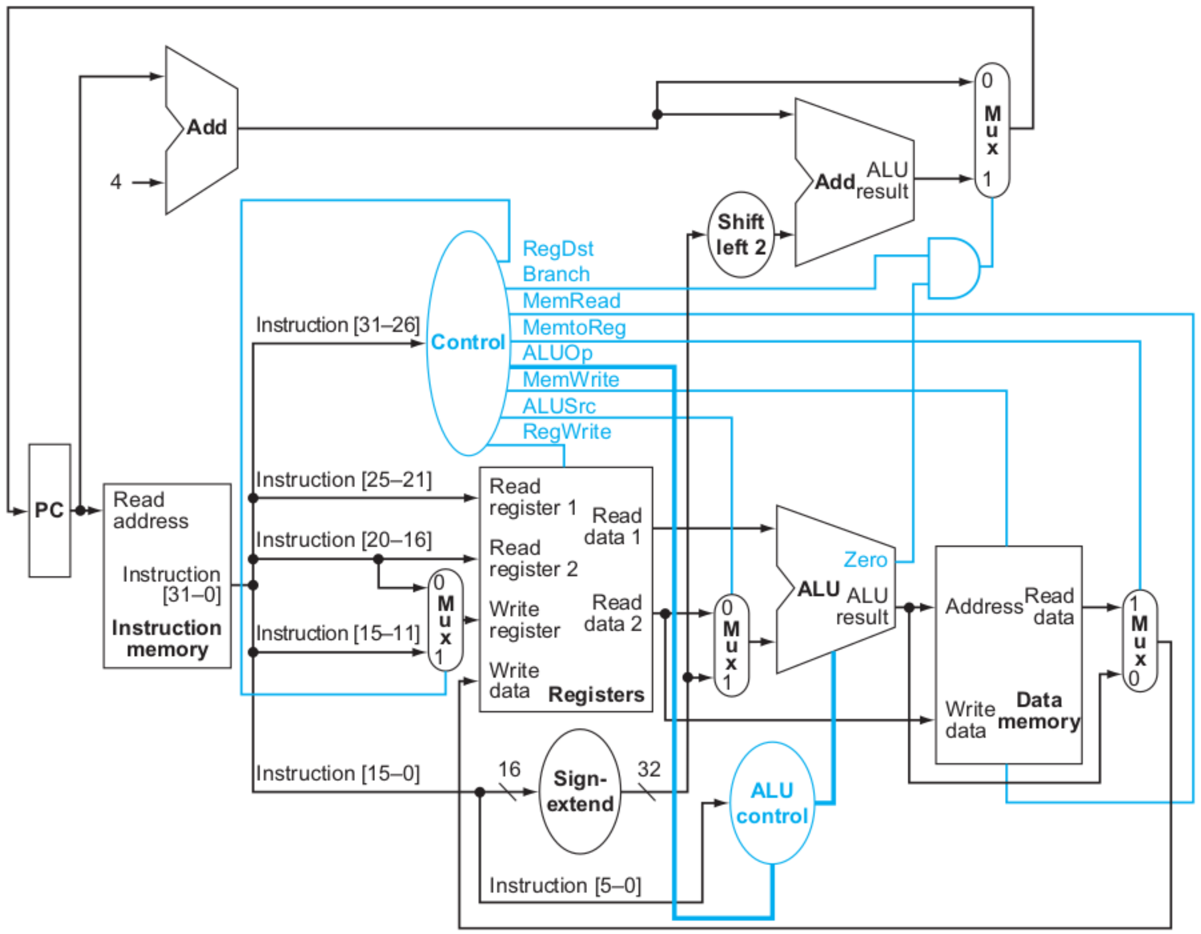
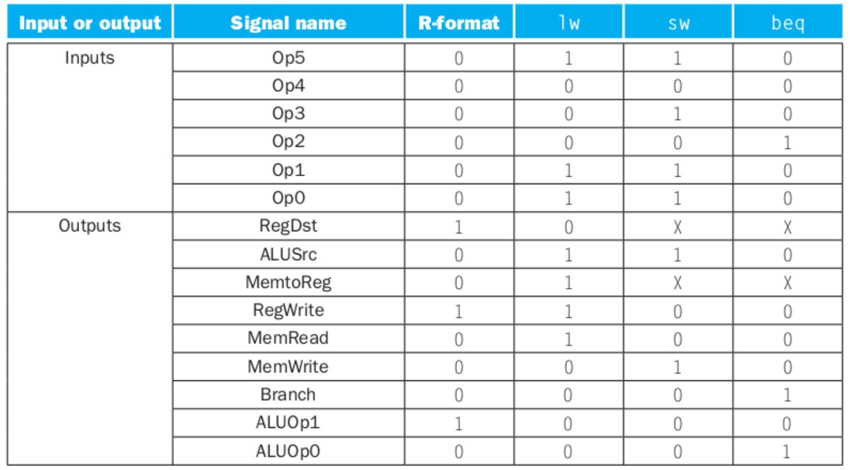
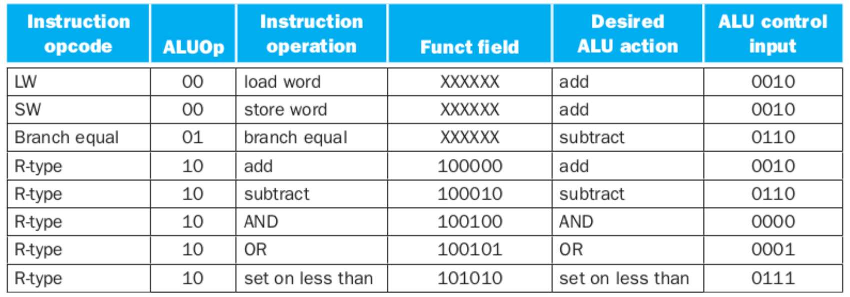
1 Single Cycle Datapath





Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1



Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

1.1 R-Format

31-26	25-21	20-16	15-11	10-6	5-0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Datapath Flow for R-Type Instruction

R-Format : PC ⟶ Instruction Memory ⟶ Register File (Read) ⟶ ALU ⟶ Register File (Write)

