CS 152 Computer Architecture and Engineering

Final Exam
May 12, 2020
Professor Krste Asanović

Name:_____
SID:_____

180 Minutes, 27 pages.

Notes:

- 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.

Question	Topic	Point Value
1	Parallelism	**32**
2	Virtual Memory	**21**
3	Branch Prediction	**28**
4	Cache Coherence	**24**
5	Memory Consistency	**30**
6	Synchronization	**25**
TOTAL		**160**

Problem 1: Parallelism (32 points)

In this problem, we will explore how out-of-order processors, VLIW machines, and vector machines extract parallelism from the following code:

```
// Assume that N is large
for (i = 0; i < N; i++) {
      a = A[i];
      b = B[i];
      C[i] = (a * a) + (b * b);
}
```

Problem 1.A: Out-of-Order Execution (8 points)

This loop is translated into the following scalar code:

```
# a0 points to A
# a1 points to B
# a2 points to C
# a3 points to C+N
loop:
      fld f1, 0(a0)
      fld f2, 0(a1)
      fmul.d f1, f1, f1
      fmul.d f2, f2, f2
      fadd.d f1, f1, f2
      fsd f1, 0(a2)
      addi a0, a0, 8
      addi a1, a1, 8
      addi a2, a2, 8
      bltu a2, a3, loop
```

Consider an out-of-order processor with the following characteristics:

- Up to 6 instructions can be dispatched, issued, and committed per cycle
- 128-entry ROB
- 96-entry unified physical register file
- 2 integer ALUs, 1-cycle latency
- 2 load/store units, 2-cycle latency (assume all loads and stores hit in the cache)
- 1 floating-point adder, 2-cycle latency
- 1 floating-point multiplier, 3-cycle latency
- All functional units are fully pipelined
- Scheduler always selects the oldest ready instructions to issue
- Assume perfect branch prediction and memory disambiguation

What is the steady-state throughput in floating-point operations (FLOPs) per cycle? Count only arithmetic operations.

Problem 1.B: VLIW (8 points)

Consider a VLIW machine with the following characteristics:

- 2 integer ALUs, 1-cycle latency
- 2 load/store units, 2-cycle latency
- 1 floating-point adder, 2-cycle latency
- 1 floating-point multiplier, 3-cycle latency
- All functional units are fully pipelined

Instructions are statically scheduled with no interlocks; all latencies are exposed in the ISA. All register operands are read before any writes from the same instruction take effect (i.e., no WAR hazards between operations within a single VLIW instruction).

Schedule the operations from Part 1.A using software pipelining. You may rewrite the code to remove WAR/WAW hazards, but do not unroll the loop. Show only the software-pipelined loop; you **do not** need to include the prologue or epilogue. For each operation, also indicate which iteration that it belongs to (i, $i+1$, $i+2$, etc.).

What is the steady-state throughput in FLOPs per cycle?

Note: Not all rows may be needed.

ALU0	ALU1	MEM0	MEM1	FADD	FMUL

FLOPs per cycle:

Problem 1.C: Vector Machines (8 points)

The loop is translated into the following vector assembly code:

```
# a0 points to A
# a1 points to B
# a2 points to C
# a3 holds N
loop:
      vsetvli t0, a3, e64
      vle.v v0, 0(a0)
      vfmul.vv v1, v0, v0
      vle.v v2, 0(a1)
      vfmul.vv v3, v2, v2
      vfadd.vv v4, v1, v3
      vse.v v4, 0(a2)
      sub a3, a3, t0
      slli t0, t0, 3
      add a0, a0, t0
      add a1, a1, t0
      add a2, a2, t0
      bnez a3, loop
```

For this question, consider a vector machine with the following characteristics:

- 16 elements per vector register
- 4 vector lanes
- 1 load/store unit per lane, 2-cycle latency
- 1 floating-point adder per lane, 2-cycle latency
- 1 floating-point multiplier per lane, 3-cycle latency
- All functional units are fully pipelined
- All functional units have dedicated read/write ports into the vector register file
- No dead time between vector instructions
- Vector instructions execute in order
- Scalar instructions execute separately on a decoupled control processor

First, we compare the performance of the vector processor with and without chaining. Vector chaining is performed through the vector register file. An element can be read on the same cycle that it is written back, or it can be read on any later cycle – the chaining is flexible.

