Computer System Architecture
6.823 Quiz #1
March 4th, 2016
Professors Daniel Sanchez and Joel Emer

Name: _____Solutions_____

This is a closed book, closed notes exam.
80 Minutes
16 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.
- 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 17 and 18 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

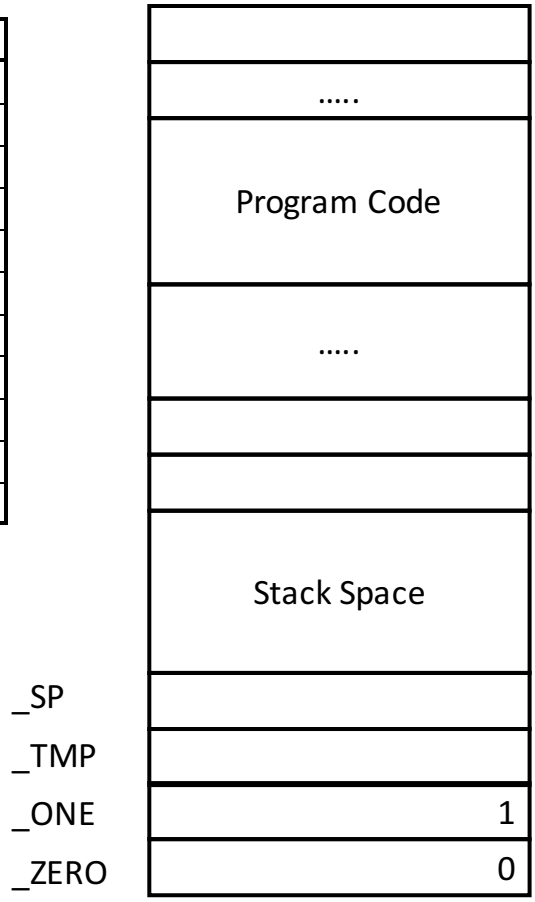
Part A	_____	25 Points
Part B	_____	40 Points
Part C	_____	35 Points
TOTAL	_____	**100 Points**

Part A: Self-modifying Code (25 points)

In this question, you will implement simple stack operations using self-modifying code on an EDSACjr machine. The memory layout is shown in the figure on the right. You have access to the named memory locations as indicated. _SP contains the current position of the stack pointer (i.e., it holds the address of the current top of the stack). You may create new local and global labels as explained in the EDSACjr handout.

Table A-1 shows the EDSACjr instruction set.

Opcode	Description	Bit Representation
ADD *n*	Accum ← Accum + M[*n*]	00001 *n*
SUB *n*	Accum ← Accum - M[*n*]	10000 *n*
STORE *n*	M[n] ← Accum	00010 *n*
CLEAR	Accum ← 0	00011 00000000000
OR *n*	Accum ← Accum | M[*n*]	00000 *n*
AND *n*	Accum ← Accum & M[*n*]	00100 *n*
SHIFTR *n*	Accum ← Accum shiftr *n*	00101 *n*
SHIFTL *n*	Accum ← Accum shiftl *n*	00110 *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*	00111 *n*
BLT *n*	If Accum < 0 then PC ← *n*	01000 *n*
END	Halt machine	01010 00000000000



You may also use the following macros if required.

Macro	Description
STOREADR *n*	Replace the address field of location *n* with the contents of the accumulator
LOADADR *n*	Load the address field of location *n* into the accumulator

Question 1 (10 points)

