Problem M1.1: Self Modifying Code on the EDSACjr

This problem gives us a flavor of EDSAC-style programming and its limitations. Please, read Handout #1 (EDSACjr) and Lecture 2, before answering the following questions (You may find local labels in Handout #1 useful for writing self-modifying code.)

Problem M1.1.A — Writing Macros For Indirection

With only absolute addressing instructions provided by the EDSACjr, writing self-modifying code becomes unavoidable for almost all non-trivial applications. It would be a disaster, for both you and us, if you put everything in a single program. As a starting point, therefore, you are expected to write *macros* using the EDSACjr instructions given in Table H1-1 (in Handout #1) to *emulate* indirect addressing instructions described in Table M1.1-1. Using macros may increase the total number of instructions that need to be executed because certain instruction level optimizations cannot be fully exploited. However, the code size *on paper* can be reduced dramatically when macros are appropriately used. This makes programming and debugging much easier.

Please use following global variables in your macros.

```
_orig_accum:   CLEAR           ; temp. storage for accum
_store_op:     STORE 0         ; STORE template
_bge_op:       BGE 0           ; BGE template
_blt_op:       BLT 0           ; BLT template
_add_op        ADD 0           ; ADD template
```

These global variables are located somewhere in main memory and can be accessed using their labels. The `_orig_accum` location will be used to temporarily store the accumulator's value. The other locations will be used as "templates" for generating instructions.

Opcode	Description
ADDind *n*	Accum ← Accum + M[M[*n*]]
STOREind *n*	M[M[n]] ← Accum
BGEind *n*	If Accum ≥ 0 then PC ← M[n]
BLTind n	If Accum < 0 then PC ← M[n]

Table M1.1-1: Indirection Instructions

Problem M1.1.B **Subroutine Calling Conventions**

A possible subroutine calling convention for the EDSACjr is to place the arguments right after the subroutine call and pass the return address in the accumulator. The subroutine can then get its arguments by offset to the return address.

Describe how you would implement this calling convention for the special case of one argument and one return value using the EDSACjr instruction set. What do you need to do to the subroutine for your convention to work? What do you have to do around the calling point? How is your result returned? You may assume that your subroutines are in set places in memory and that subroutines cannot call other subroutines. You are allowed to use the original EDSACjr instruction set shown in Handout #1 (Table H1-1), as well as the indirection instructions listed in Table M1.1-1.

To illustrate your implementation of this convention, write a program for the EDSACjr to iteratively compute `fib(n)`, where n is a non-negative integer. `fib(n)` returns the nth Fibonacci number (`fib(0)=0, fib(1)=1, fib(2)=1, fib(3)=2`…). Make `fib` a subroutine. (The C code is given below.) In few sentences, explain how could your convention be generalized for subroutines with an arbitrary number of arguments and return values?

The following program defines the iterative subroutine `fib` in C.

```
int fib(int n) {
  int i, x, y, z;
  x=0, y=1;
  if(n<2)
    return n;
  else{
    for(i=0; i<n-1; i++){
      z=x+y;
      x=y;
      y=z;
    }
    return z;
  }
}
```

Problem M1.1.C — Recursive Subroutines

Design a calling convention for *recursive* 1 argument/1 return value subroutines using a stack. Your convention should support subroutines that can call themselves. How are the arguments passed in and how is the result returned? Again, you may use the indirection instructions defined in Table M1.1-1.

Include the code for any macros you rely upon. You may find it helpful to write the following macros (where SP is the stack pointer and is stored in the memory location _SP):

ADDstack n	accum <- accum + M[M[n]+SP]
STOREstack n	M[M[n]+SP] <- accum
ADJUSTstack n	SP <- SP+M[n]

To illustrate the use of your convention, write a paragraph explaining how you would implement a recursive `fib(n)` subroutine on the EDSACjr. (The C code for a recursive version of `fib(n)` is given below). Include diagrams of the stack as it would appear on each call to `fib(n)` when n=4.

In few sentences, explain how your convention would be generalized for procedures with a different number of arguments and a different number of returned values?

The following program defines the recursive subroutine `fib` in C.

```
int fib (int n){
  if(n<2)
    return n;
  else{
    return(fib(n-1) + fib(n-2));
  }
}
```

Problem M1.2: CISC, RISC, and Stack: Comparing ISAs

This problem requires the knowledge of Handout #2 (6.823 Stack ISA), Handout #3 (CISC ISA—x86jr), Handout #4 (RISC ISA—MIPS64), and Lectures 2 and 3. Please, read these materials before answering the following questions.

Problem M1.2.A — CISC

Let us begin by considering the following C code:

```
int b;  //a global variable

void multiplyByB(int a){
  int i, result;
  for(i = 0; i<b; i++){
    result=result+a;
  }
}
```

Using gcc and objdump on a Pentium III, we see that the above loop compiles to the following x86 instruction sequence. (On entry to this code, register `%ecx` contains `i`, and register `%edx` contains `result`, and register `%eax` contains `a`. `b` is stored in memory at location `0x08049580`.) A brief explanation of each instruction in the code is given in Handout #3.

```
           xor    %edx,%edx
           xor    %ecx,%ecx
 loop:     cmp    0x08049580,%ecx
           jl     L1
           jmp    done
 L1:       add    %eax,%edx
           inc    %ecx
           jmp    loop
 done:    ...
```

How many bytes is the program? For the above x86 assembly code, how many bytes of instructions need to be fetched if b = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem M1.2.B — RISC

Translate each of the x86 instructions in the following table into one or more MIPS64 instructions in Handout #4. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed. Assume that upon entry, R1 contains `b`, R2 contains `a`, R3 contains `i`. R4 should receive `result`. If needed, use R5 to hold the condition value and R6, R7, etc., for temporaries. You should not need to use any floating point registers or instructions in your code.

x86 instruction	label	MIPS64 instruction sequence
`xor %edx,%edx`		
`xor %ecx,%ecx`		
`cmp 0x08049580,%ecx`		
`jl L1`		
`jmp done`		
`add %eax,%edx`		
`inc %ecx`		
`jmp loop`		
...	*done:*	...

How many bytes is the MIPS64 program using your direct translation? How many bytes of MIPS64 instructions need to be fetched for b = 10 using your direct translation? How many bytes of data memory need to be fetched? Stored?