However, with no chaining, a dependent vector instruction must stall until the previous vector instruction finishes writing back all elements. As an example, the pipeline timing would proceed as follows for two dependent `vfadd` instructions if not using chaining:

Instruction	1	2	3	4	5	6	7	8	9
`vfadd v2, v0, v1`	R	X1	X2	W					
		R	X1	X2	W				
			R	X1	X2	W			
				R	X1	X2	W		
`vfadd v4, v2, v3`								R	X1

Complete the following table for one stripmine iteration, which shows the cycle numbers at which each vector instruction begins execution (starting from the vector register read). The first column corresponds to the baseline vector design with no chaining. The second column adds flexible chaining to the processor. Assume that `vl` is set to the maximum vector length, and the first vector instruction executes in cycle 1. Ignore scalar instructions.

Instruction	**Cycle number**	
	Without chaining	With chaining
`vle v0`	1	1
`vfmul v1`		
`vle v2`		
`vfmul v3`		
`vfadd v4`		
`vse v4`		

Assuming that the scalar instruction overhead of the stripmine loop is entirely hidden by the control processor executing separately, what is the FLOPs per cycle per lane with and without chaining? Consider the loop in steady state.

Problem 1.D: Impact on Performance (4+4 points)

For each of the processors (OoO, VLIW, vector) introduced in Parts 1.A to 1.C, discuss how the following hardware changes would impact performance on the given loop. Assume that all other design parameters remain unchanged.

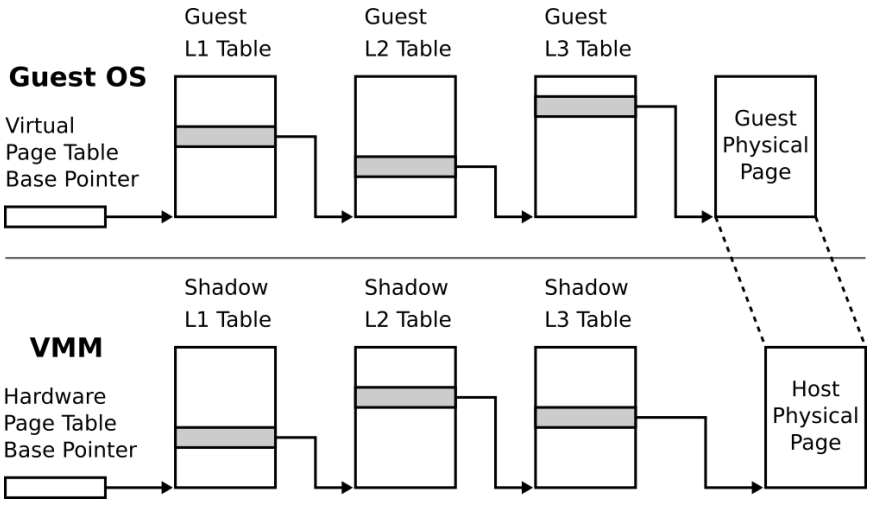
(i) Doubling the register file size (i.e., doubling the size of the unified physical register file for OoO, doubling the number of architectural registers for VLIW, doubling the length of the vector registers). Does taking advantage of the expanded register file capacity require changing the code?

(ii) Adding another floating-point multiplier. Does taking advantage of this new functional unit require changing the code?

Problem 2: Virtual Memory and Virtualization (21 points)

A virtual machine monitor (VMM) runs several guest OSs on a single host machine. The guest OSs run in user (unprivileged) mode, whereas the VMM runs in supervisor (privileged) mode. The OS in each guest virtual machine manages its own set of page tables, which reflect the mapping of the guest virtual address space to the guest physical address space ("virtual-to-real"). The guest physical addresses must then be mapped to host physical addresses.

To reuse the hardware TLB, the VMM maintains a set of *shadow page tables* that map directly from the guest virtual address space to the host physical address space ("virtual-to-physical"). When running the guest OS in user mode, the VMM sets the hardware page table base pointer to point to the shadow page table. The TLB works as if there were no virtualization.



Problem 2.A: TLB Miss Latency (3 points)

Suppose that the guest and host machines both use three-level page tables. The host has a hardware-refilled TLB. When running the guest OS, what is the TLB miss latency if the TLB access takes 1 cycle and the memory latency is 50 cycles per access?

Problem 2.B: TLB Miss Latency Revisited (3 points)

Now suppose that the guest machine uses two-level page tables instead, while the host continues to use three-level page tables. When running the guest OS, what is the TLB miss latency if the TLB access takes 1 cycle and the memory latency is 50 cycles per access?

