

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 <i>n</i>	Accum \leftarrow Accum + M[M[<i>n</i>]]
STOREind <i>n</i>	M[M[<i>n</i>]] \leftarrow Accum
BGEind <i>n</i>	If Accum \geq 0 then PC \leftarrow M[<i>n</i>]
BLTind <i>n</i>	If Accum $<$ 0 then PC \leftarrow M[<i>n</i>]

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 $\text{fib}(n)$, where n is a non-negative integer. $\text{fib}(n)$ returns the n th Fibonacci number ($\text{fib}(0)=0$, $\text{fib}(1)=1$, $\text{fib}(2)=1$, $\text{fib}(3)=2\dots$). 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

Subroutine Calling Other Subroutines

The following program defines a *recursive* version of the subroutine `fib` in C.

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

In a few sentences, explain what happens if the subroutine calling convention you implemented in Problem M1.1.B is used for `fib_recursive`.

Problem M1.2: CISC and RISC: Comparing ISAs

This problem requires the knowledge of Handout #2 (CISC ISA—x86jr), Handout #3 (RISC ISA—MIPS32), 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, register %edx contains result and register %eax contains a. b is stored in memory at location 0x08047580.) A brief explanation of each instruction in the code is given in Handout #2.

```
                xor    %edx,%edx
                xor    %ecx,%ecx
loop:           cmp    0x08047580,%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 MIPS32 instructions in Handout #3. Place the L1 and loop labels where appropriate. You should use the minimum number of instructions needed. Assume that upon entry R2 contains a and R3 contains i. R1 should be loaded with the value of b from memory location 0x08047580, while 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	MIPS32 instruction sequence
<code>xor %edx,%edx</code>		
<code>xor %ecx,%ecx</code>		
<code>cmp 0x08049580,%ecx</code>		
<code>jl L1</code>		
<code>jmp done</code>		
<code>add %eax,%edx</code>		
<code>inc %ecx</code>		
<code>jmp loop</code>		
<code>...</code>	<code>done:</code>	<code>...</code>

How many bytes is the MIPS32 program using your direct translation? How many bytes of MIPS32 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

Optimization

To get more practice with MIPS32, 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: Addressing Modes on MIPS ISA

Ben Bitdiddle is suspicious of the benefits of complex addressing modes. So he has decided to investigate it by incrementally removing the addressing modes from our MIPS ISA. Then he will write programs on the “crippled” MIPS ISAs to see what the programming on these ISAs is like.

Problem M1.3.A

Displacement addressing mode

As a first step, Ben has discontinued supporting the displacement (base+offset) addressing mode, that is, our MIPS ISA only supports register indirect addressing (without the offset).

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why.

LW R1, 16(R2) **→**

Problem M1.3.B

Register indirect addressing

Now he wants to take a bolder step by completely eliminating the register indirect addressing. The new load and store instructions will have the following format.

```
LW R1, imm16    ; R1 <- M[imm16]
SW R1, imm16    ; M[imm16] <- R1
```

6	5	5	16
Opcode	Rs		Offset

Can you still write the same program as before? If so, please translate the following load instruction into an instruction sequence in the new ISA. If not, explain why. (Don't worry about branches and jumps for this question.)

LW R1, 16(R2) **→**

Problem M1.3.C

Subroutine

Ben is wondering whether we can implement a subroutine using only absolute addressing. He changes the original ISA such that all the branches and jumps take a 16-bit absolute address (the 2 lower orders bits are 0 for word accesses), and that `jr` and `jalr` are not supported any longer.

With the new ISA he decides to rewrite a piece of subroutine code from his old project. Here is the original C code he has written.

```
int b; //a global variable

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

The C code above is translated into the following instruction sequence on our original MIPS ISA. Assume that upon entry, R1 and R2 contain `b` and `a`, respectively. R3 is used for `i` and R4 for `result`. By a calling convention, the 16-bit word-aligned return address is passed in R31.

```
Subroutine: xor  R4, R4, R4    ; result = 0
            xor  R3, R3, R3    ; i = 0
loop:      slt  R5, R3, R1
            bnez R5, L1        ; if (i < b) goto L1
return:    jr   R31           ; return to the caller
L1:        add  R4, R4, R2     ; result += a
            addi R3, R3, #1    ; i++
            j   loop
```

If you can, please rewrite the assembly code so that the subroutine returns without using a `jr` instruction (which is a register indirect jump). If you cannot, explain why.

