Computer System Architecture
6.823 Quiz #1
March 8th, 2019

Name: _____SOLUTIONS_____

This is a closed book, closed notes exam.
80 Minutes
14 Pages (+2 Scratch)

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.
- Pages 15 and 16 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

Part A	_____	20 Points
Part B	_____	40 Points
Part C	_____	40 Points
TOTAL	_____	**100 Points**

Part A: Self-Modifying Code (20 points)

In this problem we will use the EDSACjr instruction set from Handout 1, shown in Table A-1.

Table A-1. EDSACjr instruction set

Opcode	Description
ADD *n*	Accum ← Accum + M[*n*]
SUB *n*	Accum ← Accum - M[*n*]
STORE *n*	M[n] ← Accum
CLEAR	Accum ← 0
OR *n*	Accum ← Accum | M[*n*]
AND *n*	Accum ← Accum & M[*n*]
SHIFTR *n*	Accum ← Accum shiftr *n*
SHIFTL *n*	Accum ← Accum shiftl *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*
BLT *n*	If Accum < 0 then PC ← *n*
END	Halt machine

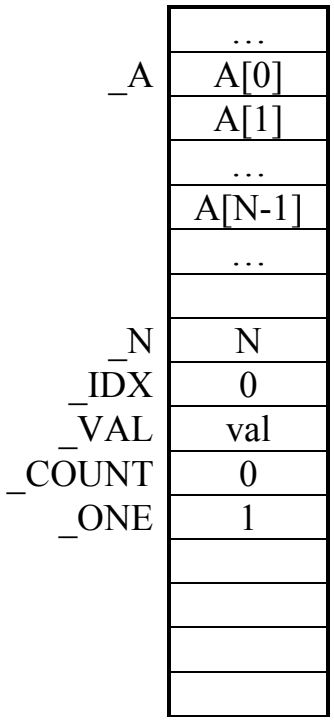
Question 1 (10 points)

Implement a program that, given a value, returns the number of entries in the N-element array A that are greater than or equal to the value. In C:

```
int A[N];
int count = 0;
for (int i = 0; i < N; i++)
    if (A[i] >= val)
        count++;
```

We have provided a portion of the code for you. To simplify your job, you can assume the code will be executed only once. We have also provided the initial memory layout. Array A is stored contiguously in memory, starting from location _A. Memory locations _N, _IDX, _VAL, and _COUNT hold the values of N, i, val, and count, respectively. Assume that N is greater than 0 (i.e., array A has at least 1 element). If you need to, you can use additional memory locations for your own variables. You should label each variable and define its initial value.

Memory:



Program:

```
        CLEAR
loadA:  ADD     _A
        SUB     _VAL
        BLT     skip

        CLEAR
        ADD     _COUNT
        ADD     _ONE
        STORE   _COUNT

skip:   CLEAR
        ADD     loadA
        ADD     _ONE
        STORE   loadA









        CLEAR
        ADD     _IDX
        ADD     _ONE
        STORE   _IDX
        SUB     _N
        BLT     loop
done:   END
```

Question 2 (10 points)

Ben Bitdiddle got an early Christmas present in the form of EDSACjr-II, which augments EDSACjr with an index register (see accompanying Handout on EDSACjr-II). Using the index register in EDSACjr-II, rewrite the program from Question 1 without using self-modifying code. For maximum points, you should use at most 12 instructions per iteration of the loop.

Memory:

	…
_A	A[0]
	A[1]
	…
	A[n-1]
	…
_N	N
_IDX	0
_VAL	val
_COUNT	0
_ONE	1
_ZERO	0

Program:

```
        LOADi   _N
        SUBi    1
loop:   LOADIX  _A
        SUB     _VAL
        BLT     skip

        CLEAR
        ADD     _COUNT
        ADD     _ONE
        STORE   _COUNT

skip:   SUBi    1
        BGEi    loop

done:   END
```

We accepted answers that used *n* in ADDi *n* and SUBi *n* as memory addresses instead of literals (i.e., ADDi *n* becomes IX ← IX + M[n]).