Problem 2.C: Page Table Base Pointers (5 points)

When the guest OS begins to run a guest OS user process, it attempts to change the page table base pointer to point to the guest page table of the process. Since the guest OS itself is running in unprivileged mode, this causes a trap into the VMM. What action does the VMM take when it encounters this trap?

Problem 2.D: Page Table Updates (5 points)

From the perspective of the guest OS, the guest page tables live in guest physical memory. How does the VMM ensure that the shadow page table is updated when the corresponding guest page table is modified by the guest OS?

Problem 2.E: Different Page Sizes (5 points)

Describe how it is possible to support a guest virtual machine with an 8 KiB page size on a host machine with a 4 KiB page size.

Problem 3: Branch Prediction (28 points)

The following loop iterates through two arrays of integers and compares their elements. The code contains four branches labeled **B1**, **B2**, **B3**, and **B4**. Assume that the arrays X and Y are populated with uniformly random values.

```
c = 0;
for (i = 0; i < N; i++) { // B4
      x = X[i];
      y = Y[i];
      if (x == 0)       // B1
            c++;
      if (y == 0)       // B2
            c--;
      if (x != y)       // B3
            c += (x - y);
}
```

```
      la x1, X
      la x2, Y
      li x3, N
      li x4, 0          # c
loop:
      lw x5, (x1)       # x
      lw x6, (x2)       # y
      bnez x5, skip1          # B1
      addi x4, x4, 1
skip1:
      bnez x6, skip2          # B2
      addi x4, x4, -1
skip2:
      beq x5, x6, skip3       # B3
      sub x5, x5, x6
      add x4, x4, x5
skip3:
      addi x1, x1, 4
      addi x2, x2, 4
      addi x3, x3, -1
      bnez x3, loop           # B4
```

Problem 3.A: Branch Correlation (2+2 points)

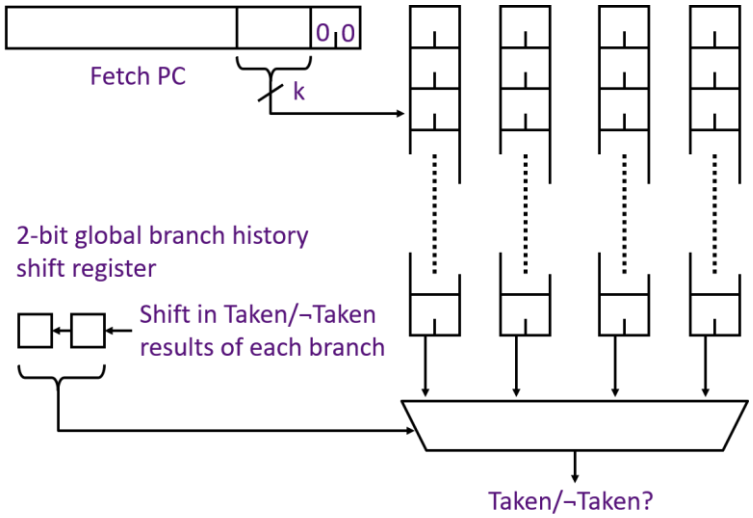
In contrast to *spatial correlation*, a branch may also demonstrate *temporal correlation* such that the present outcome of the branch is related to the previous outcomes of the same branch.

(i) For the code above, briefly explain which branches exhibit spatial correlation, if any.

(ii) For the code above, briefly explain which branches exhibit temporal correlation, if any.

Problem 3.B: Two-Level Predictors (8 points)

Consider the two-level branch predictor presented in lecture, which consists of four branch history tables (BHTs) indexed by PC. Each BHT entry contains a 2-bit saturating counter. The prediction is selected from one of the four ways based on two bits of global branch history. The outcome of the most recent branch is shifted into the global history register from right to left (1 for "taken" and 0 for "not-taken").



The most significant bit of the bimodal counter provides the prediction: 1 for "taken" and 0 for "not-taken". The counter that was used for the prediction is incremented if the branch is taken and decremented if not taken.

Complete the table on the following page for the first three iterations of the loop. The predictor columns show the state of the counters and the history register before the branch is resolved. For each row, only the counter value that was last updated needs to be written. Assume that:

- N>3, and the contents of the arrays are X = { 0, 1, 2, … } and Y = { 2, 0, 1, … }.
- All counters are initialized to the weakly "not-taken" state (01), and the global history is initialized to 00.
- Each branch is resolved before the next branch is predicted.
- The BHTs are large enough to avoid aliasing of PCs.

