Pipeline Technique & Branch Prediction Implementation

Computer Architecture 101 – Day 1

Hai Cao

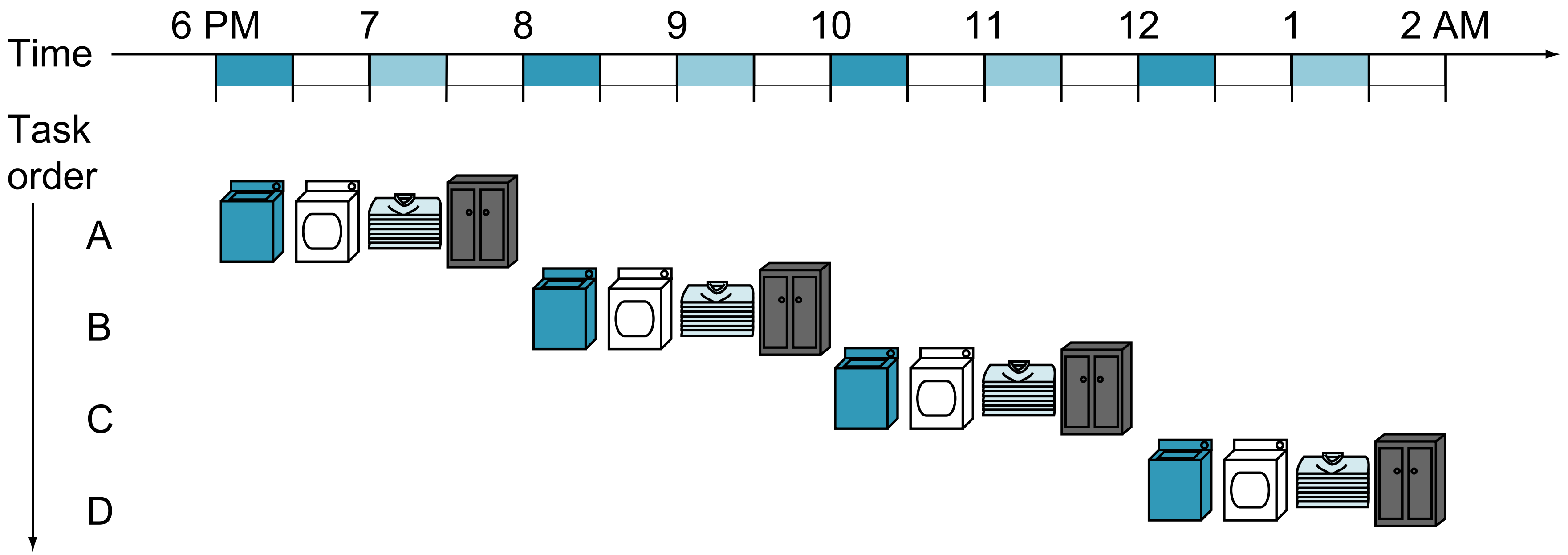
Dept. of Electronics
HCMC University of Technology – VNU-HCM

rev 1.0

Table of contents

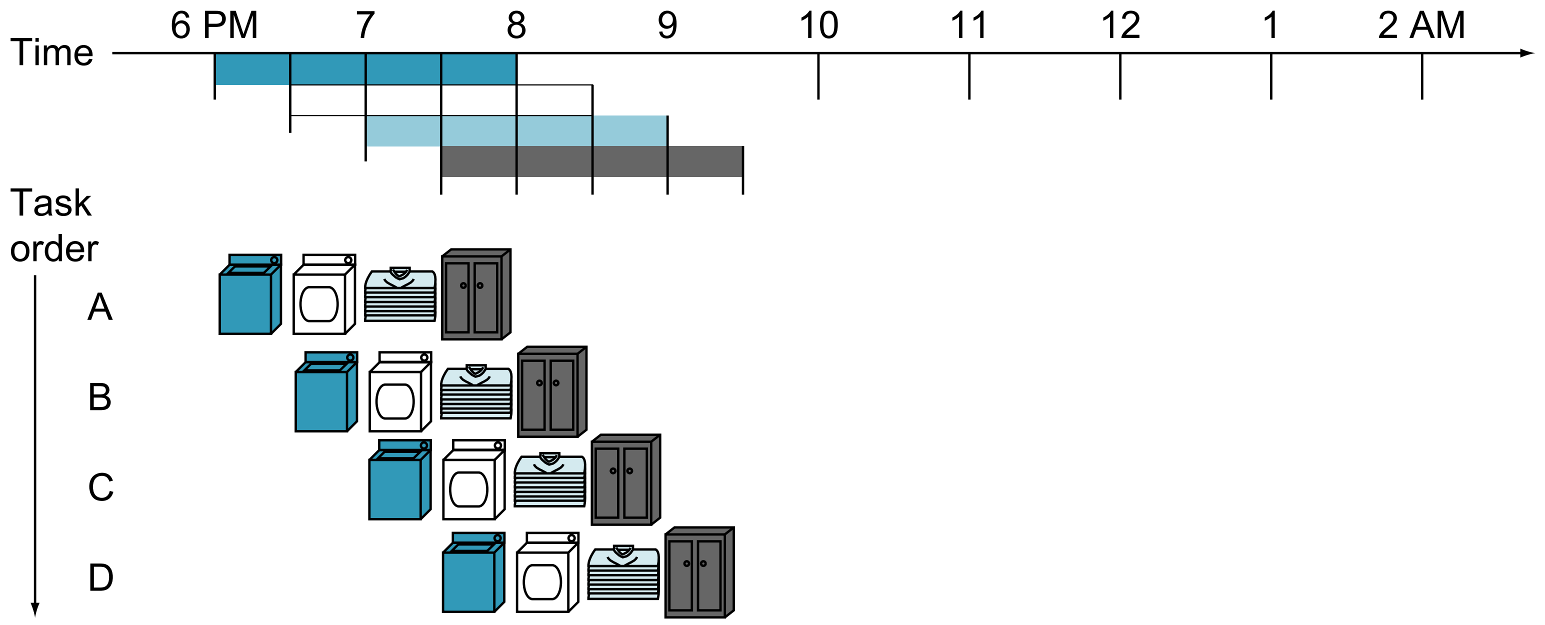
Pipeline Idealism

Laundry Analogy



1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

Laundry Analogy

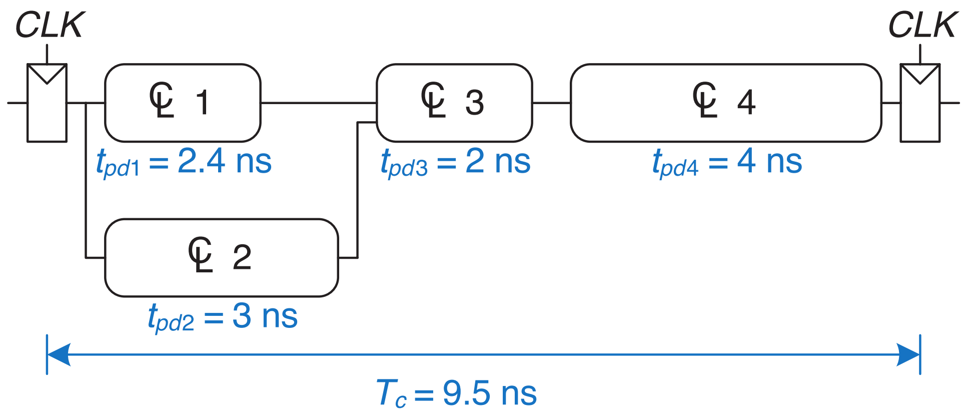


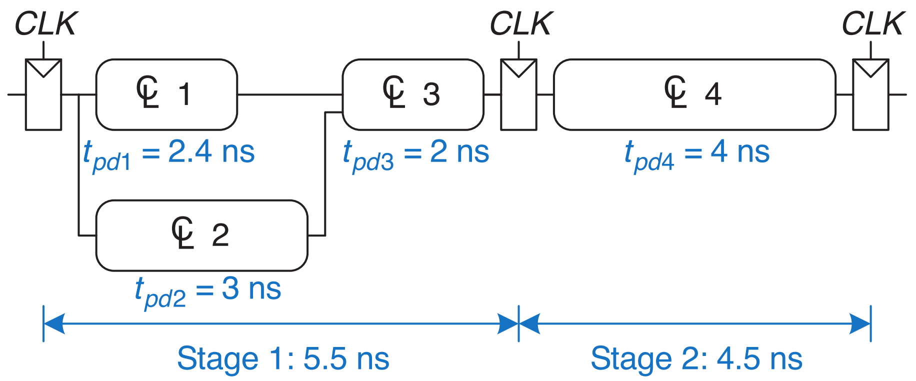
1. **Throughput** increased by 4
2. **Latency** of a load remains unchanged.