Write a macro for **PUSH**, which pushes the contents of the accumulator to the top of the stack. **PUSH** increments the stack pointer and stores the contents of the accumulator to the top of the stack. Implement the macro using the EDSACjr instruction set and macros provided above. You do not have to worry about stack overflow bound checking.

```
.macro PUSH
     STORE _TMP       ;; Store accumulator value in _TMP
     CLEAR            ;; clear contents of accumulator
     ADD _SP          ;; accum <- M[_SP];  Loads the current stack
                      ;; pointer in accumulator
     ADD _ONE         ;; accum <- accum + 1; Increment the stack
                      ;; pointer value
     STORE _SP        ;; M[_SP] <- accum; Store incremented stack
                      ;; pointer in _SP
     STOREADR _ST     ;; Address field of location _ST has the updated
                      ;; stack pointer
     CLEAR            ;; Clear accumulator
     ADD _TMP         ;; accum <- M[_TMP]; Restore original value in
                      ;; accumulator
_ST: STORE 0          ;; 0 will be replaced with the incremented stack
                      ;; pointer location. This will move the contents
                      ;; of the accumulator to the top of the stack
.end
```

It is possible to implement the PUSH macro without using the STOREADR macro.

Question 2 (10 points)

We will now implement the **POP** macro, which stores the contents of the top of the stack into the accumulator, and decrements the stack pointer. Implement the **POP** macro using the EDSACjr instruction set and macros provided. You can ignore empty-stack bound checks.

```
.macro POP
     CLEAR            ;; clear accumulator contents
     ADD _SP          ;; accum <- M[_SP]; accumulator holds stack
                      ;; pointer
     STOREADR _LD     ;; Address field in location _LD holds the stack
                      ;; pointer
     SUB _ONE         ;; accum <- accum – 1; decrement stack pointer
     STORE _SP        ;; M[_SP] <- accum; Store decremented stack
                      ;; pointer in _SP
     CLEAR            ;; clear contents of accumulator
_LD: ADD 0            ;; 0 will be replaced by address of stack pointer
                      ;; This will move contents of top of stack to
                      ;; accumulator
.end
```

It is possible to implement the POP macro without using the STOREADR macro.

Question 3 (5 points)

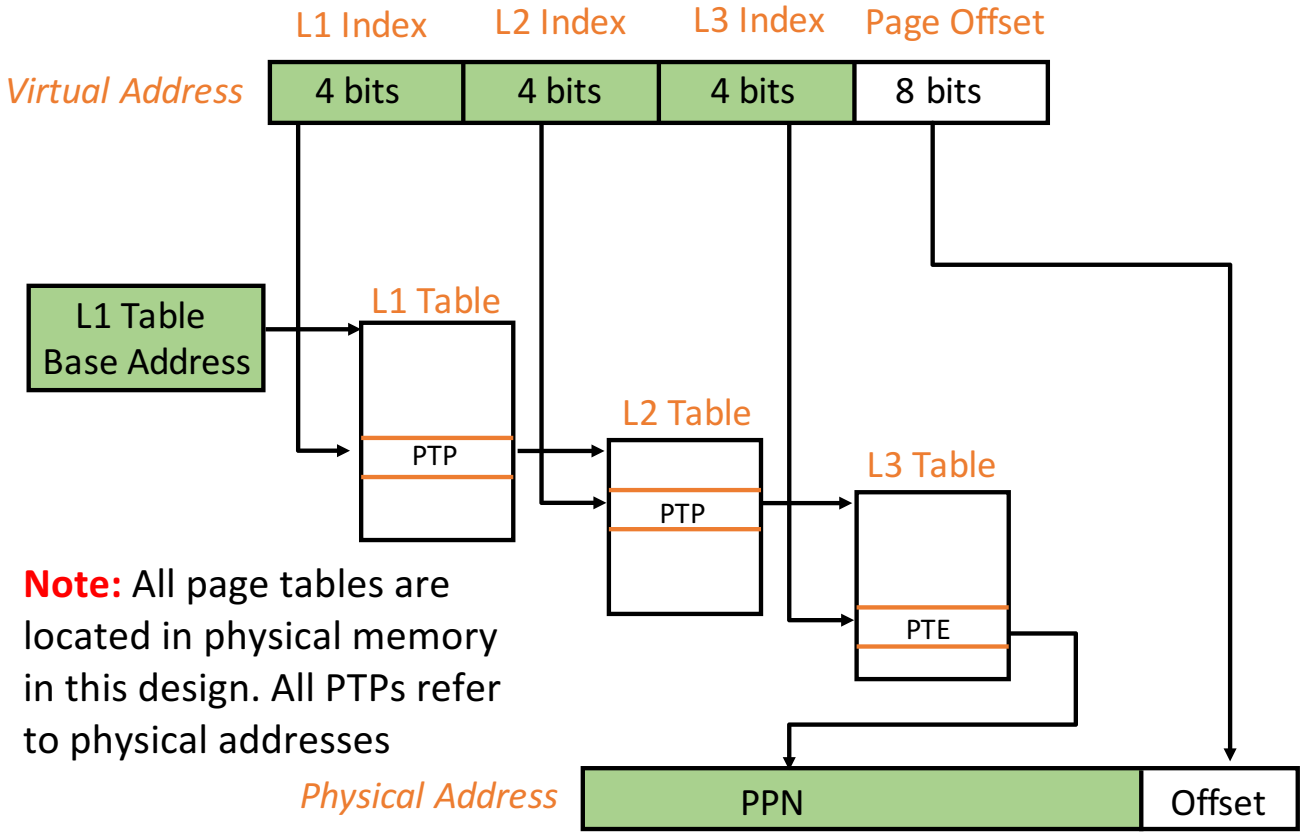
Write assembly code using the EDSACjr instruction set to pop two values from the top of the stack, and push their sum to the top of the stack. You may use the **PUSH** and **POP** macros in your code.

```
POP             ;; accum <- val1; val1 is the value in top of stack
STORE _TMP      ;; M[_TMP] <- val1;
POP             ;; accum <- val2; val2 is the next value in top of
                ;; stack
ADD _TMP        ;; acccum <- val2 + val1
PUSH            ;; top of stack contains val1 + val2
```