Problem M1.4: Fully-Bypassed Simple 5-Stage Pipeline

We have reproduced the fully bypassed 5-stage MIPS processor pipeline from Lecture 5 in Figure M1.4-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 M1.4.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 M1.4.B

Bypass Signal

In Lecture 5, 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 the 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 signals have the highest and the lowest priorities.

Bypass_{EX->ID} ASrc = (rs_D=ws_E).we-bypass_E.re1_D (given in Lecture 5)

Bypass_{MEM->ID} =

Bypass_{WB->ID} =

Priority:

Problem M1.4.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.

Problem M1.5: Basic Pipelining

Unlike the Harvard-style (separate instruction and data memories) architectures, machines using the Princeton-style have a shared instruction and data memory. In order to reduce the memory cost, Ben Bitdiddle has proposed the following two-stage Princeton-style MIPS pipeline to replace a single-cycle Harvard-style pipeline from our lectures.

Every instruction takes exactly two cycles to execute (i.e., instruction fetch and execute) and there is no overlap between two sequential instructions; that is, fetching an instruction occurs in the cycle following the previous instruction's execution (no pipelining).

Assume that the new pipeline does not contain a branch delay slot. Also, don't worry about self-modifying code for now.

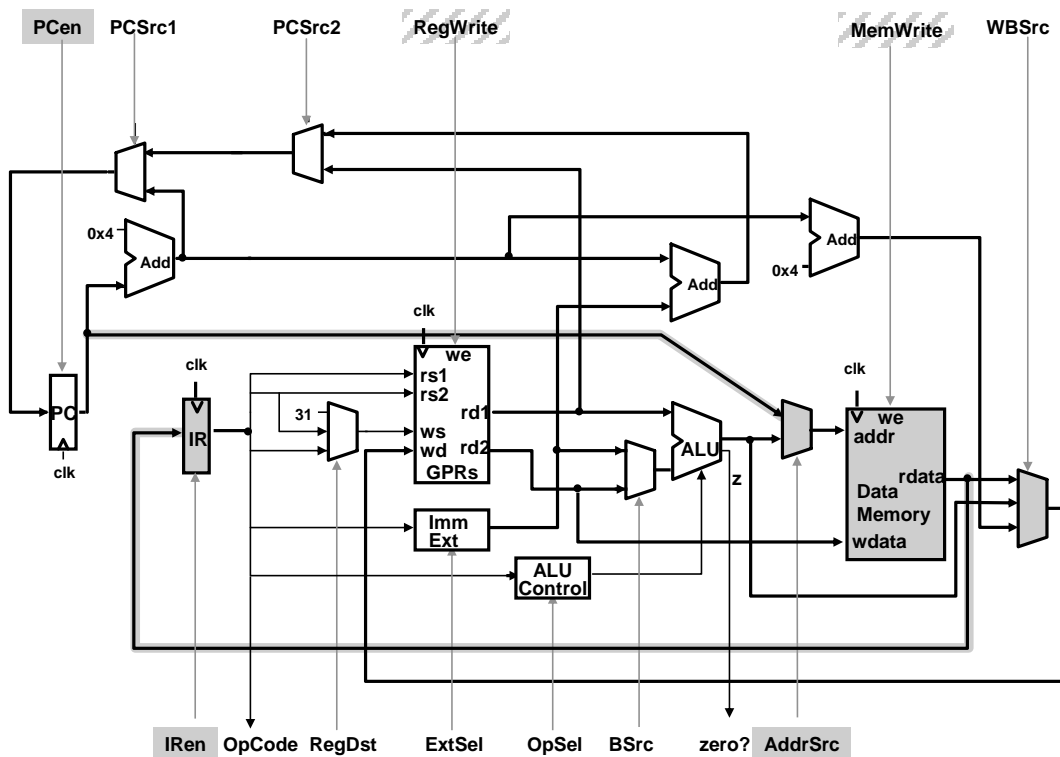


Figure M1.5-A: Two-stage pipeline, Princeton-style

Problem M1.5.A

Mux Control Signals (1)

Please complete the following control signals. You are allowed to use any internal signals (e.g., OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.).