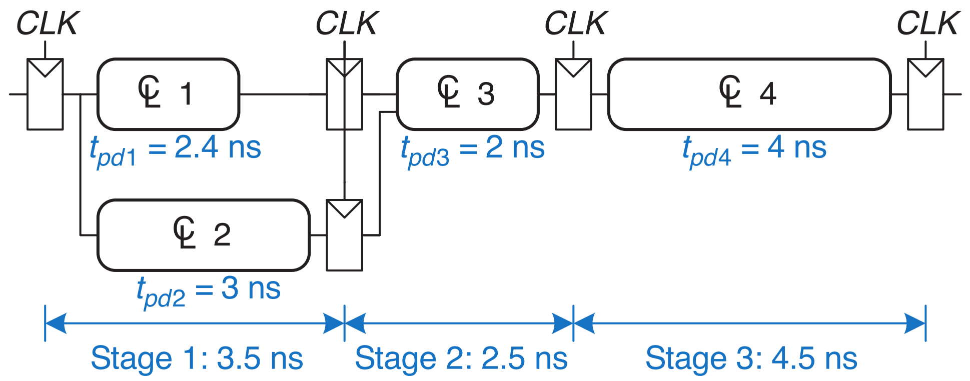
⇒ If the number of loads increases, with higher **throughput**, the completion time decreases.

Motivation

- Increase throughput without additional hardware.
- Allow multiple instructions to overlap in execution.







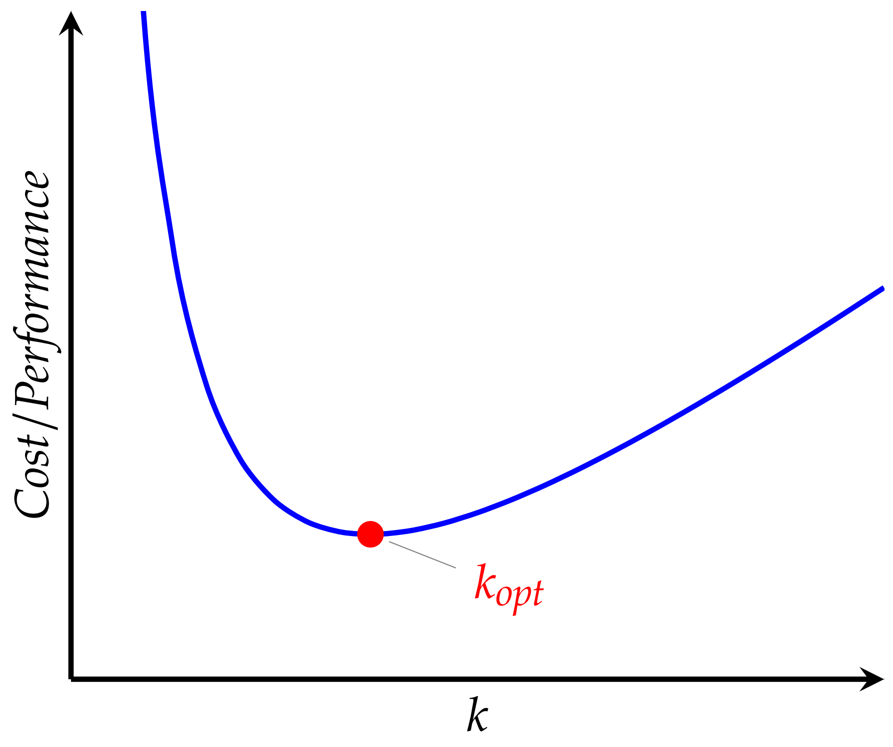
Performance and Cost

Given: A system requires G gates and takes T time to execute. If the system is pipelined, each stage introduces an additional L flip-flops/latches, which require S time. With k is the number of stages:

1. **Throughput/Performance** is the number of tasks processed in a second.
   - Nonpipelined:
   $$P_{nonpilelined} = \frac{1}{T+S}$$
   - Pipelined:
   $$P_{pipelined} = \frac{1}{T/k+S}$$
2. **Cost** refers to the total count of combinational logic gates, which influences the power consumption.
   - Nonpipelined:
   $$C_{nonpilelined} = G + L$$
   - Pipelined:
   $$C_{pipelined} = G + Lk$$

Trade-off

$$
\begin{aligned}
y &= \frac{Cost}{Performance} \\
&= (Lk + G)\left(\frac{T}{k} + S\right) \\
&= LT + GS + LSk + \frac{GT}{k} \\
\frac{\mathrm{d}y}{\mathrm{d}k} &= LS - \frac{GT}{k^2} \\
\Rightarrow k_{opt} &= \sqrt{\frac{LS}{GT}}
\end{aligned}
$$



Pipelined Processor

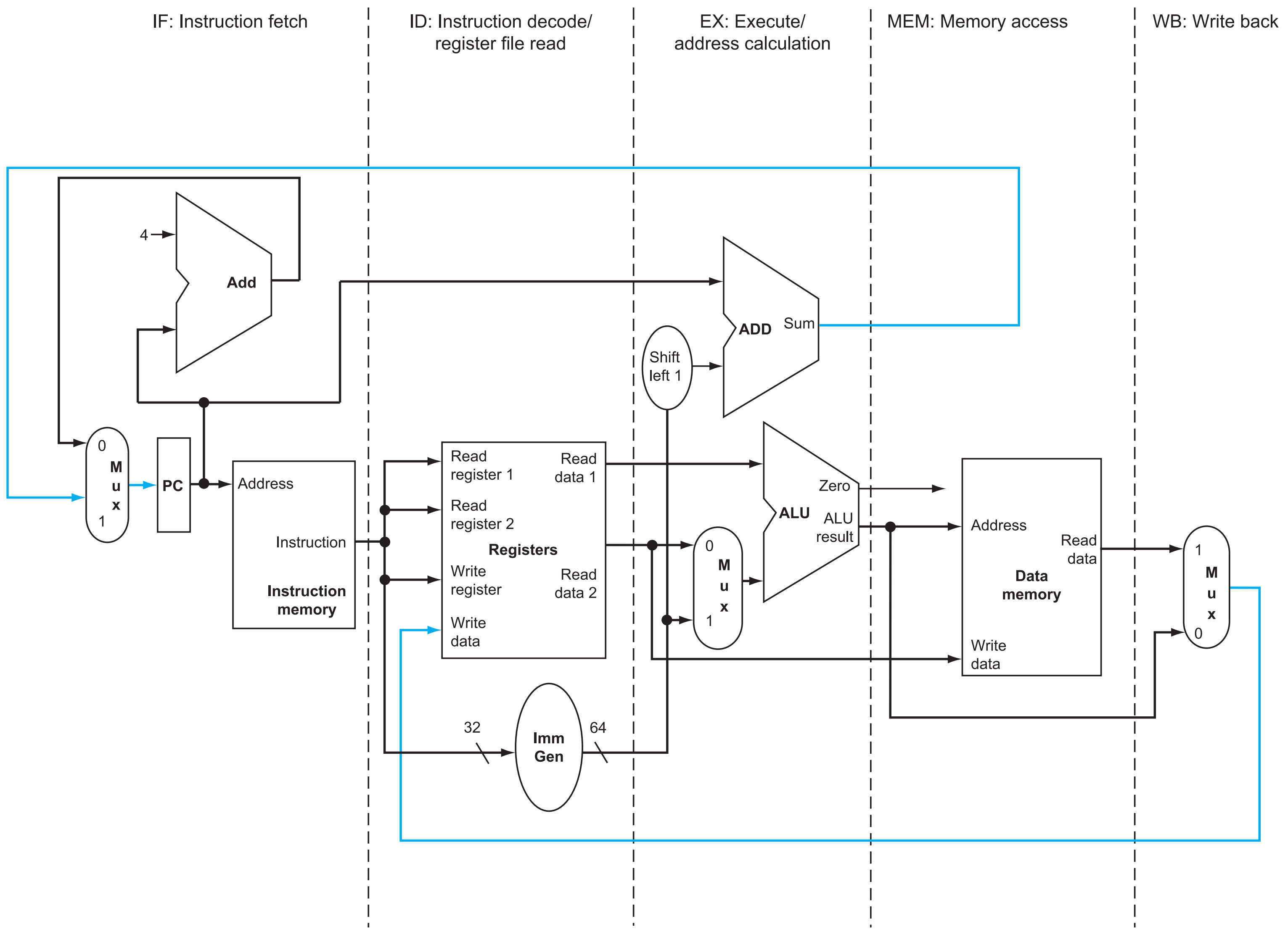
Instruction Pipelining

Each RISC-V instruction is simple and generally needs 5 steps/stages:

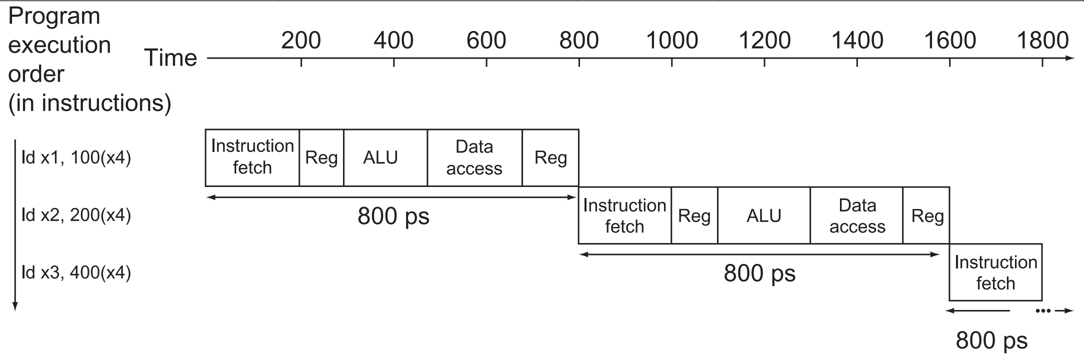
- *F* **Fetch** instruction from memory.
- *D* **Decode** instruction and read data from Regfile.
- *X* **Execute** the operation or compute an address for memory related instructions.
- *M* **Memory** is accessed for loading or storing data.
- *W* **Write Back** data to Regfile.

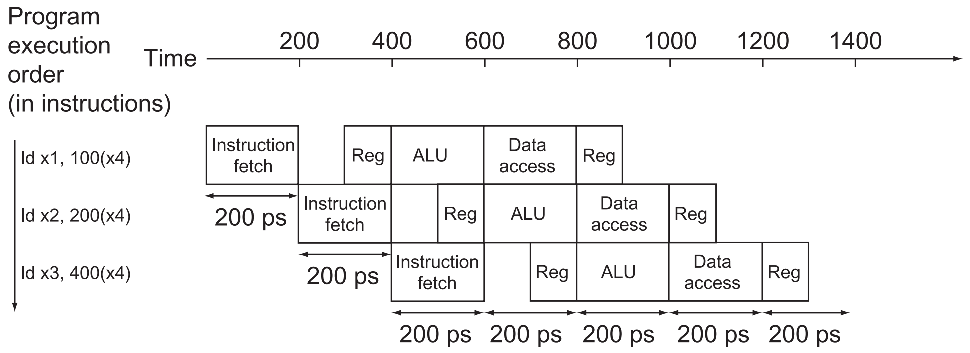
Instructions	`F`	`D`	`X`	`M`	`W`
ALU	✓	✓	✓		✓
Load	✓	✓	✓	✓	✓
Store	✓	✓	✓	✓	
Branch	✓	✓	✓		✓
JAL	✓		✓		✓
JALR	✓	✓	✓		✓

Pipelined Processor



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps





Speed-up

Given:

m	the number of instructions
n	the number of pipelined stages
t	the required time of each stage to complete

