential consistency (SC)***, assuming the stores make no progress, can each load in Table 3.3 complete? If so, what value does it read?

Instruction Number	Address	Can complete?	Value
1	0x2000	Y	0xBADE2140
5	0x3000	N	
7	0x1000	N	
8	0x4000	N	
9	0x2000	N	
16	0x3000	N	
18	0x3000	N	

Table 3.3 Load Queue

Question 3.B (5 points)

Under ***total store ordering (TSO)***, assuming the stores make no progress, can each load in Table 3.4 complete? If so, what value does it read?

Instruction Number	Address	Can complete?	Value
1	0x2000	Y	0xBADE2140
5	0x3000	Y	0x1234ABCD
7	0x1000	Y	0xFOOD3ABC
8	0x4000	N	
9	0x2000	N	
16	0x3000	N	
18	0x3000	N	

Table 3.4 Load Queue

Question 3.C (5 points)

Under the ***weak multi-copy-atomic memory model***, assuming the stores make no progress, can each load in Table 3.5 complete? If so, what value does it read?

Instruction Number	Address	Can complete?	Value
1	0x2000	Y	0xBADE2140
5	0x3000	Y	0x1234ABCD
7	0x1000	Y	0xF00D3ABC
8	0x4000	N	
9	0x2000	N	
16	0x3000	Y	0xDEADBEEF
18	0x3000	N	

Table 3.5 Load Queue

Question 3.D (5 points)

Now, simultaneous multithreading (SMT) is implemented in each out-of-order core to support multithreading in a single processor. Assume all stores in Table 3.1 have been issued by Thread 1 while all loads in Table 3.6 are issued by Thread 2. Assume the stores make no progress. Under the ***weak multi-copy-atomic memory model***, can each load in Table 3.6 complete? If so, what value does it read?

Instruction Number	Address	Can complete?	Value
1	0x2000	Y	0xBADE2140
5	0x3000	Y	0x1234ABCD
7	0x1000	Y	0xAACCBDAF
8	0x4000	N	
9	0x2000	Y	0xBADE2140
16	0x3000	Y	0x1234ABCD
18	0x3000	Y	0x1234ABCD

Table 3.6 Load Queue in Thread 2

Name: ______________________________

Question 4: VLIW Machines [15 points]

Question 4.A (10 points)

In this problem, we consider the execution of the following code segment on a VLIW processor:

```
double A[N], B[N], C, D;
...
for (i = 0 ; i < N ; i++) {
  B[i] = (A[i] * C) + D;
}
```

```
# t0: i, f0: C, f1: D
# a0: N, a1: pointer of A[i], a2: pointer of B[i]

loop:   fld f2, 0(a1)           # load A[i]
        fmul f3, f2, f0         # A[i] * C
        fadd f4, f3, f1         # B[i] = A[i] * C + D
        fsd f4, 0(a2)           # store B[i]
        addi a1, a1, 8          # bump A
        addi a2, a2, 8          # bump B
        addi a0, a0, -1         # decrement N
        bgtz a0, loop           # loop
```

Now we have a VLIW machine with five execution units:

- two ALU units, latency one cycle, also used for branch operations.
- one memory unit, latency two cycles, fully pipelined, each unit can perform either a store or a load.
- one FADD unit and one FMUL unit, both have latency three cycles and are fully pipelined

Assume N is a large number and there are no exceptions during the execution.

Name: ______________________________

Schedule instructions for the VLIW machine with software pipelining but without loop unrolling in Table 4-1 including the prologue and the epilogue.

Label	ALU1	ALU2	MEM	FADD	FMUL
	addi a0, a0, -1	addi a1, a1, 8	fld f2, 0(a1)		
					fmul f3, f2, f0
	addi a0, a0, -1	addi a1, a1, 8	fld f2, 0(a1)		
				fadd f4, f3, f1	fmul f3, f2, f0
	addi a0, a0, -1	addi a1, a1, 8	fld f2, 0(a1)		
loop:		addi a2, a2, 8	fsd f4, 0(a2)	fadd f4, f3, f1	fmul f3, f2, f0
	addi a0, a0, -1	addi a1, a1, 8	fld f2, 0(a1)		
	bgzt a0, loop				
		addi a2, a2, 8	fsd f4, 0(a2)	fadd f4, f3, f1	fmul f3, f2, f0
		addi a2, a2, 8	fsd f4, 0(a2)	fadd f4, f3, f1	
		addi a2, a2, 8	fsd f4, 0(a2)		

Table 4-1: VLIW Scheduling with Software Pipelining

Question 4.B (5 points)

Explain why VLIW machines were not successful in general-purpose systems, but have been successful in embedded DSP systems.