Unique ID `funct`, as all `op = 000000`

Format `add rd, rs, rt`
Example `add $s1, $s2, $s3`
Effect `$s1 <- $s2 + $s3`

`RegDst = 1`: destination comes from `rd [15:11]`
`ALUSrc = 0`: second ALU input comes from register file
`MemtoReg = 0`: not interacting with data memory
`RegWrite = 1`: modifying register file
`MemRead = 0`: not interacting with data memory
`MemWrite = 0`: not interacting with data memory
`Branch = 0`: not a branch operation
`ALUOp = 10`: ALU control unit will generate ALU control signal

1.2 I-Format

31-26	25-21	20-16	15-0
op	rs	rt	immediate/address
6 bits	5 bits	5 bits	16 bits

Datapath Flow for `lw` Instruction

`lw` : PC ⟶ Instruction Memory ⟶ Register File (Read) ⟶ ALU ⟶ Data Memory (Read) ⟶ Register File (Write)

Unique ID `op = 100011`

Format `lw rt, imm(rs)`
Example `lw $s1, 100($s2)`
Effect `$s1 <- M[100+$s2]`

`RegDst = 0`: destination comes from `rt [20:16]`
`ALUSrc = 1`: second ALU input comes from sign-extended offset
`MemtoReg = 1`: loading data from memory to register
`RegWrite = 1`: modifying register file
`MemRead = 1`: loading data from memory to register
`MemWrite = 0`: not modifying data memory
`Branch = 0`: not a branch operation
`ALUOp = 00`: ALU control signal set to `0010`, i.e. addition

Datapath Flow for `sw` Instruction

`sw` : PC ⟶ Instruction Memory ⟶ Register File (Read) ⟶ ALU ⟶ Data Memory (Read)

Unique ID `op = 101011`

Format `sw rt, imm(rs)`
Example `sw $s1, 100($s2)`
Effect `M[100+$s2] <- $s1`

`RegDst = X`: destination register is garbage as `RegWrite = 0`
`ALUSrc = 1`: second ALU input comes from sign-extended offset
`MemtoReg = X`: destination register is garbage as `RegWrite = 0`
`RegWrite = 0`: not modifying register file
`MemRead = 0`: not reading data memory
`MemWrite = 1`: storing data from register to data memory
`Branch = 0`: not a branch operation
`ALUOp = 00`: ALU control signal set to `0010`, i.e. addition

Datapath Flow for `beq` Instruction

$$\texttt{beq} : \text{PC} \longrightarrow \text{Instruction Memory} \longrightarrow \left\{ \begin{array}{l} \text{Register File (Read)} \longrightarrow \text{ALU} \\ \text{Sign Extension Unit} \longrightarrow \text{Shift Left 2 Unit} \longrightarrow \text{Adder} \end{array} \right. \longrightarrow \text{Branch MUX} \longrightarrow \text{PC}$$

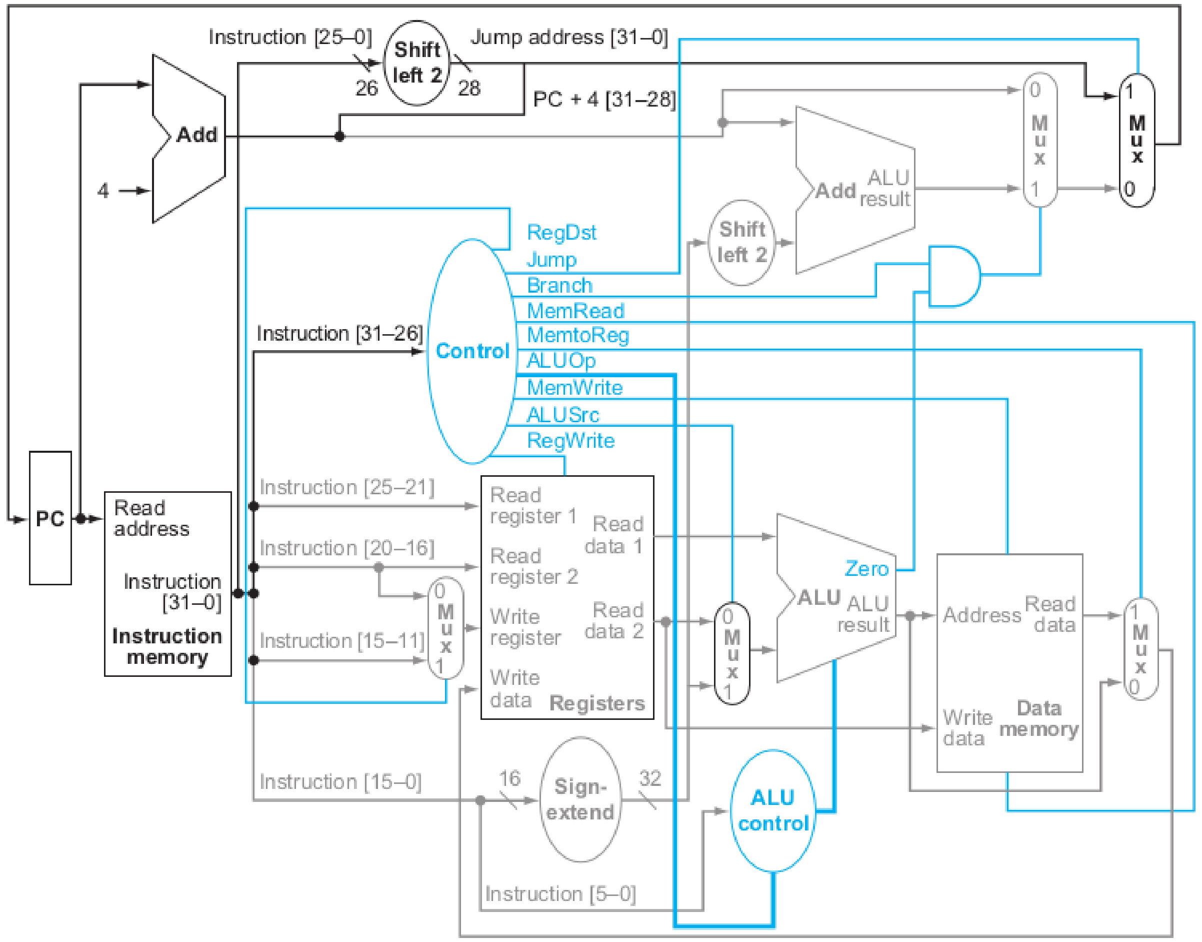
Unique ID `op = 000100`

Format `beq rs, rt, imm`
Example `beq $s1, $s2, 100`
Effect `if ($s1 == $s2) then PC <- PC + 4 + 100 * 4`

