CS 152 Computer Architecture and Engineering
CS 252 Graduate Computer Architecture

Midterm #1
SOLUTIONS
March 1, 2021
Professor Krste Asanović

Name:________________________

SID:________________________

80 Minutes, 5 Questions

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	CS152 Point Value	CS252 Point Value
1	12	12
2	21	12
3	26	26
4	14	14
5	12	12
TOTAL	85	76

Problem 1: (12 Points) Iron Law of Processor Performance

Mark whether the following modifications will cause each term in the Iron Law to **increase** or **decrease**, or whether the modification will have a **negligible** effect. Assume all other parameters of the system are unchanged whenever possible. Explain your reasoning. Be explicit if you are relying on any specific assumptions.

	Instructions / Program	Cycles / Instruction	Time / Cycle
Adding a second data bus to a single-bus microcoded machine	**Negligible** The second data bus is a microarchitectural feature not visible at the ISA level.	**Decrease** The second data bus avoids some structural hazards compared to a single shared bus, reducing the number of microinstructions by enabling more parallel operations.	**Increase** The second data bus increases fanout and wire congestion. Additional muxes are needed to select between busses for each consumer. Driving the second bus requires more control signals, increasing the microcode ROM width.
Adding instructions with register-operand indexing: R[rd] = R[R[rs1]] + R[R[rs2]]	**Decrease** – Code that performs dynamic array indexing may sometimes be replaced with fewer instructions. OR **Negligible** – A compiler is unlikely to use this addressing mode, since arrays are usually allocated in memory, not in the scalar register file.	**Increase** Each operand involves two register file reads, which may require occupying the decode stage for two cycles or introducing an additional pipeline stage. Structural hazards can arise from the limited number of register file read ports. More data hazards are possible.	**Increase** More read ports may have to be added to the register file. The control logic becomes more complex to sequence the second round of register file reads.

	Instructions / Program	Cycles / Instruction	Time / Cycle
Using a software page table walker, instead of a hardware PTW	**Increase** For each TLB miss, additional instructions are executed by an exception handler to walk the page tables and refill the TLB.	**Decrease** A TLB miss does not incur a long-latency stall for a hardware page table walk. Bubbles are replaced with additional instructions.	**Decrease** – The hardware complexity is reduced. OR **Negligible** – A hardware PTW is a relatively simple state machine and is unlikely to be a critical path in typical implementations.
Removing support for precise exceptions	**Increase** Resuming from an imprecise exception may require more instructions to repair/restore microarchitectural state.	**Negligible** – Precise exceptions have minimal impact on the latencies of operations: Exception information is propagated in parallel to the pipeline, and bypassing allows results to be used before architectural state is updated at the commit point. OR **Decrease** – Exception handling latency is marginally reduced, as control can be redirected to the handler as soon as an exception is detected, rather than waiting for commit.	**Decrease** – The hardware complexity is reduced. OR **Negligible** – The simplification of commit logic is balanced by the introduction of hardware mechanisms to save microarchitectural state to memory for restartable exceptions.

	Instructions / Program	Cycles / Instruction	Time / Cycle
Changing the base page size from 4 KiB to 8 KiB	**Decrease** There are fewer page faults to handle and fewer pages for the OS to manage. (It is possible that larger pages exacerbate waste from internal fragmentation, increase memory pressure, and trigger more frequent swapping, but the difference is modest enough that this is unlikely).	**Decrease** Doubling the TLB reach reduces the number of TLB misses.	**Negligible** The page table walk is unchanged since the base page size affects only the leaf page table entries.
Removing byte load and store instructions from the ISA	**Increase** Additional bitwise instructions are required to emulate byte accesses with wider load and store instructions.	**Decrease** The shift and mask operations to pack/unpack a byte within a word increases the proportion of simple arithmetic instructions.	**Decrease** A smaller shifter is needed to align the load/store data in a cache line. (Simplified ECC circuitry in the cache subsystem was another justification cited by the architects of the Alpha ISA.)

Problem 2: (21 Points) Microprogramming (CS152)

Consider the REVLL complex instruction. This instruction reverses a linked list in memory, where the rs1 operand to this instruction is the memory address of the first node in the linked list. This instruction has no destination register, but the instruction zeroes the register specified by rs1 upon completion (it does not preserve rs1).