Problem M1.2.C — Stack

In a stack architecture, all operations occur on top of the stack. Only push and pop access memory, and all other instructions remove their operands from the stack and replace them with the result. The 6.823 stack-based instruction set for this question is available in Handout #2. *(Assume that the MUL instruction does not exist.)* The hardware implementation we will assume for this problem set uses stack registers for the top two entries; accesses that involve other stack positions (e.g., pushing or popping something when the stack has more than two entries) use an extra memory reference. Assume each opcode is a single byte. Offsets, constants and addresses require two bytes.

Translate the `multiplyByB` loop to the stack ISA. For uniformity, please use the same control flow as in parts **A** and **B**. Assume that when we reach the loop, `a` is the only thing on the stack. Assume `b` is now at address `B` (it fits within a 2 byte address specifier). If needed, please use `A` as the temporary memory address to hold `a`, `I` as the temporary memory address to hold `i`, and `RESULT` as the temporary memory address to hold `result`.

How many bytes is your program? Using your stack translations from part **C**, how many bytes of stack instructions need to be fetched for b = 10? How many bytes of data memory need to be fetched? Stored? If you could push and pop to/from a four-entry register file rather than memory (the Java virtual machine does this), what would be the resulting number of bytes fetched and stored?

Problem M1.2.D — Conclusions

In just a few sentences, compare the three ISAs you have studied with respect to code size, number of instructions fetched, and data memory traffic.

Problem M1.2.E — Optimization

To get more practice with MIPS64, optimize the code from part **B** so that it can be expressed in fewer instructions. Your solution should contain commented assembly code, a paragraph which explains your optimizations, and a short analysis of the savings you obtained.

Problem M1.3: Stack Architecture

This problem requires the knowledge of Handout #2 (6.823 Stack ISA) and Lecture 3. Please, read these materials before answering the following questions.

Problem M1.3.A — Program Execution on Stack Architecture

By analyzing the program in Table M1.3-1, please write the equation that was implemented on the stack architecture. Assume `A` is the memory address to hold `a`, `B` is the memory address that holds `b`, `C` is the memory address that holds `c`, `D` is the memory address that holds `d`, `E` is the memory address that holds `e`, `F` is the memory address that holds `f`, `G` is the memory address that holds `g`, and `H` is the memory address that holds `h`.

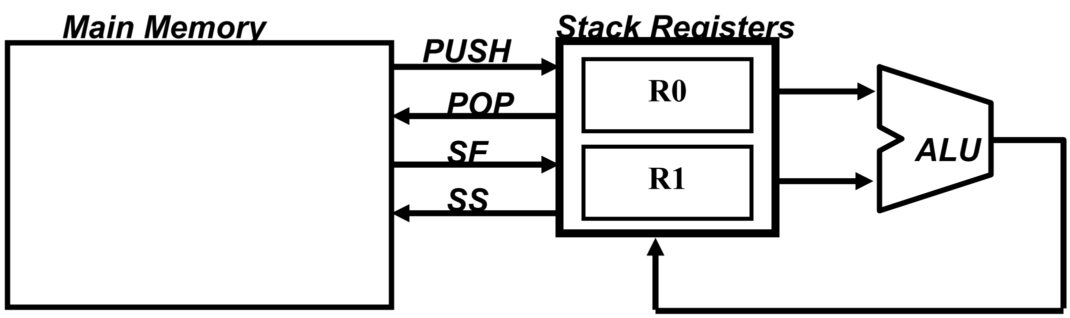
a =

PUSH	B
PUSH	C
PUSH	D
MUL	
PUSH	E
MUL	
PUSH	F
PUSH	G
MUL	
PUSH	H
ADD	
SUB	
ADD	
POP	A

Table M1.3-1

Problem M1.3.B **Optimization (1)**

The hardware implementation we assume for this question is a stack machine with two stack registers. The top 2 stack entries are **always** held in registers and the rest of the stack is held in memory. When the depth of the stack is 2 or more, a push causes a Stack Store (**SS**) to store an element of the stack to memory. When the depth of the stack is 3 or more, a pop causes a Stack Fetch (**SF**) to fetch an element of the stack from memory. All the ALU instructions have been optimized to minimize the number of memory references. For example, when the depth of stack is 3 or more, an INC operation would cause **0** (not 2) memory references.



Please complete Table M1.3-2. Do not include instruction fetches in the Number of Memory References.

Program	Stack Depth	Number of Stack Entries Stored In The Registers	Number of Memory References
PUSH B	1	1	1
PUSH C	2	2	1
PUSH D	3	2	2
MUL	2	2	
PUSH E	3	2	
MUL	2	2	
PUSH F	3	2	
PUSH G	4	2	
MUL	3	2	
PUSH H	4	2	
ADD	3	2	
SUB	2	2	
ADD	1	1	
POP A	0	0	

Table M1.3-2

Problem M1.3.C **Optimization (2)**

In this question, we want to improve performance of the stack machine from Question M1.3.B by initiating additional memory references only when the stack registers overflow or underflow. The 2 stack registers are used to hold the top 0, 1, or 2 stack entries. When the registers hold 2 stack entries, a push causes a SS to store an element of the stack to memory. When the registers hold 0 stack entries, a pop causes a SF to fetch an element of the stack from memory. In contrast to the stack machine from Part B, a pop does not cause a SF when the registers hold 1 or 2 stack entries, even when the depth of the stack is 3 or more.

Based on this optimized stack machine, please complete Table M1.3-3. Do not include instruction fetches in the Number of Memory References.

Program	Stack Depth	Number of Stack Entries Stored In The Registers	Number of Memory References
`PUSH B`	1	1	1
`PUSH C`	2	2	1
`PUSH D`	3	2	2
`MUL`	2		
`PUSH E`	3		
`MUL`	2		
`PUSH F`	3		
`PUSH G`	4		
`MUL`	3		
`PUSH H`	4		
`ADD`	3		
`SUB`	2		
`ADD`	1	1	
`POP A`	0	0	

Table M1.3-3

Problem M1.3.D **Optimization (3)**

Comparing the results in Problem M1.3.B and Problem M1.3.C, do we save any memory references by using the optimized stack machine for this particular program (Table M1.3-1)? If so, how many?

Problem M1.4: Microprogramming and Bus-Based Architectures

In this problem, we explore microprogramming by writing microcode for the bus-based implementation of the MIPS machine described in Handout #5 (Bus-Based MIPS Implementation). Read the instruction fetch microcode in Table H5-3 which was reproduced at the end of this problem (Worksheet M1-1) for readers' convenience. Make sure that you understand how different types of data and control transfers are achieved by setting the appropriate control signals before attempting this problem.

