Computer System Architecture
6.823 Quiz #1
March 10th, 2017
Professors Daniel Sanchez and Joel Emer

Name: _____

This is a closed book, closed notes exam.
80 Minutes
18 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 19 and 20 are scratch pages. Use them if you need more space to answer one of the questions, or for rough work.

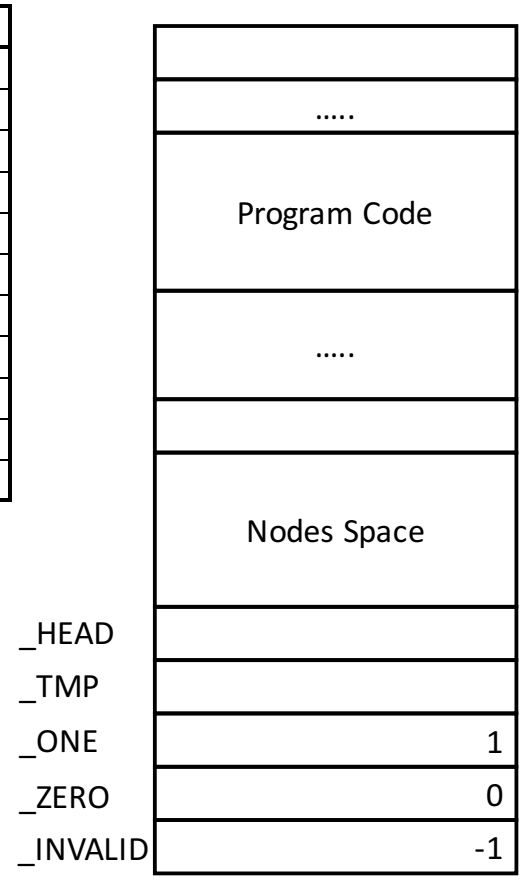
Part A	_____	25 Points
Part B	_____	35 Points
Part C	_____	40 Points
TOTAL	**_____**	**100 Points**

Part A: Self-modifying Code (25 points)

In this question, you will implement linked-list 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. Linked-list nodes consist of two words: the first is an integer `value`, the second is an address pointing to the `next` node. `_HEAD` contains the address of the first node of the list (or `_INVALID` if it is empty). The `next` field of the last node is `_INVALID`. All valid addresses are positive. 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

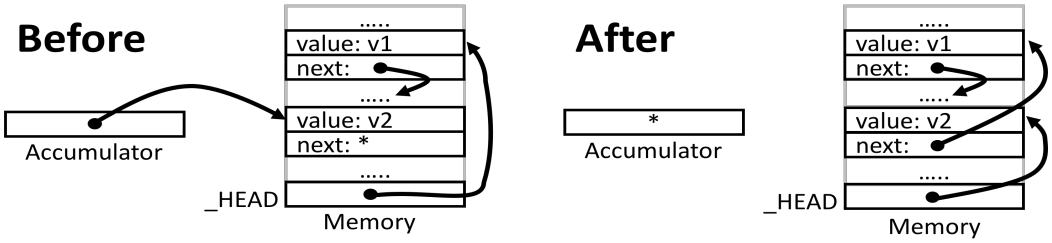


You may also use the following macros if required.

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 **LISTPUSH**, which pushes the node pointed to by the accumulator to the head of the list. **LISTPUSH** takes one argument, the memory address of the new node, which is available in the accumulator. As shown in the figure below, **LISTPUSH** stores the current `_HEAD` pointer in the new node's `next` field, and updates the `_HEAD` pointer to point to the new node. Implement the macro using the EDSACjr instruction set and macros provided above. Do not refer to "`value`" or "`next`"; they are for illustration only. You need not worry about memory allocation; the new node's address is provided in the accumulator.




```
.macro LISTPUSH
     STORE _TMP      ;; store accumulator (address of the new node)
```

```
.end
```

Question 2 (10 points)