Alternate: This instruction reverses a linked list in memory, where the rs1 operand to this instruction is the memory address of a pointer to the first node in the linked list (a pointer to a pointer).

```
REVLL rs1
```

Every node in the linked list has the following structure. Assume that pointers are 32 bits wide in this architecture. The *next* pointer is either the memory address of the next node in the list or is equal to 0 (NULL) to indicate the end of the linked list.

```
struct node
{
        void *value;
        struct node *next;
}
```

```
struct node // Alternate
{
        struct node *next;
        void *value;
}
```

For reference, the equivalent C and assembly code for this instruction are provided below.

```
void REVLL(struct node *head) {
  struct node *prev = NULL;
  struct node *curr = head;
  while (curr != NULL) {
    struct node *next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
  }
}
```

```
        # head is passed in a0
        # t0 holds prev
        # t1 holds next
        beqz a0, done
        addi t0, t0, 0
loop:
        lw t1, 4(a0)
        sw t0, 4(a0)
        addi t0, a0, 0
        addi a0, t1, 0
        bnez t1, loop
done:
```

```
void REVLL(struct node **head) {
  struct node *prev = NULL;
  struct node *curr = *head;
  while (curr != NULL) {
    struct node *next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
  }
}
```

```
        # head is passed in a0
        # t0 holds prev
        # t1 holds next
        lw a0, 0(a0)
        beqz a0, done
        addi t0, t0, 0
loop:
        lw t1, 0(a0)
        sw t0, 0(a0)
        addi t0, a0, 0
        addi a0, t1, 0
        bnez t1, loop
done:
```

2.A (2 points) Unpipelined CPI

Consider the execution of the assembly linked-list reversal code on an unpipelined RISC-V core with a CPI of 1 for every instruction, except for loads and stores, which take 2 cycles each. How many cycles does this program take to reverse a linked list with length 4 on this core?

Prologue: 0 loads/stores, 2 other instructions
Loop: 1 load, 1 store, 3 other instructions
2 + (2 + 2 + 3) * 4 = 30 cycles

Prologue: 1 load, 2 other instructions
Loop: 1 load, 1 store, 3 other instructions
2 + 2 + (2 + 2 + 3) * 4 = 32 cycles

2.B (16 points) Microprogramming

In the attached microcode table, write microcode to implement the REVLL instruction for a bus-based RISC-V machine. This microarchitecture is identical to the one described in Handout #1 and Problem Set 1.

The final solution should be efficient with respect to the number of microinstructions used. Make sure to use logical descriptions of data movement in the "pseudocode" column for clarity. Credit will be awarded for optimizing signals using "don't care" or * values as appropriate, but this is less important than producing a correct implementation. Please comment your code clearly. If the pseudocode for a line does not fit in the space provided, or if you have additional comments, you may write neatly in the margins.

Reference material on the microcoded datapath is provided on the following page.

Three temporaries are needed to maintain the `curr`, `prev`, and `next` pointers during the linked list traversal, but as only two operand registers (A and B) are provided by the datapath, it is necessary for the microcode to clobber rs1. The solution uses R[rs1] to hold `curr`, B to hold `prev` (initialized to 0 before entering the microcode loop), and A to hold `next`.

The solution for the alternate version optimizes for fewer microinstructions in the loop body. There is also a simpler implementation that is nearly identical to the solution for first version, except for an extra microinstruction with a memory operation to deference R[rs1] prior to the loop.

2.C (2 points) Microcoded performance

How many cycles does your implementation take to reverse a linked list with length 4? Assume that the memory access time is 4 cycles.

Fetch: 3
Prologue: 2
Loop: 1 + 4 + 4 + 2 (4 iterations)
Epilogue: 1
3 + 2 + 4*11 + 1 = 50 cycles

Fetch: 3
Prologue: 1 + 4 + 1 + 1
Loop: 4 + 4 + 2 (4 iterations)
Epilogue: 1
3 + 7 + 4*10 + 1 = 51 cycles

2.D (1 point) Implementation Comparison

Compare the performance of your implementation with that of the unpipelined RISC-V core from 2.A. Assume that both processors are using the same memory system, such that the unpipelined core has 4x the cycle time of the microcoded machine to accommodate the memory latency.

1 cycle of the unpipelined core is equivalent to 4 cycles of the microcoded machine.

