Problem M1.5: Fully-Bypassed Simple 5-Stage Pipeline [0.5 Hours]

We have reproduced the fully bypassed 5-stage MIPS processor pipeline from Lecture 5 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 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 $_{EX\text{->}ID}$ ASrc = $(rs_D=ws_E).we\text{-}bypass_E.re1_D$ (given in Lecture L5)

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

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

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.

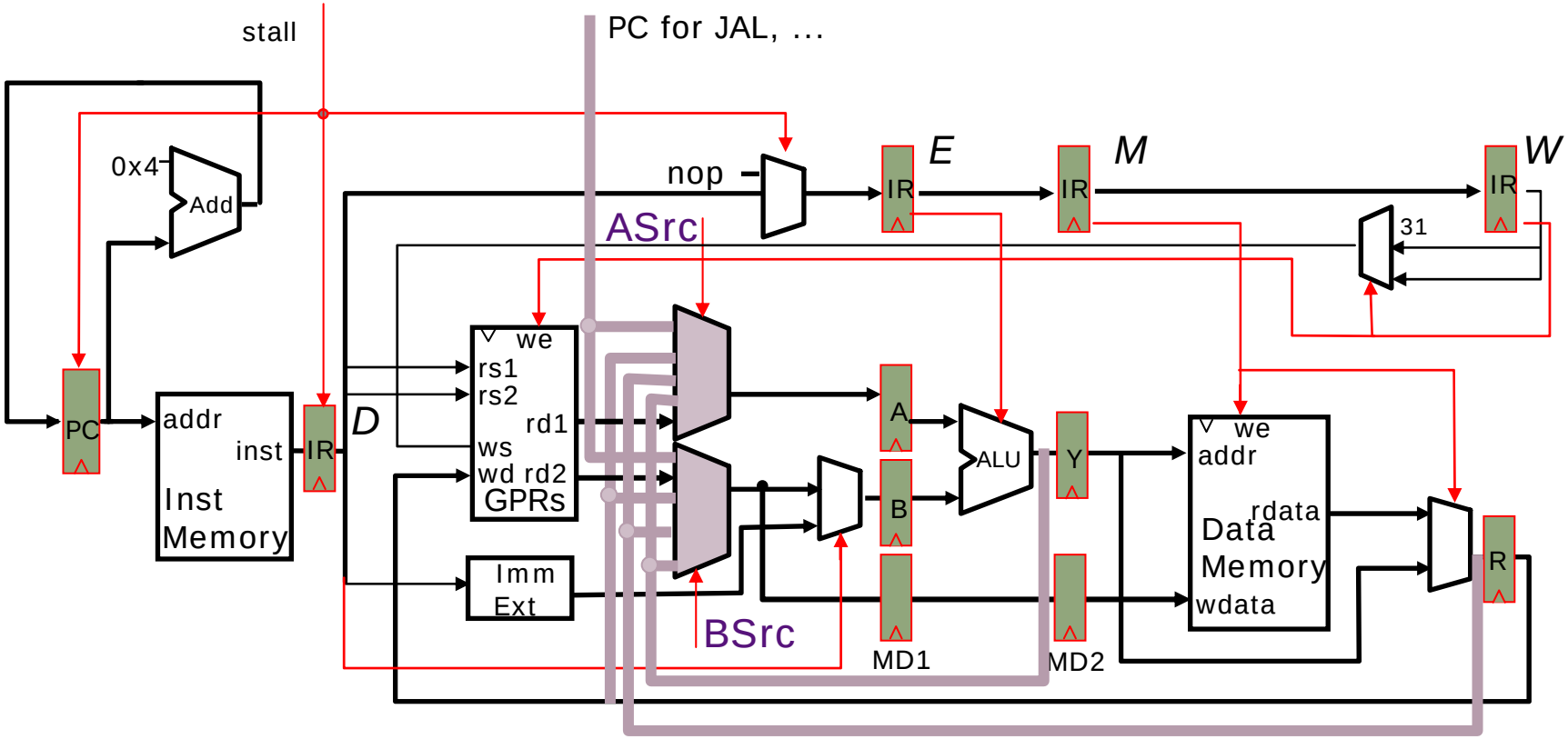


Figure M1.5-A: Fully-Bypassed MIPS Pipeline

Problem M1.6: Basic Pipelining [1 Hour]

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.

