CS 152: Computer Architecture and Engineering

Final Exam
May 14th, 2019
Professor Krste Asanović

Name:______________________
SID:______________________

This is a closed book, closed notes exam.
170 Minutes, 24 pages.

- Not all questions are of equal difficulty, so look over the entire exam!
- Please carefully state any assumptions you make.
- Please write your name on every page in the exam.
- Do not discuss the exam with other students who haven't 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.
- You may detach the appendences provided at the reverse of the exam.

Question	**Topic**	**Point Value**
1	Virtual Memory	**28**
2	Memory Hierarchy	**24**
3	Pipelining	**20**
4	Cache Coherence	**28**
5	Multithreading	**30**
6	Memory Consistency	**22**
7	Vector Machines	**18**
TOTAL:		**170**

Problem 1: Virtual Memory

Multiple choice: Check one unless otherwise noted

A) **(2 Point)** In a virtually indexed, physically tagged cache, what part of the virtual address is used to select the cache set?

[ ] VPN **[ ]** PPN **[ ]** Page offset **[ ]** Tag

B) **(2 Point)** A page-table walker (PTW) handles what kind of events?

[ ] SegFaults **[ ]** Demand paging **[ ]** Page faults. **[ ]** TLB misses

C) **(3 Points)** Which of the following are advantages of page-based virtual memory over a base-and-bound scheme? **Check all that apply.**

[ ] Protecting one process from other processes on the system
[ ] Translating addresses to virtualize the resource of physical memory
[ ] Reduced external fragmentation
[ ] Reduced address translation cost

D) **(3 Points)** Explain why TLBs are critical for good performance in a paged virtual memory system.

E) **(4 points)** Consider a system with 4096B pages and page-table entries that are 32 bits at all levels of the page table hierarchy. How many levels of page tables are needed to support a 32-bit virtual address space?

F) **(9 points)** Consider a system with the following specifications:

- 8192B page sizes
- 64-bit virtual address space
- 48-bit physical address space
- 32KiB, 4-way set-associative L1 cache with 64B block size
- 256KiB, 4-way set-associative L2 cache with 64B block size
- TLB entries contain 8 bits of metadata

Fill in the table. Show ***organized*** work outside the table so partial credit may be assigned.

	# of bits
Physical page number	
Virtual page number	
Page offset	
TLB entry	
Cache block offset	
L1 cache tag	
L1 cache index	
L2 cache tag	
L2 cache index	

G) **(5 points)** In the system from (F), what would be a reasonable technique(s) to apply to avoid aliasing in the L1 cache while minimizing the overhead of TLB lookups? The L2 cache? Justify your response.

Problem 2: Uniprocessor Caches and Memory Hierarchy

A) **(3 Points)** How does total capacity, access latency, and bandwidth scale as we move between layers of a uniprocessor's memory hierarchy? Indicate the general trend each with an arrow pointing in the direction of an **increase** in that parameter.

Memory	Example Parameter	Capacity	Access Latency	Bandwidth
Register File	↑			
On-chip Caches				
Off-Chip Memory				

B) **(3 Points)** Suppose we build a processor with a 1-cycle L1 hit latency, a 10-cycle L1 miss penalty, and a 20-cycle L2 miss penalty. Assuming a L1 hit rate of 90% and a L2 local hit rate of 80%, what is the average memory access time seen by this processor?

cycles

C) **(4 Points)** What is the difference between an inclusive and exclusive multi-level cache? Give one advantage of each approach.

The remainder of this problem evaluates cache performance for different loop orderings. Consider the following two loops, written in C, which calculates the sum of all elements of a 16 by 512 matrix of 32-bit integers:

Loop A	***Loop B***
`int sum = 0;`	`int sum = 0;`
`for (i = 0; i < 16; i++)`	`for (j = 0; j < 512; j++)`
`  for (j = 0; j < 512; j++)`	`  for (i = 0; i < 16; i++)`
`    sum  += A[i][j];`	`    sum += A[i][j];`

The matrix `A` is stored contiguously in memory in row-major order. Row-major order means that elements in the same row of the matrix are adjacent in memory. You may assume A starts at 0x0, thus `A[i][j]` resides in memory location `[4*(512*i + j)]`.

Assume:
- caches are initially empty.
- only accesses to matrix `A` cause memory references and all other necessary variables are stored in registers.
- instructions are in a separate instruction cache.