In order to further simplify this problem, *ignore* the busy signal, and assume that the memory is as fast as the register file.

The final solution should be elegant and efficient (e.g. number of new states needed, amount of new hardware added).

Problem M1.4.A — Implementing Memory-to-Memory Add

For this problem, you are to implement a new memory-memory add operation. The new instruction has the following format:

ADDm r_d, r_s, r_t

ADDm performs the following operation:

$$M[r_d] \leftarrow M[r_s] + M[r_t]$$

Fill in Worksheet M1-1 with the microcode for ADDm. Use *don't cares* (*) for fields where it is safe to use don't cares. Study the hardware description well, and make sure all your microinstructions are legal.

Please comment your code clearly. If the pseudo-code for a line does not fit in the space provided, or if you have additional comments, you may write in the margins as long as you do it neatly. Your code should exhibit "clean" behavior and not modify any registers (except r_d) in the course of executing the instruction.

Finally, make sure that the instruction fetches the next instruction (i.e., by doing a microbranch to FETCH0 as discussed above).

Problem M1.4.B Implementing DBNEZ Instruction

DBNEZ stands for Decrease Branch Not Equal Zero. This instruction uses the same encoding as conditional branch instructions on MIPS:

6	5	5	16
opcode	**rs**		**Offset**

DBNEZ decrements register **rs** by 1, writes the result back to **rs**, and branches to (**PC**+4)+**offset**, if result in **rs** is not equal to 0. Offset is sign extended to allow for backward branches. This instruction can be used for efficiently implementing loops.

Your task is to fill out Worksheet M1-2 for DBNEZ instruction. You should try to optimize your implementation for the minimal number of cycles necessary and for which signals can be set to don't-cares. You do not have to worry about the busy signal.

(Note that the microcode for the fetch stage has changed slightly from the one in the Problem M1.4.A, to allow for more efficient implementation of some instructions.)

Problem M1.4.C Instruction Execution Times

How many cycles does it take to execute the following instructions in the microcoded MIPS machine? Use the states and control points from MIPS-Controller-2 in Lecture 4 and assume Memory will not assert its busy signal.

Instruction	Cycles
`SUB  R3,R2,R1`	
`SUBI R2,R1,#4`	
`SW   R1,0(R2)`	
`BEQZ R1,label  # (R1 == 0)`	
`BNEZ R1,label  # (R1 != 0)`	
`J    label`	
`JR   R1`	
`JAL  label`	
`JALR R1`	

Which instruction takes the most cycles to execute? Which instruction takes the fewest cycles to execute?

Problem M1.4.D **Exponentiation**

Ben Bitdiddle needs to compute the power function for small numbers. Realizing there is no multiply instruction in the microcoded MIPS machine, he uses the following code to calculate the result when an unsigned number `m` is raised to the `n`th power, where `n` is another unsigned number.

```
if (m == 0) {
    result = 0;
}
else {
    result = 1;
    i = 0;

    while (i < n) {
        temp = result;
        j = 1;
        while (j < m) {
            result += temp;
            j++;
        }
        i++;
    }
}
```

The variables `i`, `j`, `m`, `n`, `temp`, and `result` are unsigned 32-bit values.

Write the MIPS assembly that implements Ben's code. Use only the MIPS instructions that can be executed on the microcoded MIPS machine (ALU, ALUi, LW, SW, J, JAL, JR, JALR, BEQZ, and BNEZ). The microcoded MIPS machine does not have branch delay slots. Use `R1` for `m`, `R2` for `n`, and `R3` for `result`. At the end of your code, only `R3` must have the correct value. The values of all other registers do not have to be preserved.

How many MIPS instructions are executed to calculate the power function? How many cycles does it take to calculate the power function? Again, use the states and control points from MIPS-Controller-2 and assume Memory will not assert its busy signal.

m, n	Instructions	Cycles
0, 1		
1, 0		
2, 2		
3, 4		
M, N		

Problem M1.4.E **Microcontroller Jump Logic**

Now we will fill in a gap in the microcontroller implementation. In the lecture on microprogramming, we did not explain the implementation of the jump logic of the microcontroller. Your task in this problem is to implement that logic. Use AND gates, OR gates and inverters to implement the combinational logic that realizes the control equations for the jump logic of the MIPS microcontroller below. The control equations for the jump logic:

μPCSrc = *Case* μJumpTypes

next	=>	**μPC+1**
spin	=>	**μPC.busy + (μPC+1).~busy**
fetch	=>	**absolute**
dispatch	=>	**op-group**
feqz	=>	**absolute.zero + (μPC+1).~zero**
fnez	=>	**absolute.~zero + (μPC+1).zero**

The selection bits for each input of the μPCSrc mux, as well as the μJumpTypes encoding are given in the tables below. Your task is to create combinational logic that translates between them, according to the control equations. Assume that the busy and zero signals follow positive logic (so they are true if the wire is carrying a 1 and false if the wire is carrying a 0). Your design will be judged on its correctness, clarity and organization. These factors are more important than the efficiency of your design.

μJumpTypes	Encoding
next	000
spin	001
feqz	110
fnez	111
fetch	010
dispatch	100

Table M1.4-1: μJumpTypes Encoding

μPCSrc	Selection bits
μPC+1	00
μPC	01
absolute	10
op-group	11

Table M1.4-2: μPCSrc Selection bits

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	ld MA	Mem W	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	N	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDM0:																

Worksheet M1-1

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Imm	µBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
DBNEZ:																

Worksheet M1-2

Problem M1.5: Fully-Bypassed Simple 5-Stage Pipeline

We have reproduced the fully bypassed 5-stage MIPS processor pipeline from Lecture 6 in Figure M1.5-A. In this problem, we ask you to write equations to generate correct bypass and stall signals. Feel free to use any symbol introduced in the lecture.

Problem M1.5.A — Stall

Do we still need to stall this pipeline? If so, explain why. (1) Write down the correct equation for the stall condition, and (2) give an example instruction sequence which causes a stall.

Problem M1.5.B — Bypass Signal

In Lecture L6, we gave you an example of bypass signal (ASrc) from EX stage to ID stage. In the fully bypassed pipeline, however, the mux control signals become more complex, because we have more inputs to the muxes in ID stage.