Write a macro for **LISTPOP**, which removes the node at the head of the list and stores its address in the accumulator, or stores `_INVALID` (-1) in the accumulator if the list is empty. Implement the macro using the EDSACjr instruction set and macros provided above.

```
.macro LISTPOP
     CLEAR           ;; accumulator is not an input









.end
```

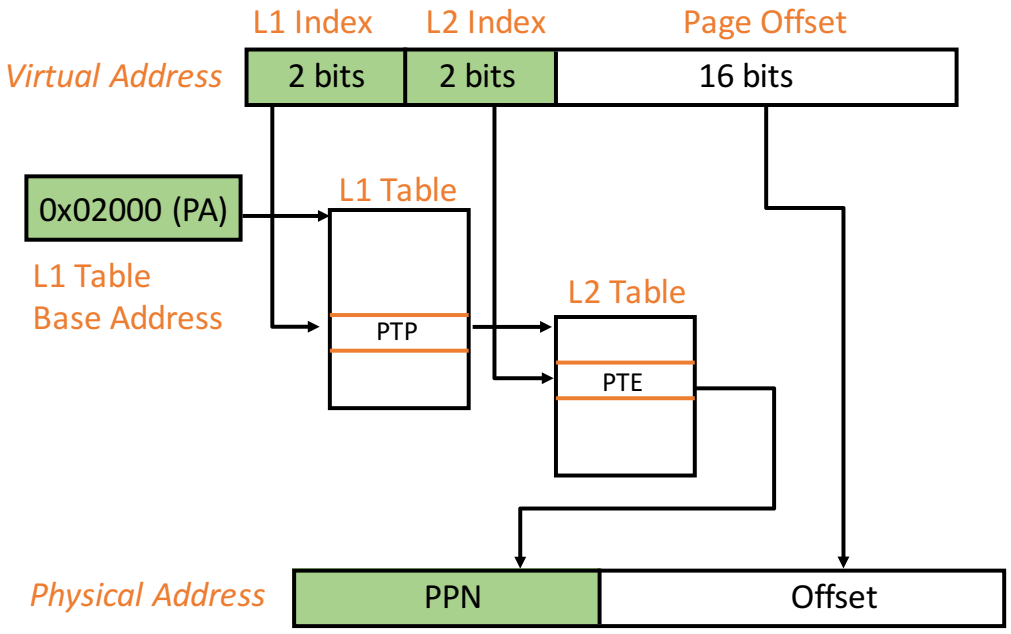
Question 3 (5 points)

Assume there exists a macro called **FREE** that takes an address as input in the accumulator and deallocates it (just like `free(void* ptr)` in C). Write a macro for **LISTCLEAR**, which uses the **FREE** macro and your **LISTPOP** macro to remove and deallocate all nodes in the list. Assume all valid node addresses are positive, or else a pointer is `_INVALID` (-1). Implement the macro using the EDSACjr instruction set and macros provided above.