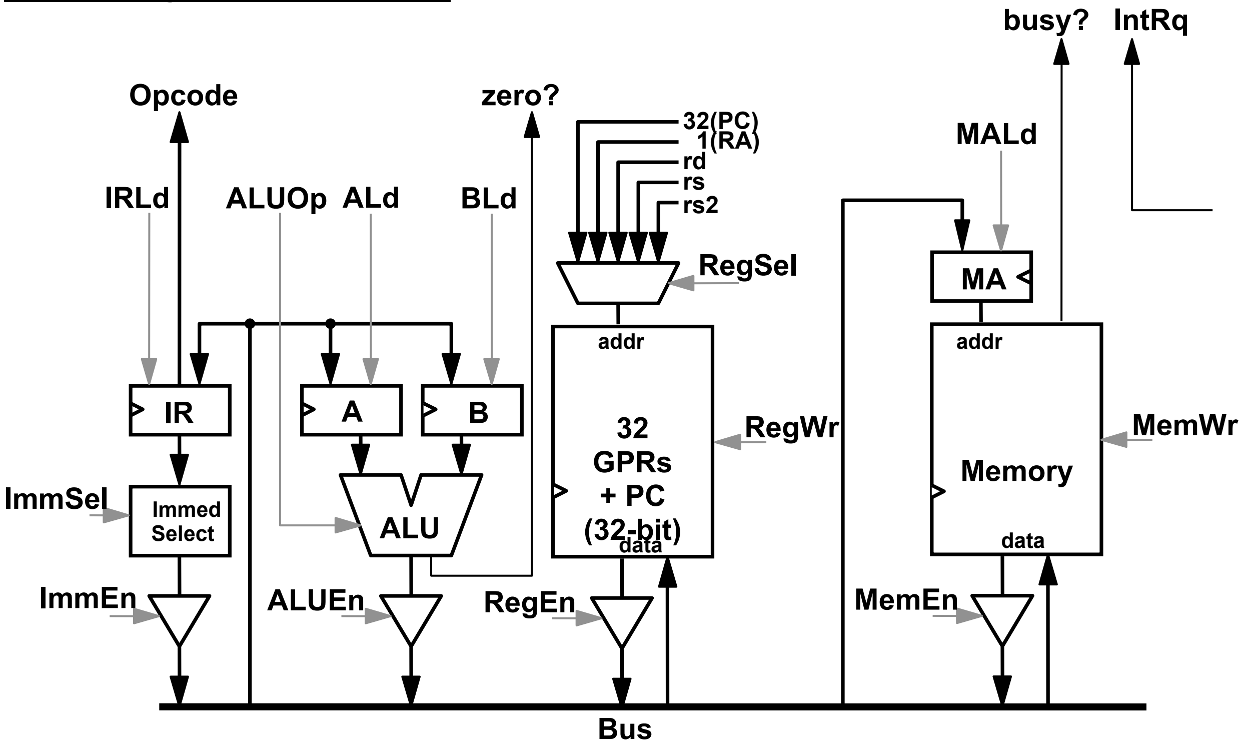
Reversing a linked list with length 4 takes the unpipelined core 30*4 = 120 equivalent cycles, which is 120/50 = 2.4 times longer than the microcoded REVLL instruction. (This demonstrates how a microcoded machine with CPI > 1 can achieve better performance compared to an unpipelined implementation clocked at a lower frequency.)

Reversing a linked list with length 4 takes the unpipelined core 32*4 = 128 equivalent cycles, which is 128/51 ≈ 2.51 times longer than the microcoded REVLL instruction.

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	1	0	*	INC_A_4	1	*	0	0	*	0	D	*
…																
REVLL0:	A, B ← R[rs1]	0	rs1	0	1	1	1	*	0	*	0	0	*	0	N	*
	if (A == 0) μBr to FETCH0	0	*	0	0	0	0	COPY_A	0	*	0	0	*	0	EZ	FETCH0
	B ← A - B (== 0)	0	*	0	0	0	1	SUB	1	*	0	0	*	0	N	*
REVLL1:	MA ← A + 4	0	*	0	0	*	0	INC_A_4	1	1	0	0	*	0	N	*
	A ← Mem	0	*	0	0	1	0	*	0	0	0	1	*	0	S	*
	Mem ← B	0	*	0	0	0	0	COPY_B	1	0	1	0	*	0	S	*
	B ← R[rs1]	0	rs1	0	1	0	1	*	0	*	0	0	*	0	N	*
	R[rs1] ← A; if (A != 0) μBr to REVLL1	0	rs1	1	0	0	0	COPY_A	1	*	0	0	*	0	NZ	REVLL1
	μBr to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En	A Ld	B Ld	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	1	0	*	INC_A_4	1	*	0	0	*	0	D	*
…																
REVLL0:	MA ← R[rs1]	0	rs1	0	1	*	*	*	0	1	0	0	*	0	N	*
	A ← Mem	0	*	0	0	1	*	*	0	0	0	1	*	0	S	*
	MA, B ← A; R[rs1] ← A; if (A == 0) μBr to FETCH0	0	rs1	1	0	0	1	COPY_A	1	1	0	0	*	0	EZ	FETCH0
	B ← A - B (== 0)	0	*	0	0	*	1	SUB	1	0	0	0	*	0	N	*
REVLL1:	A ← Mem	0	*	0	0	1	0	*	0	0	0	1	*	0	S	*
	Mem ← B	0	*	0	0	0	0	COPY_B	1	0	1	0	*	0	S	*
	B ← R[rs1]	0	rs1	0	1	0	1	*	0	*	0	0	*	0	N	*
	R[rs1], MA ← A if (A != 0) μBr to REVLL1	0	rs1	1	0	1	0	COPY_A	1	1	0	0	*	0	NZ	REVLL1
	μBr to FETCH0	*	*	0	0	*	*	*	0	*	0	0	*	0	J	FETCH0

Microcoding Reference Material



Arithmetic Logic Unit:

ALUOp	ALU Result Output
`COPY_A`	A
`COPY_B`	B
`INC_A_1`	A+1
`DEC_A_1`	A-1
`INC_A_4`	A+4
`DEC_A_4`	A-4
`ADD`	A+B
`SUB`	A-B
`SLT`	Signed(A) < Signed(B)
`SLTU`	A < B

