CS152 Computer Architecture and Engineering

Midterm 1
March 1, 2021

Name: _____
SID: _____

This is a closed book, closed notes exam
80 minutes, 4 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 of 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.

Question	Point Value
1	12
2	20
3	26
4	
5	
Total	

Name: _____

Problem 1: (12 points) Iron Law

Mark whether the following modifications will cause each of the first three categories 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 per program	Cycles per instruction	Seconds per cycle
Adding a second data bus to a single-bus microcoded machine			
Adding instructions with register-operand indexing: R[rd] = R[R[rs1]] + R[R[rs2]]			
Using a software page-table-walker, instead of a hardware PTW			
Remove support for precise exceptions			
Changing the base page size from 4 KiB to 8 KiB			
Removing byte load and store instructions from the ISA			

Name: _____

Problem 2: (20 points) Microprogramming

Consider the REVLL complex instruction. This instruction reverses a linked list, where the rs1 operand to this instruction is a pointer to the first element of the linked list. This instruction has no destination register, but the instruction will zero the register specified in rs1 upon completion.

```
REVLL rs1
```

This instruction assumes that every node in the linked list has the following structure. Assume addresses are 32 bits wide in this architecture. The end of the linked list is reached when the *next* pointer has a value of 0 (NULL).

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

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:
```

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?

2.B (12 points) Microprogramming

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

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 next page.

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.

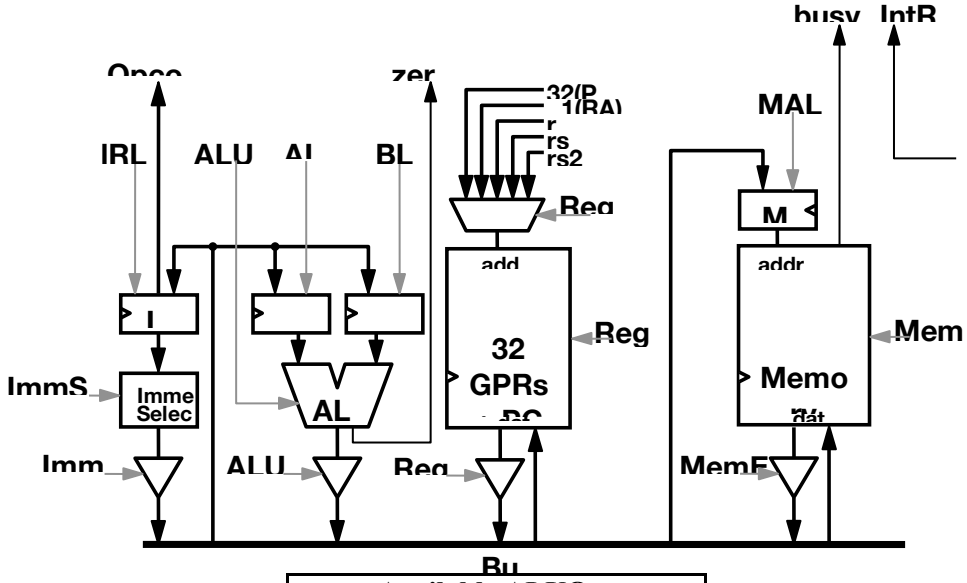
Compare the performance of your implementation with that of the unpipelined RISC-V core from 2.A, using the same memory system for both processors. Assume the unpipelined core has 4x the cycle time of the microcoded machine to accommodate the same memory latency.

Name: ____________________

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	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																
REVLL0:																

Name: _____

Microcoding Reference Material:



Available ALUOps	
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:

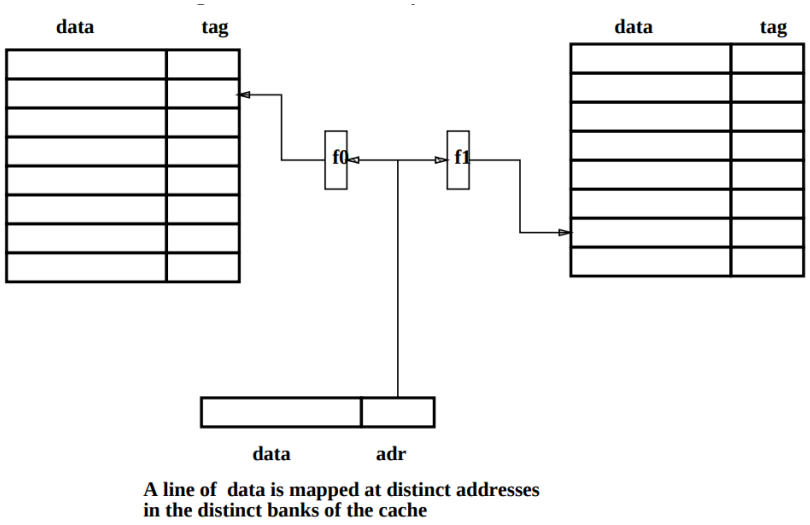
The **μBr** column represents a 3-bit field with six possible values: N, J, EZ, NZ, D, and S.

- N (next): The next μPC is simply (current μPC + 1)
- J (jump): The next μPC is unconditionally the state specified in the Next State column
- EZ (branch-if-equal-zero): The next μPC depends on the value of the ALU's zero output signal. If zero is asserted ($zero = 1$), then the next μPC is that specified in the Next State column, otherwise, it is (current μPC + 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 μPC +1).

Name: _____

Problem 2: (20 points) Skewed-Associative Caches

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



Seznec, André. "A case for two-way skewed-associative caches." ACM SIGARCH computer architecture news 21.2 (1993): 169-178.

2.A. (2 points)