Write down the bypass condition for each bypass path in Mux 1. Please, indicate the priority of the signals; that is, if all bypass conditions are met, indicate which one has the highest and the lowest priorities.

Bypass $_{\text{EX->ID}}$ ASrc = $(rs_D{=}ws_E)$.we-bypass$_E$.re1$_D$ (given in Lecture L6)

Bypass $_{\text{MEM->ID}}$ =

Bypass $_{\text{WB->ID}}$ =

Priority:

Problem M1.5.C — Partial Bypassing

While bypassing gives us a performance benefit, it may introduce extra logic in critical paths and may force us to lower the clock frequency. Suppose we can afford to have only one bypass in the datapath. How would you justify your choice? Argue in favor of one bypass path over another.

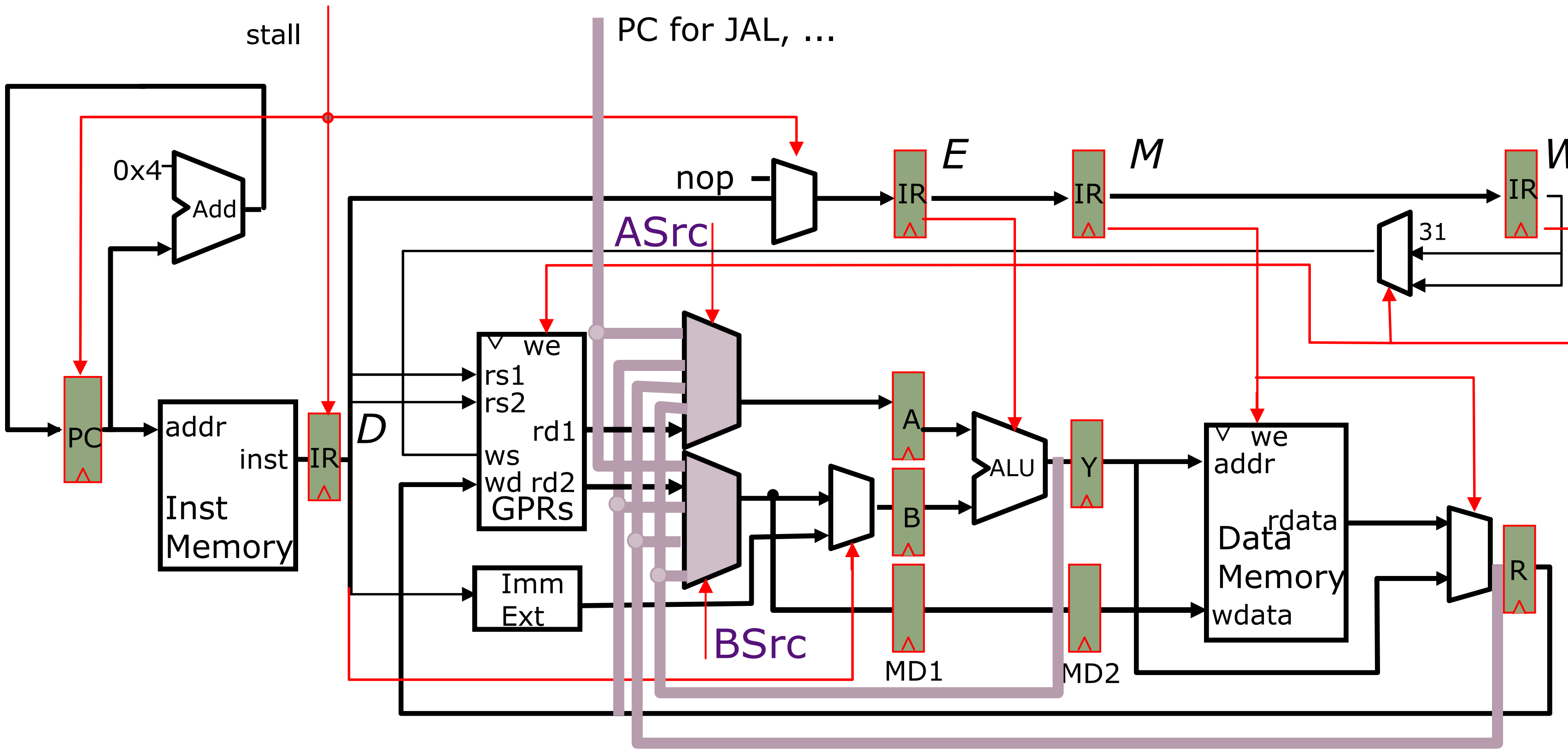


Figure M1.5-A. Fully-Bypassed MIPS Pipeline

Problem M1.6: Basic Pipelining

After having studied a single-cycle, Harvard-style (separate instruction and data memories) MIPS processor, Ben Bitdiddle decides to build a two-stage pipelined (i.e., *instruction fetch* and *execute*) **Princeton-style** architecture (shared instruction and data memory). He proposes the microarchitecture shown in the figure below. Assume our ISA contains a branch delay slot: instructions that follow branches and jumps are executed regardless of whether control flow has changed. (Note: the behavior of a branch/jump in the delay slot is undefined.)