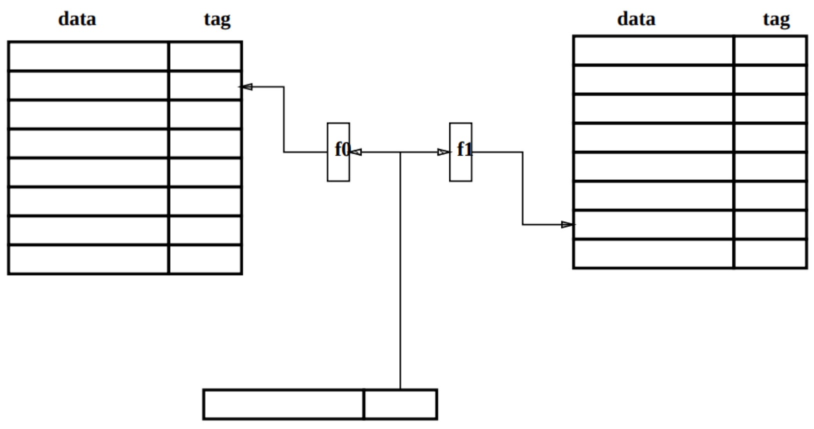
Immediate Selector:
Five immediate types are supported by **ImmSel**: `ImmI`, `ImmU`, `ImmS`, `ImmJ`, and `ImmB`.

Microbranches:
The **µBr** column represents a 3-bit field with six possible values: `N`, `J`, `EZ`, `NZ`, `D`, and `S`.

- `N` (next): The next state is simply (*current state* + 1).
- `J` (jump): The next state is *unconditionally* the state specified in the Next State column (i.e., it's an unconditional microbranch).
- `EZ` (branch-if-equal-zero): The next state depends on the value of the ALU's *zero* output signal (i.e., a conditional microbranch). If *zero* is asserted ($zero = 1$), then the next state is that specified in the Next State column, otherwise, it is (*current state* + 1).
- `NZ` (branch-if-not-zero): This behaves exactly like EZ but instead performs a microbranch if *zero* is not asserted ($zero \neq 0$).
- `D` (dispatch): The FSM looks at the opcode and function fields in the IR and goes to the corresponding state.
- `S` (spin): The µPC stalls if *busy?* is asserted; otherwise, it goes to (*current state* +1)

Problem 2: (14 points) Skewed-Associative Caches (CS252)

A skewed-associative cache uses a hash function on the index bits of the address to determine the cache set that it belongs to. Each way of a skewed associative cache uses a unique hash function.



A line of data is mapped at distinct addresses in the distinct banks of the cache

2.A (3 points)

Describe a code sequence that demonstrates higher performance on a skewed-associative cache than on an equivalent set-associative cache.

A sequence of constant-stride accesses that have the same index bits conflicts in a regular set-associative cache, whereas the hash function of a skewed-associative cache uniformly distributes the lines among all sets.

2.B. (3 points)

Describe a code sequence that demonstrates reduced performance on a skewed-associative cache than on an equivalent set-associative cache.

An access pattern which causes only compulsory misses in a set-associative cache or consists of all different indices (such as lines in close spatial proximity) may conflict in a skewed-associative cache due to hash collisions.

2.C (4 points)
Explain why a skewed-associative cache is typically implemented with unique hashing functions for each way of the cache. In other words, what is the advantage of using a unique hashing function for each way of the cache, compared to a single hashing function for all ways?

This is the concept of inter-way dispersion, or "inter-bank dispersion" as discussed in the original Seznec paper. Two cache lines which conflict in one way of the cache would not conflict in the other ways. This avoids cache thrashing behavior.

2.D (4 points)
Explain how virtual memory aliasing can be prevented in a skewed-associative cache when part of the virtual page number is used for the index bits.

The usual anti-aliasing mechanisms for a VIPT cache also apply here:

- On a miss, probe all sets in which aliases can potentially be located. If *n* bits of the VPN overlap with the index, there are 2^n indices through which an alias may exist. Hash each possibility and evict the corresponding line if present.
- Use a physically-indexed inclusive L2 cache to store the VPN bits that are part of the L1 index or the hash itself. On an L1 miss that hits in the L2, evict the line from the L1 that the L2 entry identifies.
- Rely on the OS to ensure that all virtual pages which map to the same physical page match in the VPN bits that overlap with the cache index (page coloring).

Problem 3: (26 Points) Pipelining

Note: The solutions for the alternate version of this question are at the end.

Note: The questions in 3.B, 3.C, and 3.D can be answered independently of each other.

Consider the standard fully-bypassed 5-stage RISC pipeline.

- A simple static branch predictor which always predicts PC+4 (branch not taken) in the Fetch stage (not-taken branches do not induce bubbles)
- Unconditional direct jumps (JAL) redirect from the **Decode** stage
- Conditional branches and unconditional indirect jumps (JALR) redirect from the **Execute** stage
- Bypass paths bypass into the operand registers before the **Execute** stage (the bypass select muxes are in the **Decode** stage)

The subsections of this question will consider the execution of the following loop on this pipeline.