Part B: Virtual Memory and Caches (40 points)

Question 1 (5 points)

Consider a **direct-mapped cache** with **64-byte blocks** and **4 sets**. The table below shows a timeline of how the cache metadata (tags and valid bits) changes after a series of memory accesses. The leftmost column indicates the address of the memory access, and the rest of the row should indicate the metadata **after** the access is performed.

Fill in the table below by showing how cache metadata changes after each access. If an entry remains unchanged after the memory access, you may leave that entry blank. As an example, we have filled in the corresponding entries for the first memory access (0xA4C1).

State	Set 0		Set 1		Set 2		Set 3	
	Valid	**Tag**	**Valid**	**Tag**	**Valid**	**Tag**	**Valid**	**Tag**
Initial state	0	-	0	-	0	-	0	-
After 0xA4C1							1	0xA4
After 0x2673			1	0x26				
After 0xB51A	1	0xB5						
After 0xA4FF								
After 0x4232	1	0x42						

The 16-bit address is divided into 6 bits of block offset, 2 bits of index, and 8 bits of tag. So, we use the top 2 bits of the second to last nibble to calculate the set the access belongs to.

Access to 0xA4FF is a cache hit, so it does not change any tag or valid bit.

Question 2 (5 points)

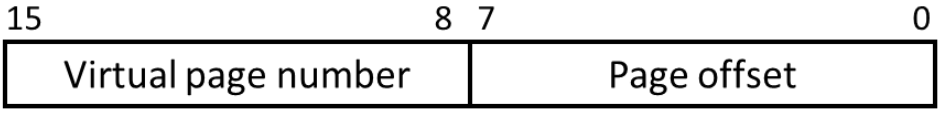
For the same memory access pattern, fill in the table below for a **2-way set-associative cache** with **128-byte blocks** and **2 sets**. Assume that the cache uses a least recently used (LRU) replacement policy (the table does not include LRU metadata). Again, we have filled in the appropriate entries for the first memory access.

State	Set 0				Set 1			
	Way 0		Way 1		Way 0		Way 1	
	Valid	Tag	Valid	Tag	Valid	Tag	Valid	Tag
Initial state	0	-	0	-	0	-	0	-
After 0xA4C1					1	0xA4		
After 0x2673	1	0x26						
After 0xB51A			1	0xB5				
After 0xA4FF								
After 0x4232	1	0x42						

LRU policy means that we kick out the older entry within Set 0, which is the entry containing the tag 0x26.

Question 3 (3 points)

Ben Bitdiddle recently bought a processor that has **16-bit virtual addresses**. The following figure shows the virtual address format:



What is the size of a page in this system?

2^8 = 256B page

Question 4 (8 points)

Ben's processor has an **8-entry direct-mapped TLB**.

Ben writes the program below, which sums the entries of matrix. A has 4 rows and 256 columns, holding 32-bit integers in row-major order (i.e., consecutive elements on the same row are in contiguous memory locations). Assume that A starts at virtual address 0x0000, and sum is already held by a register. Ignore instruction fetches.

```
int sum = 0;
for (int i = 0; i < 256; i++)
  for (int j = 0; j < 4; j++)
    sum += A[j][i];
```

(a) (4 points) How many TLB misses will this program incur?

Since the TLB has 8 entries and is direct-mapped, we use bits 8-10 for indexing into the TLB. Each row of A is 1KB or 4 pages wide, and we are accessing each page of a different row for each iteration of the inner loop. Thus, each access is 4 pages apart, meaning that the index to the TLB alternates between 0 and 4.
So access to the first row causes a miss on the third row, and vice versa (same for rows 2 and 4).
=> 256*4 = 1024 misses

(b) (4 points) How many misses would the program incur if the TLB were fully associative? Assume a least recently used (LRU) replacement policy.