`RegDst = X`: destination register is garbage as `RegWrite = 0`
`ALUSrc = 0`: second ALU input comes from register file
`MemtoReg = X`: destination register is garbage as `RegWrite = 0`
`RegWrite = 0`: not modifying register file
`MemRead = 0`: not interacting with data memory
`MemWrite = 0`: not interacting with data memory
`Branch = 1`: is a branch operation
`ALUOp = 01`: ALU control signal set to `0110` , i.e. subtraction.

1.3 Modifying Datapath for J-Format

31-26	25-0
op	target address
6 bits	26 bits



Unique ID `op = 000010`

Format `j imm`
Example `j 3000`
Effect `PC <- 3000 * 4 = 12000`

New Hardware Performing bitwise `shift left 2` to the 26-bit immediate recovers the 28-bit jump target.

New MUX When `jump` signal is on, we feed the jump target to `PC`; otherwise we proceed with `PC + 4` or branch target.

New Control Signal When control unit sees the instruction is `j`, `jump` is turned on and all other control signals are as follows:

`RegDst = X`: destination register is garbage as `RegWrite = 0`
`ALUSrc = X`: jump completed before going through ALUSrc MUX
`MemtoReg = X`: destination register is garbage as `RegWrite = 0`
`RegWrite = 0`: avoid writing garbage data
`MemRead = 0`: avoid reading garbage data
`MemWrite = 0`: avoid writing garbage data
`Branch = X`: jump MUX is after branch
`ALUOp = X`: no ALU operation needed

1.4 Weakness of Single Cycle Datapath

The single cycle design is too inefficient for modern processors. Since we need to be able to execute any instruction in one cycle, the clock cycle is essentially determined by the longest path in the processor. In MIPS instruction set, `lw` takes the longest path, as it uses all five functional units in series: fetching from the instruction memory, reading the register file, calculating the address using the ALU, reading the data memory, and finally writing back to the register file. Thus, the overall performance of a single-cycle implementation is likely to be poor. In contrast, modern processors use the **multicycle datapath** design with an implementation technique called **pipelining**.

2 Multicycle Datapath and Pipelining

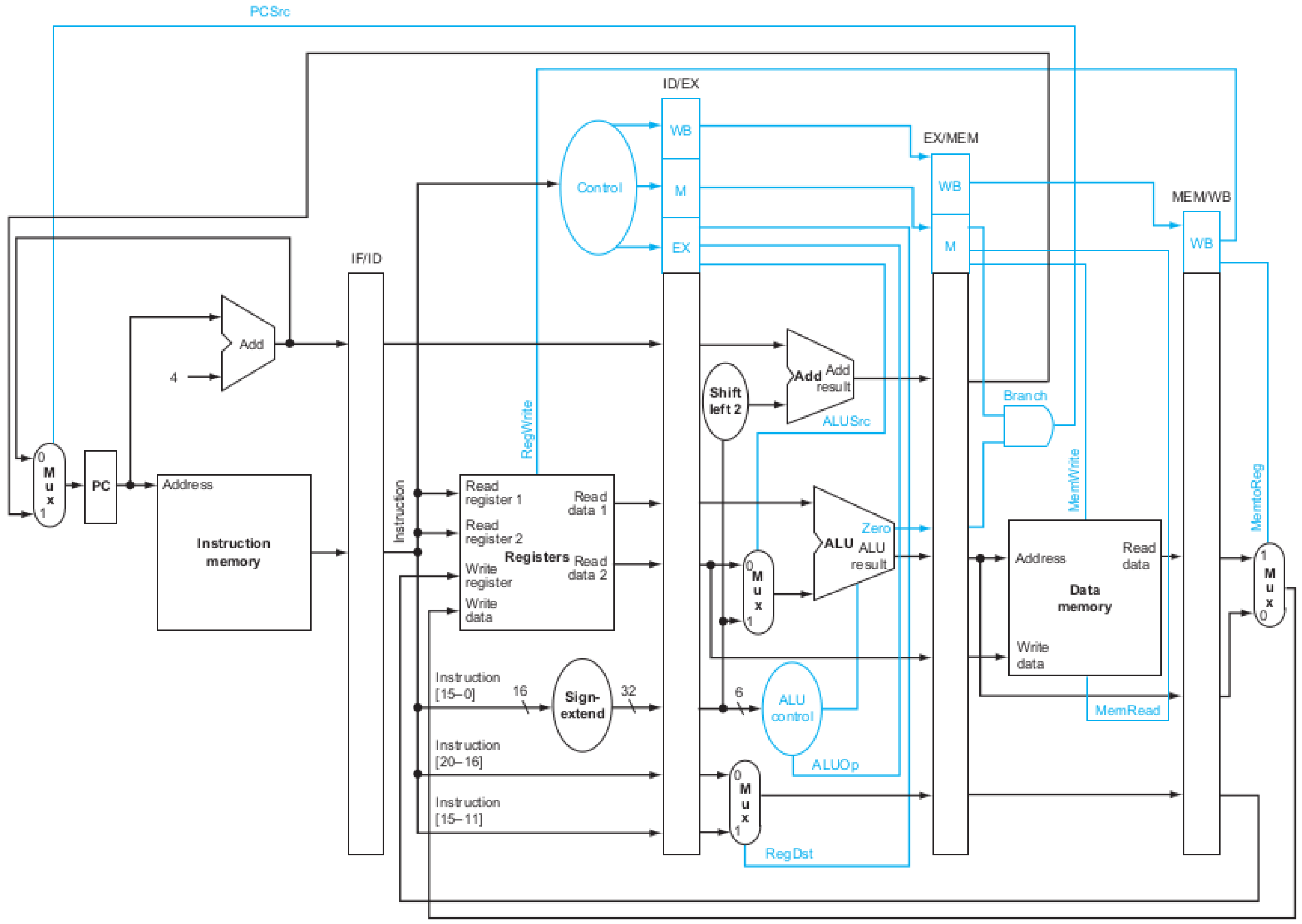


Figure 2.1 Naive Pipelining with Control Signals Labeled (branch in MEM and no hazard handling)

2.1 Overview

Multicycle Datapath

- Single long clock cycle ⟶ several shorter clock cycles.
- Each instruction takes several clock systems to execute.
- Intermediate results are stored in pipeline registers.
- Execution Time: each $cc = 200ps$; total $= 200 \cdot 5 = 1000ps$.
- Conclusion: slower than the original single cycle datapath.