The first iteration has been done as an example.

Let X = { 0, 1, 2, … } and Y = { 2, 0, 2, … }.

Loop		**Branch Predictor**					**Branch Behavior**	
Iteration	Branch	History	Way 00	Way 01	Way 10	Way 11	Predicted	Actual
0	B1	00	01	01	01	01	NT	NT
	B2	00	01	01	01	01	NT	T
	B3	01	01	01	01	01	NT	NT
	B4	10	01	01	01	01	NT	T
1	B1							
	B2							
	B3							
	B4							
2	B1							
	B2							
	B3							
	B4							

Problem 3.C: Expected Accuracy (6 points)

Suppose that the elements in arrays X and Y are randomly and uniformly distributed over the set of integers 0, 1, and 2 (each possibility is equally likely). With the two-level predictor, what is the **expected accuracy** in predicting branch B3 correctly (`beq x5, x6, skip3`) as the loop approaches an infinite number of iterations? Show your work.

Hint: Consider the combined outcomes of branches B1 and B2, their probabilities, and how they contribute to the bimodal counters for B3. It may be helpful to separate the cases as such:

Conditions		Possibilities of (x, y)
$x \neq 0$	$y \neq 0$	(1, 1), (1, 2), (2, 1), (2, 2)
$x = 0$	$y \neq 0$	(0, 1), (0, 2)
$x \neq 0$	$y = 0$	(1, 0), (2, 0)
$x = 0$	$y = 0$	(0, 0)

Problem 3.D: Trace Scheduling (5 points)

Now consider a different microarchitecture without a dynamic branch predictor. The processor statically predicts that branches are never taken, and taken branches incur a multi-cycle penalty.

Although originally conceived in a VLIW context, trace scheduling is a general compiler technique for removing control hazards that can also be applied to conventional scalar architectures. Assuming the contents of arrays X and Y follow the same uniform distribution as Part 3.C (all elements are equally likely to be either 0, 1, or 2), reschedule the assembly code to minimize the branch penalty along the most frequently executed code path.

Problem 3.E: Predication (5 points)

In high-performance processor implementations, one hardware technique to reduce the impact of frequent branch mispredictions is to internally convert short forward branches into sequences of predicated operations. This can be performed at the microarchitectural level without requiring any software modifications. For example, the original instruction sequence on the left can be executed as the predicated form on the right, where p1 represents an internal predicate register.

`bnez x5, skip`
`addi x4, x4, 1`
`skip:`
	`sneq p1, x5, x0`
	`addi x4, x4, 1, p1.t`

However, this scheme nonetheless incurs some overhead in that the predicated operation would still be executed as a NOP even when the predicate is false, whereas a correctly predicted taken branch would avoid fetching and executing that instruction.

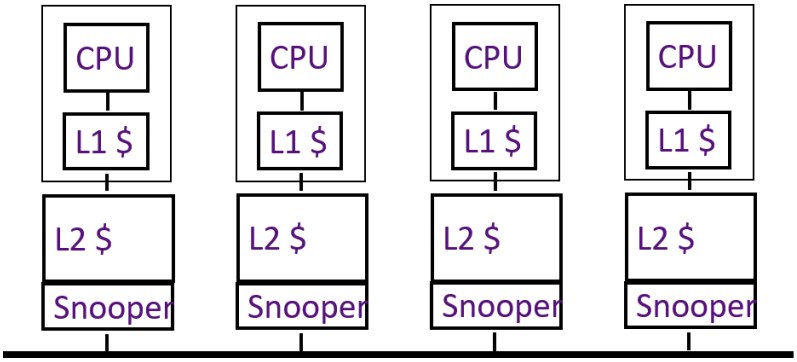
For the above sequence, how accurate does the branch predictor have to be in the non-predicated version to attain the same performance (latency) as the predicated version? Assume that the branch is taken 50% of the time, `bnez` takes 1 cycle to execute, and the misprediction penalty is 3 cycles. All other operations (`addi`, `sneq`) also each take 1 cycle to execute.

Hint: The average latency of this sequence would be 1+(1)(0.5) = 1.5 cycles in the cases where the branch is correctly predicted and 3+(1)(0.5) = 3.5 cycles if mispredicted.

Problem 4: Cache Coherence (24 points)

Problem 4.A: Inclusion Policy (4+4 points)

In lecture, it was mentioned that an inclusive L2 cache can act as a filter to reduce the amount of L1 coherence traffic in a snoopy cache-coherence protocol. If a coherence request misses in the L2 cache, there is no need to probe the L1 cache for the given line.