D) (7 points) Consider a 4KiB direct-mapped data cache with 64-byte cache lines. Calculate the number of cache misses that will occur when running Loop A. Calculate the number of cache misses that will occur when running Loop B. You must show your work for full credit!

The number of cache misses for Loop A:

The number of cache misses for Loop B:

E) (7 points) Consider a 4KiB fully-associative data cache with 64-byte cache lines. This data cache uses a first-in/first-out (FIFO) replacement policy. Calculate the number of cache misses that will occur when running Loop A, and when running Loop B. You must show your work for full credit!

The number of cache misses for Loop A:

The number of cache misses for Loop B:

Problem 3: Pipelining and The Iron Law

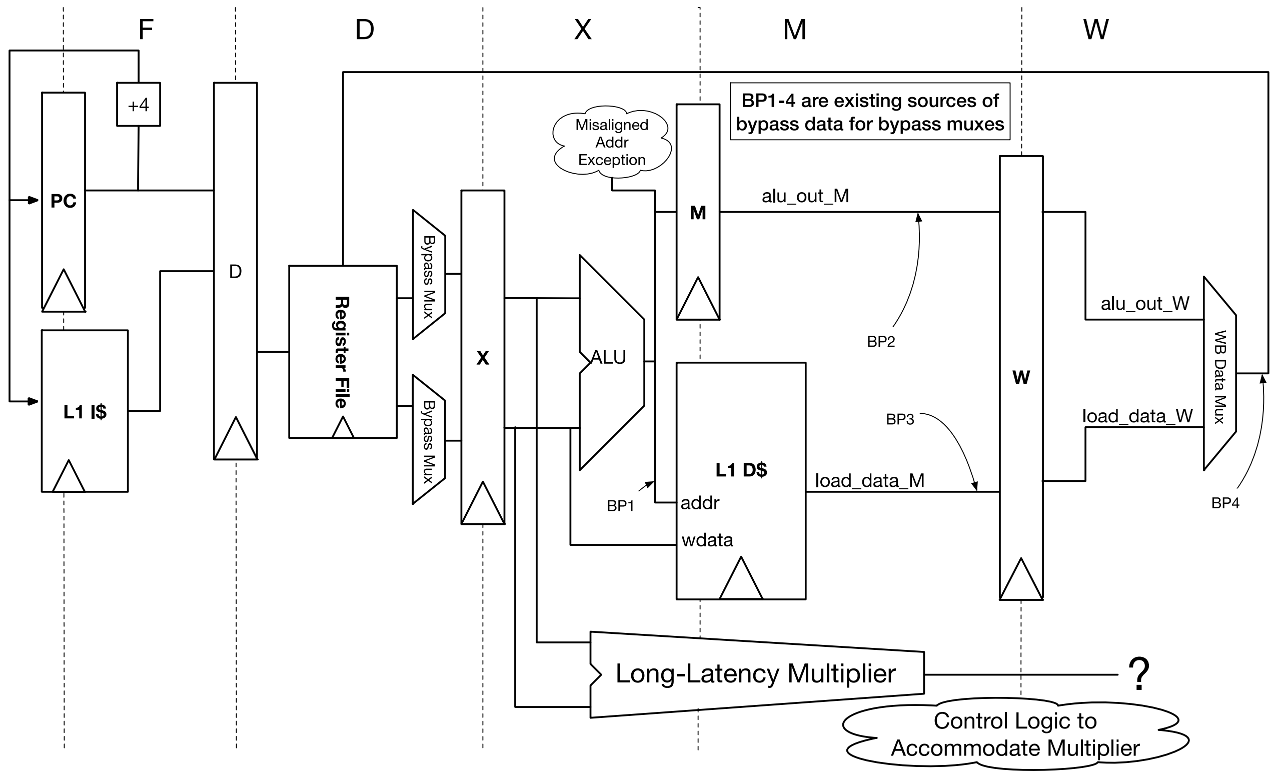
Iron Law (15 points total)

Mark whether the following modifications will cause each of the *first three* categories to **increase, decrease**, or whether the modification will have **negligible effect**. Assume all other parameters of the system are unchanged whenever possible. Explain your reasoning.