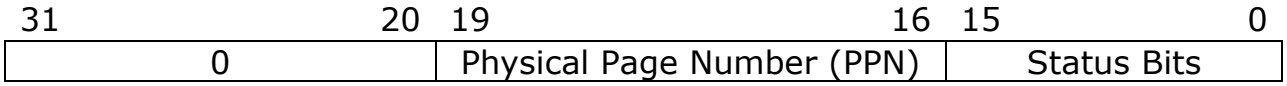
```
.macro LISTCLEAR




.end
```

Part B: Virtual Memory

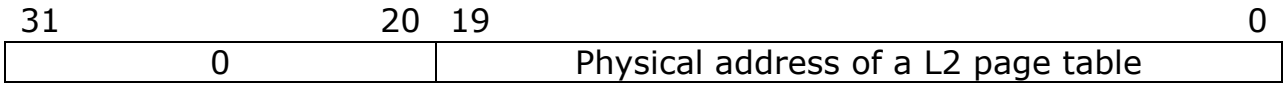
Ben Bitdiddle purchases a new processor to run his 6.823 labs. The processor manual indicates that the machine is byte-addressed with 20-bit virtual addresses and 20-bit physical addresses. The following figure summarizes the 2-level page table structure and shows the breakdown of a virtual address in this system. The physical address of the base of the Level 1 page table (0x02000) is stored in the L1 Table Base Address register. The L1 and L2 page tables are located in physical memory. The size of both L1 and L2 page table entries is 4 bytes. Each entry of the L1 page table contains the physical address of the base of each Level 2 page table (a PTP), and each of the L2 page table entries holds the PTE of the data page.



A PTE in L2 page tables can be broken into the following fields. (Don't worry about status bits).



A PTP in the L1 page table appears as follows.



Question 1 (10 points)

Assuming the initial memory state is as shown to the right, what is the physical page number (PPN) of virtual address (VA) 0xB29A0? What is the physical address (PA)? Show and explain your work for full credit. **For your convenience, we separate the page number from the offset with a colon ":".**

Virtual Address

0xB:29A0 = 0b 1011:0010100110100000

Address (PA)	
0x0:2000	0x0:2048
0x0:2004	0x0:2010
0x0:2008	0x0:2038
0x0:200C	0x0:2028
0x0:2010	0x1:0084
0x0:2014	0x5:0DA8
0x0:2018	0x6:11A0
0x0:201C	0xB:9944
0x0:2020	0xC:7FFF
0x0:2024	0x4:B000
0x0:2028	0x7:30B1
0x0:202C	0xD:2E5C
0x0:2030	0x3:A000
0x0:2034	0x6:010C
0x0:2038	0xA:74C0
0x0:203C	0x8:A524
0x0:2040	0x9:FFEE
0x0:2044	0x2:93A4
0x0:2048	0xA:74D0
0x0:204C	0x3:FD40

Snapshot of physical memory

VPN 0xB => PPN _____

VA 0xB29A0 => PA _____

Unable to run Pin in his own environment, Ben's friend, Alyssa P. Hacker, refers him to the Nested Paging handout to learn how to run his labs in a virtual machine (much to the TA's dismay!) However, Ben is frustrated by the worst-case performance. Let's find out why.

Question 2 (4 Points)

Ben starts his foray into virtualization by thinking about `gPA=>hPA` translation.

a) Assuming Ben's host physical memory has the same snapshot as in Question 1, what is the host physical address (`hPA`) of guest physical address (`gPA`) `0xB29A4`? Explain.

`gPA 0xB29A4 => hPA` _____

b) Assuming no TLB, how many accesses to host physical memory are required to access the data associated with a `gPA` (i.e., perform the `gPA=>hPA` translation and fetch the data)? Explain.

Question 3 (12 Points)

Given a guest virtual address (gVA), the first step of a nested page table walk is to load the relevant guest L1 page table PTP. This provides the base gPA of the guest L2 table. Ben is shocked at how much work is required!

a) Assume host physical memory is initialized as in Question 1 and as shown to the right, the Guest Table Base Address register holds 0xB29A0 (a gPA), and the Host Table Base Address register holds 0x02000 (a hPA). During a nested page table walk of guest virtual address (gVA) 0x61EAC, what are the contents of the guest L1 page table PTP entry? **For your convenience, we separate the page number from the offset with a colon ":".**

Address (PA)	
0x0:2000	0x0:2048
0x0:2004	0x0:2010
0x0:2008	0x0:2038
0x0:200C	0x0:2028
0x0:2010	0x1:0084
...	...
0x2:2998	0xD:2E5C
0x2:299C	0x3:A000
0x2:29A0	0x6:010C
0x2:29A4	0xA:74CC
0x2:29A8	0x7:30B1
...	...
0x5:2994	0x6:11A0
0x5:2998	0xB:F149
0x5:299C	0xC:7BFF
0x5:29A0	0x4:B020
0x5:29A4	0xF:A120
...	...
0xA:299C	0x8:A624
0xA:29A0	0x9:FEED
0xA:29A4	0x2:93A4
0xA:29A8	0xA:7440
0xA:29AC	0x3:FD40