- Constant memory latency (scratchpad)
- More predictable branches

Name: ______________________________

Question 5: Vector Machines [20 points]

Problem 5.A (10 points)

In this problem, we will vectorize the following code with the RISC-V Vector ISA:

```
double A[N], B[N];
char C[N]; # C[i] = 0 or 1
int X[N], Y[N]; # permutation of {1, …, N}
...
for (i=0; i<N; i++) {
  if (!C[i]) { B[Y[i]] += A[X[i]] * B[Y[i]]; }
}
```

Fill out the blanks below (the code spans to the next page). You do not need to fill out all blanks for correct code.

```
# a0: N
# a1: A pointer, a2: B pointer, a3: C pointer
# a4: X pointer, a5: Y pointer
# v1: 8 bit int vector for mask
# v0, v2-v7: 32-bit int vectors
# v8-v16: 64-bit float vectors

loop:     setvl t0, a0________________  # set VL for loop, t0 = VL

          # set mask from C

          vld v1, 0(a3)________________________

          _____________________________________

          _____________________________________

          # load A[X[]], B[Y[i]]

          vld v0, 0(a4), v1.f__________________

          vld v2, 0(a5), v1.f__________________

          vsli v0, v0, 3, v1.f_________________

          vsli v2, v2, 3, v1.f_________________

          vldx v8, 0(a1), v0, v1.f_____________
```

Name: ______________________

```
vldx v9, 0(a2), v2, v1.f____________
____________________________________
____________________________________
____________________________________

# compute A[X[i]] * B[Y[i]] + B[Y[i]]
vmadd v10, v8, v9, v9, v1.f__________
____________________________________
____________________________________

# store B[Y[i]]
vstx v10, 0(a2), v2, v1.f____________
____________________________________
____________________________________

# bump pointers and loop
sll t1, t0, 3
sll t2, t0, 2
add a1, a1, t1              # bump A
add a2, a2, t1              # bump B
add a3, a3, t0              # bump C
add a4, a4, t2              # bump X
add a5, a5, t2              # bump Y
sub a0, a0, t0              # decrement N
bnez a0, loop               # loop
```

Question 5.B (10 points)

Compare GPUs against vector processors:

	GPU	Vector
Programming Model	SPMD(CUDA, OpenCL)	Scalar + Auto vectorization SPMD
Hardware Execution Model	SIMT	SIMD
Memory Operations	All memory operations are scatter/gather + HW coalescing Unit	Unit-stride, strided, and indexed
Conditional Execution	Thread masks + Divergence Stack	Vector masks

Which machine is more complex? Explain briefly.

GPU incurs additional hardware overhead (HW coalescing unit, divergence stack) due to multithreading for a single instruction.

Question 6: Synchronization [20 points]

In the following question, you will implement semaphores in C using various synchronization primitives. A semaphore is a data structure used to control access to a critical section so that only N threads can enter the critical section at a time.

The semaphore is a shared integer initialized to N. Two functions, “wait” and “signal”, must be implemented. The wait function spins until the integer is >0. It then decrements the integer and finishes. The “signal” function increments the integer. In this way, only N threads can pass through the “wait” function before the “signal” function is called.

```
// Example non-atomic implementations of wait and signal

void wait(int *sem)
{
  while (*sem <= 0) {}
  *sem = *sem - 1;
}

void signal(int *sem)
{
  *sem = *sem + 1;
}
```

Question 6.A (4 points)

The following code attempts to implement wait and signal using atomic add operations.

```
/* atomically reads the value of *dst, adds inc to it,
 * and writes the value back */
void atomic_add(int *dst, int inc);

void wait(int *sem)
{
  while (*sem <= 0) {}
  atomic_add(sem, -1);
}

void signal(int *sem)
{
  atomic_add(sem, 1);
}
```

Assuming the memory system is sequentially consistent, does this code correctly implement wait and signal? If not, how might this implementation fail to guarantee that only N threads enter the critical section?