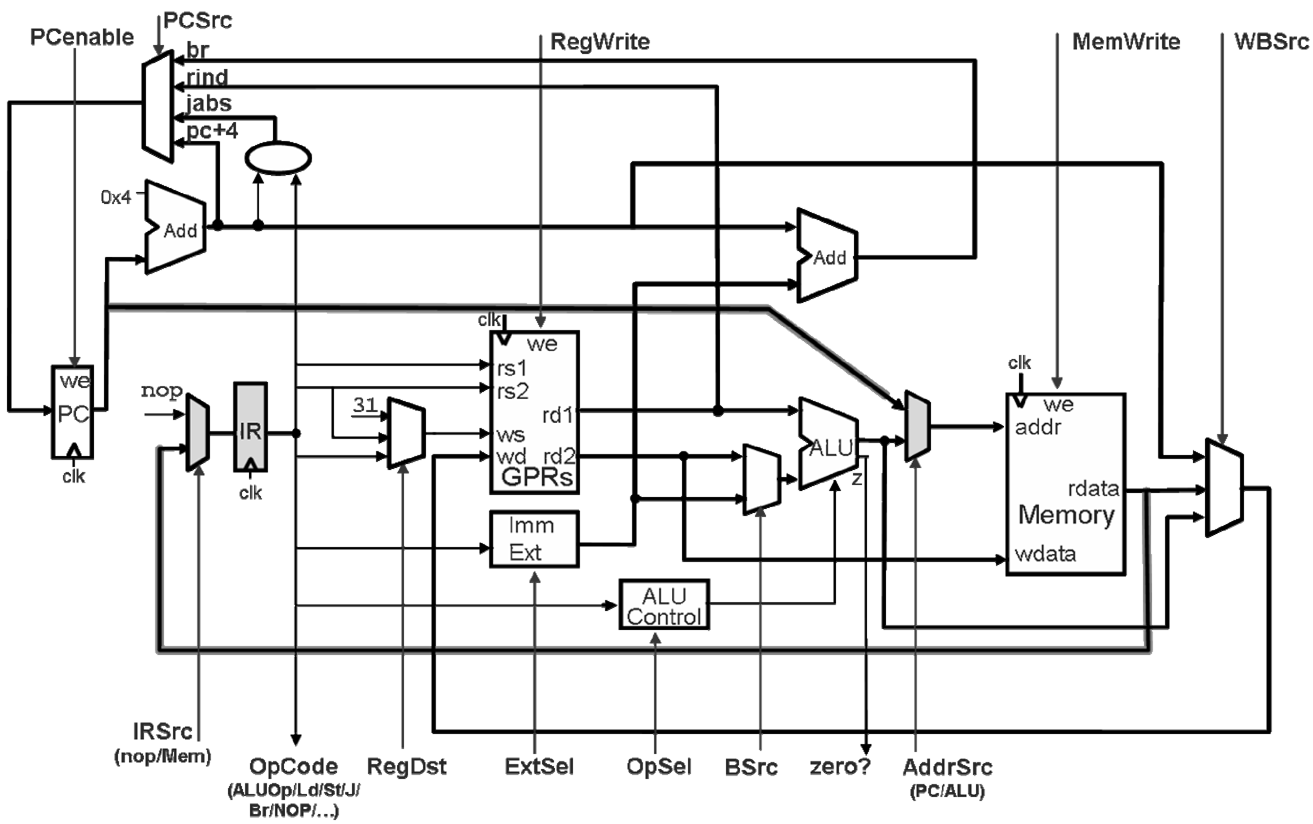


Figure M1.6-A. Two-stage pipeline, Princeton-style

Problem M1.6.A **Instruction Fetch (1)**

Help Ben determine the logic for stalling the *instruction fetch* stage assuming self-modifying code is not allowed. In one sentence explain why the pipeline might need to stall.

Problem M1.6.B **Instruction Fetch (2)**

Write the logic equation to determine the value of PCenable. (PCenable indicates whether the PC should be loaded with a new value (True) or should hold its old value (False)).

Example syntax: PCenable = (OpCode == ALUOp) or ((ALU.zero?) and (not (PC == 17)))

You may use any internal signals (e.g. OpCode, zero?, PC, IR, rd1, rdata, etc.) but may not express PCenable as a function of other control signals (e.g. ExtSel, IRSrc, PCSrc, etc.).

PCenable =

____

Problem M1.6.C **MUX Control Signals**

Fill in the blanks to complete the MUX control signals AddrSrc and IRSrc.

AddrSrc	IRSrc
AddrSrc = *Case* ____________	IRSrc = *Case* ____________
____________ => ALU	____________ => nop
____________ => PC	____________ => Mem

Problem M1.6.D

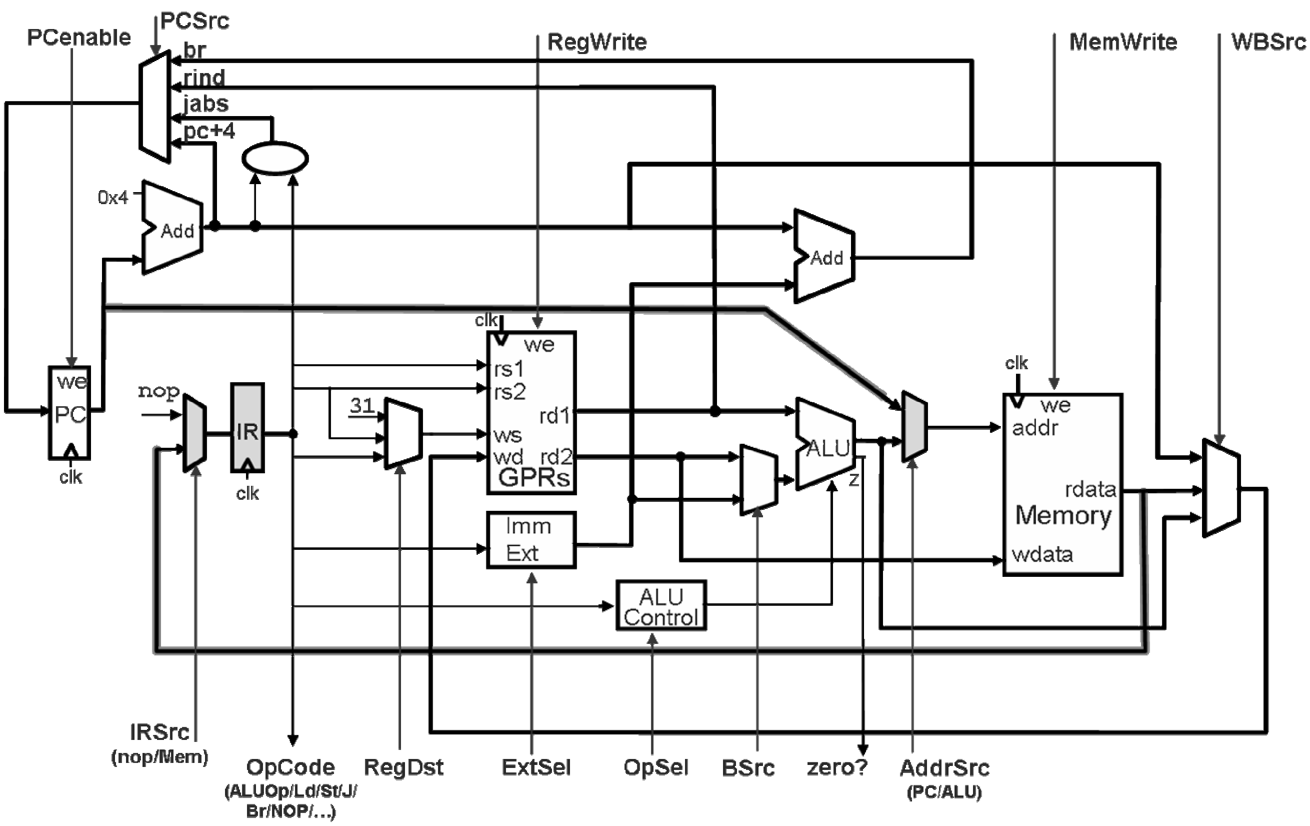
Now we are ready to put Ben's machine to the test. We would like to see a cycle-by-cycle animation of Ben's two-stage pipelined, Princeton-style MIPS machine when executing the instruction sequence below. In the following table, each row represents a snapshot of some control signals and the content of some special registers for a particular cycle. Ben has already finished the first two rows. Complete the remaining entries in the table. Use * for "don't care."