For the final column "Execution Time," mark whether the following modifications **increase**, **decrease**, have **negligible effect**, or whether the modification will have a potentially significant but **ambiguous** effect. Explain your reasoning. If the modification has an **ambiguous** effect, describe the tradeoff in which it would be a significantly beneficial modification or in which it would a significantly detrimental modification (i.e., as an engineer would you suggest using the modification or not and why?).

	Instructions / Program	Cycles / Instruction	Seconds / Cycle	Execution Time
Pipelining a single-cycle implementation				
Pipelining an unpipelined multi-cycle implementation, while keeping the *latency* of each instruction constant				
Reducing the number of stages in a pipeline				

Modifying the 5-stage Pipeline: Long Latencies



A) **(2 Point)** If we allow non-dependent operations to proceed while a multiply is outstanding, which parts of execution are proceeding "out-of-order" in the processor? Assume that multiply operations cannot generate exceptions based on their input or output values. **Check all that apply.**

[ ] Issue **[ ]** Completion **[ ]** Commit **[ ]** None of these

B) **(3 Points)** Suppose we want to add a multiplier with an 8-cycle latency and an 8-cycle occupancy to a 5-stage pipeline, as shown above. We must keep around some information about registers whose values are pending an outstanding multiply operation. What type of microarchitectural structure is commonly used for this purpose?

Problem 4: Cache Coherence

A) **(2 Point)** What is the fewest number of metadata bits (i.e., not including the line's tag and data) a writeback cache must track per cache line to implement an MSI coherence protocol (assume no transient states).

[ ] 1 bit **[ ]** 2 bits **[ ]** 3 bits **[ ]** 4 bits

B) **(3 Point)** Which sequence of memory operations would produce less coherency traffic under MESI vs MSI. Assume all memory operations act on the same cache line, and that neither processor P1 nor P2 have the line in cache initially. **Check all that apply.**

```
[   ]  P1.read,  P2.read,  P1.write
[   ]  P1.read,  P1.write, P2.read
[   ]  P1.write, P2.read,  P1.read
```

C) **(3 Points)** In general, which of the following are advantages of snoopy cache coherence over directory-based cache coherence. **Check all that apply.**

[ ] Simpler to implement
[ ] Scalable to more cores
[ ] Lower latency on cache misses
[ ] Uses less interconnect bandwidth

D) **(4 Points)** In the table below, indicate which memory operations experience a hit, true sharing miss, or false sharing miss under an MSI protocol. Assume x1, x2 are in the same cache line and the line is initially cached in a shared state by both processors (P1 and P2). The first row is filled out for you.

Time	P1	P2	Hit	True Sharing Miss	False Sharing Miss
1		Write x2		**X**	
2	Read x1				
3		Read x1			
4	Write x2				
5		Write x1			

In the remainder of this question, we'll study a simplified version of the directory-based cache coherence scheme from HW5 (**detach Appendix B**, **attached to the reverse of the exam).** Our system consists of 16 cores, each with write-back, write-allocate private caches. All caches are connected to a single directory. Specifically, we're interested the series of coherence events that transpire to service a CPU's load or store under different initial conditions. For example, consider a load-miss in CPU0 to a line is currently not cached by any CPU in the system:

Time	**Agent**	**Current State**	**Event/Message Received**	**Next State**	**Message(s) Sent**
0	CPU 0	C-nothing	Load	C-pending	ShReq(0)
1	Directory	R(ε) *(empty)*	ShReq(0)	R(0)	ShResp(0)
2	CPU 0	C-pending	ShResp(0)	C-shared	N/A
Total Messages Sent:					2

Simplifications:
- assume agents can send and receive multiple messages simultaneously
- assume messages take one time step to reach their destinations
- **if a set of events *may* happen in parallel, indicate this by setting the "time" field to the same value.**
- include only the ID field of messages (i.e., neglect data, address fields)

E) **(6 Points)** In the table below, indicate the series of events that occur to service CPU0, load-miss when CPU 4 has the line cached in the C-exclusive state.