Part B: Caches and Virtual Memory (45 points)

Ben Bitdiddle purchases a new processor to run his 6.823 lab experiments. The processor manual informs Ben that the machine is byte-addressed with 20-bit virtual addresses and 16-bit physical addresses.

The processor manual only specifies that the machine uses a 3-level page table with the following virtual-address breakdown.



Question 1 (4 points)

What is the page size of Ben's machine? _____256 bytes_____

Demarcate the physical address into the following fields: Physical Page Number (PPN), Page Offset

PA

___8___ bits	___8___ bits
PPN	Page Offset

Ben executes the following snippet of code on his new processor. Assume integers are 4-bytes long, and the array elements are mapped to virtual addresses `0x0000` through `0x1ffc`. Assume `array` and `sum` have been suitably initialized.

```
1     int array[2048];
2     while (1) {
3       for (int i = 0; i < 4; i++)
4           sum += array[i * 256];
5     }
```

The processor manual states this machine has a TLB with 4 entries. Assume that variables `i` and `sum` are stored in registers, and ignore address translation for instruction fetches; only accesses to `array` require address translation.

Question 2 (8 points)

In steady state, how many misses from the TLB will Ben observe per iteration of the `while` loop (lines 3, 4) on average, if (state your reasoning):

a) the TLB is direct-mapped _____4_____

The array elements accessed have VPNs 0x0, 0x4, 0x8, 0xC. In a direct-mapped TLB, these map to the same index and hence the same entry in the TLB. Every access to each array element, replaces the translation already in the TLB. Hence every access results in a TLB miss in steady state.

b) the TLB is fully-associative _____0_____
(assume LRU replacement policy)

Here translations to all 4 array elements can reside in the TLB simultaneously. Hence in steady state, there are no misses in the TLB.