1	loop:	`beq a1, x0, end`
2		`lw t0, 0(a0)`
3		`sw t0, 0(a1)`
4		`addi a0, a0, 0x8`
5		`lw a1, 4(a1)`
6		`j loop`
7	end:	

3.A Basic Pipelining

3.A.i (4 points) Pipeline diagram

Complete the pipeline diagram for the first iteration of this loop (all instructions in the table).

beq	F	D	X	M	W													
lw		F	D	X	M	W												
sw			F	D	D	X	M	W										
addi				F	F	D	X	M	W									
lw						F	D	X	M	W								
j							F	D	X	M	W							
beq									F	D	X	M	w					

3.A.ii: (2 points) CPI
Compute the CPI of the loop as the number of iterations approaches infinity.

Count number of cycles between two branches in different iterations
8 / 6

3.B: Improving Performance

3.B.i: (3 points) Bypass paths
Is there a bypass path you can add to improve the CPI of this code? If so, describe the bypass, and the new CPI as the number of iterations approaches infinity? Otherwise, explain why not.

Bypass load data at the end of the M stage into the store data wire at the end of the X stage. This avoids the load-use-delay between the LW and the SW which writes that data into memory.
New CPI is 7 / 6

3.B.ii: (3 points) Load-delay slots
Instead of adding a bypass path, you decide to introduce a load-delay slot to this architecture. Describe how to modify the code to take maximum advantage of the load-delay slot, and calculate the new CPI.

Reorder the addi instruction after the load. This avoids the load-use-delay as well.
New CPI is 7 / 6

3.C CISC Instructions

Instead of adding any of the features in Q3.B, you instead decide to add support for a new instruction to improve the performance of this code sequence. The new instruction you add is MMOV: memory-memory move.

```
MMOV: memory-memory move
M[R[rs2]] = M[R[rs1]]
```

To support this new instruction, you redesign the pipeline to be IF,ID,EX,LD,ST/WB. Now stores are performed in parallel with writeback, while loads are still performed in the 4th stage.

3.C.i (3 points) Structural Hazards
Does the pipeline modification introduce new structural hazards into the machine? If so, describe the hazard, provide a code example that demonstrates the hazard, and suggest a modification that resolves this hazard with minimal performance penalty. If not, explain why.

Structural hazard on access to memory, as a two instruction sequence

SW
LW

will have both instructions trying to access the data memory through the ST, LD stages respectively.

Interlocking only when there is back-to-back SW/LW will resolve this.
Alternatively, making the data memory dual ported will remove the structural hazard.

3.C.ii (3 points) Data Hazards
After resolving the structural hazard, if any, does the pipeline modification still introduce new data hazards into the machine? If so, describe the hazard, provide a code example that demonstrates the hazard, and suggest a modification that resolves this hazard with minimal performance penalty. If not, explain why.

If interlock was suggested for resolving the structural hazard, then no data hazard is possible.
If dual-porting was suggested, then there is potential for RAW through memory. Need to bypass store data in ST stage to load data output in LD stage when store address and load address match.

3.C.iii (3 points) Control Hazards
After resolving the structural and data hazards, if any, does the pipeline modification still introduce new control hazards into the machine? If so, describe the hazard, provide a code example that demonstrates the hazard, and suggest a modification that resolves this hazard with minimal performance penalty. If not, explain why.

No control hazard, since handling of branch/jump instructions are not affected, and they are all done in the F/D/X stages.

3.C.iv (3 points) Precise exceptions
Does the pipeline still support precise exceptions after this modification? If not, provide a code example demonstrating a case where precise exceptions are not possible, and suggest an interlock to preserve precise exceptions with minimal performance penalty. If yes, explain why.

Yes, the pipeline still supports precise exceptions. Register writes and stores still happen behind the commit point. Alternatively, if the assumption that the misaligned store address detection were moved to ST as well, then the commit point would be in WB instead of MEM.
Both yes or no were accepted with sufficient justification

3.C.v (2 points) CISC CPI
Given your answers to i-iv above, what is the peak CPI the instruction sequence can achieve when modified to use your new CISC instruction?

This eliminates the load-use delay, as well as the additional cycle to retire the store.
So cycles is reduced by 2, and instructions is reduced by 1.

6/5 CPI

Problem 3: (26 Points) Pipelining

Note: The questions in 3.B, 3.C, and 3.D can be answered independently of each other.

Consider the standard fully-bypassed 5-stage RISC pipeline.

- A simple static branch predictor which always predicts PC+4 (branch not taken) in the Fetch stage (not-taken branches do not induce bubbles)
- Unconditional direct jumps (JAL) redirect from the **Decode** stage
- Conditional branches and unconditional indirect jumps (JALR) redirect from the **Execute** stage
- Bypass paths bypass into the operand registers before the **Execute** stage (the bypass select muxes are in the **Decode** stage)

The subsections of this question will consider the execution of the following loop on this pipeline.