The completed time

without	pipelining mnt
with	pipelining $(m+n-1)t$

$$S(n) = \frac{mnt}{(m+n-1)t} = \frac{mn}{m+n-1}$$

$$\lim_{m\to\infty} S(n) = n$$

Exercise

Given: A 5-stage pipelined processor with completed time of each stage as below

t_F = 200 ps

t_D = 300 ps

t_X = 250 ps

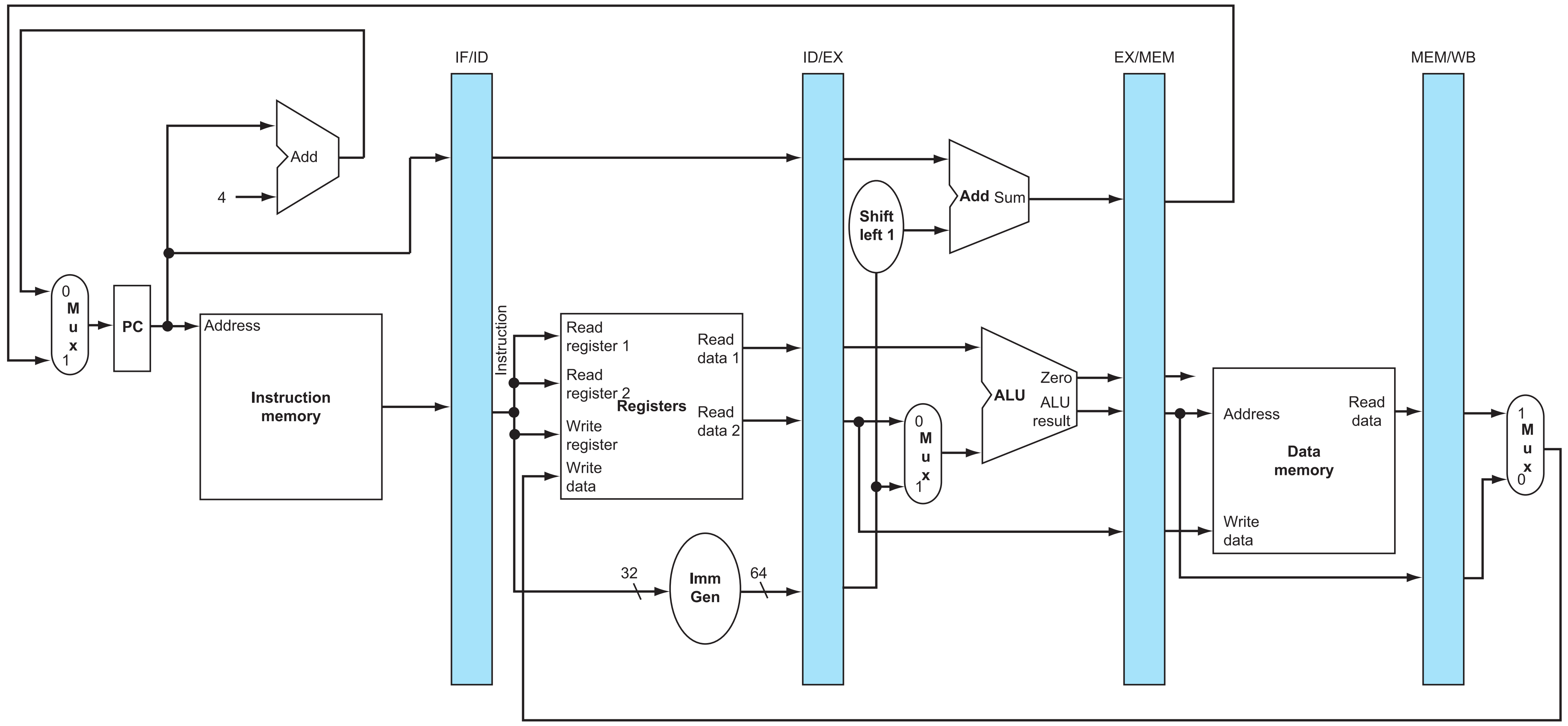
t_M = 200 ps

t_W = 150 ps

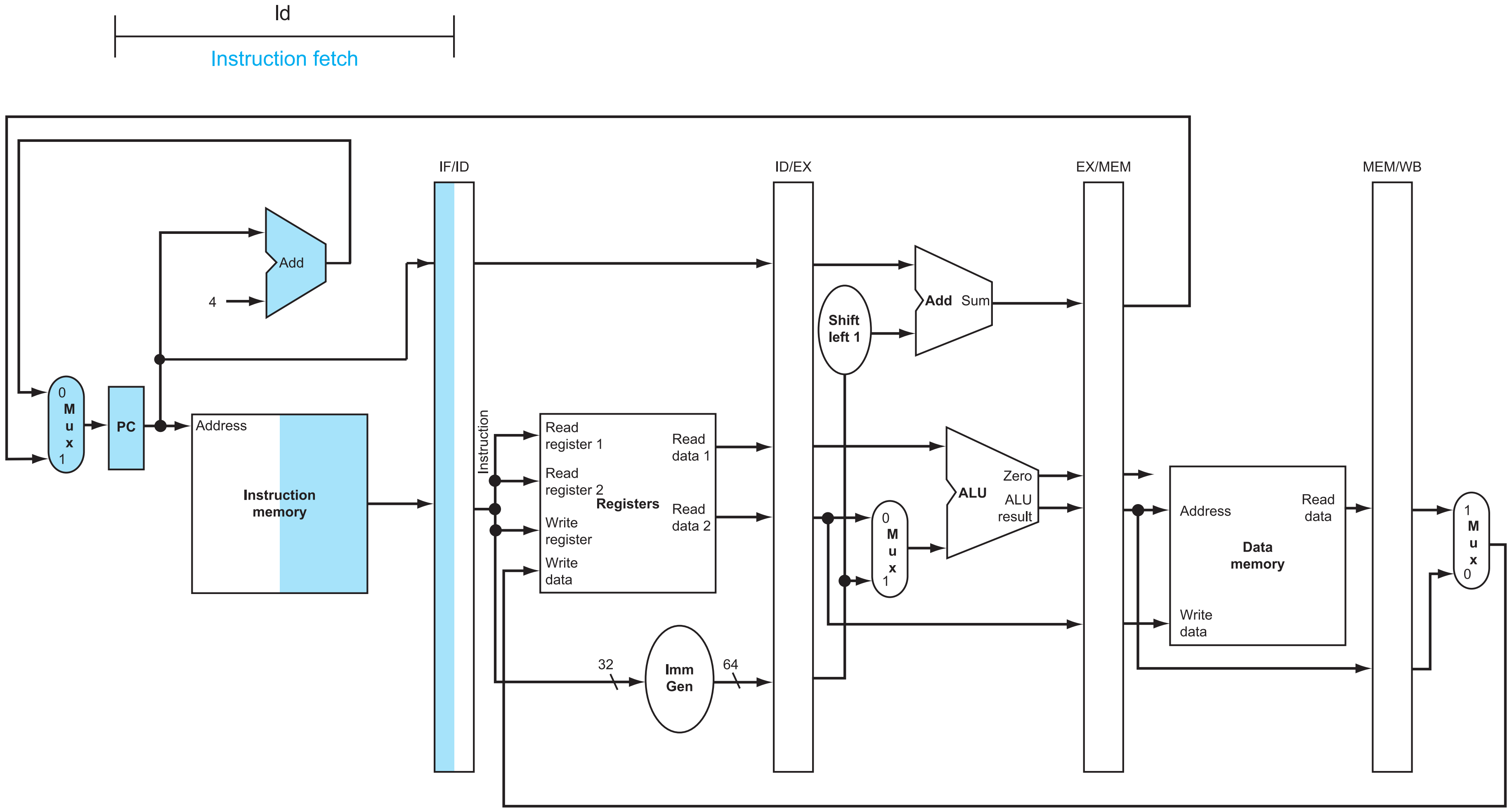
Questions:

1. Before pipelined, what is the maximum frequency of this processor (f_{SC})?
2. What is the maximum frequency of this processor (f_{PL})?
3. If this pipelined processor runs a program of 2300 lines of instruction, how long will it take to complete the program (T)?