Example syntax: $PCEn = (OpCode == ALUOp) \text{ or } ((ALU.zero?) \text{ and } (not (PC == 17)))$

You may also use the variable S which indicates the pipeline's operation phase at a given time.

$S := I\text{-Fetch} \mid \text{Execute} \quad (\text{toggles every cycle})$
--

PCEn =

IREn =

AddrSrc = Case _____ _____ _____ => PC _____ => ALU
--

Problem M1.5.B

Modified pipeline

After having implemented his proposed architecture, Ben has observed that a lot of datapath is not in use because only one phase (either I-Fetch or Execute) is active at any given time. So he has decided to fetch the next instruction during the Execute phase of the previous instruction.

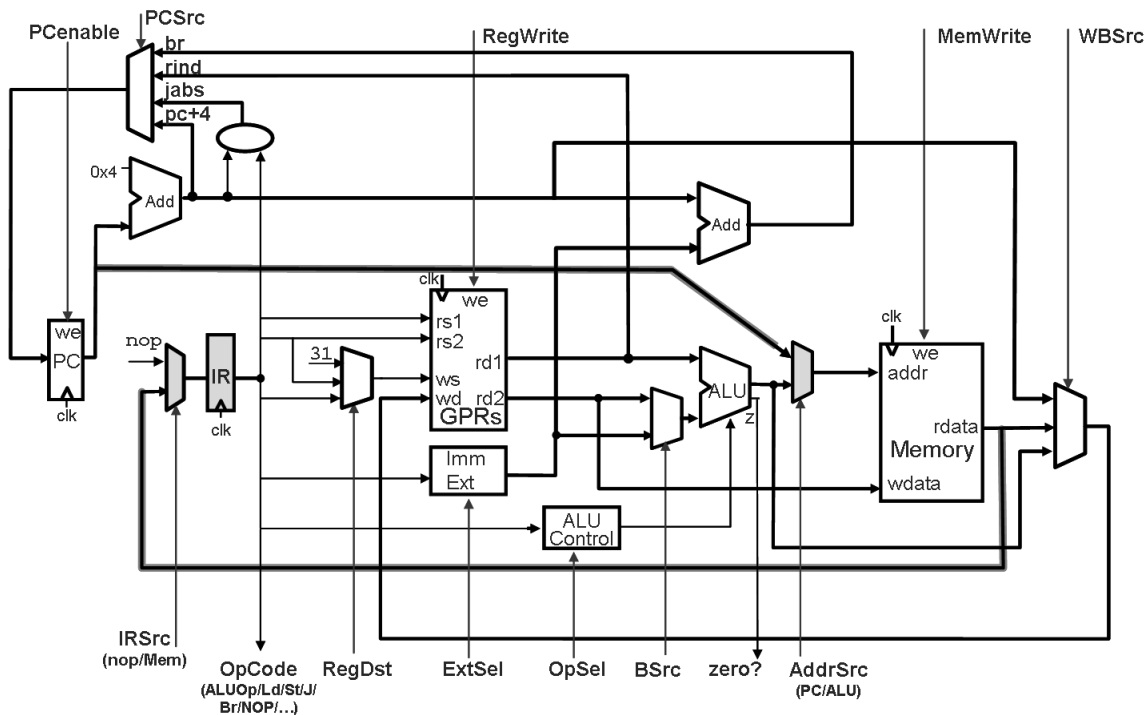


Figure M1.5-B: Modified Two-stage Princeton-style MIPS Pipeline

Do we need to stall this pipeline? If so, for each cause (1) write down the cause in one sentence and (2) give an example instruction sequence. If not, explain why. (Remember there is **no** delay slot.)