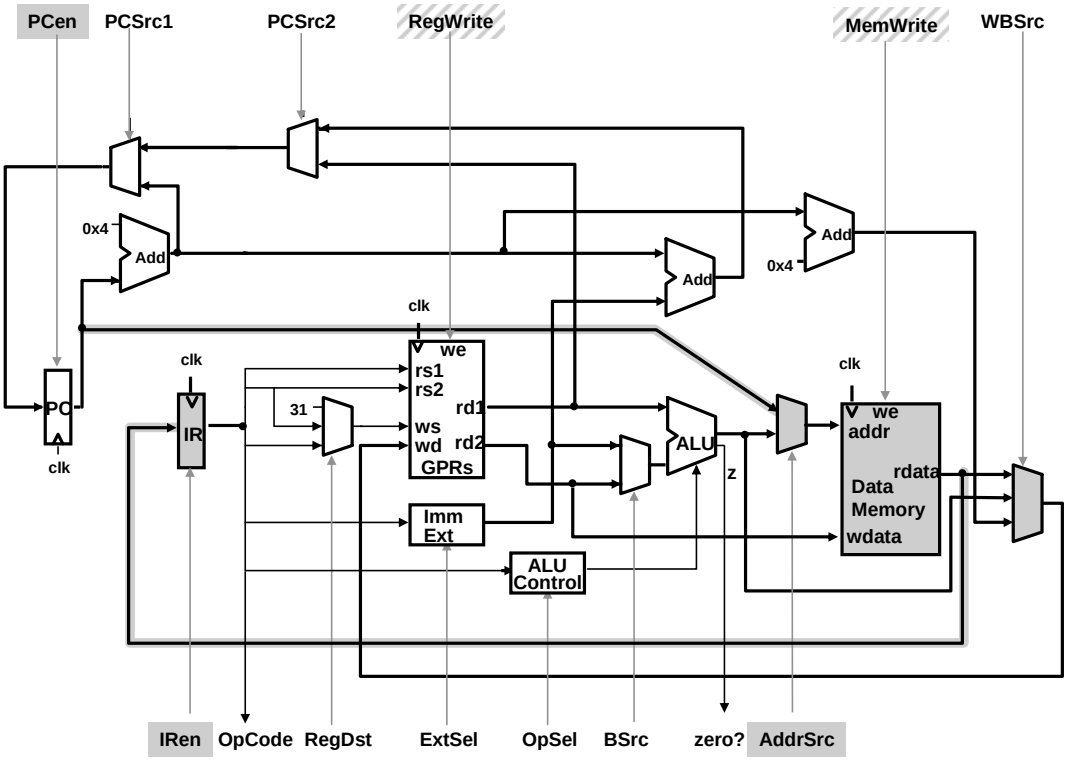


Figure M1.6-A: Two-stage pipeline, Princeton-style

Problem M1.6.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) or ((ALU.zero?) and (not (PC == 17)))

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

```
S := I-Fetch | Execute  (toggles every cycle)
```

PCEn =

IREn =

AddrSrc = Case _____

_____ => PC

_____ => ALU

Problem M1.6.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.