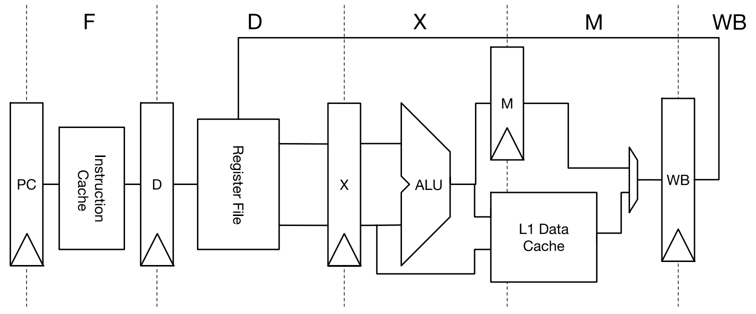
Time	**Agent**	**Current State**	**Event/Message Received**	**Next State**	**Message(s) Sent**
0	CPU 0	C-nothing	Load	C-pending	ShReq(0)
1	Directory				
Total Messages Sent:					

F) **(10 Points)** In the table below, indicate the series of events that occur to service CPU0's store-miss when CPUs 0, 4 and 8 have the line cached in a shared state.

Time	**Agent**	**Current State**	**Event/Message(s) Received**	**Next State**	**Message(s) Sent**
0	CPU 0	C-Shared	Store	C-pending	ExReq(0)
1	Directory				
Total Messages Sent:					

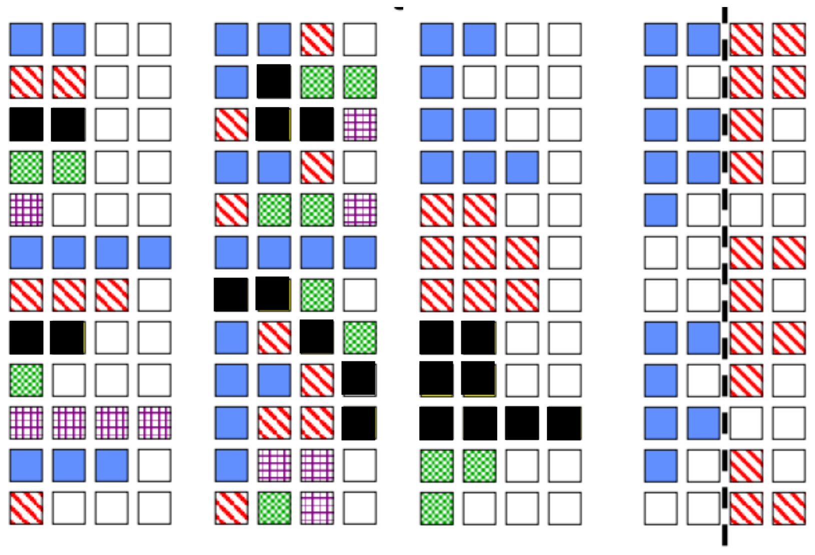
Question 5: Multithreading

A) **(2 Points)** If we multithread a classic 5-stage RISC pipeline with no bypassing (the result of an instruction can be read in decode one cycle after writeback) using a fixed-interleave policy, how many threads are required to avoid interlocks (assume no other hazards)?



[ ] 3 [ ] 4 [ ] 5 [ ] 6

B) **(4 Points)** The following diagrams indicate how the slots of a four execution units are being utilized, with each row corresponding to a different cycle and each column corresponding to a functional unit slot of the machine. Instructions belonging to different threads are indicated with a different color and texture. A white square indicates an empty slot. Label each illustration with letter (**A – G**) of the corresponding multithreading scheme.



[ ] [ ] [ ] [ ]

Labels
A: Simultaneous MT
B: Coarse-grained MT
C: Vertical MT
D: Multiprocessing
E: Fine-grained MT
F: Horizontal MT
G: Parallel MT

C) **(3 Points)** Which of the following **must** be duplicated per-thread in a multithreaded processor. **Check all that apply.**

[ ] GPRs
[ ] TLBs
[ ] PCs
[ ] Branch Predictors
[ ] Reorder Buffers

In remainder of this question, we will consider the execution of the following C kernel:

```
void kernel(float * A, float * B, float * C, int N) {
     for (int i = 0; i < N; i++)
               C[i] += A[i] * B[i];
}
```

The code above can be translated into the following RISC-V assembly code:

```
# a1, a2, a3 hold A, B, and C respectively
# a4 holds N
# t0 is initially 0
loop:
     flw f1, 0(a1)
     flw f2, 0(a2)
     flw f3, 0(a3)
     fmul f4, f1, f2
     fadd f5, f4, f3
     fst f5, 0(a3)
     addi a1, a1, 4
     addi a2, a2, 4
     addi a3, a3, 4
     addi t0, t0, 1
     bne  t0, a4, loop
```

Each cycle, the processor can fetch and issue one instruction that performs any of the following operations:
- load/store, 20-cycle latency (fully pipelined)
- integer add, 1-cycle latency
- floating-point add, 1-cycle latency
- floating-point multiply, 5-cycles latency (fully pipelined)
- branch, no delay slots, 1-cycle latency

The processor does not have a cache. Each memory operation directly accesses main memory. 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. Assume N is very large.