Snapshot of **host** physical memory

Guest Virtual Address (gVA)

0x6:1EAC = 0b 0110:0001111010101100

Guest L1 Table PTP = _____

b) Assume no TLB. Starting from some gVA, how many accesses to host physical memory are required to determine the guest L1 PTP entry of a guest virtual address (gVA)? Explain.

Question 4 (4 Points)

For the 2-level nested page table in the Nested Paging handout, assuming no TLB, how many accesses to host physical memory are required to perform a guest memory access? (i.e. given a gVA, find its corresponding hPA **and fetch the data**). Explain.

Question 5 (5 Points)

For an *M*-level hierarchical guest page table and an *N*-level hierarchical host page table, assuming no TLB, how many accesses to host physical memory are required to perform a guest memory access? (i.e. given a gVA, find its corresponding hPA **and fetch the data**). Explain.

Part C: ISA/Instruction Pipelining

In this problem we study a pipelined implementation of BigMIPS. The BigMIPS ISA is given as a handout.

Question 1: ISA Design (5 points)

The BigMIPS ISA uses consecutive 32-bit registers to form a 64-bit base address for Load Word (`LW`) and Store Word (`SW`) instructions. Could the `LW` and `SW` instruction formats incorporate an arbitrary pair of source registers to form the base address? If yes, are there any consequences? If no, why not? (One to two sentence response.)

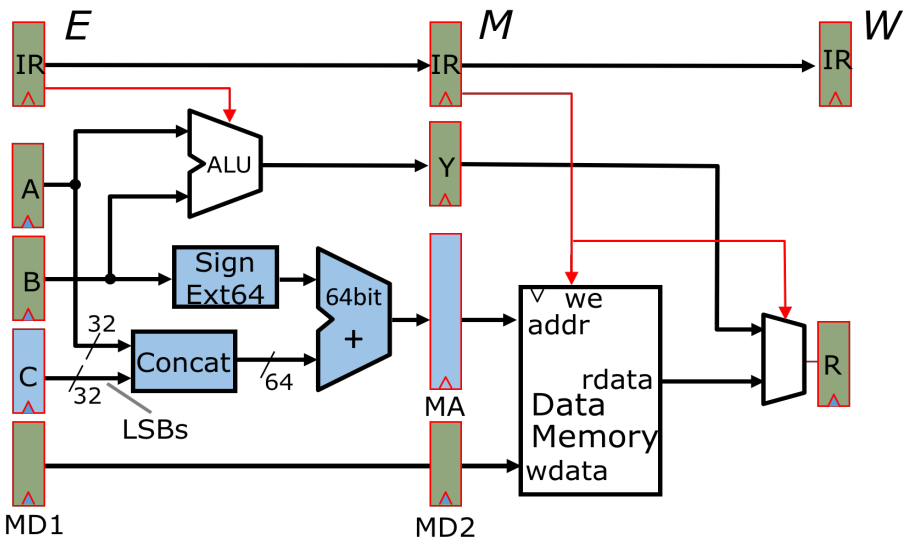


Figure C.1: Updates to the MIPS Execute and Memory stages to support the BigMIPS ISA.