Label	Address	Instruction
I_1	100	ADD
I_2	104	LW
I_3	108	J I_7
I_4	112	LW
I_5	116	ADD
I_6	120	SUB
I_7	312	ADD
I_8	316	ADD

Time	PC	"IR"	PCenable	PCSrc1	AddrSrc	IRSrc
t_0	I_1:100	-	1	pc+4	PC	Mem
t_1	I_2:104	I_1	1	Pc+4	PC	Mem
t_2						
t_3						
t_4						
t_5						
t_6						

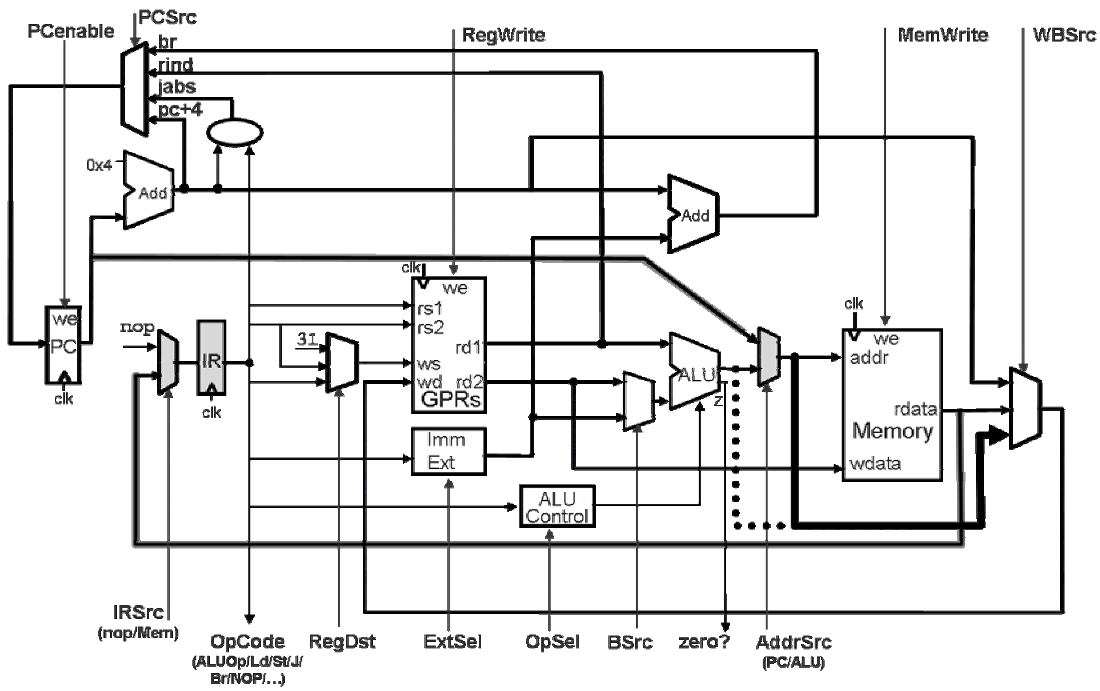
Problem M1.6.E **Self-Modifying Code**

Suppose we allow self-modifying code to execute, i.e. store instructions can write to the portion of memory that contains executable code. Does the two-stage Princeton pipeline need to be modified to support such self-modifying code? If so, please indicate how. You may use the diagram below to draw modifications to the datapath. If you think no modifications are required, explain why.



Problem M1.6.F

To solve a chip layout problem Ben decides to reroute the input of the WB mux to come from after the AddrSrc MUX rather than ahead of the AddrSrc MUX. (The new path is shown in bold, the old as a dotted line.) The rest of the design is unaltered.



How does this break the design? Provide a code sequence to illustrate the problem and explain in one sentence what goes wrong:

Problem M1.6.G **Architecture Comparison**

List one advantage of the Princeton architecture over the Harvard architecture:

List one advantage of the Harvard architecture over the Princeton architecture:

Problem M1.7: A 5-Stage Pipeline with an Additional Adder

In this problem we consider a new datapath to improve the performance of the fully-bypassed 5-stage 32-bit MIPS processor datapath given in Lecture 6 (reproduced in Figure M1.5-A). In the new datapath the ALU the Execute stage is replaced by a simple adder and the original ALU is moved from the Execute stage to the Memory stage (See Figure M1.7-A). The adder in the 3rd stage (formerly Execute) is used only for address calculations involving load/store instructions. For all other instructions, the data is simply forwarded to the 4th stage.

The ALU will now run in parallel with the data memory in the 4th stage of the pipeline (formerly Mem). During a load/store instruction, the ALU is inactive, while the data memory is inactive during the ALU instructions. *In this problem we will ignore jump and branch instructions.*

Problem M1.7.A — Elimination of a hazard

Give an example sequence of MIPS instructions (five or fewer instructions) that would cause a pipeline bubble in the original datapath, but not in the new datapath.

Problem M1.7.B — New hazard

Give an example sequence of MIPS instructions (five or fewer instructions) that would cause a pipeline bubble in the new datapath, but not in the original datapath.

Problem M1.7.C — Comparison

Compare the advantages and disadvantages of the new datapath. Which one would you recommend? Justify your choice.