Describe a code sequence that demonstrates higher performance on a skewed-associative cache than on an equivalent set-associative cache.

2.B. (2 points)
Describe a code sequence that demonstrates reduced performance on a skewed-associative cache than on an equivalent set-associative cache.

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?

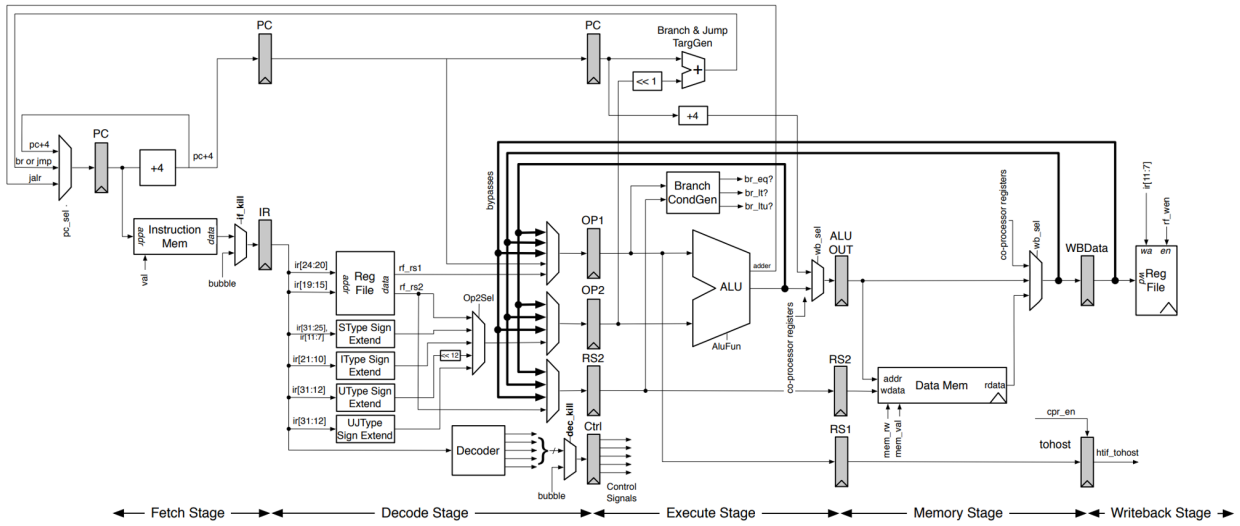
2.D (4 points)

Explain how virtual memory aliases can be prevented from co-existing within a skewed-associative cache when part of the virtual page number is used for the index bits.

Name: _____

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, depicted above. This pipeline additionally contains a simple branch predictor which always predicts PC+4 (branch not taken) in the Fetch stage. Note that branches and jumps redirect from the **Execute** stage, and that the bypass paths bypass into the operand pipeline 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																		
sw																		
addi																		
lw																		
j																		
beq																		

3.A.ii: (2 points) CPI

Compute the CPI of the loop as the number of iterations approaches infinity.

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.

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.

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.

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.

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.

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.

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?

Q4: (13 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. The array elements are 32-bit ints.

```
int A[N][M]; // N = 32, M = 256
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 this cache needs such that this code will only produce compulsory misses?

4.B: (2 points) VIPT

Suppose the cache is virtually indexed and physically tagged. Does the number of sets you came up with in 4.A introduce a virtual aliasing issue assuming a 4 KiB page size? Explain.

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 you replace the prefetch instruction with a NO-OP, the first 512 iterations of the inner loop take 50 cycles each to complete. What should the OFFSET of the prefetch instruction be to maximize timeliness?

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

4.D: (3 points) SW 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 machine mode (no virtual memory). Assume the cache has a fully-associative TLB with 4 entries and LRU replacement, the page size is 4 KiB, and that the matrix is aligned to a page boundary. Suggest a reason why performance degrades and a potential solution that retains the same TLB design.

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 machine 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.

Q5: (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.

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;
   }
}
```

4.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.

4.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 a L2 TLB?

4.C: (4 points) AMAT with Virtual Memory

Derive an equation 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. 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
- 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