Pipeling

- Overlapping execution of multiple instructions.
- New instruction is fetched every clock cycle.
- Benefit: improving the overall throughput of instructions (more instructions completed in a given time) however not decreasing execution time of individual instruction.

Pipeline Hazards Overview

- **Structural Hazard**: if instruction and data are in the same memory, instruction fetch cannot overlap with load/store.
  - Solution: instruction memory and data memory.
- **Data Hazard**: result of one instruction is needed by next instruction before it is written back to the register file.
  - Solution: Stalling
  - Solution: Forwarding
- **Control Hazard**: conditional branch instructions may change sequence of instructions executed.
  - Solution I: Flushing
  - Improvement: Branch in `ID`
  - Solution II: Code Rearrangement

2.2 Data Hazard

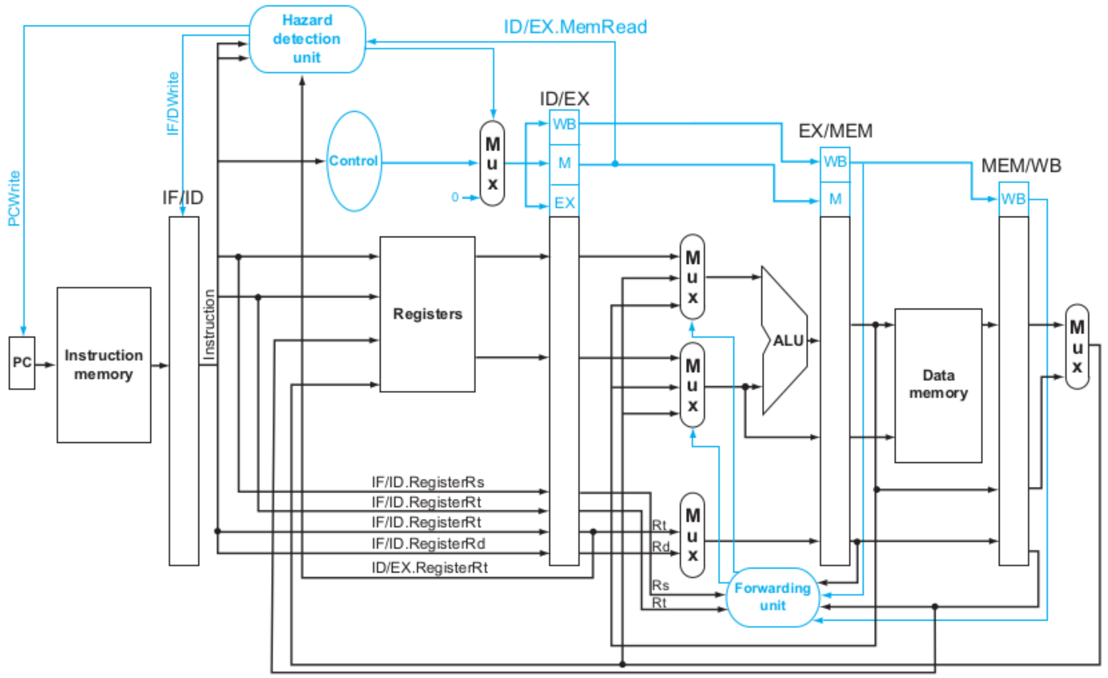
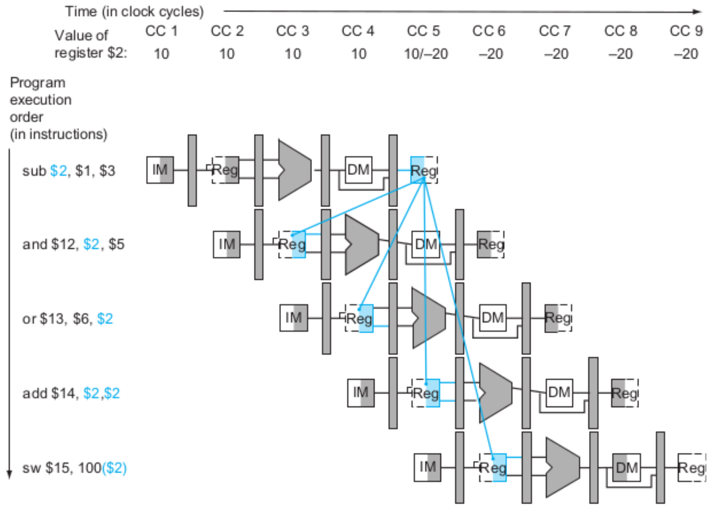


