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

### Problem M1.5.A — Stall

We still need the logic for stalls, because we cannot prevent load-use hazard. If a load instruction is followed by an instruction which takes the loaded value as a source operand, we cannot avoid stalling for a cycle. The following instruction sequence illustrates this hazard.

```
LW  R1, 0(R2)     # R1 <- M[R2]
ADD R3, R5, R1    # R1 is a source operand of ADD (data dependency)
                  # The correct value of R1 is not available when
                  # ADD is in ID stage.  So it has to stall for a cycle.
```

### Problem M1.5.B — Bypass Signal

Here are the bypass conditions.

Bypass $_{EX->ID}$ ASrc = $(rs_D=ws_E).we\text{-}bypass_E.re1_D$ (given in Lecture L5)

Bypass $_{MEM->ID}$ = $(rs_D=ws_M).we_M.re1_D$

Bypass $_{WB->ID}$ = $(rs_D=ws_W).we_W.re1_D$

Priority: Bypass $_{EX->ID}$ > Bypass $_{MEM->ID}$ > Bypass $_{WB->ID}$
(In order to execute a given program correctly, the value from the latest producer must be taken if multiple bypass paths are active.)

### Problem M1.5.C — Partial Bypassing

It is an open question and there is no single correct answer. Here are a couple of issues to consider as a guideline.

First, you may consider the penalty for not having all the bypass paths. If we don't have the bypass path EX→ID, we have to stall for three cycles for the hazard to be resolved. Likewise, not having MEM→ID results in a stall of two cycles, and not having WB→ID, in one. Therefore, you can conclude that the bypass path between EX→ID is the most beneficial.

Secondly, the best bypass path depends on the access patterns of data. The EX→ID bypass path is effective if a producer instruction is followed by a consumer, except load-use cases (See solution for M1.5.A). On the other hand, the MEM→ID bypass path works best if there are many load-use cases or many (producer, consumer) pairs have an independent instruction between them. Likewise, the WB→ID bypass path helps when many (producer, consumer) pairs are separated by exactly two independent instructions.

## Problem M1.6: Basic Pipelining

**Problem M1.6.A** **Mux Control Signals (1)**

PCEn = (S==Execute)

IREn = (S==I-Fetch)

AddrSrc = Case S

I-Fetch => PC

Execute => ALU

**Problem M1.6.B** **Modified pipeline**

A stall can occur in 2 different cases.

1. A structural hazard in the shared memory.
   LD R1, 16(R2)
   Any instruction following this LD instruction should be stalled.

2. The other is caused by a control hazard, because we don't have a delay slot.
   J 200
   Any instruction following this J instruction should be flushed.

**Problem M1.6.C** **Mux Control Signals (2)**

PCEnable = not ((opcode == LW) or (opcode == SW))

AddrSrc = Case opcode

not (LW or SW) => PC

(LW or SW) => ALU

IRSrc = Case opcode

LW or SW or Jump or $Br_{taken}$ => nop

Else => Mem

**Problem M1.6.D**

| 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 |
| $t_2$ | **$I_3$:108** | **$I_2$** | **0** | ***** | **ALU** | **Nop** |
| $t_3$ | **$I_3$:108** | **-** | **1** | **pc+4** | **PC** | **Mem** |
| $t_4$ | **$I_4$:112** | **$I_3$** | **1** | **jabs** | **PC** | **Nop** |
| $t_5$ | **$I_7$:312** | **-** | **1** | **pc+4** | **PC** | **Mem** |
| $t_6$ | **$I_8$:316** | **$I_7$** | **1** | **pc+4** | **PC** | **Mem** |

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

The answer is no. The hazard is resolved by the datapath itself because (1) memory accesses are serialized by the stall logic at the shared memory and (2) memory write takes only one cycle.

**Problem M1.6.F**

Due to this rerouting we will now have to stall even if it is an ALU instruction.

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

The Princeton architecture is cheaper than the Harvard architecture, but the Harvard architecture is faster than the Princeton architecture.

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

### Problem M1.7.A — Elimination of a hazard

The new datapath is trying to eliminate the hazard that occurs when a load instruction is immediately followed by an ALU instruction that requires the value that was loaded. In the original datapath, a pipeline interlock (stall) is needed for this type of an instruction sequence, an example of which is shown below. In Ben's datapath, this load-use interlock is not required because the data from the load instruction can be immediately forwarded to the ALU.

```
LW R1, 0(R3)
ADDI R1, R1, #5
```

### Problem M1.7.B — New Hazard

The new hazard occurs when the result of an ALU operation is needed to calculate the address of a load or store instruction.

```
ADDI R1, R1, #5
LW R3, 3(R1)
```