Now, there is only a miss the first time you access a page of a given row. So 4 pages * 4 rows = 16 misses.

Question 5 (5 points)

Alyssa P. Hacker suggests that a larger page size would eliminate a majority of misses in the direct-mapped TLB. What should the **minimum** page size be to have **at most (i.e., ≤) 16 TLB misses** in Ben's program? Compute the new number of TLB misses for this page size.

Using 512B pages gives us bits 9-11 for the index into the TLB, which now resolves the previous conflict misses.

Question 6 (5 points)

Alyssa takes a look at Ben's program, and modifies it as follows:

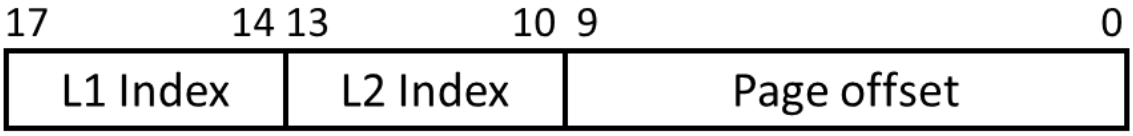
```
int sum = 0;
for (int i = 0; i < 4; i++)
  for (int j = 0; j < 256; j++)
    sum += A[i][j];
```

How many TLB misses does this program incur on the 8-entry direct-mapped TLB?

Now you are sequentially streaming through 4KB of data, so a miss only occurs when you encounter a new page. So it's 16 misses.

Question 7 (9 points)

Ben modifies the processor to use 18-bit virtual addresses and a two-level hierarchical page table. The new virtual address format is as follows:

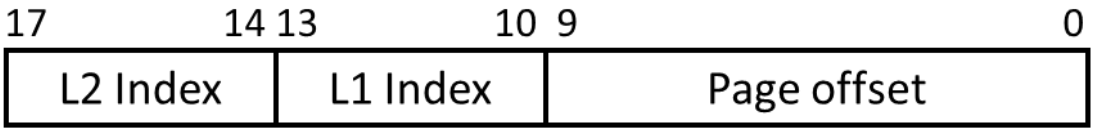


Now, assume that matrix A has 4 rows and 4096 (2^{12}) columns. We want to run Ben's program with the new matrix A on Ben's modified processor. Assume that all page tables have been swapped out to disk, and don't worry about the pages needed for code.

(a) (5 points) How many total L1 and L2 **page tables** will be resident in memory after the loop in Ben's program runs? Note that Ben's program only loops up to column 256, so it no longer traverses the entire matrix A.

Each row of A is now 16KB (4K entries * 4B), and each page is 1KB. The inner loop of Ben's program now iterates over entries 16 pages apart, which means that the L1 index will change for every access, while the L2 index remains the same. This gives us 1 L1 page table + 4 L2 page tables = 5 page tables.

(b) (4 points) Ben now swaps the L1 and L2 index bits as follows:

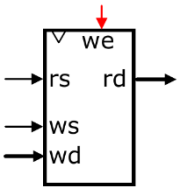


How many total L1 and L2 **page tables** will be resident in memory after Ben's loop with the new virtual address breakdown?