Problem M1.5.C

Mux Control Signals (2)

Please complete the following control signals in the modified pipeline. As before, you are allowed to use any internal signals (e.g., OpCode, PC, IR, zero?, rd1, data, etc.) but not other control signals (ExtSel, IRSrc, PCSrc, etc.)

PCEnable =

AddrSrc = Case _____ _____ => PC _____ => ALU
IRSrc = Case _____ _____ => nop _____ => Mem

Problem M1.5.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”.

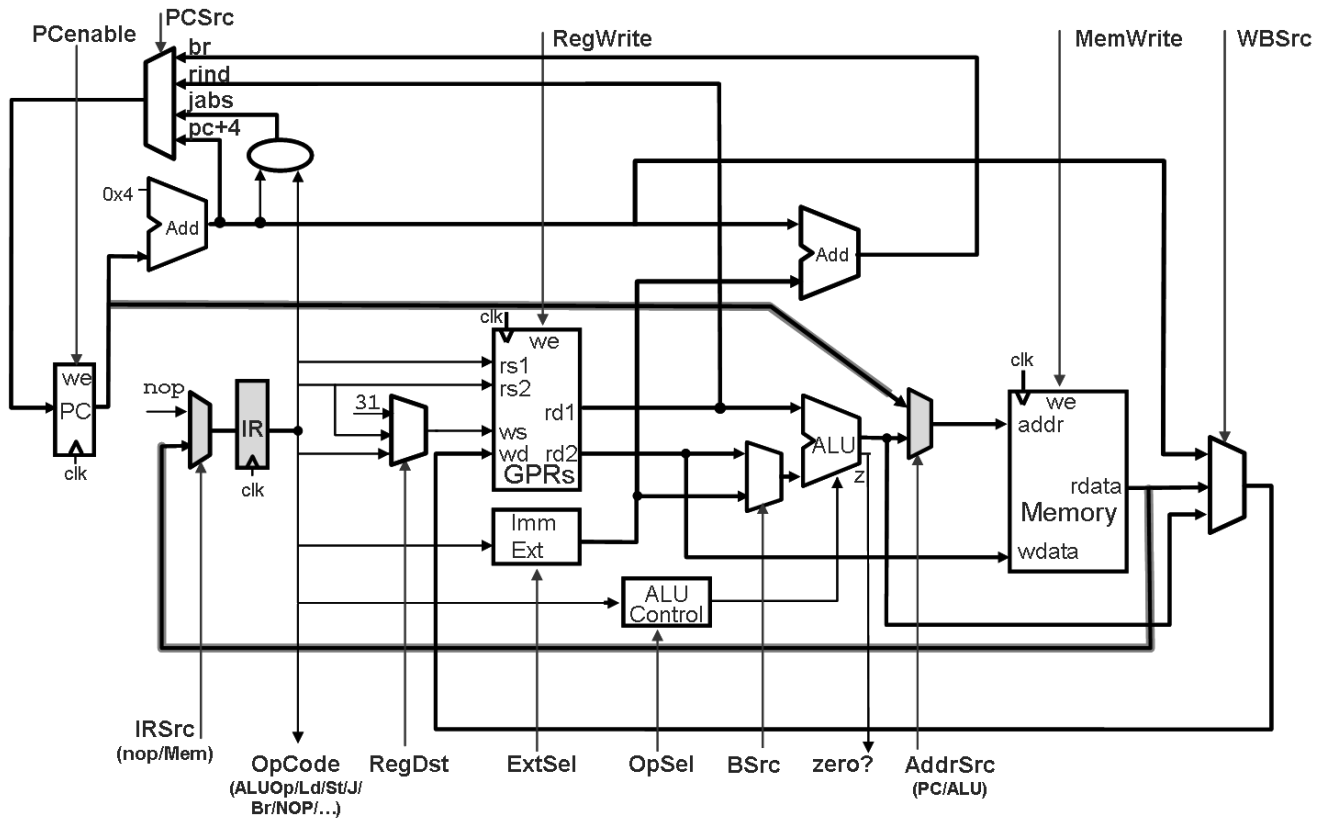
Label	Address	Instruction
I₁	100	ADD
I₂	104	LW
I₃	108	J I₇
I₄	112	LW
I₅	116	ADD
I₆	120	SUB
I₇	312	ADD
I₈	316	ADD