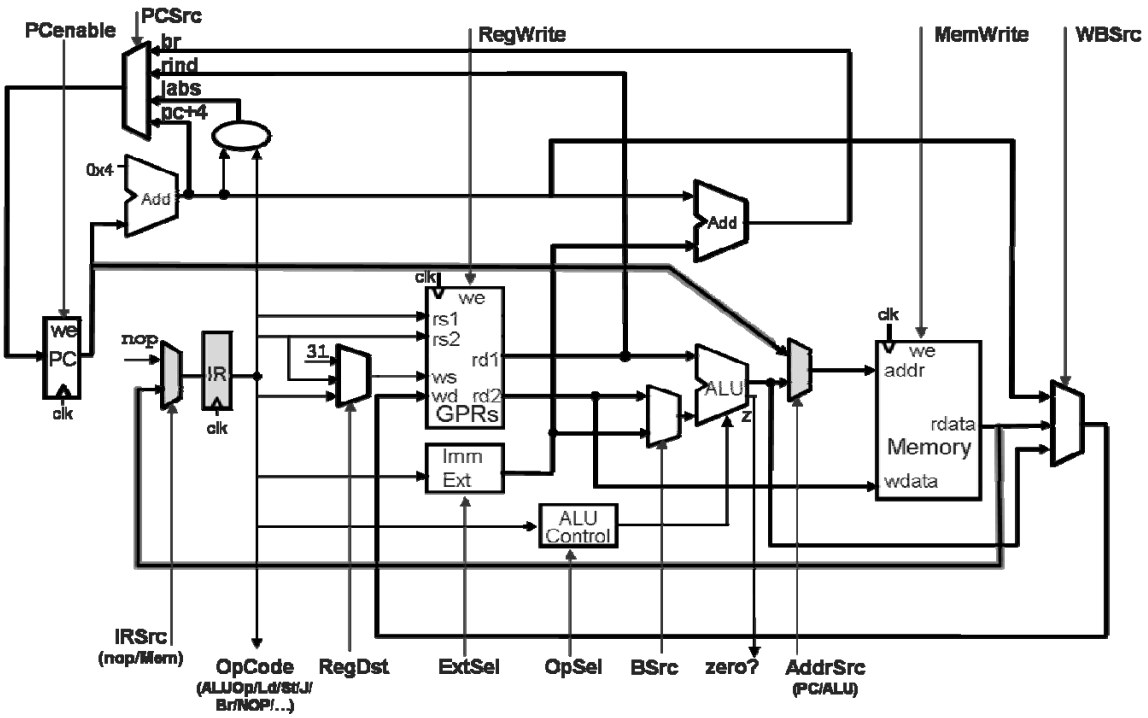


Figure M1.6-B: Modified Two-stage Princeton-style MIPS Pipeline

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 M1.6.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 M1.6.D