IF	ID	AC	EX/MEM	WB
Instruction fetch	Instruction decode and register read	Address calculation	ALU execution and memory access	Writeback to register file

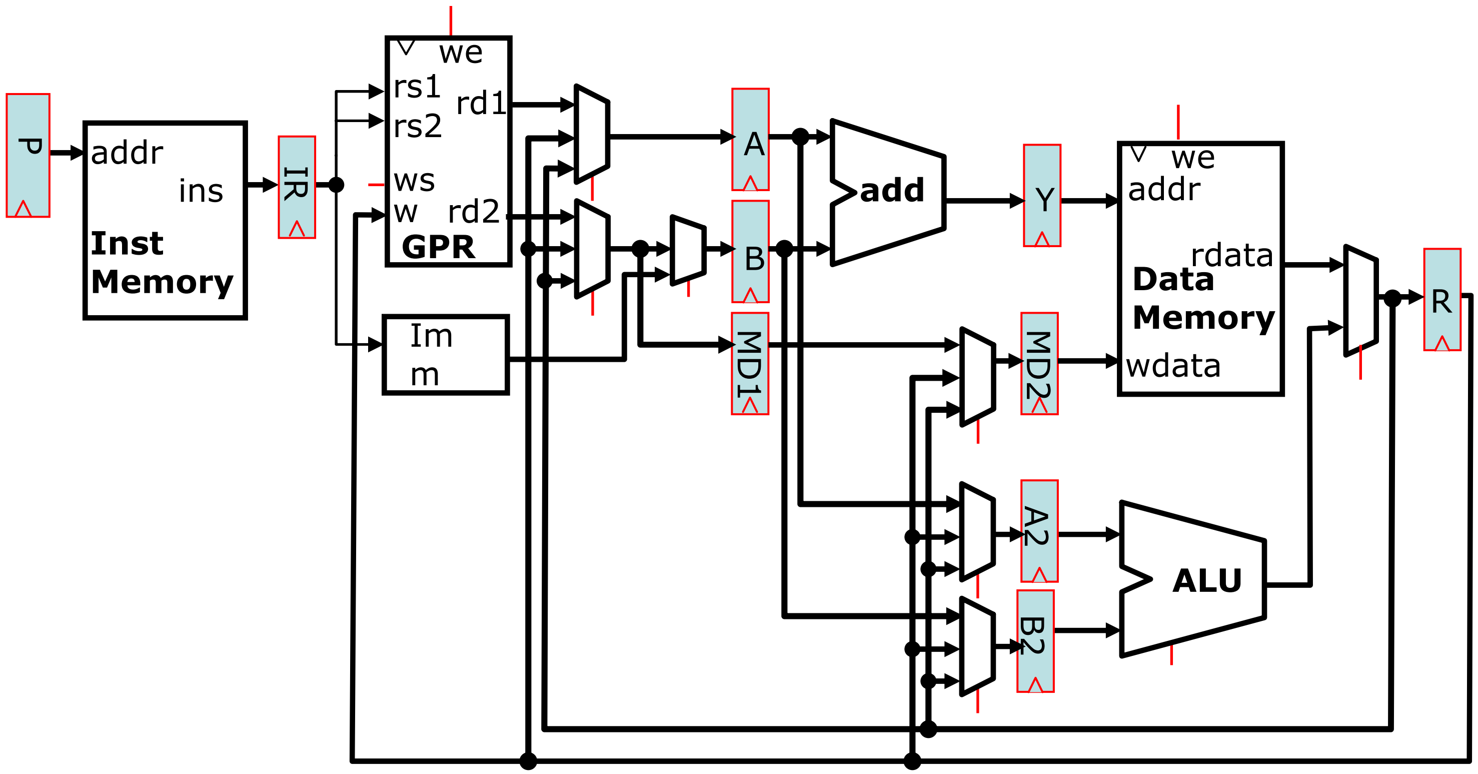


Figure M1.7-A. 5-Stage Pipeline with an Additional Adder

Problem M1.7.D **Stall Logic**

Write the stall condition (in the style of Lecture L6) for the new hazard arising from the modification to the data path. Please make use of the following signal names when writing your stall equations:

C_{dest}	C_{re}
ws = ***Case*** **opcode** **ALU ⇒ rd** **ALUi, LW ⇒ rt** **JAL, JALR ⇒ R31** **we =** ***Case*** **opcode** **ALU, ALUi, LW ⇒ (ws ≠ 0)** **JAL, JALR ⇒ on** **... ⇒ off**	**re1 =** ***Case*** **opcode** **ALU, ALUi, LW,** **SW, BZ,** **JR, JALR ⇒ on** **J, JAL ⇒ off** **re2 =** ***Case*** **opcode** **ALU, SW ⇒ on** **... ⇒ off**

Problem M1.7.E **Datapath Improvement**

Consider a MIPS ISA that only supports register indirect addressing, i.e. has no displacement (base+offset) addressing mode. Assuming the new machine only had to support this ISA, how could the datapath be improved? Draw the new datapath showing your design. (You do not have to show everything -- just the important features like pipeline registers, major components, major connections, etc.) Compare the hazards in this new datapath with the hazards in datapaths shown in Figure M1.7-A. And the original datapath in Lecture 6 (Figure M1.5-A). Justify the new datapath.

Problem M1.7.F **Displacement Addressing Synthesizing**

If the MIPS ISA did not have displacement addressing, what would programmers do? Could you still write the same programs as before? Explain.

Problem M1.7.G **Jumps and Branches**

Now we will consider jumps and branches for the pipeline shown in part A of this problem. Assume that the branch target calculation is performed in the Instruction Decode stage. In what pipeline stages can you put the logic to determine whether a conditional branch is taken? (don't worry about duplicating logic) What are the advantages and disadvantages between the different choices? For each choice, consider the number of cycles for the branch delay, any additional stall conditions, and any potential changes in the clock period.

Problem M1.8: Dual ALU Pipeline

In this problem we consider further improvements to the fully bypassed 5-stage MIPS processor pipelines presented in Lecture 6 and Problem M1.7. In this new pipeline we essentially replace the Adder in stage 3 (Figure M1.7-A) by a proper ALU with the goal of eliminating all hazards (Please see Figure M1.8-A).