This code does not correctly implement wait because checking the value of *sem in the while loop is not atomic with decrementing sem. If the starting value of *sem is 1, it is possible that two threads will simultaneously read that value, fall through the while loop, and decrement *sem. The value will be correct, -1, but one more thread entered the critical section than was allowed by the semantics of wait/signal.

Name: ______________________________

Question 6.B (8 points)

Implement the wait and signal functions using a compare and swap instruction.

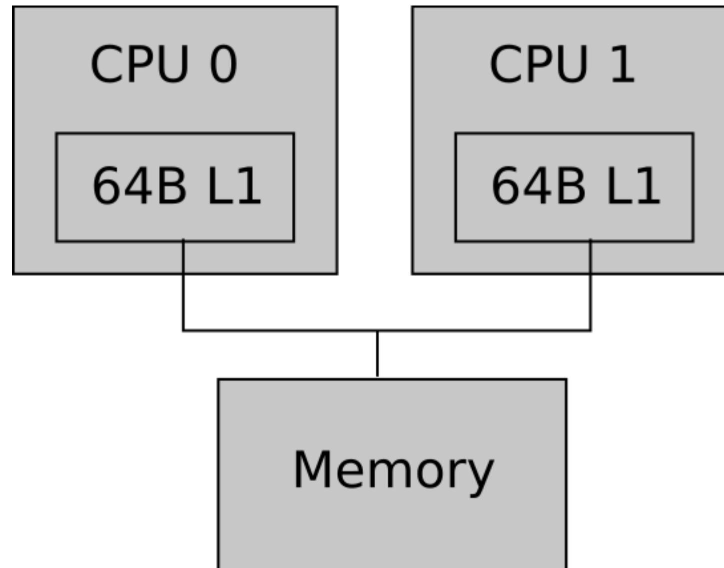
```
/* Atomically checks if (*dst == old) and writes new to *dst if they match.
 * Returns 1 on success and 0 on failure */
int compare_and_swap(int *dst, int old, int new);

void wait(int *sem)
{
  int old, success;
  do {



    while ((old = *sem) <= 0) {}
    success = compare_and_swap(sem, old, old - 1);



  } while (!success);
}

void signal(int *sem)
{
  int old, success;
  do {



    old = *sem;
    success = compare_and_swap(sem, old, old + 1);



  } while (!success);
}
```

Question 6.C (8 points)
Now implement the function using load-reserved and store-conditional.

```
/* Load and return the value from *src and set a reservation */
int load_reserved(int *src);
/* Store new to *dst if the reservation is still set.
 * Return 1 on success and 0 on failure */
int store_conditional(int *dst, int new);

void wait(int *sem)
{
  int success, old;
  do {



    while ((old = load_reserved(sem)) <= 0) {}
    success = store_conditional(sem, old - 1);



  } while (!success);
}

void signal(int *sem)
{
  int success, old;
  do {



    old = load_reserved(sem);
    success = store_conditional(sem, old + 1);



  } while (!success);
}
```

Question 7: Cache Coherence [22 points]

Question 7.A (10 points)
Consider a dual-core system in which each CPU has an L1 cache with a single 64-byte line. The caches implement the MOESI protocol on a snoopy bus.



The following table shows a sequence of memory events that occur. For each event, show the messages that will be sent on the bus and the subsequent states of the caches. Assume that both caches start out with invalid lines.