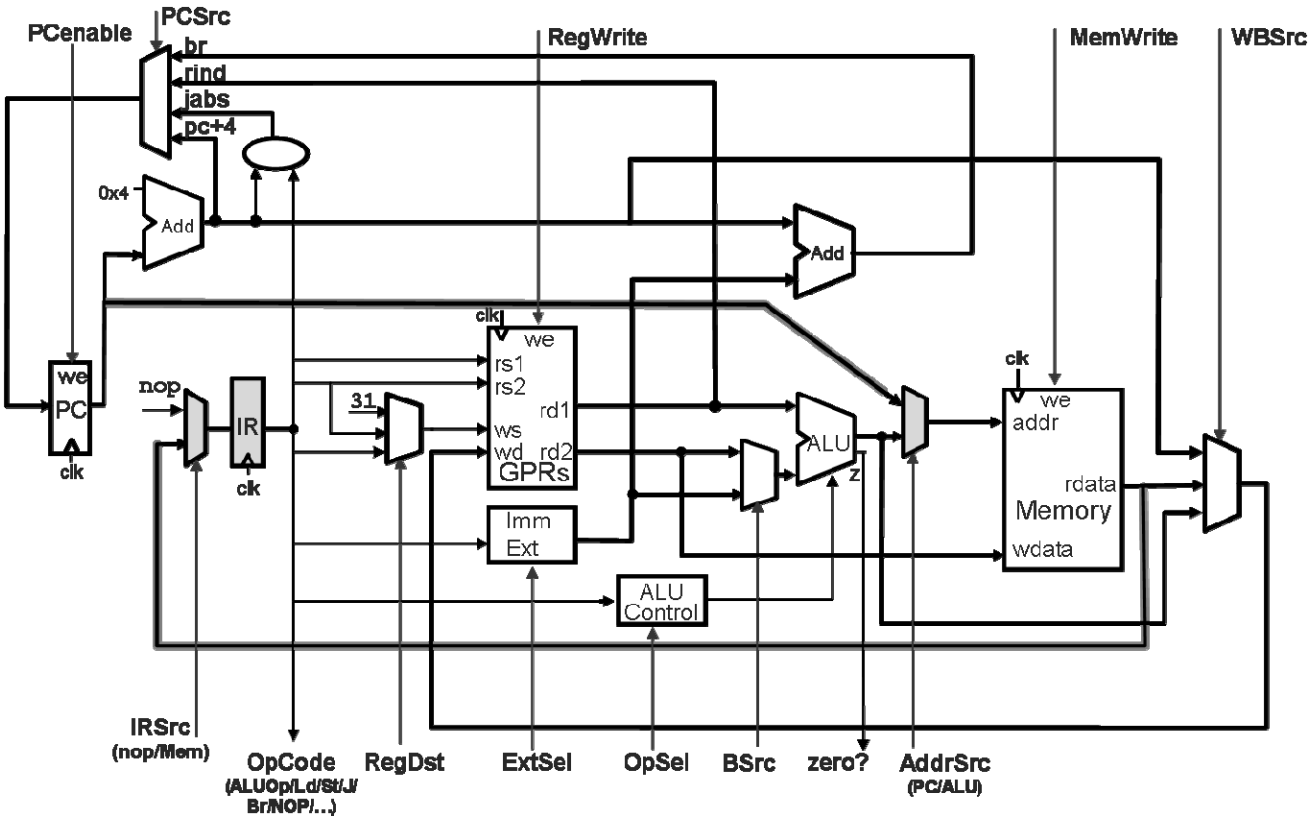
Now we are ready to put Ben's machine to the test. We would like to see a cycle-by-cycle animation of Ben's two-stage pipelined, Princeton-style MIPS machine when executing the instruction sequence below. In the following table, each row represents a snapshot of some control signals and the content of 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						