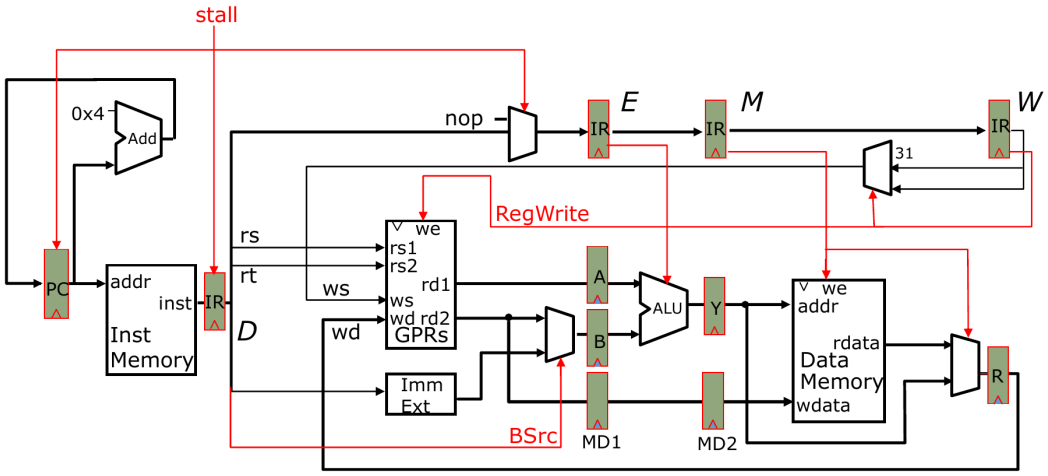
Now, the L1 index stays the same and the L2 index changes. Since we are always using the same L1 entry to find the L2 page table, we only bring in 1 L2 page table to memory. Thus, a total of 2 page tables are resident in memory.

Part C: Instruction Pipelining (40 Points)

We want to implement a 5-stage pipelined MIPS processor. Unfortunately, we are targeting a new fabrication technology that does not have register files with two read ports—we can only use register files with one read and one write port, as shown to the right.

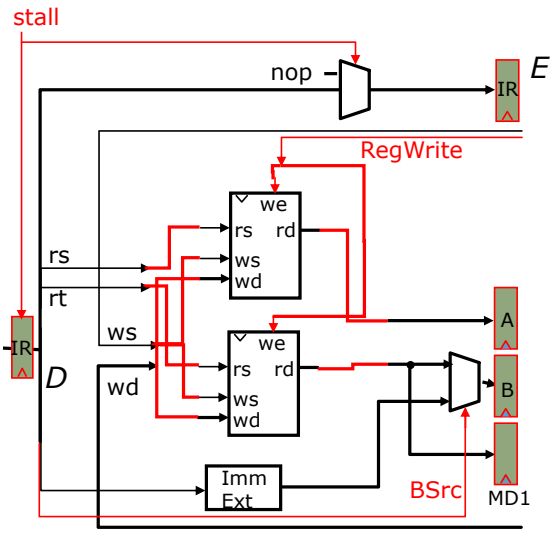


The figure below shows the standard 5-stage MIPS pipeline without bypasses, with correct stall logic, and with the usual dual-ported register file. We will develop different implementations of the decode stage to use single-port register files instead, and study their impact on performance.

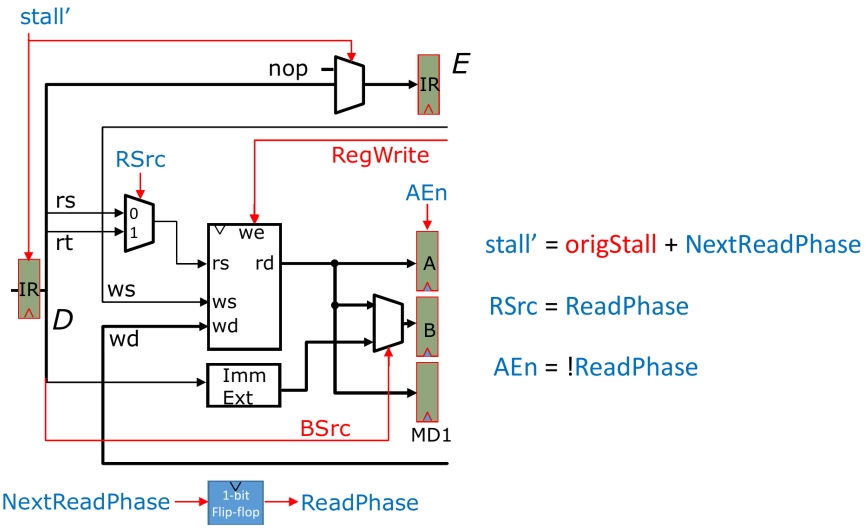


Question 1 (5 points)

Ben Bitdiddle proposes to use two register files in parallel to achieve the same behavior as a dual-ported register file. The diagram below shows an incomplete decode stage with two single-ported register files. Connect their inputs and outputs to the appropriate signals to achieve the same behavior as register file with two read ports and one write port. Your implementation should not add any new control signals.