1	`loop:`	`lw t0, 0(a0)`
2		`lw t1, 0(a2)`
3		`sw t0, 0(t1)`
4		`addi a0, a0, 0x8`
5		`lw a2, 4(a2)`
6		`bne a2, a1, loop`
7	`end:`	

3.A Basic Pipelining

3.A.i (4 points) Pipeline diagram

Complete the pipeline diagram for the first iteration of this loop (all instructions in the table).

lw	F	D	X	M	W													
lw		F	D	X	M	W												
sw			F	D	D	X	M	W										
addi				F	F	D	X	M	W									
lw						F	D	X	M	W								
bne							F	D	D	X	M	W						
lw											F	D	X	M	W			

3.A.ii (2 points) CPI
Compute the CPI of the loop as the number of iterations approaches infinity.

Count number of cycles between two branches in different iterations
10 / 6

3.B Improving Performance

3.B.i (3 points) Bypass paths
Is there a bypass path you can add to improve the CPI of this code? If so, describe the bypass, and the new CPI as the number of iterations approaches infinity? Otherwise, explain why not.

Yes. Bypass load data in M stage to store address in X stage if offset of ST instruction is decoded to be 0. This eliminates the load-use delay.
CPI is 9 / 6

3.B.ii (3 points) Load-delay slots
Instead of adding a bypass path, you decide to introduce a load-delay slot to this architecture. Describe how to modify the code to take maximum advantage of the load-delay slot, and calculate the new CPI.

Reorder the loads to remove delays after instructions 2 and 5.
New order could be 1,2,5,3,4,6
CPI is 8 / 6

3.C: CISC Instructions

Instead of adding any of the features in Q3.B, you instead decide to add support for a new instruction to improve the performance of this code sequence. The new instruction you add is SWI: store word indirect.

```
SWI: store word indirect
M[M[R[rs1]]] = rs2
```

To support this new instruction, you redesign the pipeline to be IF,ID,EX,LD,ST/WB. Now stores are performed in parallel with writeback, while loads are still performed in the 4th stage.

3.C.i (3 points) Structural hazards
Does the pipeline modification introduce new structural hazards into the machine? If so, describe the hazard, provide a code example that demonstrates the hazard, and suggest a modification that resolves this hazard with minimal performance penalty. If not, explain why.

Structural hazard on access to memory, as a two instruction sequence

SW
LW

will have both instructions trying to access the data memory through the ST, LD stages respectively.

Interlocking only when there is back-to-back SW/LW will resolve this.
Alternatively, making the data memory dual ported will remove the structural hazard.

3.C.ii (3 points) Data hazards
After resolving the structural hazard, if any, does the pipeline modification still introduce new data hazards into the machine? If so, describe the hazard, provide a code example that demonstrates the hazard, and suggest a modification that resolves this hazard with minimal performance penalty. If not, explain why.

If interlock was suggested for resolving the structural hazard, then no data hazard is possible.
If dual-porting was suggested, then there is potential for RAW through memory. Need to bypass store data in ST stage to load data output in LD stage when store address and load address match.

3.C.iii (3 points) Control hazards
After resolving the structural and data hazards, if any, does the pipeline modification still introduce new control hazards into the machine? If so, describe the hazard, provide a code example that demonstrates the hazard, and suggest a modification that resolves this hazard with minimal performance penalty. If not, explain why.

No control hazard, since handling of branch/jump instructions are not affected, and they are all done in the F/D/X stages.

3.C.iv (3 points) Precise exceptions
Does the pipeline still support precise exceptions after this modification? If not, provide a code example demonstrating a case where precise exceptions are not possible, and suggest an interlock to preserve precise exceptions with minimal performance penalty. If yes, explain why.

The pipeline can be made to support precise exceptions with minimal changes. The only difference would be that the commit point would be in the WB/ST stage, to catch bad addresses generated by the load in SWI that will be caught in the ST stage.
Both yes and no were accepted, with sufficient explanation.

3.C.v (2 points) CISC CPI
Given your answers to i-iv above, what is the peak CPI the instruction sequence can achieve when modified to use your new CISC instruction?

This eliminates the load-use delay, as well as the additional cycle to retire the store.
So cycles is reduced by 2, and instructions is reduced by 1.

8/5 CPI

Problem 4: (14 Points) Prefetching

We now consider the execution of the following C code, which sums across the columns of a 2D array stored in row-major order. Row-major order means that elements in the same row of the array are adjacent in memory, i.e., A[i][j] is next to A[i][j+1]. The array elements are 32-bit integers.

```
int A[N][M]; // N = 32, M = 256 // alternate: N = 64, M = 128
int sum = 0
for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
                sum += A[i][j];
        }
}
```

4.A (2 points) Cache design
Consider an 8-way set-associative cache with 16-byte cache lines and LRU replacement. What is the minimum number of sets that this cache needs such that this code will only produce compulsory misses?
Note: The alternate exam specified a 4-way set-associative cache with 32-byte cache lines.