D) **(5 points)** Consider a single-issue in-order, multithreaded pipeline, where threads are **switched every cycle** using a fixed round-robin schedule. If the thread is not ready to run on its turn, a bubble is inserted into the pipeline. Each thread executes the above algorithm, and is calculating its own independent piece of the A array (i.e., there is no communication required between threads). In steady state, how many cycles does the machine take to execute each loop iteration for a very large value of N, without rescheduling (changing) the assembly and **assuming only a single thread.** Explain.

cycles

E) **(4 points)** Without rescheduling (changing) the assembly, how many threads are required to saturate the machine from part D? Explain.

threads

F) **(12 points)** What is the **minimum** number of threads we need to achieve peak performance on the machine from part D, assuming you **may** reschedule the code as necessary without loop unrolling. Explain, giving your final schedule.

You may perform the following optimizations:

- **Reordering instructions**
- **Renaming / re-allocating registers**
- **Changing load/store immediate offsets**

threads

Problem 7: Memory Consistency Models (20 points)

A) **(3 Points)** Is it possible to translate code that assumes sequential consistency to the RISC-V weak memory consistency model? Explain.

B) **(3 Points)** Given the instruction sequences below, check all possible combinations of P1.r2, P2.r2 after both threads have executed, assuming an ISA that is sequentially consistent. X and Y are non-overlapping, and initially X = 0, Y = 0,

P1:	**P2:**
`li r1, 1`	`li r1, 2`
`st r1, X`	`st r1, Y`
`lw r2, Y`	`ld r2, X`

```
[ ] P1.r2 = 0; P2.r2 = 0
[ ] P1.r2 = 0; P2.r2 = 1
[ ] P1.r2 = 2; P2.r2 = 0
[ ] P1.r2 = 2; P2.r2 = 1
```

C) **(2 Points)** Given the same instruction sequences and initial conditions from part B, which of the following combinations are possible under an ISA with TSO memory consistency model.

```
[ ] P1.r2 = 0; P2.r2 = 0
[ ] P1.r2 = 0; P2.r2 = 1
[ ] P1.r2 = 2; P2.r2 = 0
[ ] P1.r2 = 2; P2.r2 = 1
```

D) **(4 Points)** In general, what is the difference between a weak and a strong memory consistency model? Give one advantage of each.

E) (10 points) The following RISC-V assembly encodes a request-response relationship between two threads. The requestor thread puts work in a memory location `request` and then sets `go` = 1. It then spins waiting on the responder to produce the response. The responder thread spins waiting for work from the master, by checking if `go` has been set. Once available, it reads the request data, computes the response, and writes the response data to a memory location `response` before setting `done` = 1.

Requestor thread:

```
     li a0, 1
     sw data, request
     sw a0, go
spin:
     lw a1, done
     beqz a1, spin
     lw a2, result
     sw zero, done
```

Responder thread:

```
spin:
     lw a1, go
     beq a1, spin
     lw a2, request
     sw zero, go
     … a3 = process request
     sw a3, response
     li a0, 1
     sw a0, done
```

Under a fully relaxed memory consistency model, insert fences where necessary to ensure this code functions correctly? Use the least restrictive fences for full credit. **Assume each thread executes its code only once.**

```
     li a0, 1

     sw data, request

     sw a0, go

spin:

     lw a1, done

     beqz a1, spin

     lw a2, result

     sw zero, done
```

```
spin:

     lw a1, go

     beq a1, spin

     lw a2, request

     sw zero, go

     … a3 = process request

     sw a3, response

     li a0, 1

     sw a0, done
```

Problem 8: Vector Machines

A) **(2 Point)** What is the minimum number of lanes a machine must have to exploit data-level parallelism on vector instructions when `VL` is set to 4?