Question 3 (8 points)

In steady state, how many total memory accesses will Ben observe per iteration of the `while` loop (lines 3, 4) on average, if (state your reasoning):

a) the TLB is direct-mapped ____16_____

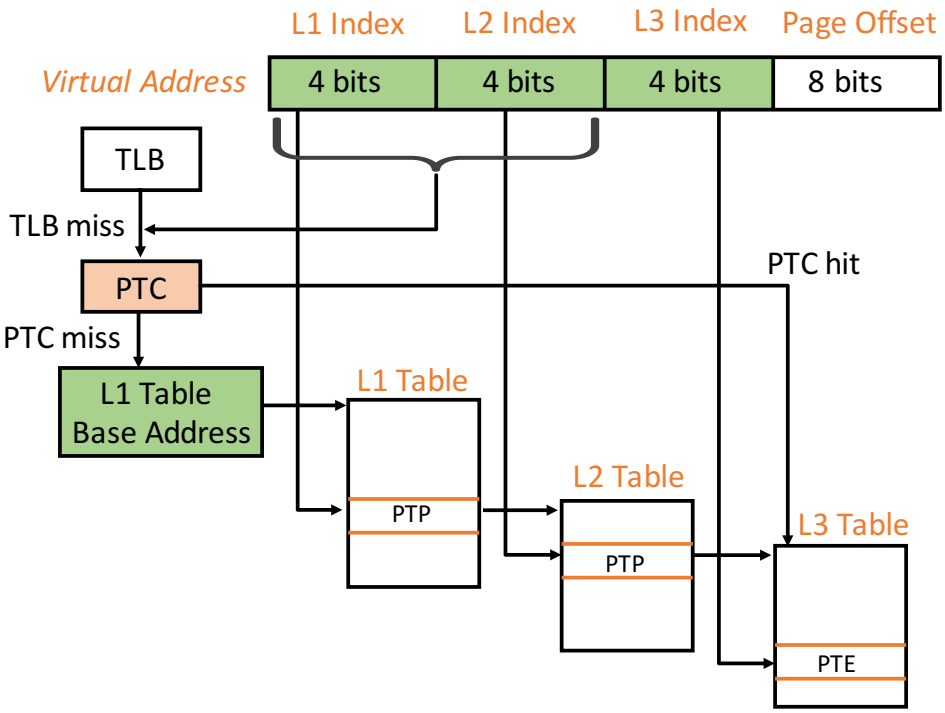
Each of the 4 misses has to do a page walk: 3 memory accesses
Plus 1 memory access for the actual data

b) the TLB is fully-associative _____4_____
(assume LRU replacement policy)

TLB accesses hit. So the only memory access is to fetch the data value.

Question 4 (10 points)

Ben wonders if he can reduce the number of memory accesses required to perform the address translations. His friend Alyssa P. Hacker suggests adding a *partial-translation cache (PTC)*, in addition to the TLB. The PTC stores a mapping of the higher-order bits of the virtual address to a L3 page table entry. If a translation misses in the TLB, but hits in the PTC, the MMU issues an access to the corresponding L3 page table directly, *skipping* the L1 and L2 page tables. On a TLB miss + PTC miss, the page walk returns the PPN and also installs a translation from VPN to L3 Page Table id in the PTC, and this incurs no additional cost.



Alyssa proposes adding a PTC with 1 entry to the processor. Does this addition benefit the code snippet in Question 2? How many total memory accesses will Ben observe now, if:

a) the TLB is direct-mapped ____8_____

The array elements have the same L1 and L2 index, but differ only in their L3 index. The PTC caches the translation from [L1 | L2] → L3. Only the very first array access does the entire page walk. The subsequent accesses (which includes all accesses in steady state) miss in the TLB, but hit in the PTC. They skip the full page walk, instead accessing only the L3 table and fetching the data value.