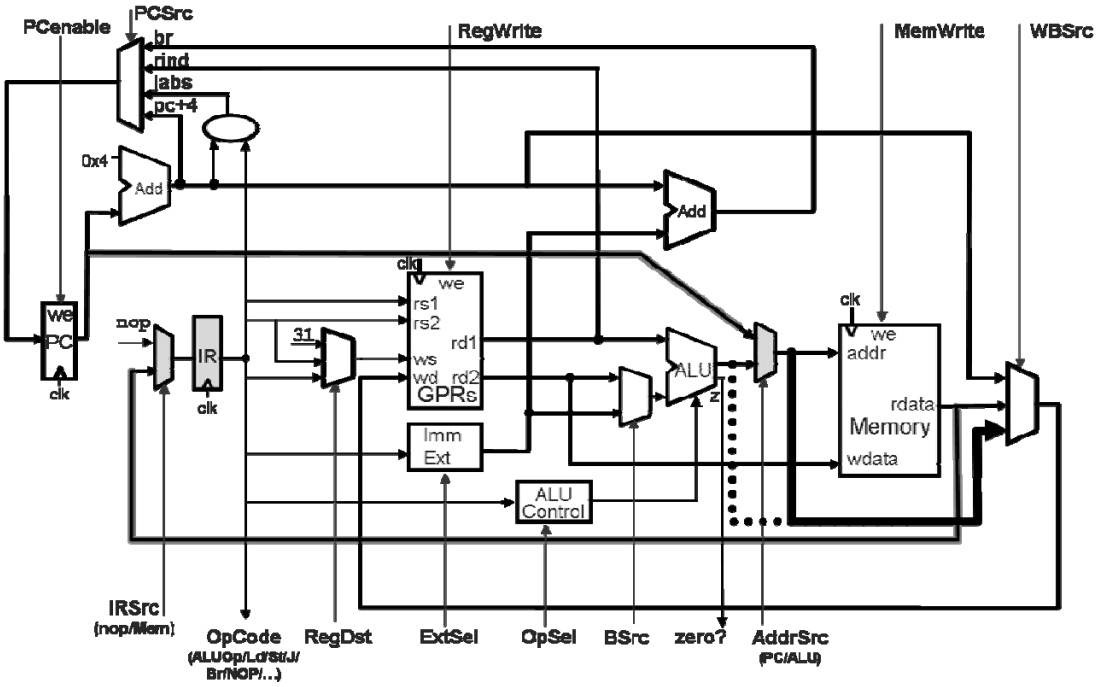
Problem M1.6.E **Self-Modifying Code**

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 with a bold line, the old in 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**

Give one advantage of the Princeton architecture over the Harvard architecture.

Give one advantage of the Harvard architecture over the Princeton architecture.

Problem M1.7: A 5-Stage Pipeline with an Additional Adder [1.5 Hours]

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 M1.5-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 M1.7-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

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 M1.7.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