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 M1.5.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: 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 5 (reproduced in Figure M1.4-A). In the new datapath the ALU in 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 M1.6-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 M1.6.A

Elimination of a hazard

What hazard is the new datapath trying to eliminate? Give an example sequence of MIPS instructions (five or fewer instructions) that would cause a hazard in the original datapath but not in the new datapath.

Problem M1.6.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 M1.6.C

Comparison

List the advantages and disadvantages of the new datapath. Which datapath 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

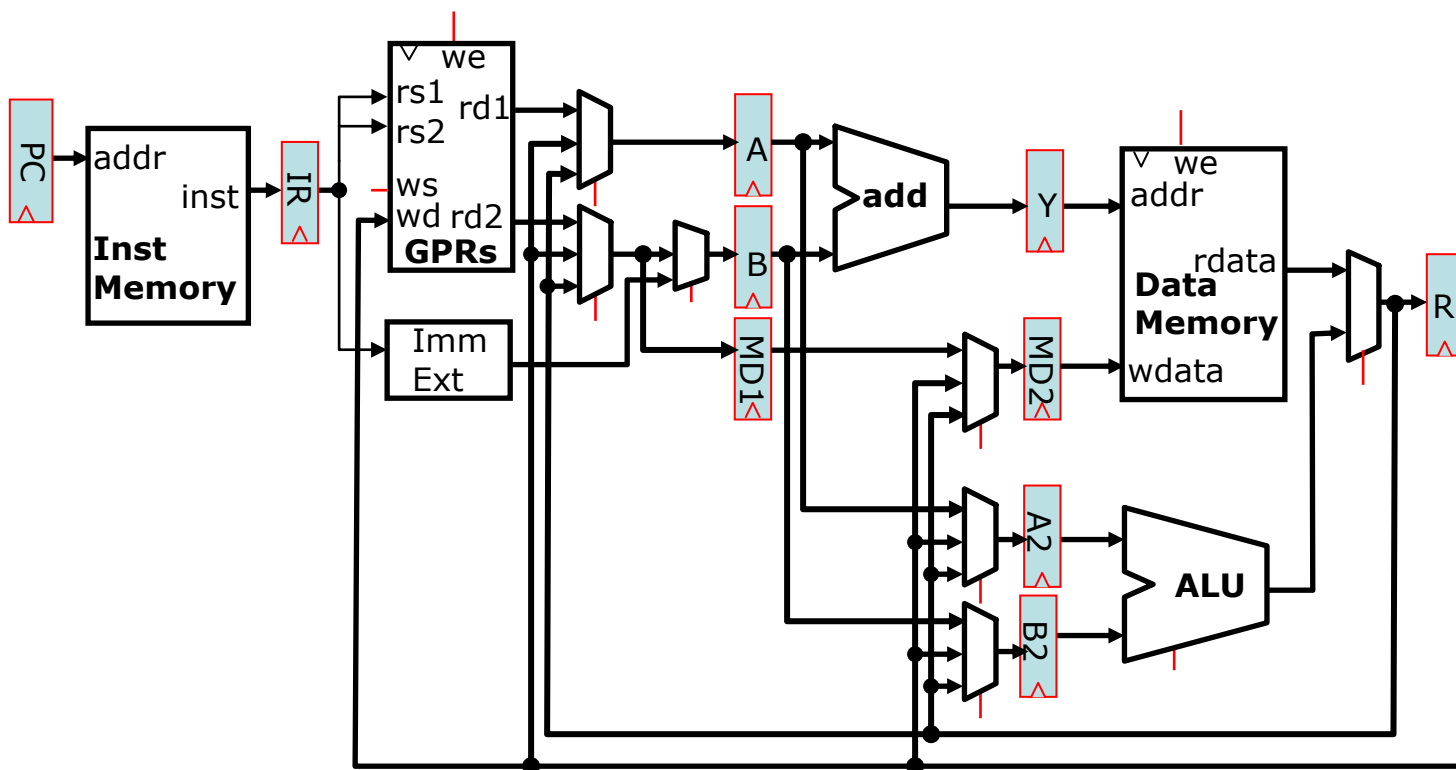


Figure M1.6-A: 5-Stage Pipeline with an Additional Adder

Problem M1.6.D

Stall Logic

Write the stall condition (in the style of Lecture 5) 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.