2 + 2 + 2+ 2 = 8

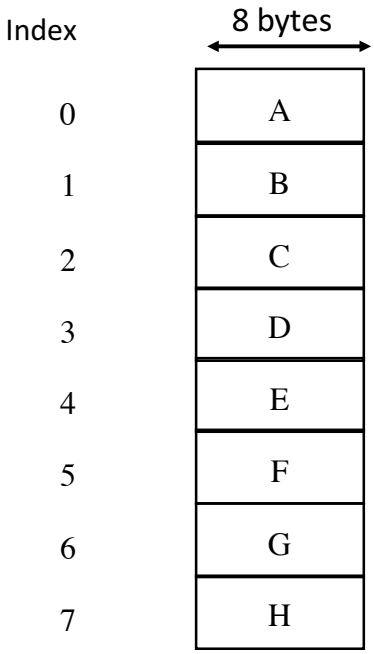
[[2 → L3 table access (1) + fetch data value (1)]]

b) the TLB is fully-associative ____4_____
(assume LRU replacement policy)

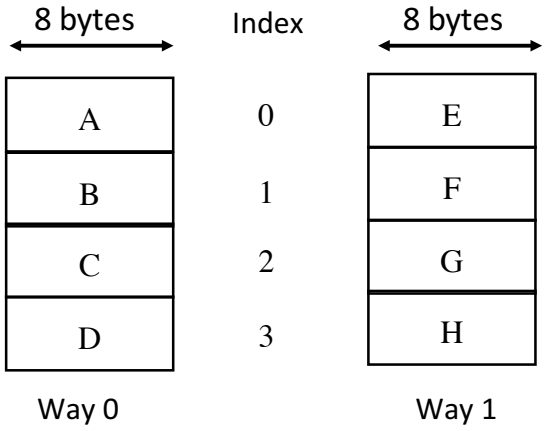
TLB accesses hit. So the only memory access is to fetch the data value.

Question 5 (10 points)

Alyssa's processor contains a 64 byte L1 cache with eight **8-byte cache blocks**, denoted A—H in the figure below. For each configuration shown in the figure, which block(s) can virtual (byte) address `0x34` be mapped to? Assume a **page size of 16 bytes**. Fill out the table at the bottom of the page, indicating each of the possible blocks by its assigned letter (A—H).



(a) Direct-Mapped Cache



(b) Two-Way Set-Associative Cache

	Virtually Indexed	Physically Indexed
Direct-mapped (a)	G	A, C, E, G
2-way set-associative (b)	C, G	A, C, E, G

Part C: Instruction Pipelining (35 points)

Consider the following MIPS code sequence:

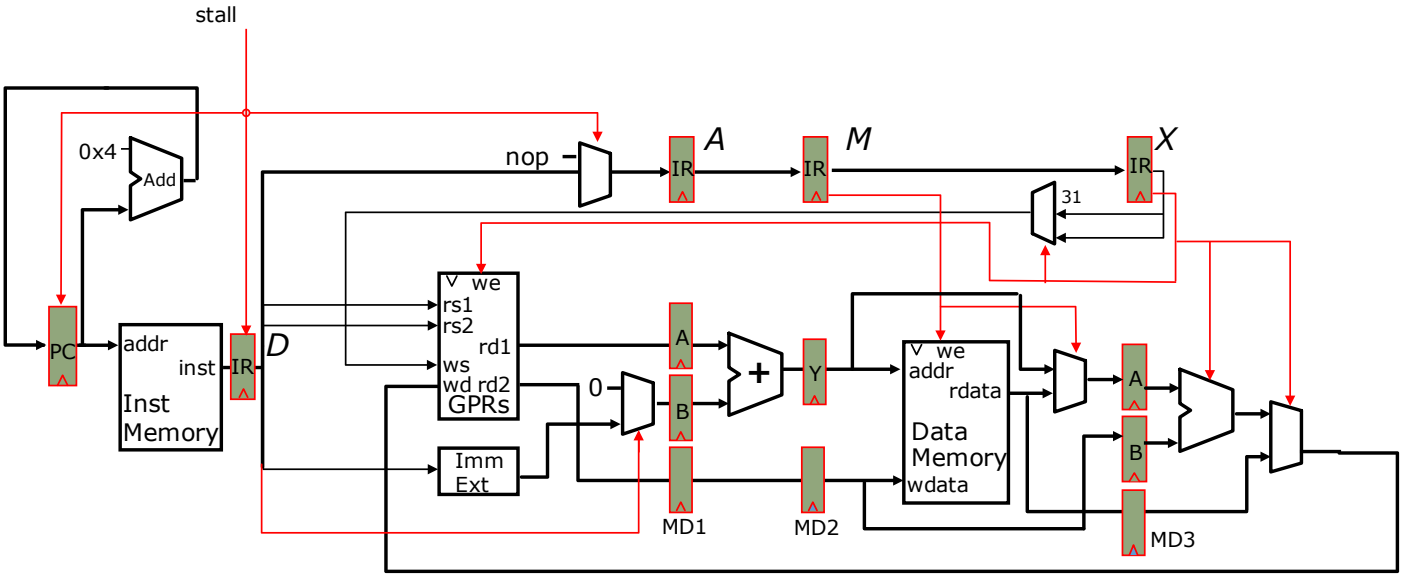
```
I1    LW    R1, 0(R3)
I2    XOR   R1, R1, R4
I3    MUL   R2, R1, R4
I4    LW    R4, 5(R2)
I5    XOR   R4, R4, R5
I6    SW    R2, 0(R3)
```