(i) Explain how a strictly exclusive L2 cache can also be used to optimize snooping by the L1 cache.

(ii) Could a non-inclusive, non-exclusive L2 cache (i.e., neither strictly inclusive nor strictly exclusive) be similarly used to optimize snooping by the L1 cache? Explain.

Problem 4.B: False Sharing (4 points)

In the following table, indicate which memory operations experience a hit, true sharing miss, or false sharing miss under an MSI coherence protocol. Assume that x1 and x2 reside in the same cache line, and both words are read by both processors P1 and P2 before this sequence. The first row has been completed for you.

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

Problem 4.C: Directory-Based Coherence (6+6 points)

The following questions explore the directory-based coherence protocol described in Appendix A (same as Handout #6 from Problem Set 5) in more detail.

As before, assume that message passing maintains FIFO order: All messages between the same source and destination are always received in the same order that they were sent. Also assume that each site has sufficient queuing capacity to buffer all incoming messages without drops.

(i) Consider the situation where a cache is sent an InvReq message for a given cache line. This occurs only if the directory state indicates that the site is a current sharer of the memory block, and the directory intends to invalidate the copy in the cache before granting exclusive access to another cache.

Typically, one expects the line to be in the *C-shared* state when the InvReq arrives. How is it also possible for the cache to receive the InvReq message while it has the line in the *C-pending* state (row #22 in Table H12-1 of Appendix A) – in other words, when the line is not actually present? Why does ignoring InvReq work out correctly in this case?

(ii) Consider the case where a cache requests a line in the *C-exclusive* state (ExReq) when the line is clean and shared by other caches. To reduce the response latency, it is a tempting idea to send the ExRep message with the data to the requestor in parallel to sending InvReq to the other caches. Does this optimization work correctly?

Problem 5: Memory Consistency (30 points)

Problem 5.A: Load/Store Queues (4+4+4 points)

Consider a multiprocessor with out-of-order cores that implement conservative out-of-order load/store execution (loads wait for memory addresses to be fully checked/disambiguated).

Table 2.1 shows the current state of the store queue in one of the cores. Stores are kept in the store queue until they commit. The instruction number indicates the order of the instructions in the program, with lower numbers being earlier in program order.

Table 2.2 shows the values present in the non-blocking data cache of the same core. Loads following cache misses can read from the data cache on a hit if the memory consistency model is not violated.

Tables 2.3, 2.4, and 2.5 show the current state of the load queue. Assume that all loads and stores access the full 32-bit word.

Table 2.1: Store Queue

Instruction #	Address	Value
5	0x100	0x12345678
7	0x200	*unknown*
11	0x300	0xABCDABCD
13	0x200	0x11001100
17	*unknown*	*unknown*

Table 2.2: Data Cache

Valid?	Address	Value
Y	0x100	0xFFFFFFFF
Y	0x200	0x1234ABCD
Y	0x300	0x87654321
N	0x400	*unknown*

(i) Under *sequential consistency* (SC), if the stores make no progress, can each load in Table 2.3 complete? If so, what value is read?

Table 2.3: Load Queue (SC)

Instruction #	Address	Can Complete?	Value
2	0x300		
6	0x100		
12	0x400		
16	0x200		
18	0x300		

(ii) Under *total store order* (TSO), if the stores make no progress, can each load in Table 2.4 complete? If so, what value is read?

Table 2.4: Load Queue (TSO)

Instruction #	Address	Can Complete?	Value
2	0x300		
6	0x100		
12	0x400		
16	0x200		
18	0x300		

(iii) Under a *fully relaxed multi-copy-atomic memory model*, if the stores make no progress, can each load in Table 2.5 complete? If so, what value is read?

Table 2.5 Load Queue (weak ordering)

Instruction #	Address	Can Complete?	Value
2	0x300		
6	0x100		
12	0x400		
16	0x200		
18	0x300		

Problem 5.B: TSO and Optimizations (4+4+4 points)

Explain whether the following microarchitectural optimizations are permitted under the TSO consistency model.

(i) Can a load from one thread on a multithreaded core bypass a value from the write buffer stored by a different thread on the same core? (Recall that a write buffer holds data from committed stores waiting to be written to the cache.)

(ii) Can a write buffer coalesce writes to the same word from the same thread? (Writes from two stores, not necessarily consecutive in program order, are merged into the same write buffer entry.)