Event	Messages Sent	Cache 0 state	Cache 0 tag	Cache 1 state	Cache 1 tag
CPU 0 store 0x00	0:CRI	M	0	I	N/A
CPU 1 load 0x20	1:CR, 0:CCI	O	0	S	0
CPU 1 store 0x40	1:CRI	O	0	M	1
CPU 0 load 0x60	0:WR, 0:CR, 1:CCI	S	1	O	1
CPU 0 load 0x80	0:CR	E	2	O	1
CPU 0 store 0x90		M	2	O	1
CPU 0 store 0x50	0:WR, 0:CRI, 1:CCI	M	1	I	N/A or 1

The following function sums all `n` integers in `array` into `result`. It runs on a multicore system, with each core processing part of the array. In the argument list, `nc` is the number of cores running and `coreid` is a number from 0 to nc-1 that is distinct for each core.

```
void sum_array(int *result, int *array, int n, int coreid, int nc)
{
  for (int i = coreid; i < n; i += nc) {
    atomic_add(result, array[i]);
  }
}
```

Question 7.B (5 points)
What are two ways in which this code is inefficient, assuming that the system has L1 caches with 64-byte cache lines and uses a simple MI coherence protocol (where the only states are Modified and Invalid).

1. Interleaving array elements between cores causes false sharing
2. Large stride for accessing array leads to poor spatial locality
3. Atomic add for each array element causes unnecessary synchronization overhead.
4. Loop not unrolled

Question 7.C (7 points)

Rewrite the program to run more efficiently. Assume that `n` is always a multiple of `nc` and `nc` is always a power of two.

```
void sum_array(int *result, int *array, int n, int coreid, int nc)
{

  int chunk = n / nc;
  int start = coreid * chunk;
  int sum = 0;

  for (int i = start; i < start + chunk; i++) {
    sum += array[i];
  }
  atomic_add(result, sum);

}
```

Name: ______________________________

Question 8: Potpourri [28 points]

Question 8.A (5 points)

Choose polling or interrupt for the following devices attached to a 1GHz RISC-V processor:

Device	Data Rate	Transfer Block Size	Polling/Interrupt?
A	80B/s	4B	Polling
B	400MB/s	4B	Interrupt
C	400MB/s	1 KiB	Interrupt

Which device benefits the most from direct memory access? Explain.

C as CPU can do meaningful work during background bulk data transfers.

Question 8.B (5 points)

Discuss disadvantages of packed-SIMD instructions.

Vector lengths are encoded in instructions
→ incompatible code for different vector lengths

Question 8.C (5 points)

What is the advantage of an MESI cache-coherence protocol over an MSI cache-coherence protocol?

MESI allows a cache to write to a line it previously loaded without sending invalidations so long as it is the only cache holding the line. This helps with data that is private to a single core (which most data is).

Question 8.D (5 points)

What is the advantage of an MOESI cache-coherence protocol over an MESI cache-coherence protocol?

MOESI allows multiple caches to share dirty data and thus reduces the number of writes and reads to memory. If a cache has data in the modified or owned state, it can serve requests for the data from new sharers through CCI without writing back to memory and downgrading to the shared state.

Name: ______________________________

Question 8.E (8 points)

Why could the amo.aq.rl version of this code perform better than the fence version of this code?

<pre> sd x1, (a1) # Unrelated store
  ld x2, (a2) # Unrelated load
  li t0, 1
again:
  amoswap.w.aq t0, t0, (a0)
  bnez t0, again
  # …
  # critical section
  # …
  amoswap.w.rl x0, x0, (a0)
  sd x3, (a3) # Unrelated store
  ld x4, (a4) # Unrelated load</pre>
	<pre> sd x1, (a1) # Unrelated store
	  ld x2, (a2) # Unrelated load
	  li t0, 1
	again:
	  amoswap.w t0, t0, (a0)
	  fence r, rw
	  bnez t0, again
	  # …
	  # critical section
	  # …
	  fence rw, w
	  amoswap.w x0, x0, (a0)
	  sd x3, (a3) # Unrelated store
	  ld x4, (a4) # Unrelated load</pre>

Using acquire/release semantics with AMOs is more fine-grained than using fences.

For the first AMO, specifying “aq” requires only the swap to complete before subsequent instructions execute, whereas a fence requires all of the instructions preceding it to complete before subsequent instructions.

For the second AMO, specifying “rl” requires preceding instructions to complete before the swap executes, whereas the fence requires preceding instructions to complete before any of the subsequent instructions execute.

Thus, the version with fences constrains the timing of the loads and stores before or after the critical section, whereas the acquire/release version does not.