While simple, using two register files is wasteful. Instead, Alyssa P. Hacker proposes the decode stage shown in the figure below, which uses a single register file. Changes are shown in **blue**.



This design works as follows: if the instruction reads two source registers, then decode takes two cycles to resolve the structural hazard on the register file:

- On the first cycle, the read port is used to read the first source operand, which is stored on register A. Then, the stall signal is asserted, so that the instruction spends another cycle in the decode stage.
- On the second cycle, the read port is used to read the second source operand, which is stored on registers B and MD1. To avoid losing register A's contents, A's enable signal is set to 0.

This decode stage takes two cycles only for instructions with two source registers; those with a single source register proceed without stalls (except those due to data hazards).

To achieve this behavior, this decode stage introduces a one-bit flip-flop, *ReadPhase*, which is 0 if the decode stage should read the instruction's first operand, and 1 if it should read the second operand. *ReadPhase* controls the new signals *RSrc* and *AEn* to implement the desired behavior. The control signal *NextReadPhase* sets the value of *ReadPhase* for the next cycle.

Finally, this design modifies the stall signal (now denoted *stall'*). For simplicity, it reuses the original stall signal, denoted *origStall*, and stalls if either *origStall* is 1 (due to a data hazard) or if an additional read is required. In other words, for instructions that require two input operands, the design **stalls until both operands are available in the register file**, then performs the reads.

Question 2 (5 points)

Fill in the table below to denote the types of instructions that require two read phases. Write **True** or **False** on each entry.

	ALU	ALUi	LW	SW	BEQZ/BNEZ	J/JAL	JR/JALR
NeedsTwoReadPhases	True	False	False	True	False	False	False

Question 3 (5 points)

Derive the Boolean expression for the control signal *NextReadPhase*. You can use the *NeedsTwoReadPhases* signal derived on the previous question, as well as any other control signals. Note that *NextReadPhase* is used to derive some control signals, so do not use those to avoid combinational cycles.

NextReadPhase = *NeedTwoReadPhases & !ReadPhase & !origStall*

Question 4 (10 points)

The loop below, shown in MIPS assembly and C, sums an array of integers. Complete the resource usage diagram below, showing the execution of this loop on the 5-stage pipeline with **no bypassing** and Alyssa's decode stage. Assume that branches are predicted not-taken and **resolved in the decode stage**. Use nops to denote stalls. How many cycles does each loop iteration take?

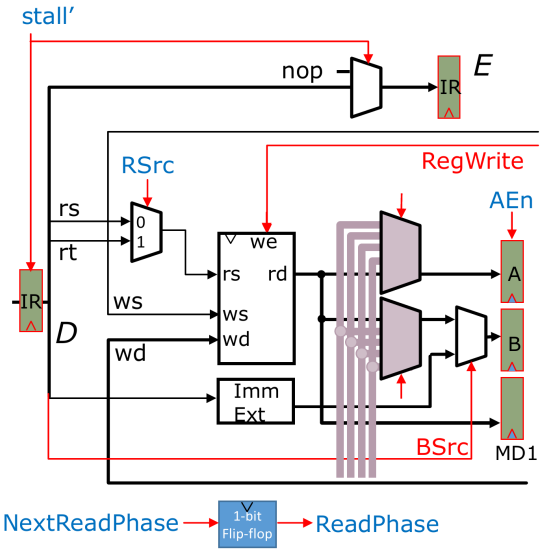
```
int array[N];
int total = 0;
for (int i = 0; i < N; i++)
   total += array[i];
```

```
# Initial values:
# R1: 0 (total)
# R2: N (loop index)
# R3: Starting address of array
loop:    LW    R4, 0(R3)
         ADD   R1, R4, R1
         ADDI  R2, R2, -1
         ADDI  R3, R3, 4
         BNEZ  R2, loop
```