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

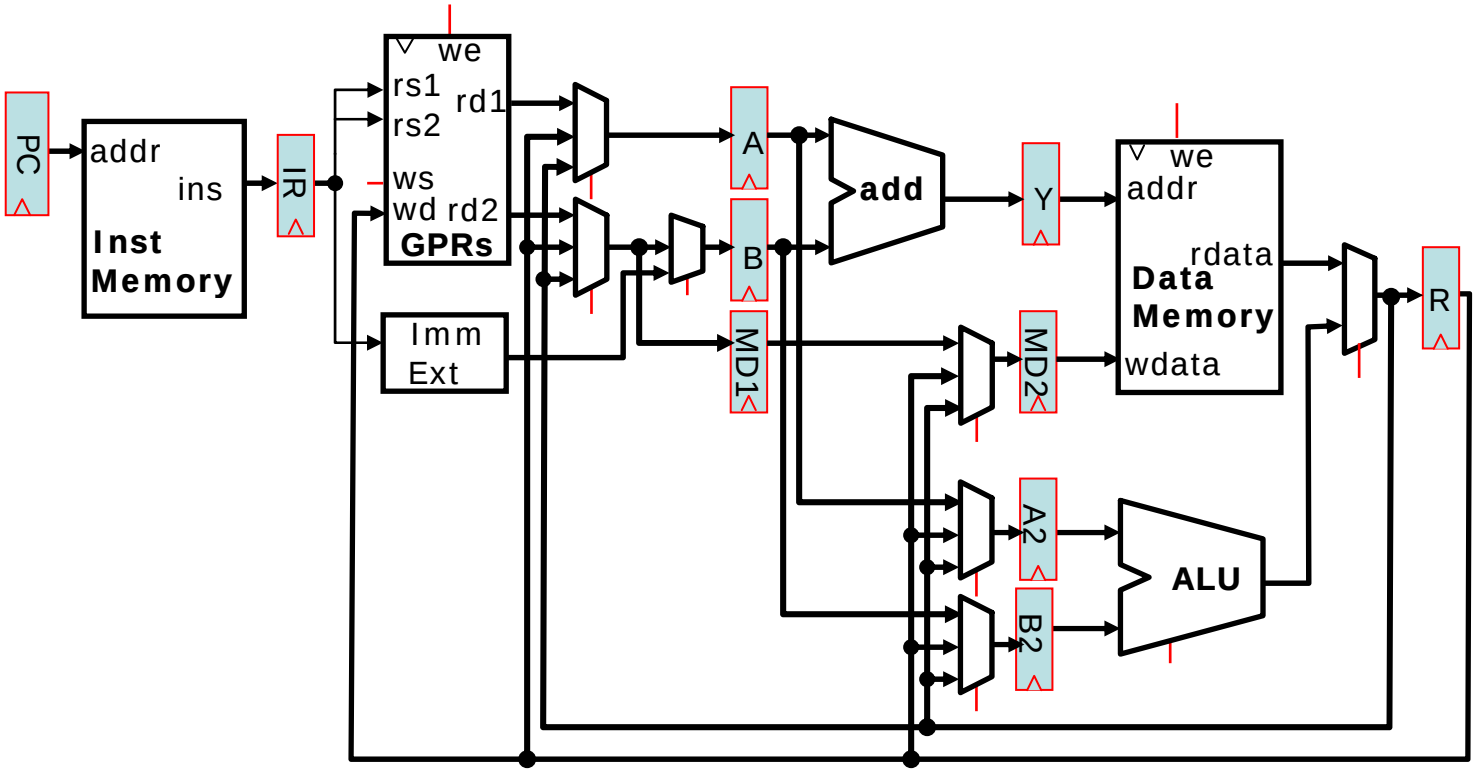


Figure M1.7-A: 5-Stage Pipeline with an Additional Adder

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

Write the stall condition (in the style of Lecture L5) 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., 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 M1.7-A and the original datapath in Lecture 5 (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 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 M1.8: Dual ALU Pipeline [1 Hour]

In this problem we consider further improvements to the fully bypassed 5-stage MIPS processor pipeline presented in Lecture 5 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

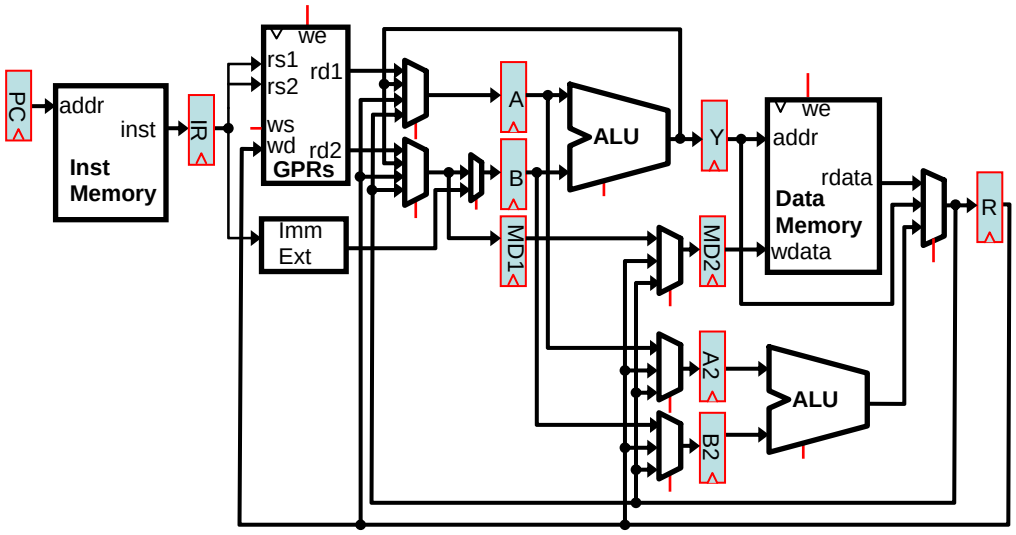


Figure M1.8-A: Dual ALU Pipeline

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.

	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 M1.8.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.

$$\begin{aligned}\mathbf{alu2_{ID}} = (\ &((OP_{ID} = ALU) + (OP_{ID} = ALUi)) \\ &\cdot((rs_{ID} = ws_{EX1}) + (rt_{ID} = ws_{EX1})\cdot re2_{ID}) \\ &\cdot(ws_{EX1} \neq 0) \\ &\cdot(\ \underline{\qquad\qquad\qquad\qquad\qquad\qquad}\ ) \\ &)\end{aligned}$$

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) [1/12 Hour]

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 M1.9.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