Question 1 (4 points)

Assume the classic 5-stage MIPS pipeline as discussed in lecture, with **full bypassing** and correct stall logic. Which instructions in the above sequence would have to stall?

I2, I5 will stall due to load-to-use hazards.

Ben is unhappy with the performance of the classic 5-stage MIPS pipeline discussed in 6.823 lectures. Ben uses the L-MIPS ISA, presented in the L-MIPS handout, and pipelines the single-cycle L-MIPS datapath in the handout as shown in the figure below. This is also a 5-stage pipeline, with the following stages: instruction fetch (F), instruction decode and register file fetch (D), address generation (A), memory access (M), and execute + write-back (X) stages. **We will ignore branches and jumps for all following questions**.



Question 2 (4 points)

Using the new class of Load-ALU instructions available in L-MIPS, rewrite the assembly sequence to produce a code sequence with minimum number of instructions. Do not change the order of any operations as you do this.

```
I1:   XORM R1, 0(R3), R4
I2:   MUL  R2, R1, R4
I3:   XORM R4, 5(R2), R5
I4:   SW   R2, 0(R3)
```

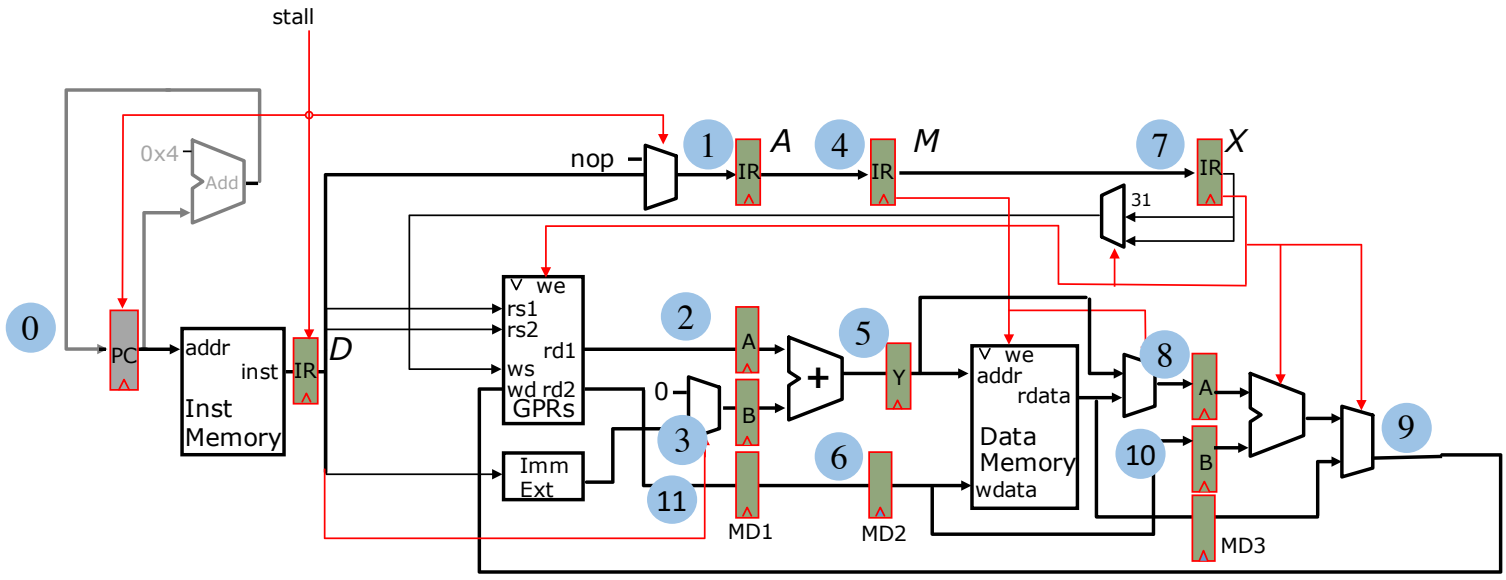
Question 3 (7 points)