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F	lw	add	addi	addi	addi	addi	addi	addi	bnez	?	?	?	lw	add
D		lw	add	add	add	add	add	addi	addi	bnez	bnez	bnez	nop	lw
E			lw	nop	nop	nop	nop	add	addi	addi	nop	nop	bnez	nop
M				lw	nop	nop	nop	nop	add	addi	addi	nop	nop	bnez
W					lw	nop	nop	nop	nop	add	addi	addi	nop	nop

Cycles per loop iteration (in steady state): _____12_____

Alyssa adds **full bypassing** to the pipeline, modifying the decode stage as shown below:



Alyssa notices that bypassing can avoid some stalls for instructions that use two source registers: if at least one of the source operands is available in a bypass path, then the single read port can be used to read the other operand, and decode does not need to stall!

Alyssa changes the control signals to implement this optimization on top of full bypassing (do not worry about how these signals are implemented). In this implementation, an instruction with two source registers stalls for a cycle (i.e., goes through two read phases) **only when none of its operands are available in bypasses**.

Question 5 (5 points)

Consider the previous loop code (shown again to the right). How many cycles does each loop iteration take on this new pipeline? Fill in the entries below, which break down the cycles per iteration spent on instructions, stalls, and branch mispredictions.

```
loop:   LW    R4, 0(R3)
        ADD   R1, R4, R1
        ADDI  R2, R2, -1
        ADDI  R3, R3, 4
        BNEZ  R2, loop
```

Note: If you'd like to use a resource usage diagram to derive these numbers, the last page of the quiz has a few copies.

Data hazard stall between LW and ADD through R4

1 cycle branch mispredict penalty since branch is resolved at decode stage

	Instructions:	5
	Stalls due to data hazards:	1
	Stalls due to read port structural hazards:	0
+	**Cycles lost to mispredicted branches:**	1
	Cycles per iteration:	7

Question 6 (10 points)

Ben specializes the code above to sum the elements of a 4-item array. His code is shown below. Assume the code runs on the new 5-stage pipeline with **full bypassing**.

```
I1:   LW    R4, 0(R3)
I2:   LW    R5, 0(R3)
I3:   LW    R6, 0(R3)
I4:   LW    R7, 0(R3)
I5:   ADD   R6, R6, R7
I6:   ADD   R4, R4, R5
I7:   ADD   R1, R6, R4
```

(a) (5 points) How many cycles does this code lose to stalls? Fill in the entries below. Use the extra resource usage diagrams in the last page to reason about the stalls if necessary.

Data hazard stall between I4 and I5 through R7
Structural hazard stall for I6 when reading R4 and R5

	Stalls due to data hazards:	**1**
+	**Stalls due to read port structural hazards:**	**1**
	Cycles lost to stalls:	**2**

(b) (5 points) Improve Ben's code by reordering instructions to minimize stalls. Write the reordered sequence of instructions (e.g., I1, I2, I3…).

The simplest solution is to swap instructions I5 and I6 – Now, I5 can bypass its values from the M and W stages, and I6 can read R4 from the register file and bypass R5 from the W stage.

Scratch Space

Use these extra pages if you run out of space or for your own personal notes. We will not grade this unless you tell us explicitly in the earlier pages.

Extra Resource Usage Diagrams

Use this as scratch space or if you need new diagrams to answer some of the questions in Part C

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F	lw	add												
D		lw												
E														
M														
W														

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F	lw	add												
D		lw												
E														
M														
W														

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F	lw	add												
D		lw												
E														
M														
W														