Figure 2.2 Pipeline with Data Forwarding Unit and Data Hazard Detection Unit

Motivation



```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

Observe $2 is written back during the fifth clock cycle but `and` and `or` need its updated value before that, two data hazards occur. The fourth instruction `add` does not create a data hazard as the register write is performed during the first half of the clock cycle and the register read is done during the second. The last instruction `sw` asks for $2 after the fifth clock cycle thus is safe.

Data Hazard Condition

Recall that the data hazard affects the ALU input at `EXE` stage, thus to detect data hazards, we want to check if any of our register read target will be updated by the instruction that is current in `MEM` or `WB` stage. More precisely, since the register read target is stored in `ID/EX` pipeline register and the `RegDst` is stored in `EX/MEM` and `MEM/WB`, we need to compare all possibilities:

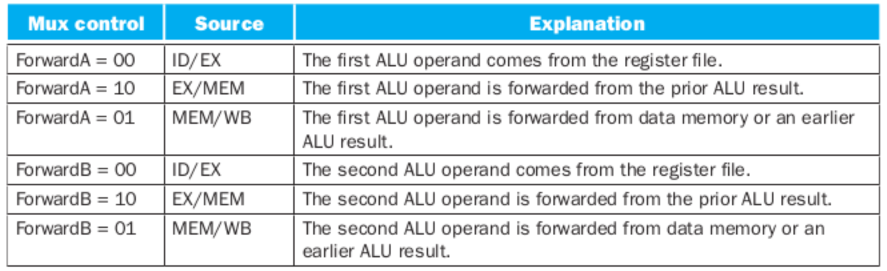
1a. `ID/EX.RegisterRs == EX/MEM.RegisterRd`
1b. `ID/EX.RegisterRt == EX/MEM.RegisterRd`
2a. `ID/EX.RegisterRs == MEM/WB.RegisterRd`
2b. `ID/EX.RegisterRt == MEM/WB.RegisterRd`

The data hazard caused by `and` is of type `1a`: `sub` is one instruction ahead of `and`, so by the time `and` asks for the first ALU input, the correct value would be stored in pipeline register `EX/MEM`. Next, `or` creates a type `2b` data hazard: `sub` is two instructions ahead of `or`, so when `or` needs its second operand, the correct value would be placed in `MEM/WB`.

Besides comparing register targets, we need to check

1. `RegWrite == 1`, i.e. we are updating the register file
2. `RegDst != 0`, i.e. `$0` is not a valid destination and should always have the value 0

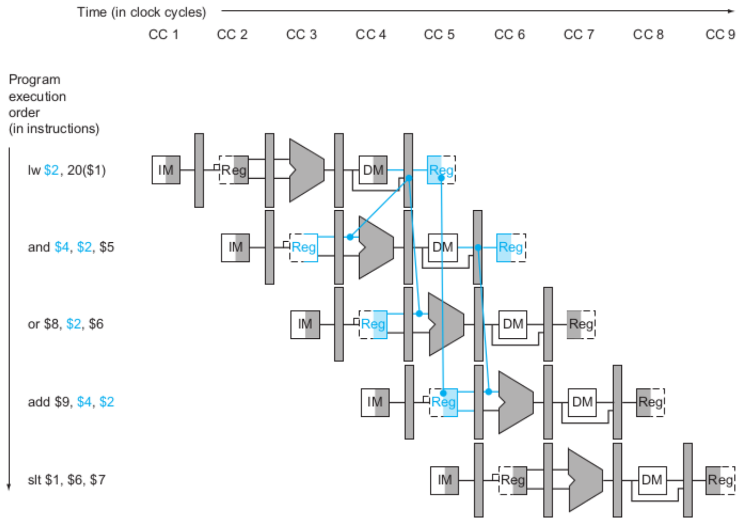
Moreover, if the two previous instructions are modifying the same register, we need to forward the more recently-updated result. In other words, before we forward `MEM/WB.RegisterRd`, we must check whether `EX/MEM` satisfies forwarding requirement. If yes, then the data in `EX/MEM` is used when forwarding.



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 2.3 Data Forwarding Unit Control Signal

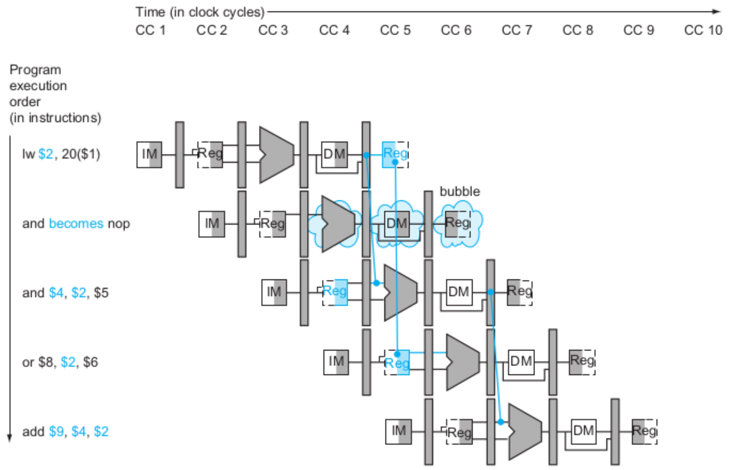
Load-use Hazard



Data forwarding can effectively solve all the data hazards except for one case: a load instruction followed by a register read reading the same register. In this case, one stall must be forced. To detect possible load-use hazards, we want to check

1. whether the instruction in `EX` stage is a load
2. if the instruction in `ID` stage wants to read the `lw` destination

If both satisfies, we stall both `ID` and `IF` stages by replacing the current `ID` instruction with an instruction that has no effect: **nop**. By deasserting all nine control signals in the `EX`, `MEM`, and `WB` stages, we create a "do-nothing" or nop instruction.



Algorithm: Data Forward Unit

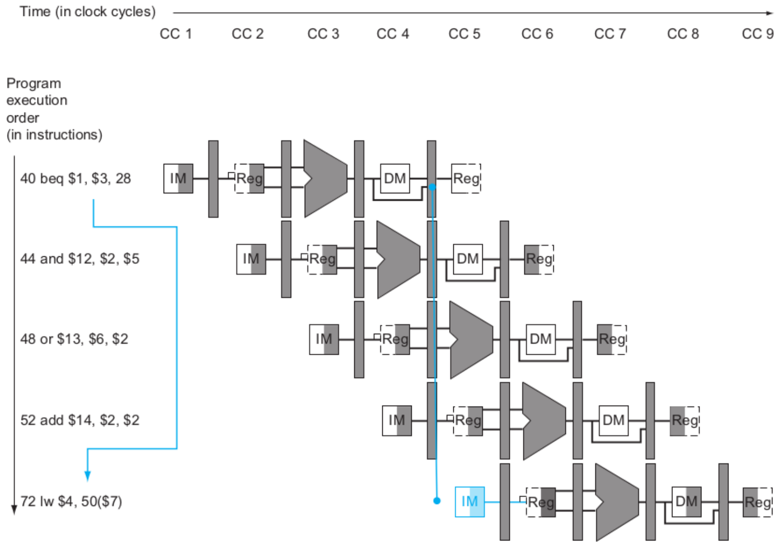
```
if ((EX/MEM.RegWrite == 1)
    and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
then ForwardA = 10