Complete the instruction flow diagram for the new sequence of instructions for Ben's pipelined L-MIPS processor. **Assume no bypassing** and correct stall logic. (In case you need it, page 18 has an extra/scratch instruction flow diagram.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I1	F	D	A	M	X														
I2		F	D	D	D	D	A	M	X										
I3			F	F	F	F	D	D	D	D	A	M	X						
I4							F	F	F	F	D	A	M	X					
I5																			
I6																			
I7																			
I8																			

Question 4 (5 points)

Ben wants to improve performance by adding bypass paths to his pipeline. Help Ben by indicating which locations he needs to insert bypass multiplexers. **Ignore any bypasses needed for control-flow instructions**.



From	To
9	8
9	10
9	5
9	6
9	11
9	2

From	To
8	5
8	6
8	11
8	2

Question 5 (10 points)

Complete the instruction flow diagram for the new sequence of instructions for the L-MIPS pipeline. **Assume full bypassing** and correct stall logic this time. Use arrows to show forwarding of values from one stage to another. (In case you need it, page 18 has an extra/scratch instruction flow diagram.)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I1	F	D	A	M	X														
I2		F	D	A	M	X													
I3			F	D	D	D	A	M	X										
I4				F	F	F	D	A	M	X									
I5																			
I6																			
I7																			
I8																			

Question 6 (5 points)

Is it possible to reorder the instructions in your code sequence (without affecting correctness) to improve performance in the fully-bypassed L-MIPS pipeline? If so, give the reordered code sequence and explain why. Otherwise, briefly explain why this is not possible.

```
I1:   XORM R1, 0(R3), R4
I2:   MUL  R2, R1, R4
I3:   XORM R4, 5(R2), R5
I4:   SW   R2, 0(R3)
```

Instructions I3 and I4 in the above sequence may be re-ordered without affecting correctness. Note that both I3 and I4 have a dependence on I2. However, I3 requires the value from I2 in the decode stage (D), whereas I4 requires the value from I2 only in the address generation stage (A).

```
I1':  XORM R1, 0(R3), R4
I2':  MUL  R2, R1, R4
I3':  SW   R2, 0(R3)
I4':  XORM R4, 5(R2), R5
```

The re-ordered sequence of instructions completes one cycle earlier as shown in the diagram below. I3' proceeds to the address generation stage one cycle earlier as compared to I3. We engage suitable bypass paths (9 → 6, 9 → 2) from I2' to I3' and I4'.

	0	1	2	3	4	5	6	7	8
I1'	F	D	A	M	X				
I2'		F	D	A	M	X			
I3'			F	D	A	A	M	X	
I4'				F	D	D	A	M	X

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 Instruction Flow Diagram

Use this as scratch space or if you need a new one to answer one of the questions in Part C.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
I1	F	D	A	M	X														
I2																			
I3																			
I4																			
I5																			
I6																			
I7																			
I8																			