[ ] 1 **[ ]** 2 **[ ]** 4 **[ ]** 8

B) **(16 points)** Vectorize the following **32-bit integer** C code using the RISC-V vector specification described in lab 4. See appendix A for the vector instruction set listing.

```
for (i = 0; i < N; i++) {
   B[i] = (A[i] < 0) ? -A[i] : A[i]; // A and B do not overlap
}

# v0, v2-v8: configured to hold 32-bit integer values
# v1: configured to hold 8-bit integers, used as mask register
# a0 and a1: hold pointers A and B, respectively
# a2: holds N.
# t0-t3 may be used as scalar temporaries

# Your code begins:



stripmine_loop:



















     bne  ______, _______, stripmine_loop
     # Your code ends
ret
```

Appendix A: Vector Architecture for Question 1

This instruction listing is identical to that in Lab 4, but with a `setvl` instruction that has identical semantics to the preprocessor macro provided in Lab 4. This instruction first sets VL to *min(maximum vector length, rs1);* and then returns the new VL.

- The two vector mask arguments are `v1t` (perform op only if `v1[i]` is true) and `v1f` (perform op only if `v1[i]` is false). Omitting the final vector mask (`vm`) argument to all instructions is legal, and treats all elements *i* < *VL* as active.
- Unlike shortening VL, lengthening VL causes the elements extending past the previous vector length to have undefined values.

Instruction	Operation
setvl rd, rs1	VL := min(MVL, rs1); (rd) := VL
vld vd, offset(rs1), vm	vd[i] := mem[(rs1) + offset + i]
vst vs3, offset(rs1), vm	mem[(rs1) + offset + i] := vs3[i]
vlds vd, offset(rs1), rs2, vm	vd[i] := mem[(rs1) + offset + i * rs2]
vsts vs3, offset(rs1), rs2, vm	mem[rs1 + offset + i * (rs2)] := vs3[i]
vldx vd, offset(rs1), vs2, vm	vd[i] := mem[(rs1) + offset + vs2[i]]
vstx vs3, offset(rs1), vs2, vm	mem[(rs1) + offset + vs2[i]] := vs3[i]
vadd vd, vs1, vs2, vm	vd[i] := vs1[i] + vs2[i]
vsub vd, vs1, vs2, vm	vd[i] := vs1[i] - vs2[i]
vmul vd, vs1, vs2, vm	vd[i] := vs1[i] * vs2[i]
vdiv vd, vs1, vs2, vm	vd[i] := vs1[i] / vs2[i]
vrem vd, vs1, vs2, vm	vd[i] := vs1[i] % vs2[i]
vmax vd, vs1, vs2, vm	vd[i] := max(vs1[i], vs2[i])
vmin vd, vs1, vs2, vm	vd[i] := min(vs1[i], vs2[i])
vsl vd, vs1, vs2, vm	vd[i] := vs1[i] << vs2[i]
vsr vd, vs1, vs2, vm	vd[i] := vs1[i] >> vs2[i]
vseq vd, vs1, vs2, vm	vd[i] := vs1[i] == vs2[i] ? 1 : 0
vsne vd, vs1, vs2, vm	vd[i] := vs1[i] != vs2[i] ? 1 : 0
vslt vd, vs1, vs2, vm	vd[i] := vs1[i] < vs2[i] ? 1 : 0
vsge vd, vs1, vs2, vm	vd[i] := vs1[i] >= vs2[i] ? 1 : 0
vaddi vd, vs1, imm, vm	vd[i] := vs1[i] + imm
vsli vd, vs1, imm, vm	vd[i] := vs1[i] << imm
vsri vd, vs1, imm, vm	vd[i] := vs1[i] >> imm
vmadd vd, vs1, vs2, vs3, vm	vd[i] := vs1[i] * vs2[i] + vs3[i]
vmsub vd, vs1, vs2, vs3, vm	vd[i] := vs1[i] * vs2[i] - vs3[i]
vnmadd vd, vs1, vs2, vs3, vm	vd[i] := -(vs1[i] * vs2[i] + vs3[i])
vnmsub vd, vs1, vs2, vs3, vm	vd[i] := -(vs1[i] * vs2[i] - vs3[i])
vslide vd, vs1, rs2, vm	vd[i] := 0 ≤ (rs2) + i < VL ? vs1[(rs2) + i] : 0
vinsert vd, vs1, rs2, vm	vd[(rs2)] := (rs1)
vextract rd, vs1, rs2, vm	(rd) := vs1[(rs2)]
vmfirst rd, vs1	(rd) := ([i for i in range(0, VL) if LSB(vs1[i]) == 1] + [-1])[0]
vmpop rd, vs1	(rd) := len([i for i in range(0, VL) if LSB(vs1[i]) == 1])
vselect vd, vs1, vs2, vm	vd[i] := vs2[i] < VL ? vs1[vs2[i]] : 0
vmerge vd, vs1, vs2, vm	vd[i] := LSB(vm[i]) ? vs2[i] : vs1[i]