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

Problem M1.6.E

Datapath Improvement

Consider a MIPS ISA that only supports register indirect addressing, i.e., it has no displacement (base+offset) addressing mode. Assuming the new machine only has to support this ISA, how can the datapath be improved? Draw the new datapath showing your design. (You do not have to show everything, only the important features like pipeline registers, major components, major connections, etc.) Compare the hazards in this new datapath with the hazards in the datapath shown in Figure M1.6-A and the original datapath in Lecture 5 (Figure M1.4-A). Justify the new datapath.

Problem M1.6.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 M1.6.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 of 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 M1.7: Dual ALU Pipeline

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

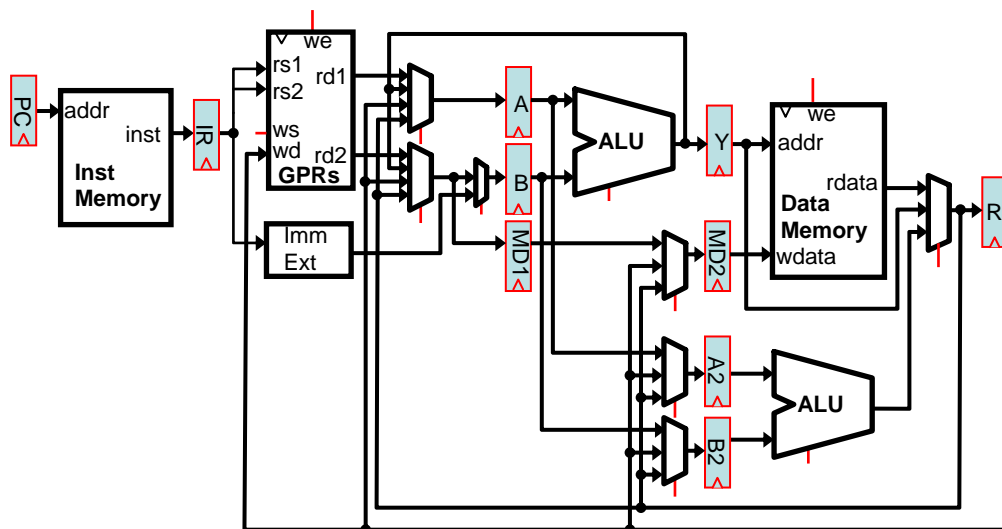


Figure M1.8-A: Dual ALU Pipeline

Problem M1.7.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.

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

Problem M1.7.B

Control Signal

Fill in the equation for the control logic signal **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 **alu2_{EX1}** and **alu2_{EX2/MEM}**, you may use these signals in your equation if needed. In the equation, “+” means logical OR and “·” means logical AND.

$$\begin{aligned}
 \text{alu2}_{ID} = & (((OP_{ID} = ALU) + (OP_{ID} = ALU_i)) \\
 & \cdot ((rs_{ID} = ws_{EX1}) + (rt_{ID} = ws_{EX1}) \cdot re2_{ID}) \\
 & \cdot (ws_{EX1} \neq 0) \\
 & \cdot (\underline{\hspace{10em}}) \\
 &)
 \end{aligned}$$

Problem M1.7.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 M1.7.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.

stall_{ID} =

Problem M1.8: Processor Design (Short Yes/No Questions)

The following statements 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, circle "**No**" otherwise. You must also briefly explain your reasoning. Ignore differences in the time that each machine takes to execute the program.

Problem M1.8.A

Interlock vs. Bypassing

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

Yes / No

Problem M1.8.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 M1.8.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