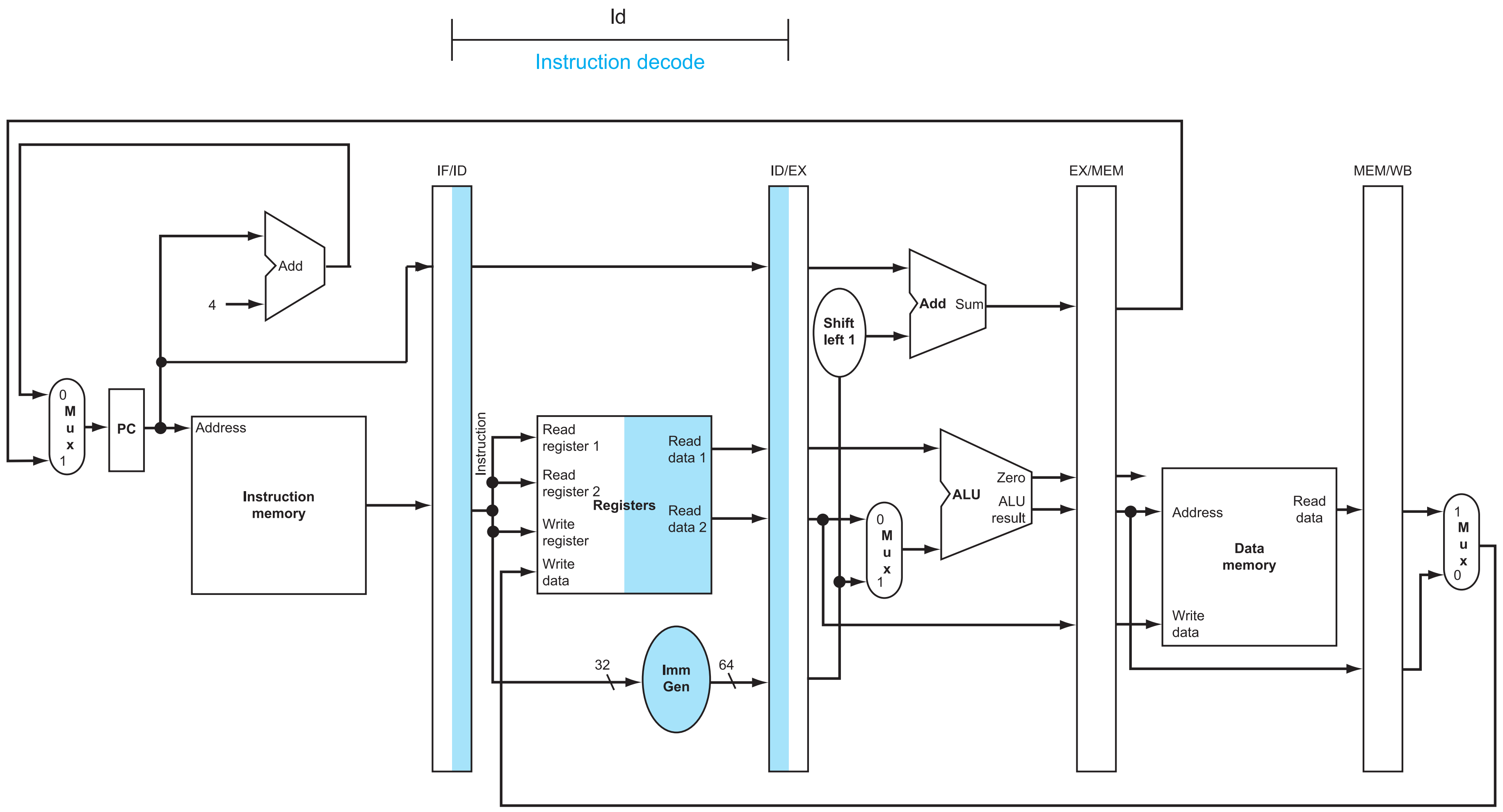
Seperating Stages



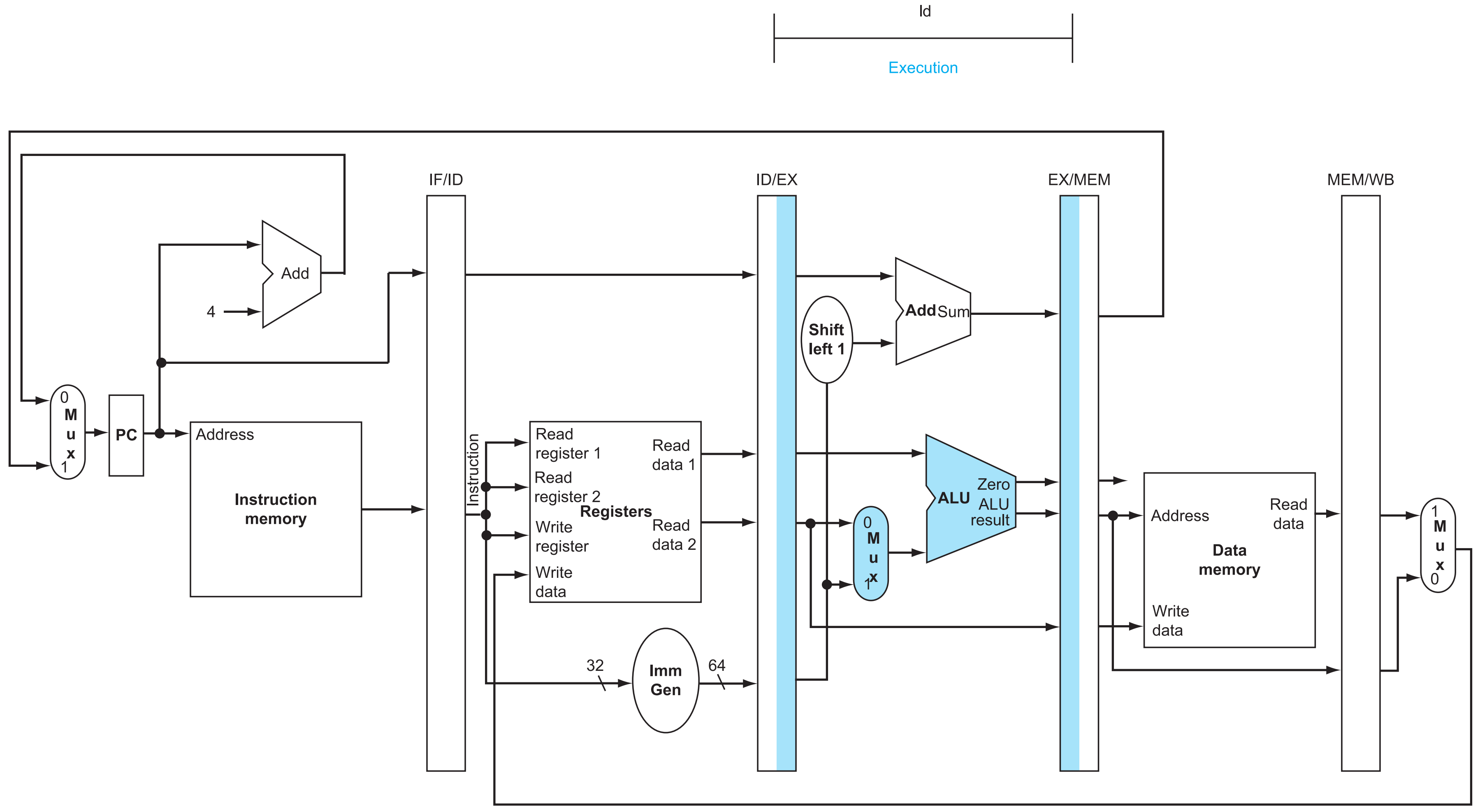
LOAD in IF Stage



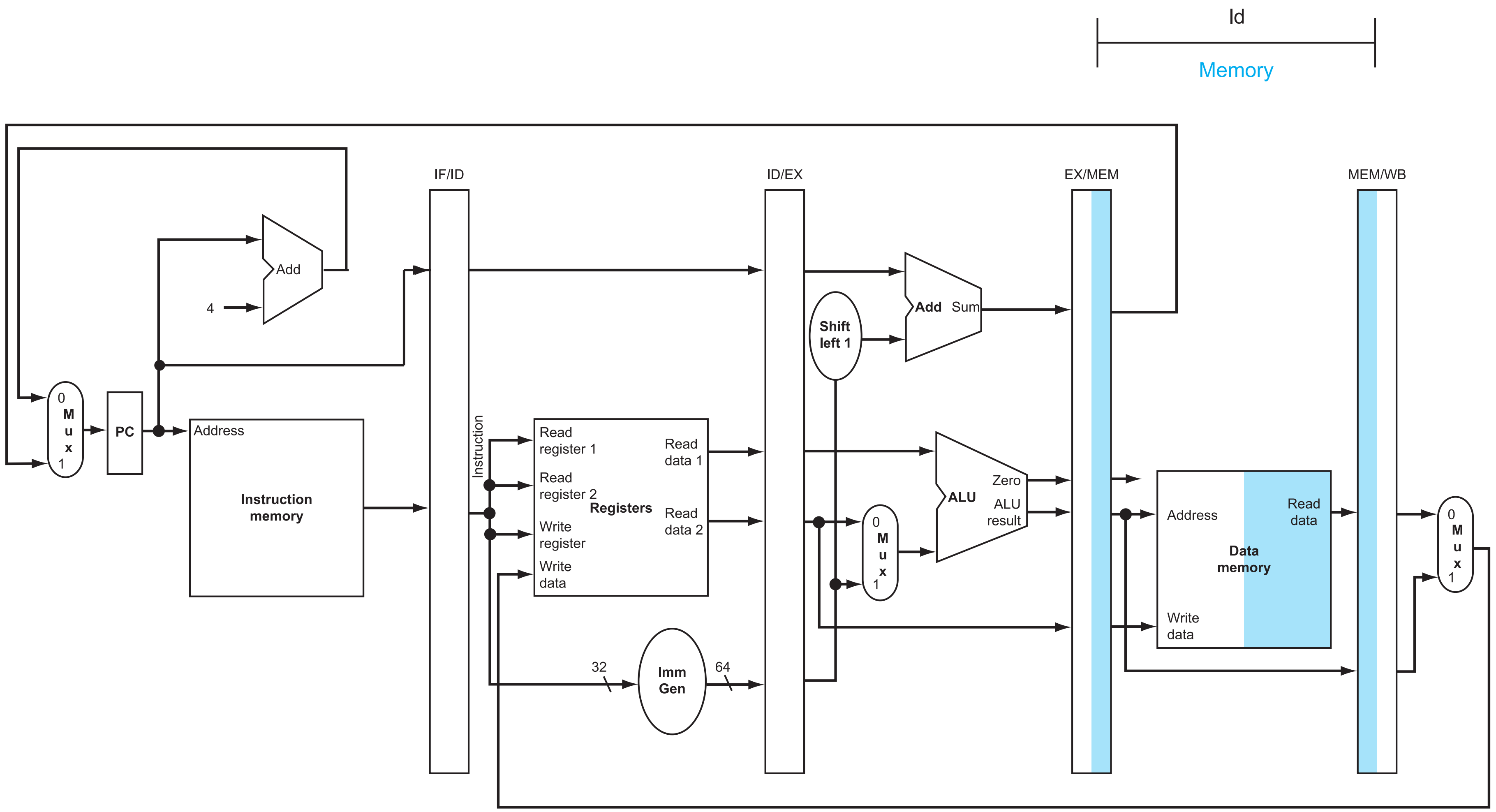
LOAD in ID Stage



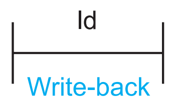
LOAD in EX Stage