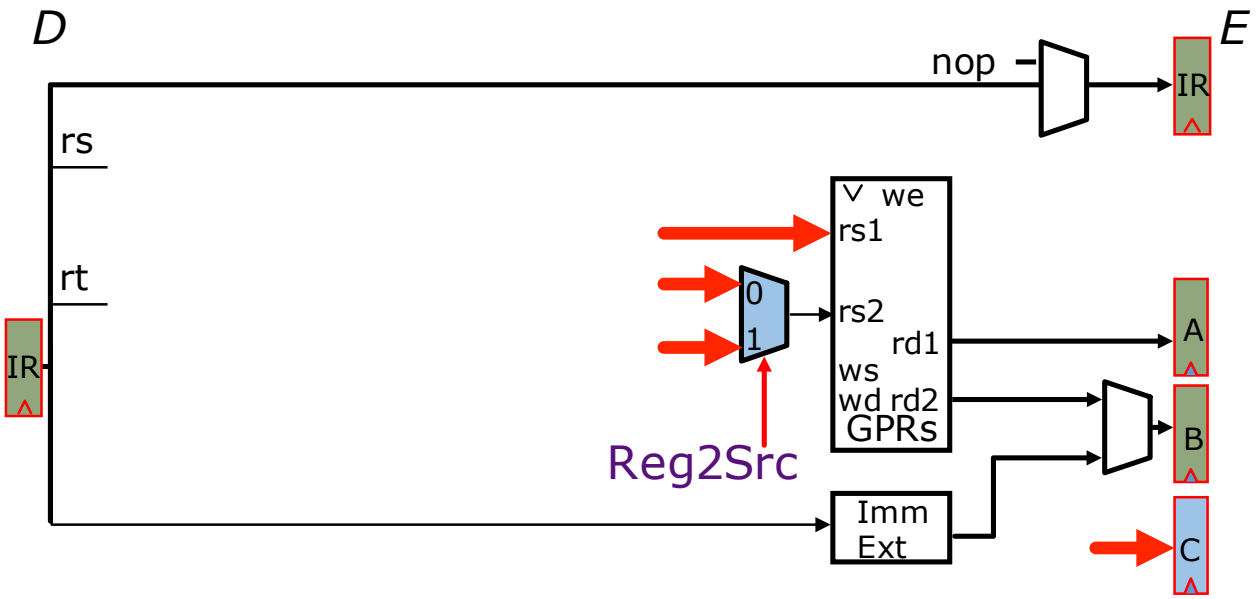
Alyssa P. Hacker extends the classic 5-stage MIPS pipeline to implement the BigMIPS ISA. She first focuses on the Execute and Memory stages, shown in Figure C.1 (we'll see the Decode stage next). The figure highlights Alyssa's additions in blue. The figure shows how she builds a 64-bit effective address for `LW` and `SW` in the Execute stage. Alyssa assumes the following about Execute-stage registers:

- *A* holds (*rs*)
- *B* holds the 32-bit extended offset
- C holds (*rs+1*)
- *MD1* holds (*rt*), the data to store (**Do not worry about getting data for stores for now**)

Alyssa adds a 64-bit adder that sums the base address and 64-bit extended offset. Register *C* (i.e. (*rs+1*)) makes up the 32 least significant bits (LSBs) of the base address. At the memory stage, Alyssa adds a 64-bit Memory Address register (*MA*).

Question 2: Loads (7 points)

Alyssa updates the Decode stage with a mux before *rs2*, as shown below. She wants your help to fill registers *A*, *B*, and *C* correctly.



a) In the figure above, draw the datapath so that it handles BigMIPS loads (`LW`) and all other instructions except stores. Connect sources to the inputs of *rs1*, *C*, and the inputs to the mux before *rs2* (the four thick red arrows). You are free to add combinational logic as necessary.