### Problem M1.7.C — Comparison

Now an address-generation interlock is needed for the LW instruction in the sequence in M1.7.B. Note that this new hazard affects both load and store instructions, while the original hazard only affected load instructions. This is a disadvantage of the modified pipeline. Also, the new datapath requires more hardware (another adder) than the original datapath. However, the load-use hazard illustrated in Problem M1.7.A has been eliminated. If we examine the behavior of typical programs, we will see that the percentage of load instructions resulting in the load-use interlock from Problem M1.7.A is higher than the percentage of all loads and stores resulting in the address-generation interlock from Problem M1.7.B. This is because many address calculations are based on values that change infrequently (e.g. the stack pointer does not change while a procedure is being executed). If a base address register has not been recently changed, then there will be no address-generation interlock. By contrast, when a load is issued, the load value is usually required within a few cycles, so a load-use interlock is much more likely. Whether performance is better on the original pipeline or on the modified pipeline will depend on the specific program.

### Problem M1.7.D — Stall Logic

The stall equation for only the new hazard is given below. The **op** signal is used to determine the instruction opcode.

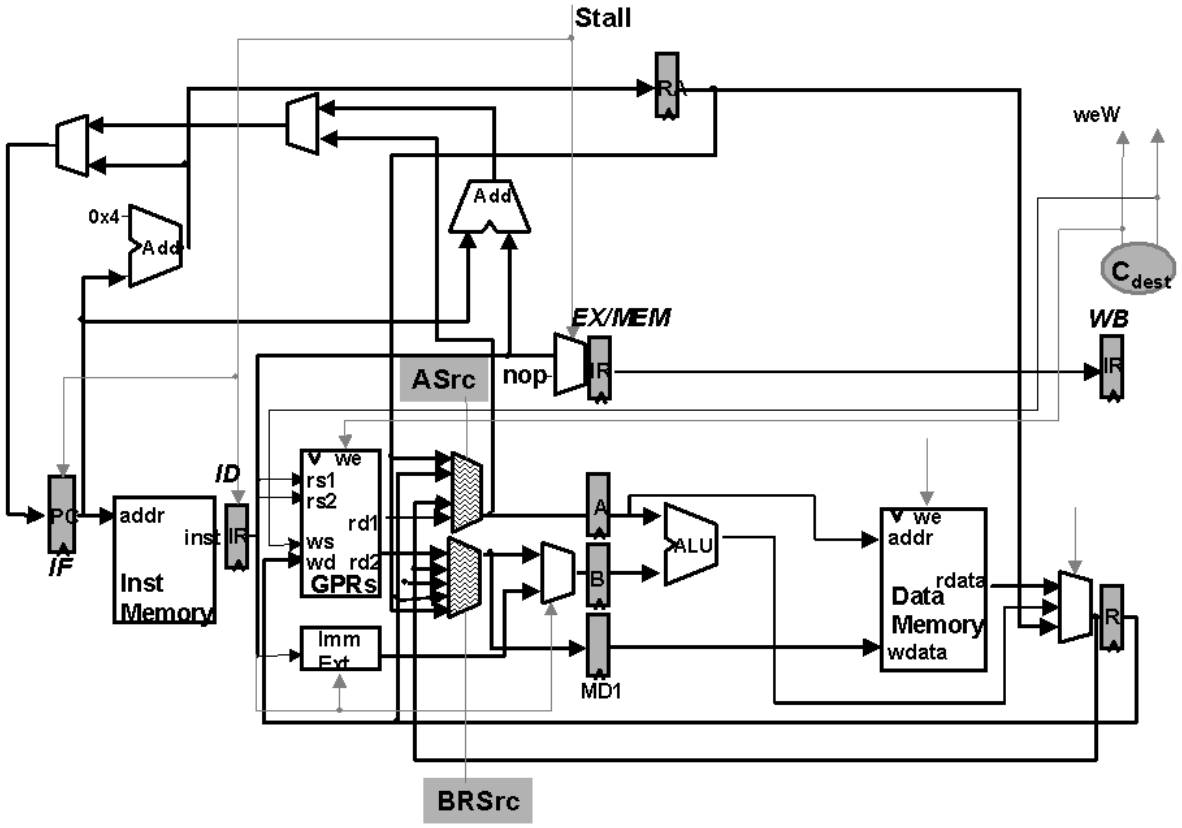
$$\textbf{Stall} = ((op_{ID} = LW) + (op_{ID} = SW)).(rs_{ID} = ws_{AC}).((op_{AC} = ALU) + (op_{AC} = ALUi)).(ws_{AC} \neq 0)$$

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

If we eliminated the displacement addressing mode from the MIPS ISA and only supported register indirect addressing, then we would no longer need to compute an effective address for loads and stores. We could improve the datapath by eliminating the AC (effective address calculation) stage from Ben's modified pipeline, resulting in the following stages

| IF | ID | EX/MEM | WB |
|---|---|---|---|
| Instruction fetch | Instruction decode and register fetch | Execution of ALU operations or memory access | Write-back to register file |

A diagram showing the new pipeline is given below.