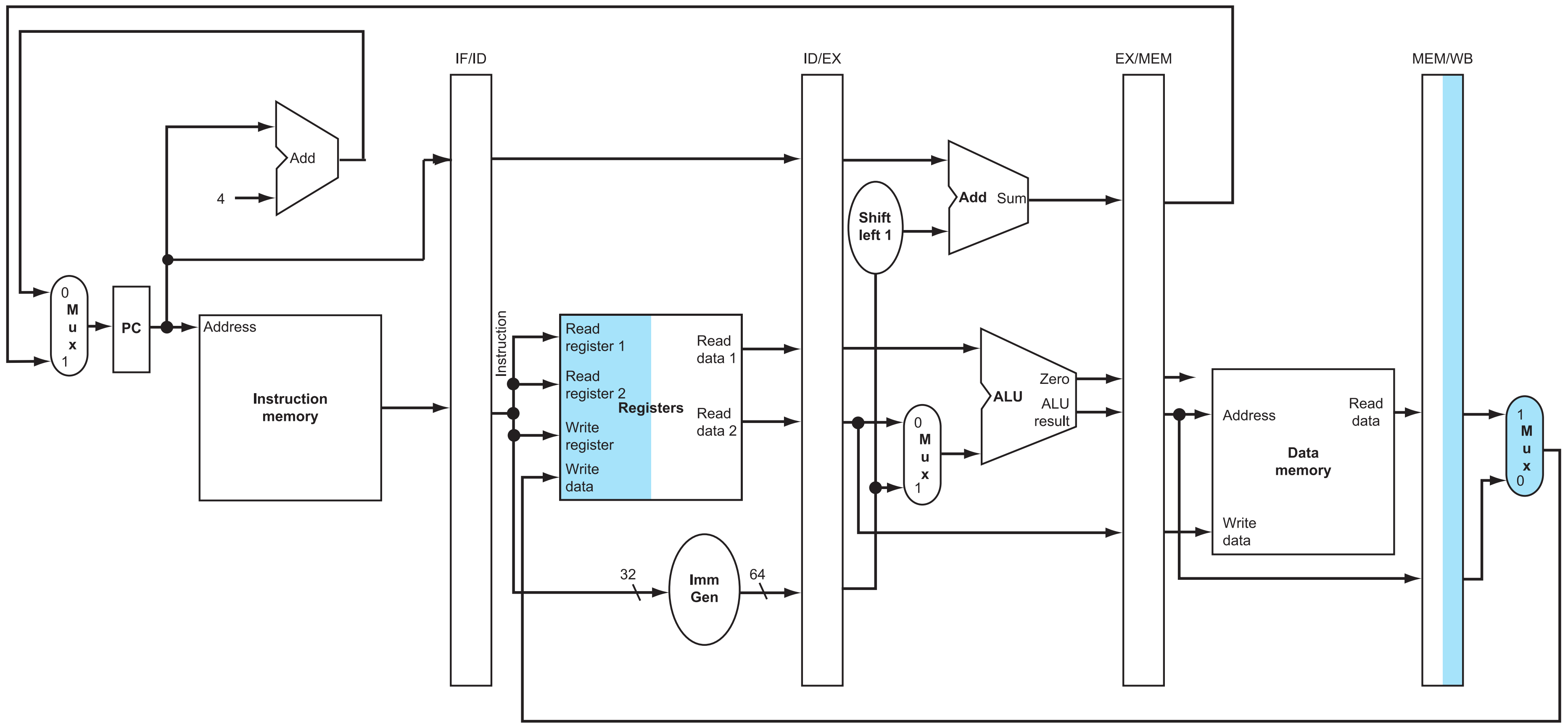


LOAD in MEM Stage

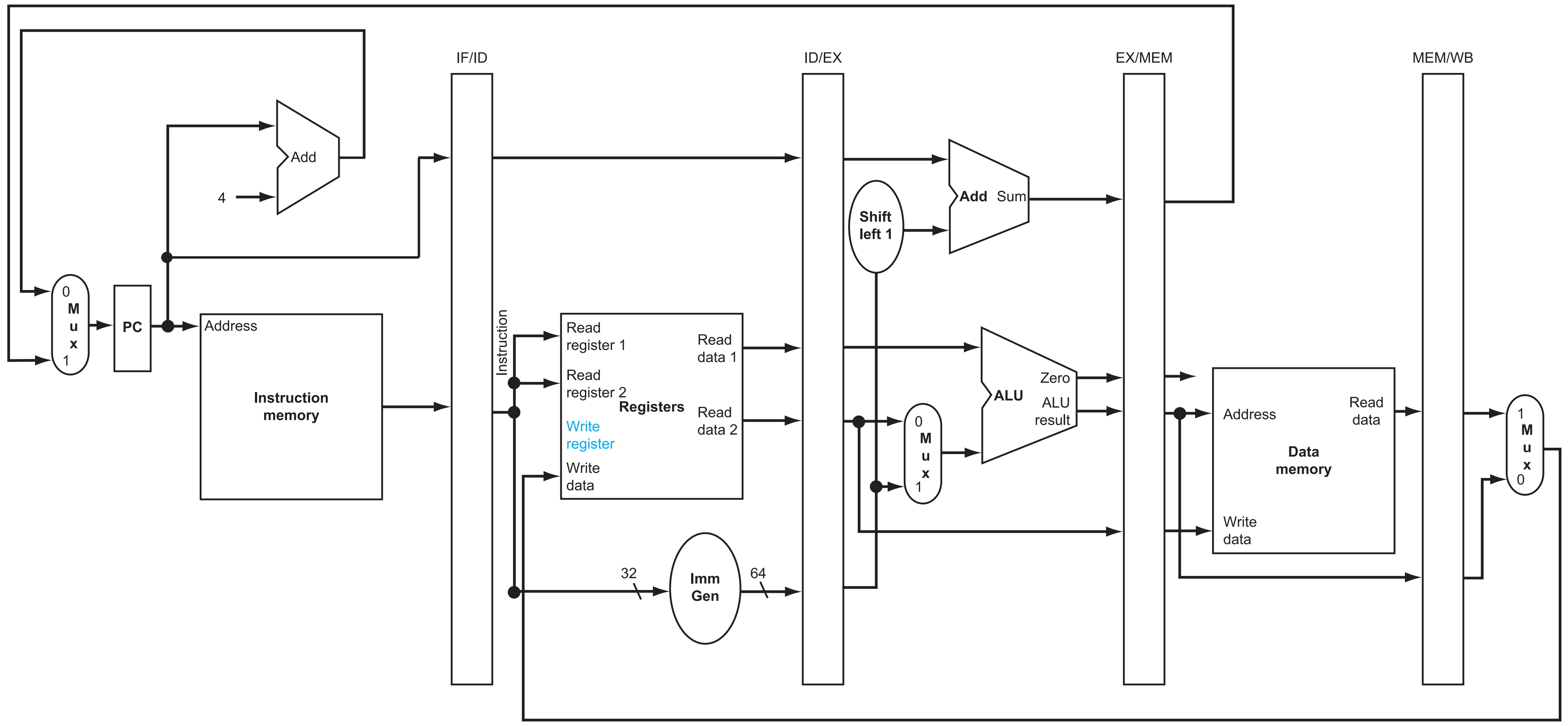


LOAD in WB Stage

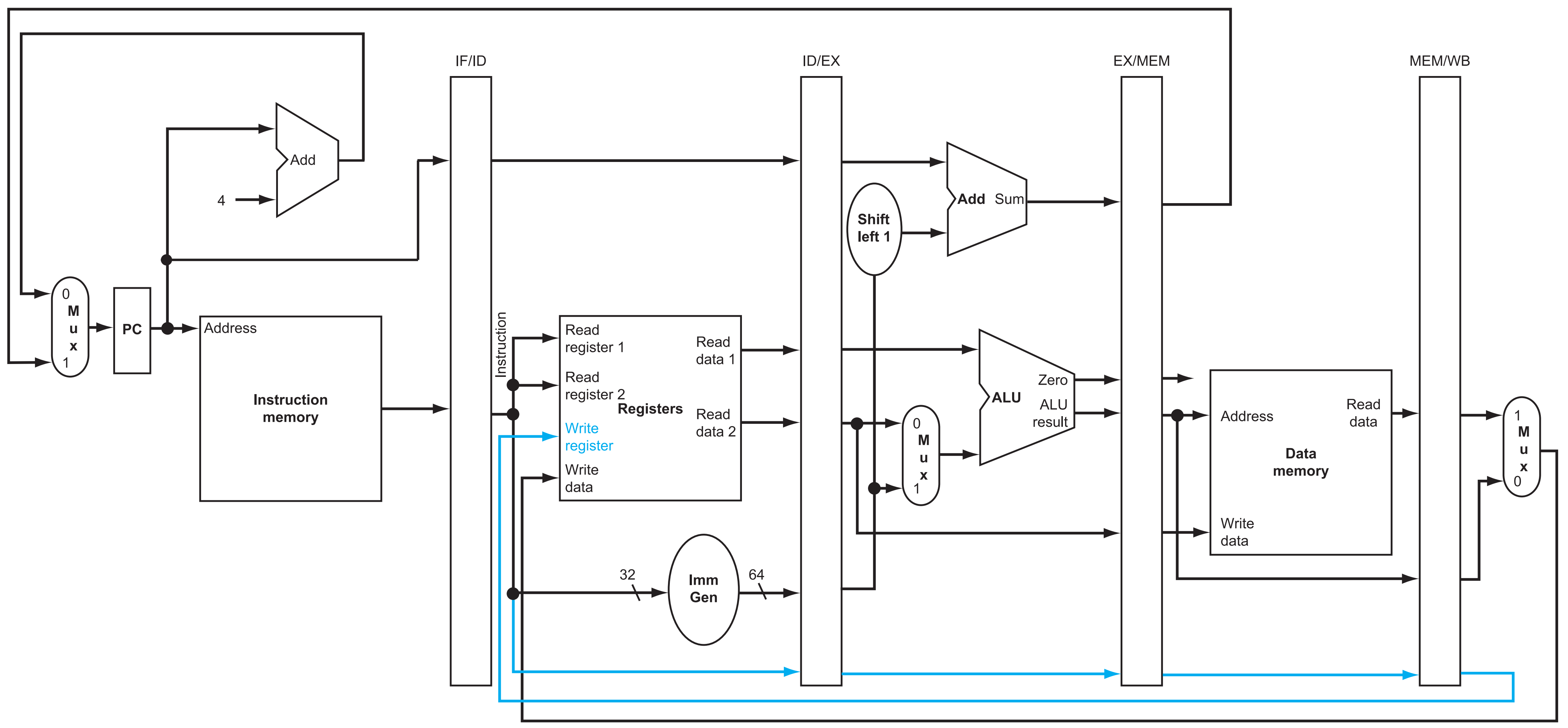




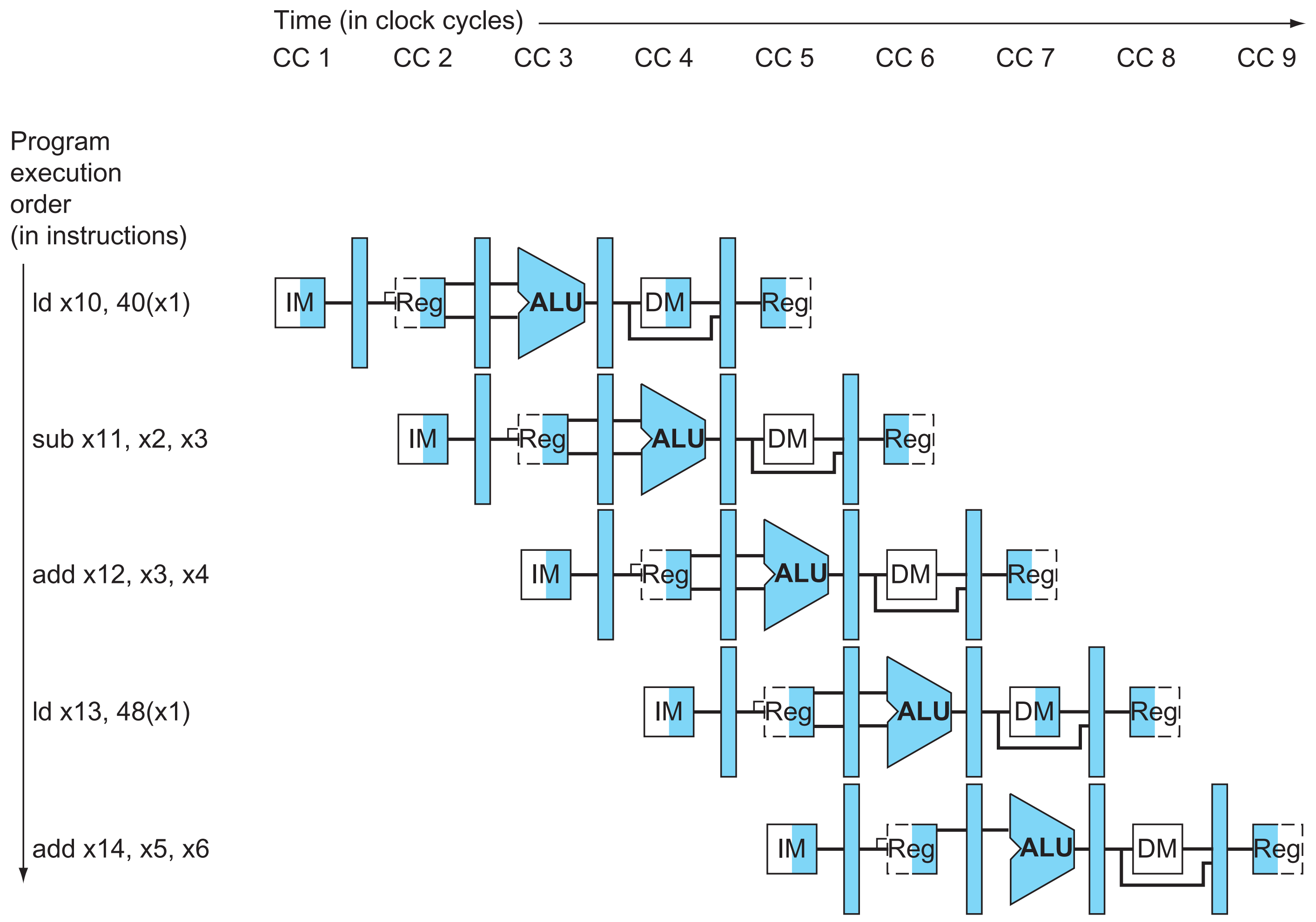
How to Wire RegWren



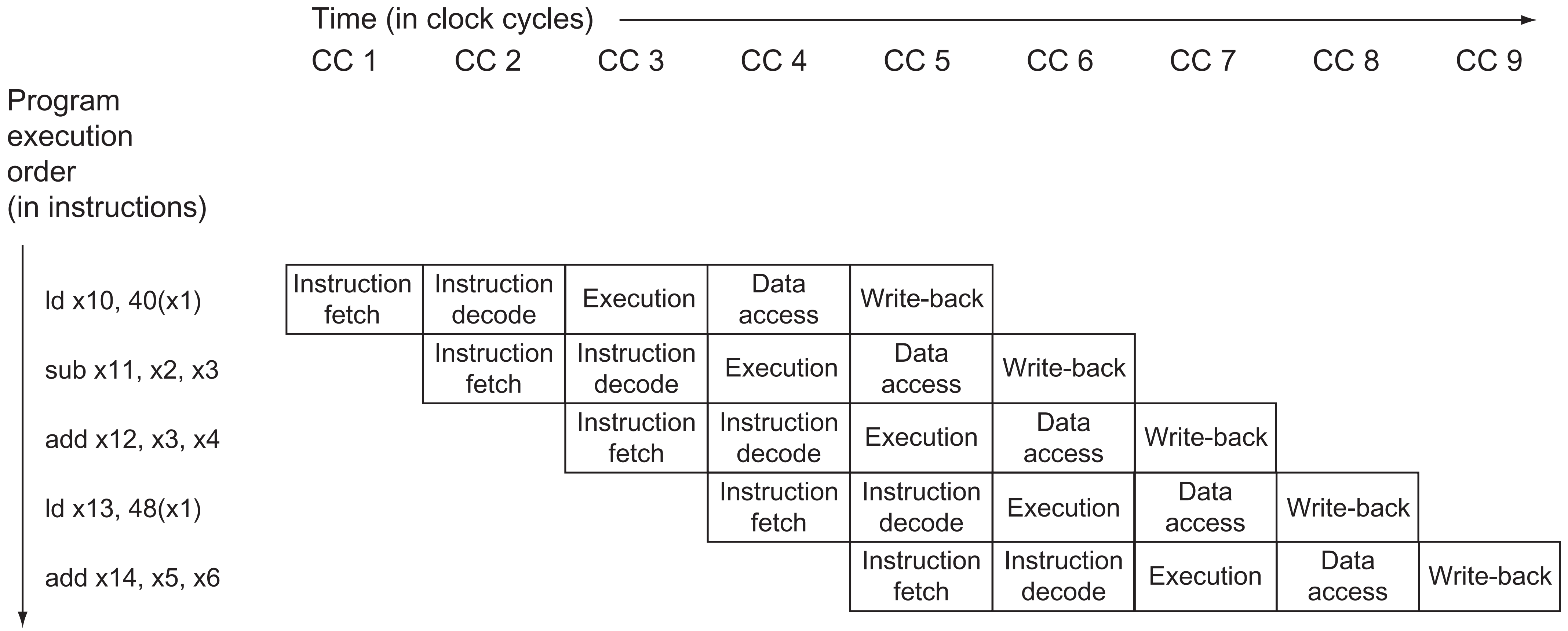
How to Wire RegWren



Pipelining


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
Program execution order (in instructions)
ld x10, 40(x1)
sub x11, x2, x3
add x12, x3, x4
ld x13, 48(x1)
add x14, x5, x6
IM
Reg
ALU
DM
Reg