The Dual ALU Pipeline has two ALUs: ALU1 is in the 3rd pipeline stage (EX1) and ALU2 is in the 4th pipeline stage (EX2/MEM). A memory instruction always uses ALU1 to compute its address. An ALU instruction uses either ALU1 or ALU2, but never both. *If an ALU instruction's operands are available (either from the register file or the bypass network) by the end of the ID stage, the instruction uses ALU1; otherwise, the instruction uses ALU2.*

In this problem, assume that the control logic is optimized to stall only when necessary. *You may ignore branch and jump instructions in this problem.*

IF	ID	EX1	EX2/MEM	WB
Instruction fetch	Instruction decode and register read	ALU1 execution and address calculation	ALU2 execution and memory access	Writeback to register file

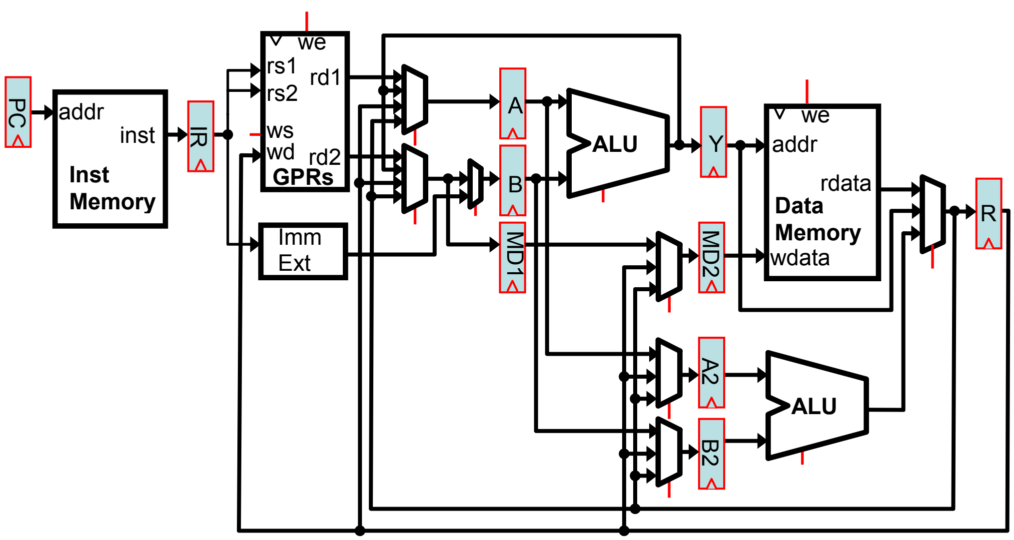


Figure M1.8-A. Dual ALU Pipeline

Problem M1.8.A — ALU Usage

For the following instruction sequence, **indicate which ALU each add instruction uses**. Assume that the pipeline is initially idle (for example, it has been executing nothing but nop instructions). Registers involved in inter-instruction dependencies are highlighted in bold for your convenience.

Instruction	ALU1 or ALU2?
`add` **r1**, r2, r3	
`lw` **r4**, 0(**r1**)	(shaded)
`add` **r5**, **r4**, r6	
`add` r7, **r5**, r8	
`add` **r1**, r2, r3	
`lw` r4, 0(**r1**)	(shaded)
`add` r5, **r1**, r6	

Problem M1.8.B — Control Signal

Fill in the equation for the control logic signal $\mathbf{alu2}_{ID}$. This signal is computed during the ID stage. It should be true if the instruction will use ALU2, or false otherwise. Like other control logic signals, **alu2** travels down the pipeline with an instruction as $\mathbf{alu2}_{EX1}$ and $\mathbf{alu2}_{EX2/MEM}$, you may use these signals in your equation if needed. In the equation, "**+**" means logical or, and "·" means logical and.

$$
\begin{aligned}
\mathbf{alu2}_{ID} = (\ & ((\mathbf{OP}_{ID} = \mathbf{ALU}) + (\mathbf{OP}_{ID} = \mathbf{ALUi})) \\
& \cdot ((\mathbf{rs}_{ID} = \mathbf{ws}_{EX1}) + (\mathbf{rt}_{ID} = \mathbf{ws}_{EX1}) \cdot \mathbf{re2}_{ID}) \\
& \cdot (\mathbf{ws}_{EX1} \neq 0) \\
& \cdot (\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\ ) \\
)\ &
\end{aligned}
$$

Problem M1.8.C — Instruction Sequences Causing Stalls

Indicate whether each of the following instruction sequences causes a stall in the pipeline. Consider each sequence separately and assume that the pipeline is initially idle (for example, it has been executing nothing but nop instructions). Registers involved in inter-instruction dependencies are highlighted in bold for your convenience.

	stall? (yes/no)
`add` **`r1`**`, r2, r3` `lw  r4, 0(`**`r1`**`)`	

`lw  `**`r1`**`, 0(r2)`
`add r3, `**`r1`**`, r4`
`lw  r5, 0(`**`r1`**`)`	

`lw  `**`r1`**`, 0(r2)` `lw  r3, 0(`**`r1`**`)`	
`lw  `**`r1`**`, 0(r2)` `sw  `**`r1`**`, 0(r3)`	

`lw  `**`r1`**`, 0(r2)`
`add `**`r3`**`, `**`r1`**`, r4`
`sw  r5, 0(`**`r3`**`)`	

`lw  `**`r1`**`, 0(r2)` `add r3, `**`r1`**`, r4`	

Problem M1.8.D — Stall Equation

Give the stall equation for the new pipeline. It should be optimized so that the pipeline only stalls when necessary to resolve data hazards. You may use the **`alu2`** logic signals from Question M1.8.B if needed.

$\mathbf{stall_{ID}} =$

Problem M1.9: Processor Design (Short Yes/No Questions)

The following questions describe two variants of a processor which are otherwise identical. In each case, circle "**Yes**" if the variants might generate different results from the same compiled program, and circle "**No**" otherwise. You must also briefly explain your reasoning. Ignore differences in the time each machine takes to execute the program.

Problem M1.9.A — Interlock vs. Bypassing

Pipelined processor A uses interlocks to resolve data hazards while pipelined processor B has full bypassing.

Yes / No

Problem M1.9.B — Delay Slot

Pipelined processor A uses branch delay slots to resolve control hazards while pipelined processor B kills instructions following a taken branch.

Yes / No

Problem M1.9.C — Structural Hazard

Pipelined processor A has a single memory port used to fetch instructions and data, while pipelined processor B has no structural hazards.

Yes / No

Problem M1.9.D — Microcode

Microcoded machine A uses 32-bit microcode instructions, while microcoded machine B uses 64-bit microcode instructions.

Yes / No

Problem M1.9.E — Stall Equation

Microcoded machine A has 32-bit data registers, while microcoded machine B has 64-bit data registers.

Yes / No