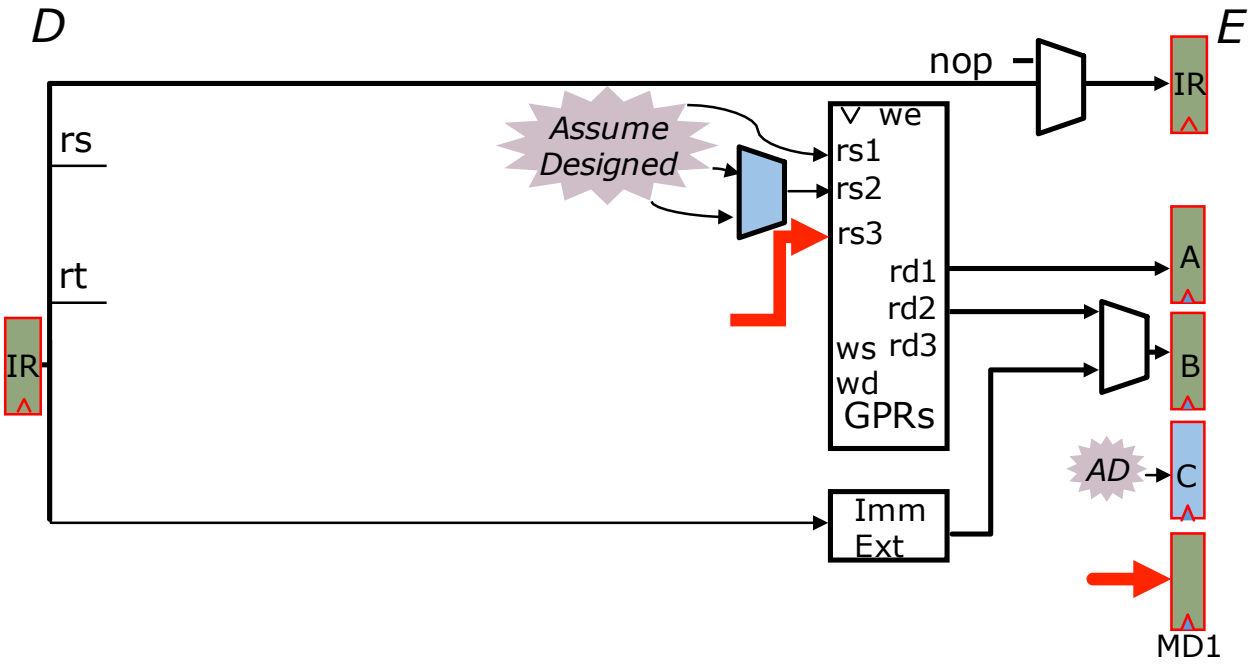
b) Derive the select signal for the mux input to *rs2*, labeled as *Reg2Src*.

You are allowed to use any internal signals (e.g., `OpCode`, `IR`, `rd1`, etc.) but not other control signals (`ExtSel`, etc.).

Reg2Src = _____

Question 3: Three Ports for Stores (5 points)

Alyssa's previous Decode stage implementation works for loads, but she realizes that stores must read three registers! (Two to form a 64-bit address and one for the data). As a first prototype, she adds a third register read port to the register file: *rs3* and *rd3*, as shown below.



In the figure above, draw the completed datapath for a store. Assume the signals from Question 2 are already designed correctly to form a 64-bit effective address. Connect sources to the inputs of *rs3* and *MD1* (the two thick red arrows). You are free to add combinational logic as necessary.

Question 4: Three-Ported BigMIPS Resource Usage (8 Points)

Consider the following BigMIPS code sequence:

```
I1    LW    R7, 8(R2)
I2    LW    R5, 24(R6)
I3    ADD   R1, R5, R7
I4    SW    R1, 8(R8)
```

Complete the instruction flow diagram for this sequence of instructions. Assume used registers have been properly initialized. **Assume full bypassing** and correct stall logic to handle all features of BigMIPS. Use arrows to show forwarding of values from one stage to another. (In case you need it, page 21 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	E	M	W														
I2																			
I3																			
I4																			
I5																			
I6																			

Alyssa finds that a third register read port is too expensive. Ben Bitdiddle suggests an ISA solution that doesn't need the third port. He replaces the Store Word instruction with two new instructions and adds a new register, *SMA*, that is part of architectural state. The new instructions and their formats are described in the table below.