if ((EX/MEM.RegWrite == 1)
    and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
then ForwardB = 10

if ((MEM/WB.RegWrite == 1)
    and (MEM/WB.RegisterRd != 0)
    and not ((EX/MEM.RegWrite == 1)
             and (EX/MEM.RegisterRd != 0)
             and (EX/MEM.RegisterRd != ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
then ForwardA = 01

if ((MEM/WB.RegWrite == 1)
    and (MEM/WB.RegisterRd != 0)
    and not ((EX/MEM.RegWrite == 1)
             and (EX/MEM.RegisterRd != 0)
             and (EX/MEM.RegisterRd != ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
then ForwardB = 01
```

Algorithm: Data Hazard Detection Unit

```
if ((ID/EX.MemRead == 1)
    and ((ID/EX.RegisterRt == IF/ID.RegisterRs) or
         (ID/EX.RegisterRt == IF/ID.RegisterRt)))
then STALL
```

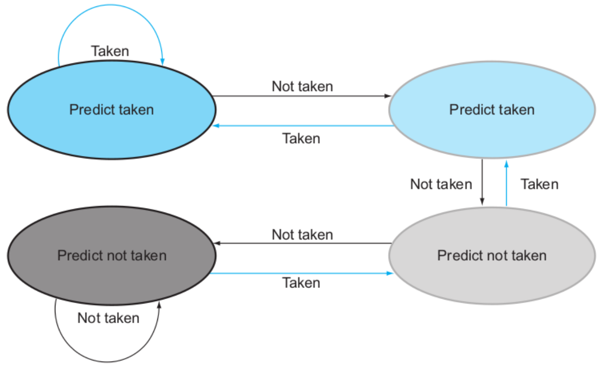
2.3 Control Hazard

Motivation



An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to take the branch doesn't occur until the `MEM` stage. This delay in determining the proper instruction to fetch is called a **control hazard** or **branch hazard** and we will look at two possible solutions.

Solution II. Dynamic Prediction: Tracking Past Results



In an aggressive pipeline, a simple static prediction might not be enough. One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed. If so, predict we will branch this time as well. This is called **dynamic branch prediction**.

One implementation is a **branch prediction buffer** or **branch history table**, which is a small memory indexed by the lower portion of the address of the branch instruction. However, the simple 1-bit prediction scheme might lead us predicting incorrectly twice (consider a while loop). To remedy this weakenss, we use a 2-bit prediction scheme, i.e. a prediction must be wrong twice before it is changed.

Solution I. Static Prediction: Assume Branch Not Taken

Since stalling until the branch decision is complete is too slow (wasting three cycles), one solution is to assume that the branch will not be taken and thus continue execution down the sequential instruction stream. If the branch is taken, however, we flush the three instructions in `IF`, `ID`, and `EX` stages by changing the control values to `0`s, then continue execution at the branch target. If branches are untaken half of the time, this optimization halves the cost of control hazards.

Currently, the branch decision happens in `MEM` stage, meaning that the penalty for wrong prediction is three flushes. To reduce the cost of the taken branch, we move the branch execution up in the pipeline to decrease the number of instructions to be flushed. This requires two actions to occur earlier: computing the branch target address and evaluting the branch decision.

Computing the branch target address is the easy part: PC and the offset are already stored in `IF/ID` register, so we just move the branch adder to the `ID` stage. The branch decision itself is harder. Take `beq` as an example, we need to compare two register reads during the `ID` stage (this can be accomplished by `XOR`ing on respective bits then `OR`ing the result). However, this may lead to additional forwarding and hazard detection unit and data in `ID` stage. Consider the following two complications.

1. During `ID`, we must decode the instruction and decide whether we need to forward the data to the equality unit. We then complete the equality comparison so if the instruction is a branch and is taken, we can set the PC to the branch target address. Recall that forwarding was formerly handled by the ALU forwarding logic, introducing the equality test unit in `ID` requires some new forwarding logic. Note that the bypassed source operands may come from either `EX/MEM` or `MEM/WB`.

2. Because the values in a branch comparison are needed during `ID` but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed. For example, if an ALU instruction immediately proceeding a branch updates one of the operands of branch, we need one stall so that `EX` for the ALU instruction is completed and can be forwarded. Similarly, if a load is immediately followed by a conditional branch depending on the load result, two stalls will be needed.

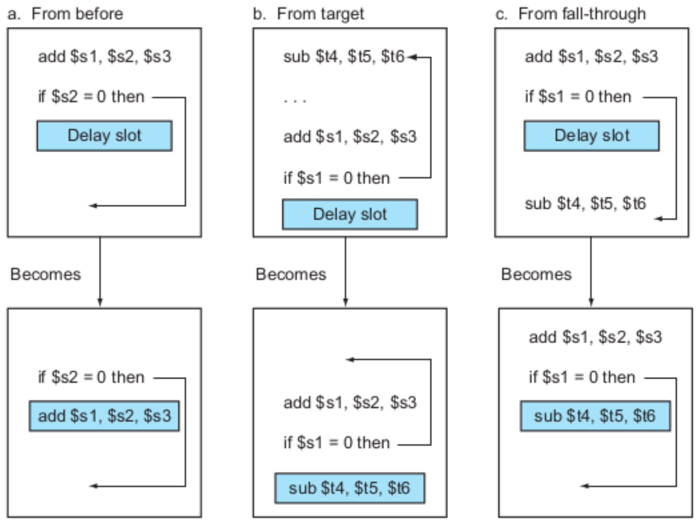
Despite these difficulties, moving the branch execution to the `ID` stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely the one immediately follows it.

To flush instructions in the `IF` stage, we introduce a new control line, `IF.Flush`, that zeros the instruction field of the `IF/ID` pipeline register and transforms the instruction into a nop as it has no action and changes no state.

Assembler/Compiler Optimization: Delayed Branch

The five-stage MIPS pipeline allows **delayed branch** to handle control hazards, meaning the compiler and assembler will try to place an instruction that does not affect the branch (that instruction always executes) after the branch in the **branch delay slot**, the slot directly after a delayed branch instruction.

The limitations on delayed branch scheduling arises from (1) the restrictions on the instruction that are scheduled into the delayed slots and (2) our ability to predict at compile time whether a branch is likely to be taken or not. Moreover, as the processors go to both longer pipelines and issuing multiple instructions per clock cycle, the branch delay becomes longer, and a single delay slot is insufficient. Hence, delayed branches has lost popularity compared to more expensive but more flexible dynamic approaches.



This figure shows three cases for scheduling the branch delay slot.

a This is the best choice as the delay slot is filled with an independent instruction from before the branch.
b We cannot move `add` because it updates `$s1`, so `sub` is moved into the slot. Note that the instruction is usually copied so it can be reached by another path. This strategy is preferred when the branch is taken with high probability, such as a loop branch.
c Finally, the branch may be scheduled from the not-taken fall through. We prefer this branch when it is taken with low probability, as we basically predict that the branch won't be taken (so `sub` is not skipped).

To make this optimization legal for (b) or (c), it must be OK (the work is wasted, but the program will still execute correctly) to execute the `sub` instruction when the branch goes in the unexpected direction.

2.4 Problem Solving

Code Rearrangement Guidelines

1. Code behavior should not be affected and the original final state should be achieved after execution.
2. Do not swap lines of code with data. dependencies.
3. Do not swap into or out of any loops.

Code Rearrangement for Data Hazard

Recall that forwarding is enough to handle ALU instructions, but load-use hazards requires one stall even with data forwarding. We can, however, rearrange the code, placing another instruction below the `lw` that would need to execute anyways.

Consider the following example:

```
lw $t1, 0($t0)
lw $t1, 0($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

```
lw $t1, 0($t0)
lw $t1, 0($t0)
lw $t4, 8($t0)
add $t3, $t1, $t2
sw $t3, 12($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

By moving `lw $t4, 8($t0)` to the third line, we solve both load-use hazards currently on line 2-3 and line 5-6.

Code Rearrangement for Control Hazard

```
100 addi $1, $0, 20
104 addi $2, $0, 0
108 lw $3, 0($4)
112 add $2, $2, $3
116 addi $4, $4, 4
120 addi $1, $1, -1
124 bne $1, $0, -5
128 slt $6, $2, $0
132 add $8, $2, $2
136 lw $7, 100($5)
```

Set up:
- 100: `$1 <- 20`
- 104: `$2 <- 0`

Loop:
- 108: read `A[0]` (`$4`: first element)
- 112: Update sum `$2`
- 116: `$4 += 4`
- 120: `$1 -= 1`
- 124: `$1 != 0` → `PC=108` `(124+4-5*4=108)`

Ending:
- 128: `$2 < 0` → `$6 = 1`
- 132 and 136: omitted

In short, this code segment sums up an array of numbers and sets `$6` to 1 if the sum if negative. We want to examine it from different perspectives:

1. Branching in `MEM`
2. Branching in `ID`
3. Using code rearrangement

1. *Branch in* `MEM`

Suppose this program runs on a pipeline that implements **data forwarding** and **load-use stalling** and branch decision is known in `MEM`. How many clock cycles are needed to execute this program?

First, classify the instructions: two instructions happen before the loop, five (`108-124`) during the loop plus one for `lw` stall (`108-112`) plus three for branch flush in each iteration. In addition to this, we need 4 clock cycles of pipeline start up time, so total clock cycles required is

$$4 + 2 + (5 + 1 + 3) \cdot 20 = 186$$

Line `124` is tested 20 times; the first 19 times `bne` fails and three instructions following it are flushed; the 20th test succeeded so `128-136` are executed normally. In total, $19 \cdot 3 = 57$ instructions are flushed.

2. *Branch in* `ID`

Suppose the branch decision is now known in `ID`. How many clock cycles are needed to execute this program? Observe that a branch data hazard now happens at line `120-124`:

```
120 addi $1, $1, -1
124 bne $1, $0, -5
```

Since line `120` modifies `$1` at its `EX` stage, the result won't be ready for `124` at `ID` stage unless one stall is inserted. Other things remain the same (4 for pipeline start up, 2 for first two instructions, 5 loop instructions plus 1 for load-use hazard, 1 for branch data hazard, and 1 for branch flushing, then 2 instructions after the loop), thus the total clock cycle required is

$$4 + 2 + (5 + 1 + 1 + 1) \cdot 20 + 2 = 168$$

3. *Using Code Rearrangement*

Suppose flushing is not supported and branch is done in `ID` stage. Can we use code rearrangement to eliminate hazards completely? Recall that code rearrangement should not affect the outcome; branch in `ID` means 1 branch delay slot; if in `MEM` then 3 slots. Our strategy is thus

- moving an instruction independent of `lw` right after it to handle load-use hazards,
- moving an instruction independent of `bne` right after it to handle branch control hazards, and
- avoid branch data hazard by seperating line `120` and `124`.