(iii) Can hardware prefetching be used with the L1 cache for both loads and stores?

Problem 5.C: Sequential Consistency and OoO Scheduling (6 points)

Suppose we design an out-of-order multicore processor that implements sequential consistency within a cache-coherent memory system, using *speculation* to improve performance. Consider two independent load instructions, ld1 and ld2, where ld1 precedes ld2 in program order. In the situation where the address of ld2 is computed before ld1, the core chooses to dynamically reorder the execution of ld2 before ld1.

After the core speculatively executes ld2, but before it can commit ld2, a coherence transaction from another core invalidates the cache line accessed by ld2. What actions, if any, should the core take to maintain sequential consistency? Explain your reasoning.

Problem 6: Synchronization (25 points)

Consider a stack data structure implemented as a singly linked list which uses non-blocking synchronization to support concurrent access by multiple threads.

Each stack entry contains a pointer to the next entry further down the stack. The shared `stack` variable points to the entry at the top of the stack. The `push()` function adds a new entry onto the stack, and the `pop()` function removes and returns the topmost entry. For thread safety, the `stack` pointer must be updated atomically. The high-level pseudocode is shown below:

```
struct entry {
      struct entry *next;
      …
};

struct entry *stack;

void push(struct entry *new) {
      ATOMIC_BLOCK {
            new->next = stack;
            stack = new;
      }
}

struct entry *pop() {
      struct entry *old, *top;
      ATOMIC_BLOCK {
            top = old = stack;
            if (top != NULL)
                  top = top->next;
            stack = top;
      }
      return old;
}
```

Problem 6.A: LR/SC Deadlock (5 points)

Suppose that the load-reserved/store conditional (LR/SC) instruction pair is the only atomic read-modify-write operation provided by our 32-bit architecture. Our initial attempt at implementing the `push()` and `pop()` functions directly using LR/SC yields the following assembly code:

```
      # a0 holds pointer to new entry
push:
      la a1, stack          # get address of stack variable
      lr.w t0, (a1)         # load pointer to top entry
      sw t0, 0(a0)          # new->next = stack
      sc.w t0, a0, (a1)     # stack = new
      bnez t0, push         # retry if SC failed
      ret

      # a0 returns pointer to previous top entry
pop:
      la a1, stack          # get address of stack variable
      lr.w t0, (a1)         # load pointer to top entry
      mv a0, t0             # old = stack
      beqz t0, update       # skip dereference if (top == NULL)
      lw t0, 0(t0)          # top = top->next
update:
      sc.w t0, t0, (a1)     # stack = top
      bnez t0, pop          # retry if SC failed
      ret
```

Each core has a private write-back/write-allocate L1 data cache, and coherence is maintained through a MESI protocol. The processor implements LR/SC based on the simple approach described in lecture:

- `lr.w` ensures that the line is present in the local cache in the Modified or Exclusive state.
- `sc.w` succeeds if and only if the line has continually remained in the Modified or Exclusive state. On failure, it writes a non-zero code to the destination register.

It turns out that our code contains a major flaw!

We notice that `push()` and `pop()` become stuck in an infinite loop even when there is no contention from other cores. What caused this to happen?

Problem 6.B: Emulating CAS (5 points)

We attempt to fix the code by rewriting it in terms of compare-and-swap (CAS). CAS compares a word in memory to an expected value and, if equal, updates the memory location to a desired value. It returns a Boolean value indicating whether the substitution was successfully performed.

While our architecture lacks a CAS instruction, its functionality can be emulated with an LR/SC sequence of four instructions:

```
int CAS(int *addr, int old, int new) {
      int status;
      ATOMIC_BLOCK {
            if (*addr == old) {
                  *addr = new;
                  status = 0; // success
            } else {
                  status = 1; // failure
            }
      }
      return status;
}
```

```
      # a0 holds addr
      # a1 holds old
      # a2 holds new
cas:
      lr.w t0, (a0)          # load original value
      bne a1, t0, fail       # fail if not equal
      sc.w t0, a2, (a0)      # attempt to update
      bnez t0, cas           # retry if SC failed
      …                      # success
fail:
      …                      # failure
```

The pseudocode for the stack data structure becomes:

```
void push(struct entry *new) {
      do {
            struct entry *old = stack;
            new->next = old;
      } while (CAS(&stack, old, new));
}

struct entry *pop() {
      struct entry *old, *top;
      do {
            top = old = stack;
            if (top != NULL)
                  top = top->next;
      } while (CAS(&stack, old, top));
      return old;
}
```