Instruction	Format and Description

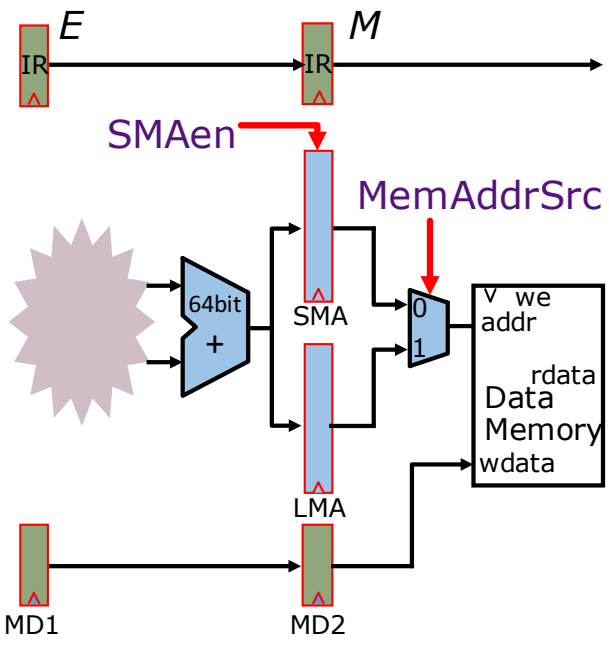
Store Word Address	`SWADDR offset(rs)`
	
	`SMA ← ((rs) << 32 | (rs+1)) + SignExt64(offset)`
	
	Calculate the 64-bit effective address as in the BigMIPS `LW` instruction. Load this address into register *SMA*. *rs* must be even.

Store Word Data	`SWDATA rt`
	
	`Mem[(SMA)] ← rt`
	
	Store the contents of register *rt* at the location addressed by *SMA*.

For Questions 5 and 6, assume that *Reg2Src* is updated appropriately for `SWADDR` and `SWDATA`.

The figure to the right shows Ben's changes to the Execute and Memory stages. The *MA* register of Figure C.1 is renamed to *LMA*. The *SMA* register is updated after the Execute stage of `SWADDR`. However, when `SWDATA` reaches the Memory stage, the *SMA* register must hold the address produced from the last `SWADDR` instruction. This behavior can be achieved through the Enable bit of the *SMA* register, *SMAen*.

With distinct memory address registers for loads and stores, Ben inserts a mux before the address field of the Data Memory, controlled by signal *MemAddrSrc*.



Question 5: Two Instructions for Stores (7 Points)

a) Derive the *SMAen* signal. Specify $\texttt{OpCode}_Z$ to indicate reading from the Z-stage *IR* register (e.g. $\texttt{OpCode}_D$ when reading the Decode stage *IR* register).

SMAen = _____

b) Derive the *MemAddrSrc* signal. Specify $\texttt{OpCode}_Z$ to indicate reading from the Z-stage *IR* register (e.g. $\texttt{OpCode}_D$ when reading the Decode stage *IR* register). Full credit will be given for use of "don't care" (*) when appropriate.

MemAddrSrc = Case *OpCode*_____

_____ => 0

_____ => 1

_____ => *

Question 6: Split-Store BigMIPS Resource Usage (8 Points)

We port the instruction sequence from Question 4 to use SWADDR and SWDATA:

```
I1    LW        R7, 8(R2)
I2    SWADDR    8(R8)
I3    LW        R5, 24(R6)
I4    ADD       R1, R5, R7
I5    SWDATA    R1
```

Complete the instruction flow diagram for this sequence of instructions. Assume used registers have been properly initialized. **Assume full bypassing** and correct stall logic to handle all features of BigMIPS. Use arrows to show forwarding of values from one stage to another. (In case you need it, page 21 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	E	M	W														
I2																			
I3																			
I4																			
I5																			
I6																			

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																			
I2																			
I3																			
I4																			
I5																			
I6																			
I7																			
I8																			