This new datapath does not have either of the hazards from Ben's original or modified pipelines. Thus, bubbles would not need to be inserted into the pipeline regardless of the instruction sequence, improving instruction throughput. As a side note, the latency of a single instruction has also been reduced since there are now only 4 stages instead of 5. Although this does not improve performance in the steady state, a fewer number of stages does help because fewer pipeline registers and bypass paths are required. However, this instruction set is limited in that it only supports register indirect addressing. This means that displacement addressing would have to be synthesized from simpler instructions (see Problem M1.7.F).

## Problem M1.7.F — Displacement Addressing Synthesizing

Programmers could synthesize a displacement load/store instruction using the ADDi instruction, a scratch register, and the register indirect load/store instruction. For example, to synthesize the following instruction with displacement addressing

```
LW R1, 4(R2)
```

we could use the following equivalent instruction sequence, where R3 is a temporary register

```
ADDI R3, R2, #4
LW R1, (R3)
```

The same programs could be written as before using this technique. However, using this limited ISA may increase the number of instructions in the program as compared to the original ISA.

## Problem M1.7.G — Jumps and Branches

If Ben uses the ALU to resolve conditional branches in both his original pipeline and his modified pipeline shown in Problem M1.7.A, then there will be an additional cycle of branch delay in the new datapath because the ALU is now one stage later in the pipeline. If we don't worry about duplicating logic, then we can put a comparator in any stage of the pipeline (except Instruction Fetch, as the register file has not yet been read in this stage) in order to resolve conditional branches. The table shown below compares each possible placement of the comparator.

| Comparator In Stage | Number of Branch Delay Cycles | Additional Stall Condition | Change in Clock Period |
|---|---|---|---|
| WB | 4 | None | Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path. |
| EX/MEM | 3 | None | Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path. |
| AC | 2 | 1 cycle stall when the ALU output or result of a load is used for the branch | Will remain unchanged since comparator is simpler than ALU operation so it cannot be the critical path. |
| ID | 1 | 2 cycle stall when the ALU output or result of a load is used for the branch | Will likely **increase** the clock period since it now could be on the critical path (fetch register value + compare) |

Obviously placing the comparator in the Write-Back stage makes no sense since this doesn't provide an advantage over placing the comparator in the Execute/Memory stage, and in fact, it increases the number of branch delay cycles by 1. Placing the comparator in the Address Calculation stage instead of the Execute/Memory stage reduces the number of branch delay cycles by 1, but introduces a potential stall condition. Since the branch delay affects all branches, while the stall condition would only affect some of the branches, placing the comparator in the Address Calculation stage is to be preferred over the Execute/Memory stage. Finally, the comparator could be placed in the Instruction Decode stage. If this doesn't lengthen the critical path, then this would be the best placement, as the number of branch delay cycles is reduced to 1. However, if it does lengthen the critical path—and it likely will—then the increased cycle time would probably not be worth the reduction in the branch delay, as now *all* instructions will run more slowly.

## Problem M1.8: Dual ALU Pipeline

**Problem M1.8.A** **ALU Usage**

```
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
```

| ALU1 or ALU2? |
|---|
| **ALU1** |
| |
| **ALU2** |
| **ALU2** |
| **ALU1** |
| |
| **ALU1** |

The following timeline shows the execution of the instructions, with the stage where each instruction produces its result highlighted in bold, and the bypassing between instructions shown by arrows.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $add_1$ | IF | ID | **EX1** | EX2 | WB | | | | | | |
| $lw_1$ | | IF | ID | EX1 | **MEM** | WB | | | | | |
| $add_2$ | | | IF | ID | EX1 | **EX2** | WB | | | | |
| $add_3$ | | | | IF | ID | EX1 | **EX2** | WB | | | |
| $add_4$ | | | | | IF | ID | **EX1** | EX2 | WB | | |
| $lw_2$ | | | | | | IF | ID | EX1 | **MEM** | WB | |
| $add_5$ | | | | | | | IF | ID | **EX1** | EX2 | WB |

The pipeline is initially idle, so the first add reads its operands from the register file in ID and uses ALU1. The second add uses the result of the lw which is not available by the end of ID; therefore the add uses ALU2, and the load data is bypassed to it at the end of EX1. The third add uses the result of the second, so its data is not available by the end of ID; it also uses ALU2, allowing the data to be bypassed to it at the end of EX1. The fourth add has no dependencies on the previous instructions; it reads its operands from the register file in ID and uses ALU1. The fifth add uses the result of the fourth add. This value is bypassed to it at the end of ID from EX2/MEM, and it uses ALU1.