Appendix B: Directory-based Cache Coherence Protocol (Abridged Handout 6)

Changes from the handout:

- Rep renamed Resp (Rep will still be accepted)
- **Tr and Tw now include the site id (id_{req}) that initialized the request**
- Removed unnecessary messages, columns, and rows for the exam
- Removed home sites – there is only one directory site

Cache states: For each cache line, there are 4 possible states:

- C-nothing: The accessed data is not resident in the cache.
- C-shared: The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-exclusive: The accessed data is exclusively resident in this cache and has been modified.
- C-pending: The accessed data is in a *transient* state.

Directory states: For each memory block, there are 4 possible states:

- R(*dir*): The memory block is shared by the sites specified in *dir* (*dir* is a set of sites). The data in memory is valid in this state. If *dir* is empty (i.e., *dir* = ε), the memory block is not cached by any site.
- W(*id*): The memory block is exclusively cached at site *id*, and has been modified at that site. Memory does not have the most up-to-date data.
- $T_R(id_{req}, dir)$: The memory block is in a transient state waiting for the acknowledgements to the invalidation requests that the directory has issued, before giving exclusive access to the site id_{req} that initiated the request.
- $T_W(id_{req}, id)$: The memory block is in a transient state waiting for a block exclusively cached at site *id* (i.e., in C-modified state) to make the memory block at the directory up-to-date, before servicing initial request made by site id_{req}.

Protocol messages:

Category	**Messages**
Cache to Memory Requests	`ShReq, ExReq`
Memory to Cache Requests	`WbReq, InvReq`
Cache to Memory Responses	`WbResp(v), InvResp`
Memory to Cache Responses	`ShResp(v), ExResp(v)`

Current State	Handling Message	Next State	Action
C-nothing	Load	C-pending	ShReq(id)
C-nothing	Store	C-pending	ExReq(id)
C-nothing	ShResp (a)	C-shared	updates cache with prefetch data
C-nothing	ExResp (a)	C-exclusive	updates cache with data
C-shared	Store	C-pending	ExReq(id)
C-shared	InvReq(a)	C-nothing	InvResp(id)
C-exclusive	WbReq(a)	C-shared	WbResp(id, data(a))
C-pending	ShResp(a)	C-shared	updates cache with data
C-pending	ExResp(a)	C-exclusive	update cache with data

Table B-1: Cache State Transitions

Current State	Message Received	Next State	Action
R(dir) & (dir = ε)	ShReq(a)	R({id})	ShResp(id, data(a))
	ExReq(a)	W(id)	ExResp(id, data(a))
R(dir) & (id ∉ dir) & (dir ≠ ε)	ShReq(a)	R(dir + {id})	ShResp(id, data(a))
	ExReq(a)	Tr(id, dir)	InvReq(dir, a) // Note: here id is also = id_{req}
R(dir) & (dir = {id})	ExReq(a)	W(id)	ExResp(id, data(a))
R(dir) & (id ∈ dir) & (dir ≠ {id})	ExReq(a)	Tr(id, dir-{id})	InvReq(dir - {id}, a) // Note: here id is also = id_{req}
W(id') *(id' ≠ id)*	ShReq(a)	Tw(id, id')	WbReq(id', a) // Note: here id is also = id_{req}
Tr(id_{req}, dir) & (id ∈ dir) & dir ≠ {id}	InvResp(a)	Tr(id_{req}, dir - {id})	None
Tr(id_{req}, dir) & (dir = {id})	InvResp(a)	W(id_{req})	ExResp(id_{req}, data(a))
Tw(id_{req}, id)	WbResp(a)	R({id_{req}, id})	data-> memory; ShResp(id_{req})

Table B-2: Directory State Transitions, messages sent from site id.

This page intentionally left blank.