```
100 addi $1, $0, 20
104 addi $2, $0, 0
108 lw $3, 0($4)
112 addi $1, $1, -1
116 add $2, $2, $3
120 bne $1, $0, -4
124 addi $4, $4, 4
128 slt $6, $2, $0
132 add $8, $2, $2
136 lw $7, 100($5)
```

The blue line elimints the load-use hazard, the pink line solves the control hazard, and finally, since we moved `addi` "out of" the branch (but not really since it still executes), we need to change the offset for `bne`.

By rearranging the code, no stall is needed and the total execution time is $4+2+20 \cdot 5+3 = 109$, which is much faster than before.

Performance of Pipelined Design

Given the following assumptions:

- 22% `lw`, 11% `sw`, 49% R-format, 16% branches, 2% jumps
- Half of all loads followed by use
- Quarter of all branches are mispredicted
- Jump and branches are determined in `ID`

Calculate the average number of cycles per instruction (CPI):

- `lw` with load-use hazards: $0.22 \cdot 1 + 0.5 \cdot 0.22 \cdot 1 \cdot$
- `sw` and R-format are safe: $(0.11 + 0.49) \cdot 1$
- Jump forces one flush: $0.02 \cdot 1 + 0.02 \cdot 1$
- Branch with control hazards: $0.16 \cdot 1 + 0.25 \cdot 0.16 \cdot 1$
- Total: 1.17

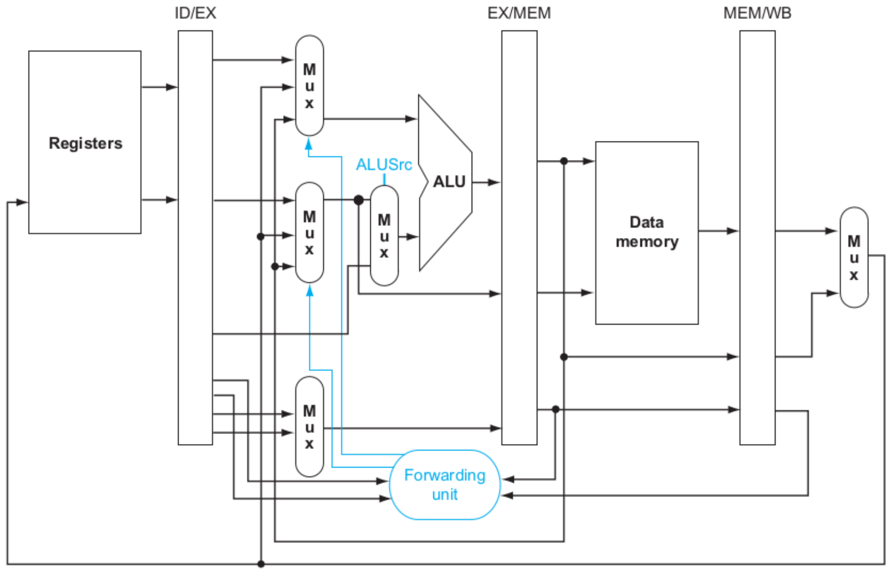


Figure 2.4 ALU Data Forwarding Zoom In (red triangle in Figure 2.5)

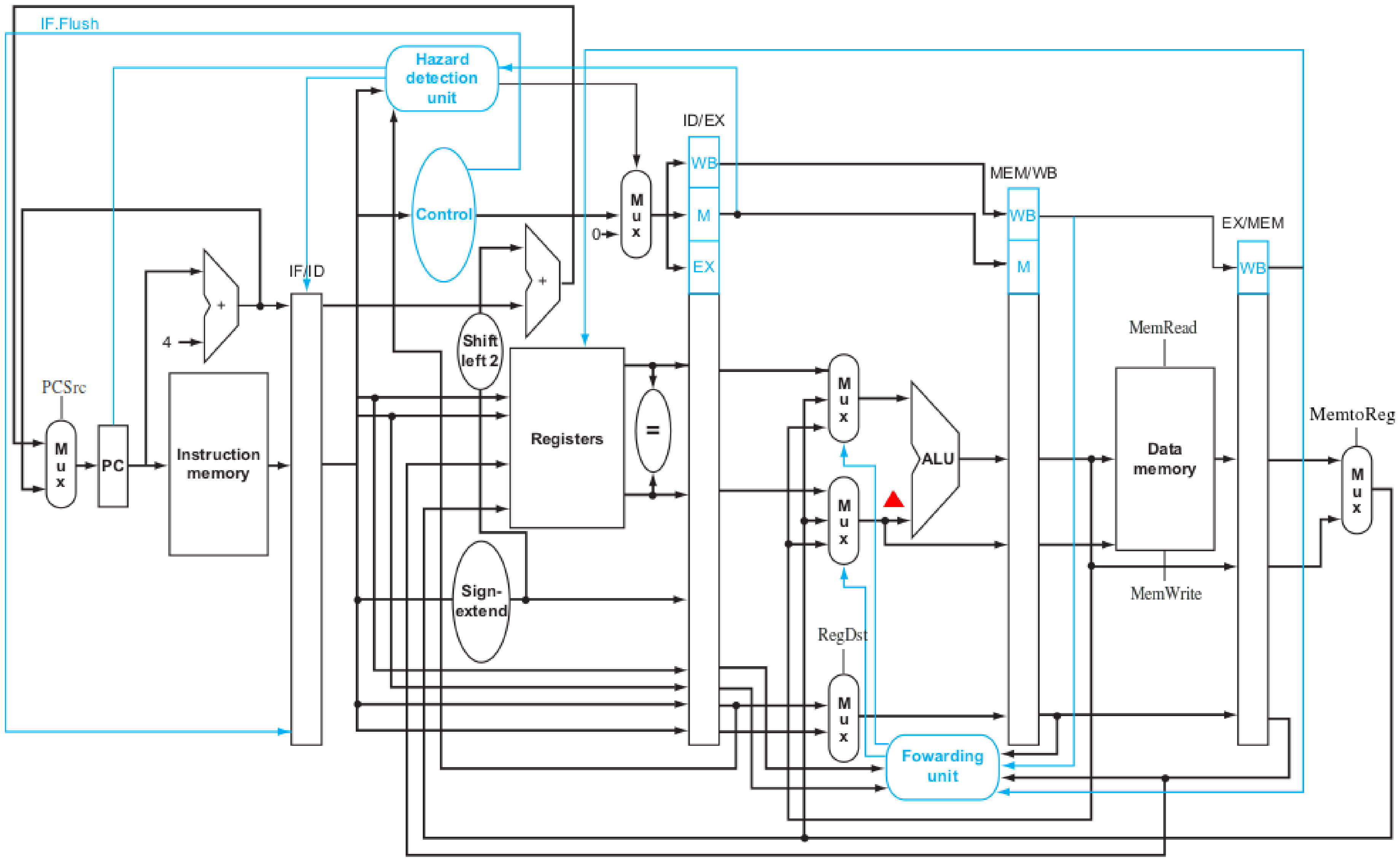


Figure 2.5 Final Pipeline Design with Some Details Missing (Control Lines and ALU)