If only compulsory misses are allowed, then the first N accesses (loading the first column across many cache lines) must not cause any evictions, since subsequent accesses would be to those (32 * 64) / 8same cache lines.

- Since our replacement policy is ideal, the first N / (8 ways) = 4 accesses should fall in one way of the cache.
- Each row of the matrix corresponds to (256 * 4 / 16 = 64) cache lines. So two consecutive accesses in the same column stride over 64 rows.
- Thus the first 4 accesses stride over 4 * 64 entries that map to consecutive sets.
- In total, we need **256 sets** to avoid conflicts.

- Since our replacement policy is ideal, the first N / (4 ways) = 16 accesses should fall in one way of the cache.
- Each row of the matrix corresponds to (128* 4 / 32 = 16) cache lines. So two consecutive accesses in the same column stride over 16rows.
- Thus the first 16 accesses stride over 16 * 16 entries that map to consecutive sets.
- In total, we need **256 sets** to avoid conflicts.

4.B (2 points) VIPT
Suppose the cache is virtually indexed and physically tagged. Does the number of sets you derived in 4.A introduce a virtual aliasing issue assuming a 4 KiB page size? Briefly explain.

256 sets * 16 bytes per line <= 4096 B page size, so no aliasing

256 sets * 32 bytes per line > 4096 B page size, so aliasing exists

4.C (3 points) Software prefetcher
You would like to reduce the frequency of compulsory misses in this code by adding software prefetching instructions. You measure the L1 miss penalty to be 40 cycles. When the prefetch instruction is replaced with a NOP, the first 32 iterations of the inner loop each take 50 cycles to complete. What should the OFFSET of the prefetch instruction be to maximize timeliness?

Note: The alternate version had L1 miss penalty = 60 cycles, and each iteration taking 70 cycles

```
int A[N][M]; // N = 32, M = 256 // alternate: N = 64, M = 128
int sum = 0;
for (int j = 0; j < M; j++) {
      for (int i = 0; i < N; i++) {
            prefetch(&A[i][j] + OFFSET); // prefetches from (A + M*i + j + OFFSET)
            sum += A[i][j];
      }
}
```

If the prefetcher behaved perfectly, then the inner loop would take 50 - 40 = 10 cycles per iteration. Thus, if prefetched data is expected to return in 40 cycles, then we want to fetch 40 / 10 = 4 iterations ahead. So OFFSET would be 4 * 256

If the prefetcher behaved perfectly, then the inner loop would take 70 - 60 = 10 cycles per iteration. Thus, if prefetched data is expected to return in 60 cycles, then we want to fetch 60 / 10 = 6 iterations ahead. So OFFSET would be 6 * 128

4.D (4 points) Software prefetching and virtual memory
After adding software prefetching, you notice that performance degrades significantly when running in user mode (with virtual memory) compared to running in physical mode (no virtual memory). Assume the L1 cache has a fully-associative TLB with 4 entries and LRU replacement, the page size is 4 KiB, and that array A is aligned to a page boundary. Suggest a reason why performance degrades under virtual memory and describe a potential solution that retains the same TLB design.

Technically, the only degradation would be caused by the penalty of prefetching pages off the bottom of the array. However, since TLB misses are unavoidable in this problem setup (TLB reach < array size), the prefetch instructions would have likely improved performance overall, as it might be able to prefetch into the TLB a PTE before the corresponding page is accessed.

Only if the PTW were blocking, and for some reason the prefetcher could not prefetch TLB entries, then the additional spurious PTWs off the bottom of the array would have overall degraded performance.

If the TLB reach were exactly equal to the array size, then the only degradation would be due to prefetching useless elements off the end of the array. This could be resolved by preventing prefetched TLB entries from evicting old entries, perhaps with a "Victim TLB", or by modifying SW to avoid prefetching off the bottom edge.

Note that if the TLB reach were greater than the array size, then the prefetched page would likely have no effect, as LRU replacement would prevent it from evicting the recently used useful pages.

4.E (3 points) Hardware prefetching and virtual memory
You realize this code has a very regular memory access pattern and thus is amenable to hardware prefetching. You implement a stride-based hardware prefetcher that observes L1 misses to DRAM and continues to prefetch along a sequence of regularly strided accesses. While your prefetcher behaves well when the code is run in physical mode (no virtual memory), you find that performance is abysmal when the code is run in user mode (with virtual memory). Suggest a reason why.

Discontinuity between virtual pages and physical pages.
Contiguous virtual pages are not guaranteed to map to contiguous physical pages.

Problem 5: (12 Points) Virtual Memory

Consider a page-based virtual memory system with 64-byte pages and 4-byte PTEs. The VPN is 8 bits wide. A two-level page table scheme uses the upper 4 bits of the VPN for the level 1 index and the lower 4 bits as the level 2 index. The TLB is 4-way fully-associative. A hardware page table walker (PTW) performs page table walks on a TLB miss.