For simplicity, we will consider only the `push()` function for the remainder of this problem. We rewrite the assembly code for `push()` with the CAS sequence inlined:

```
      # a0 holds pointer to new entry
push:
      la a1, stack          # get address of stack variable
      lw t0, (a1)           # load pointer to top entry
      sw t0, 0(a0)          # new->next = stack
      lr.w t1, (a1)         # load pointer to top entry again
      bne t0, t1, push      # retry if not the same
      sc.w t0, a0, (a1)     # stack = new
      bnez t0, push         # retry if SC failed
      ret
```

Recall from lecture that CAS nominally guarantees forward progress. Does that remain true when CAS is emulated using LR/SC, if LR/SC is implemented as described in Part 6.A? If not, give a scenario where forward progress is obstructed.

Problem 6.C: ABA Problem and CAS (5 points)

Is the revised LR/SC version from Part 6.B susceptible to the ABA problem like a CAS instruction would be? If so, describe an interleaved sequence of `push()` and `pop()` operations by two threads that results in a stack entry being permanently lost.

Problem 6.D: ABA Problem and DW-CAS (5 points)

To avoid the ABA problem, we switch to using double-width compare-and-swap (DW-CAS), a variant of CAS that supports atomic access to two contiguous words in memory.

```
int DWCAS(int *addr, int old1, int new1, int old2, int new2)
      int status;
      ATOMIC_BLOCK {
            if ((addr[0] == old1) && (addr[1] == old2)) {
                  addr[0] = new1;
                  addr[1] = new2;
                  status = 0; // success
            } else {
                  status = 1; // failure
            }
      }
      return status;
}
```

```
// Assume variables are allocated contiguously
struct entry *stack;
int count;

void push(struct entry *new) {
      do {
            struct entry *old = stack;
            new->next = old;
      } while (DWCAS(&stack, old, new, count, count+1));
}

struct entry *pop() {
      struct entry *old, *top;
      do {
            top = old = stack;
            if (top != NULL)
                  top = top->next;
      } while (DWCAS(&stack, old, top, count, count+1));
      return old;
}
```

Explain how DW-CAS overcomes the ABA problem.

Problem 6.E: Emulating DW-CAS (5 points)

Consider the following naïve attempt to emulate DW-CAS with single-word LR/SC:

```
      # a0 holds addr
      # a1 holds old1
      # a2 holds old2
      # a3 holds new1
      # a4 holds new2
dwcas:
      lr.w t0, 0(a0)         # load original value of word 1
      lr.w t1, 4(a0)         # load original value of word 2
      bne a1, t0, fail       # fail if word 1 not equal
      bne a2, t1, fail       # fail if word 2 not equal
      sc.w t0, a3, 0(a0)     # attempt to update word 1
      sc.w t1, a4, 4(a0)     # attempt to update word 2
      or t0, t0, t1
      bnez t0, dwcas         # retry if either SC failed
      …                      # success
fail:
      …                      # failure
```

Explain why this simple approach does not work, even if both words are located within the same cache line.

CS152 Computer Architecture and Design

Directory-based Cache Coherence Protocol *4/11/2011*

Before introducing a directory-based cache coherence protocol, we make the following assumptions about the interconnection network:

- Message passing is reliable, and free from deadlock, livelock and starvation. In other words, the transfer latency of any protocol message is finite.
- Message passing is FIFO. That is, protocol messages with the same source and destination sites are always received in the same order as that in which they were issued.

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

- C-invalid (= `Nothing`): The accessed data is not resident in the cache.
- C-shared (= `Sh`): The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
- C-modified (= `Ex`): The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
- C-transient (= `Pending`): The accessed data is in a *transient* state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

Home 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(*dir*): The memory block is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.
- T_W(*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 home site up-to-date.

Protocol messages: There are 10 different protocol messages, which are summarized in the following table (their meaning will become clear later).