**Problem M1.8.B** **Control Signal**

$$alu2_{ID} = ( ((OP_{ID} = ALU) + (OP_{ID} = ALUi)) \cdot((rs_{ID} = ws_{EX1}) + (rt_{ID} = ws_{EX1}) \cdot re2_{ID}) \cdot(ws_{EX1} \neq 0) \cdot( \underline{(OP_{EX1} = LW) + alu2_{EX1}} ) )$$

An ALU instruction uses ALU2 if its operands are not available by the end of ID. This occurs if the ALU instruction (in ID) uses the result of its immediately preceding instruction (in EX1) as a source, but the instruction will not produce its result until EX2/MEM. The two classes of instructions which do not produce a result until EX2/MEM are LW instructions and ALU instructions which use ALU2.

Note that the feedback dependence of $alu2_{ID}$ on $alu2_{EX1}$ means that a sequence of ALU instructions following a LW will continue to use ALU2 as long as each instruction uses the result of its predecessor.

**Problem M1.8.C** **Instruction Sequences Causing Stalls**

| | Stall? | Explanation |
|---|---|---|
| `add` **`r1`**`, r2, r3`<br>`lw r4, 0(`**`r1`**`)` | **No** | **The add (in EX1) uses ALU1 and bypasses its result to the LW (in ID).** |
| `lw` **`r1`**`, 0(r2)`<br>`add r3,` **`r1`**`, r4`<br>`lw r5, 0(`**`r1`**`)` | **No** | **The first LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2, and also to the second LW (in ID).** |
| `lw` **`r1`**`, 0(r2)`<br>`lw r3, 0(`**`r1`**`)` | **Yes** | **The result of the first LW (in EX1) is not available in time for the second LW (in ID), so the second LW must stall.** |
| `lw` **`r1`**`, 0(r2)`<br>`sw` **`r1`**`, 0(r3)` | **No** | **The LW (in EX2/MEM) bypasses its result to the SW (in EX1) in time for it to store the data in EX2/MEM.** |
| `lw` **`r1`**`, 0(r2)`<br>`add` **`r3`**`,` **`r1`**`, r4`<br>`sw r5, 0(`**`r3`**`)` | **Yes** | **The LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2. But, the result of the add (in EX1) is not available in time for the SW (in ID), so the SW must stall.** |
| `lw` **`r1`**`, 0(r2)`<br>`add r3,` **`r1`**`, r4` | **No** | **The LW (in EX2/MEM) bypasses its result to the add (in EX1) which will use ALU2.** |

Note that the base address operand for both LW and SW must be available by the end of ID, but the data operand for SW must only be available by the end of EX1.

**Problem M1.8.D** **Stall Equation**

$$\begin{aligned} stall_{ID} = (\ & ((OP_{ID} = LW) + (OP_{ID} = SW)) \\ & \cdot (rs_{ID} = ws_{EX1}) \\ & \cdot (ws_{EX1} \neq 0) \\ & \cdot ((OP_{EX1} = LW) + alu2_{EX1}) \\ & ) \end{aligned}$$

Since all instruction results are produced by the end of EX2/MEM, the operands for an instruction are always available by the end of EX1 even if it uses the result of its immediately preceding instruction as a source.

The only stall condition is when the base address operand for a memory instruction is not available by the end of ID. This occurs if the memory instruction (in ID) uses the result of its immediately preceding instruction (in EX1) as its base address, but the instruction will not produce its result until EX2/MEM. The two classes of instructions which do not produce a result until EX2/MEM are LW instructions and ALU instructions which use ALU2.

Note that ALU instructions never need to stall the pipeline. They either use ALU1 if their operands will be available by the end of ID, or ALU2 if their operands will be available by the end of EX1.

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

**Problem M1.9.A** **Interlock vs. Bypassing**

**No.** Data dependencies are preserved with either interlocks or bypassing, so the processors always generate the same results. Bypassing improves performance by eliminating stalls.

**Problem M1.9.B** **Delay Slot**

**Yes.** The instruction following a taken branch is executed on processor A, but killed on processor B so the processors can generate different results.

**Problem M1.9.C** **Structural Hazard**

**No.** Both processors retrieve the same data values. There is only a performance difference because processor A must stall an instruction fetch to allow a load instruction to access memory.

**Problem M1.9.D** **Microcode**

**No.** A wide variety of possible microcoded machines can implement the same user-level ISA semantics and generate the same results for all programs.

**Problem M1.9.E** **Stall Equation**

**Either answer is acceptable depending on assumptions about the compiler and ISA.**
**No:** The machines could always generate the same results for a 32-bit ISA. Also, machine A could implement a 64-bit ISA by using two 32-bit registers to hold each 64-bit value and carefully handling overflow conditions.
**Yes:** The machines could generate different results due to the different overflow properties of 32-bit and 64-bit registers. For example, if a value is shifted left, bits are lost using 32-bit registers that are retained with 64-bit registers.