Note: The alternate version of the exam had 128-byte pages with 8-byte PTEs. This only affects the answer for 5.A.

Consider the following code, which sequentially traverses every byte in the virtual address space.

```
void traverse() {
   char *t = (char *) 0x0;
   while (t < MAX_VADDR) {
     char l = *t;
     t += 1;
   }
}
```

5.A (4 points) Hierarchical page table
After executing the code above, what is the total size of the page table, and how many accesses to data memory did the PTW make? Assume that no pages are initially allocated in the page table.
L1 page table size = L2 page table size = (2^4) * PTE size
Total page table size = L1 page table size + Number of L2 page tables * L2 page table size
= (2^4 + 2^8) * PTE size (4 or 8 depending on version of the exam)

PTW makes 3 accesses per fault when the L2 page table for the VPN has not been allocated yet

- 1. Fetch PTE to L2 page table in L1 page table
- PTE is invalid, since page has not been allocated yet
- OS handles page fault, allocates L2 page table, allocates page for requested virtual page
- Jump back into user program to re-execute faulting instruction
- 2. Fetch PTE to L2 page table in L1 page table
- 3. Fetch PTE to leaf PTE in L2 page table

PTW makes 4 accesses per fault when the L2 page table for the VPN has already been allocated

- 1. Fetch PTE to L2 page table in L1 page table
- 2. Fetch leaf PTE in L2 page table
- PTE is invalid, since page has not been allocated yet
- OS handles page fault, allocates page for requested virtual page
- Jump back into user program to re-execute faulting instruction
- 3. Fetch PTE to L2 page table in L1 page table
- 4. Fetch leaf PTE in L2 page table

The first case happens 2^4 times, for the first fault for a virtual page in each L2 page table
The second case happens 2^4 * (2^4 - 1) times, for each subsequent virtual page in an already allocated L2 page table.

Total PTW accesses = 3 * 2^4 + 4 * (2^8 - 2^4)

5.B (4 points) L2 TLB
We now add a 16-entry fully-associative L2 TLB with LRU replacement. Unlike the normal "L1" TLB, the L2 TLB caches both leaf and hierarchical PTEs and is searched by the PTW before accessing data memory. How many requests does the PTW make to memory with an L2 TLB? Assume that no pages are initially allocated in the page table.

The difference is the second case from above. In the second case, the accesses to the PTE to the L2 page table would hit in the L2 TLB.

PTW makes 3 accesses per fault when the L2 page table for the VPN has already been allocated

- Fetch PTE to L2 page table in L1 page table.
    - ***This hits in the L2 TLB, since the first access to the first page in this L2 page table would have gone before us, and brought this PTE into the L2 TLB***
- 1. Fetch leaf PTE in L2 page table
- PTE is invalid, since page has not been allocated yet
- OS handles page fault, allocates page for requested virtual page
- Jump back into user program to re-execute faulting instruction
- Fetch PTE to L2 page table in L1 page table,
    - ***This hits in the L2 TLB, for the same reason as above***
- 2. Fetch leaf PTE in L2 page table

We never have to worry about evictions from the L2 TLB, since we are accessing pages sequentially.

Total PTW accesses = 3 * 2^4 + 2 * (2^8 - 2^4)

5.C (4 points) AMAT with virtual memory
Derive an expression to approximate AMAT for this code, accounting for the additional delay from page table walks and the additional delay when the OS is executing the page fault handler. Assume that no pages are initially allocated in the page table. You may reference the following variables in your formula:

- W – ways in cache
- S – sets in cache
- L – bytes per cache line
- P – bytes per page
- T – L1 hit time
- N – L1 miss penalty
- R – average PTW memory requests per TLB miss
- H – PTW L1 hit rate
- K – average time for OS to service a page fault

AMAT = Hit Time + Avg Penalty from Cache Misses + Avg Penalty from TLB misses + Avg Penalty from Page Faults

- Average penalty from cache misses = L1 miss rate * L1 miss penalty
  - L1 miss rate = **1 / L**, since only compulsory misses in sequential access pattern
- Average penalty from TLB misses = TLB miss rate * TLB miss penalty
  - TLB miss rate = **1 / P**, since only compulsory misses to the TLB in sequential access pattern
  - TLB miss penalty = Average PTW requests per TLB miss * AMAT for the PTW
    - AMAT for PTW = L1 hit time + PTW L1 miss rate * L1 miss penalty = **T + (1 - H) * N**
- Average penalty from page faults = page fault rate * page fault penalty
  - Page fault rate = 1 / P, since the first access to a page will cause a fault

Average penalty from cache misses = N / L

Average penalty from TLB misses = R * (T + (1 - H) * N) / P

Average penalty from page faults = K / P

Total AMAT = T + (N / L) + (R * (T + (1 - H) * N) / P) + (K / P)