Category	**Messages**
Cache to Memory Requests	`ShReq, ExReq`
Memory to Cache Requests	`WbReq, InvReq, FlushReq`
Cache to Memory Responses	`WbRep(v), InvRep, FlushRep(v)`
Memory to Cache Responses	`ShRep(v), ExRep(v)`

No.	Current State	Handling Message	Next State	Dequeue Message?	Action
1	C-nothing	Load	C-pending	No	ShReq(id,Home,a)
2	C-nothing	Store	C-pending	No	ExReq(id,Home,a)
3	C-nothing	WbReq(a)	C-nothing	Yes	None
4	C-nothing	FlushReq(a)	C-nothing	Yes	None
5	C-nothing	InvReq(a)	C-nothing	Yes	None
6	C-nothing	ShRep (a)	C-shared	Yes	updates cache with prefetch data
7	C-nothing	ExRep (a)	C-exclusive	Yes	updates cache with data
8	C-shared	Load	C-shared	Yes	Reads cache
9	C-shared	WbReq(a)	C-shared	Yes	None
10	C-shared	FlushReq(a)	C-nothing	Yes	InvRep(id, Home, a)
11	C-shared	InvReq(a)	C-nothing	Yes	InvRep(id, Home, a)
12	C-shared	ExRep(a)	C-exclusive	Yes	None
13	C-shared	(Voluntary Invalidate)	C-nothing	N/A	InvRep(id, Home, a)
14	C-exclusive	Load	C-exclusive	Yes	reads cache
15	C-exclusive	Store	C-exclusive	Yes	writes cache
16	C-exclusive	WbReq(a)	C-shared	Yes	WbRep(id, Home, data(a))
17	C-exclusive	FlushReq(a)	C-nothing	Yes	FlushRep(id, Home, data(a))
18	C-exclusive	(Voluntary Writeback)	C-shared	N/A	WbRep(id, Home, data(a))
19	C-exclusive	(Voluntary Flush)	C-nothing	N/A	FlushRep(id, Home, data(a))
20	C-pending	WbReq(a)	C-pending	Yes	None
21	C-pending	FlushReq(a)	C-pending	Yes	None
22	C-pending	InvReq(a)	C-pending	Yes	None
23	C-pending	ShRep(a)	C-shared	Yes	updates cache with data
24	C-pending	ExRep(a)	C-exclusive	Yes	update cache with data

Table H12-1: Cache State Transitions

No.	Current State	Message Received	Next State	Dequeue Message?	Action
1	R(dir) & (dir = ε)	ShReq(a)	R({id})	Yes	ShRep(Home, id, data(a))
2	R(dir) & (dir = ε)	ExReq(a)	W(id)	Yes	ExRep(Home, id, data(a))
3	R(dir) & (dir = ε)	(Voluntary Prefetch)	R({id})	N/A	ShRep(Home, id, data(a))
4	R(dir) & (id ∉ dir) & (dir ≠ ε)	ShReq(a)	R(dir + {id})	Yes	ShRep(Home, id, data(a))
5	R(dir) & (id ∉ dir) & (dir ≠ ε)	ExReq(a)	Tr(dir)	No	InvReq(Home, dir, a)
6	R(dir) & (id ∉ dir) & (dir ≠ ε)	(Voluntary Prefetch)	R(dir + {id})	N/A	ShRep(Home, id, data(a))
7	R(dir) & (dir = {id})	ShReq(a)	R(dir)	Yes	None
8	R(dir) & (dir = {id})	ExReq(a)	W(id)	Yes	ExRep(Home, id, data(a))
9	R(dir) & (dir = {id})	InvRep(a)	R(ε)	Yes	None
10	R(dir) & (id ∈ dir) & (dir ≠ {id})	ShReq(a)	R(dir)	Yes	None
11	R(dir) & (id ∈ dir) & (dir ≠ {id})	ExReq(a)	Tr(dir-{id})	No	InvReq(Home, dir - {id}, a)
12	R(dir) & (id ∈ dir) & (dir ≠ {id})	InvRep(a)	R(dir - {id})	Yes	None
13	W(id')	ShReq(a)	Tw(id')	No	WbReq(Home, id', a)
14	W(id')	ExReq(a)	Tw(id')	No	FlushReq(Home, id', a)
15	W(id)	ExReq(a)	W(id)	Yes	None
16	W(id)	WbRep(a)	R({id})	Yes	data -> memory
17	W(id)	FlushRep(a)	R(ε)	Yes	data -> memory
18	Tr(dir) & (id ∈ dir)	InvRep(a)	Tr(dir - {id})	Yes	None
19	Tr(dir) & (id ∉ dir)	InvRep(a)	Tr(dir)	Yes	None
20	Tw(id)	WbRep(a)	R({id})	Yes	data-> memory
21	Tw(id)	FlushRep(a)	R(ε)	Yes	data-> memory

Table H12-2: Home Directory State Transitions, Messages sent from site **id**