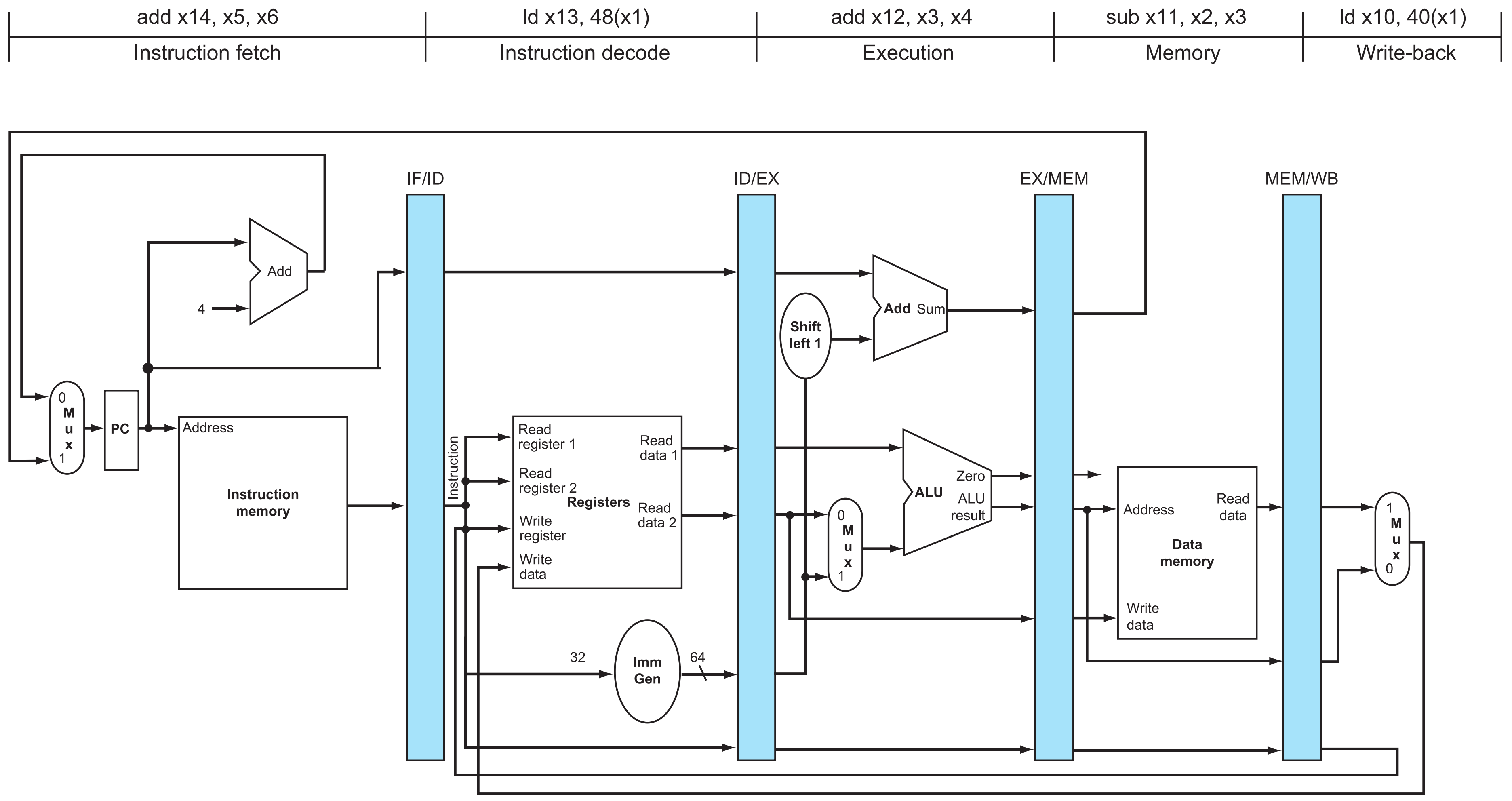
Pipelining



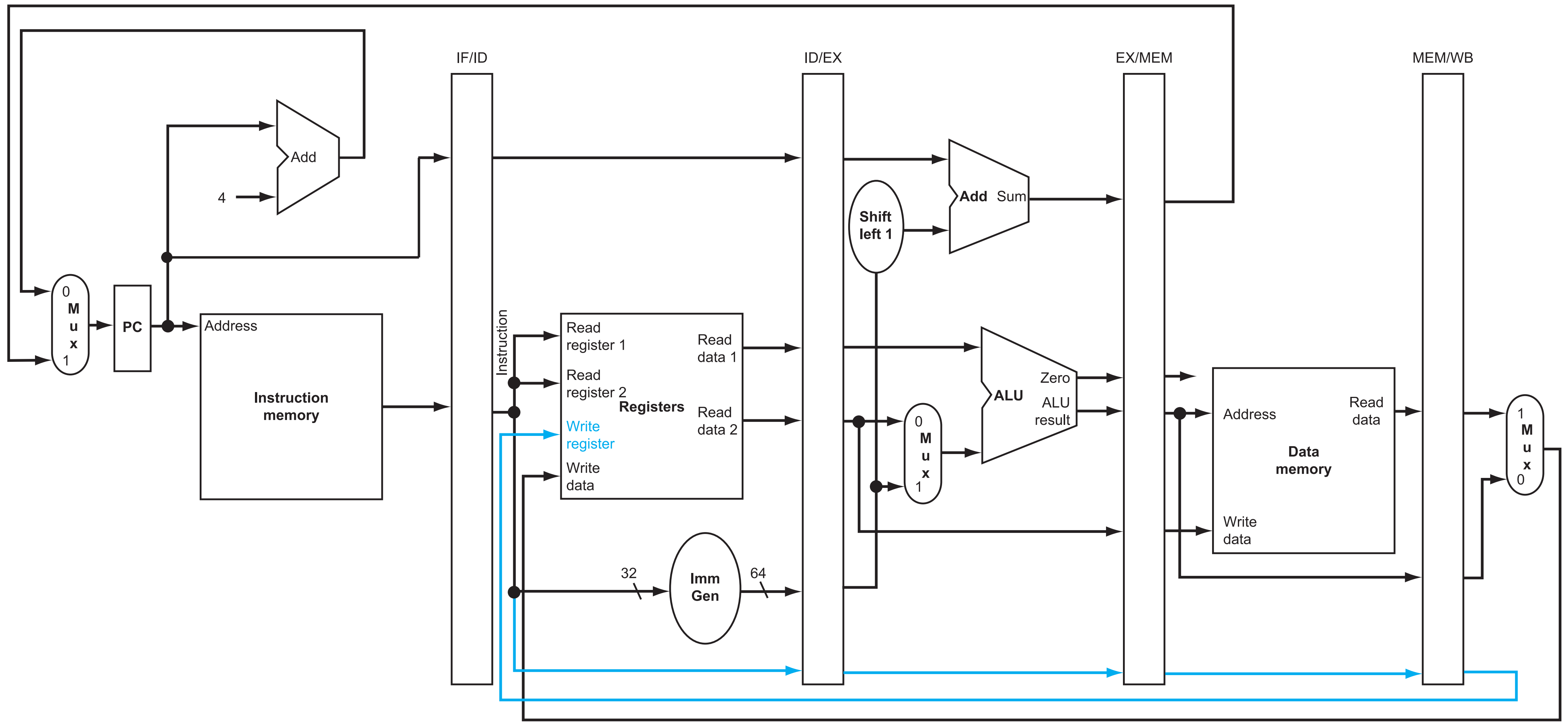
Time (in clock cycles)

Program execution order (in instructions)	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
ld x10, 40(x1)	Instruction fetch	Instruction decode	Execution	Data access	Write-back				
sub x11, x2, x3		Instruction fetch	Instruction decode	Execution	Data access	Write-back			
add x12, x3, x4			Instruction fetch	Instruction decode	Execution	Data access	Write-back		
ld x13, 48(x1)				Instruction fetch	Instruction decode	Execution	Data access	Write-back	
add x14, x5, x6					Instruction fetch	Instruction decode	Execution	Data access	Write-back

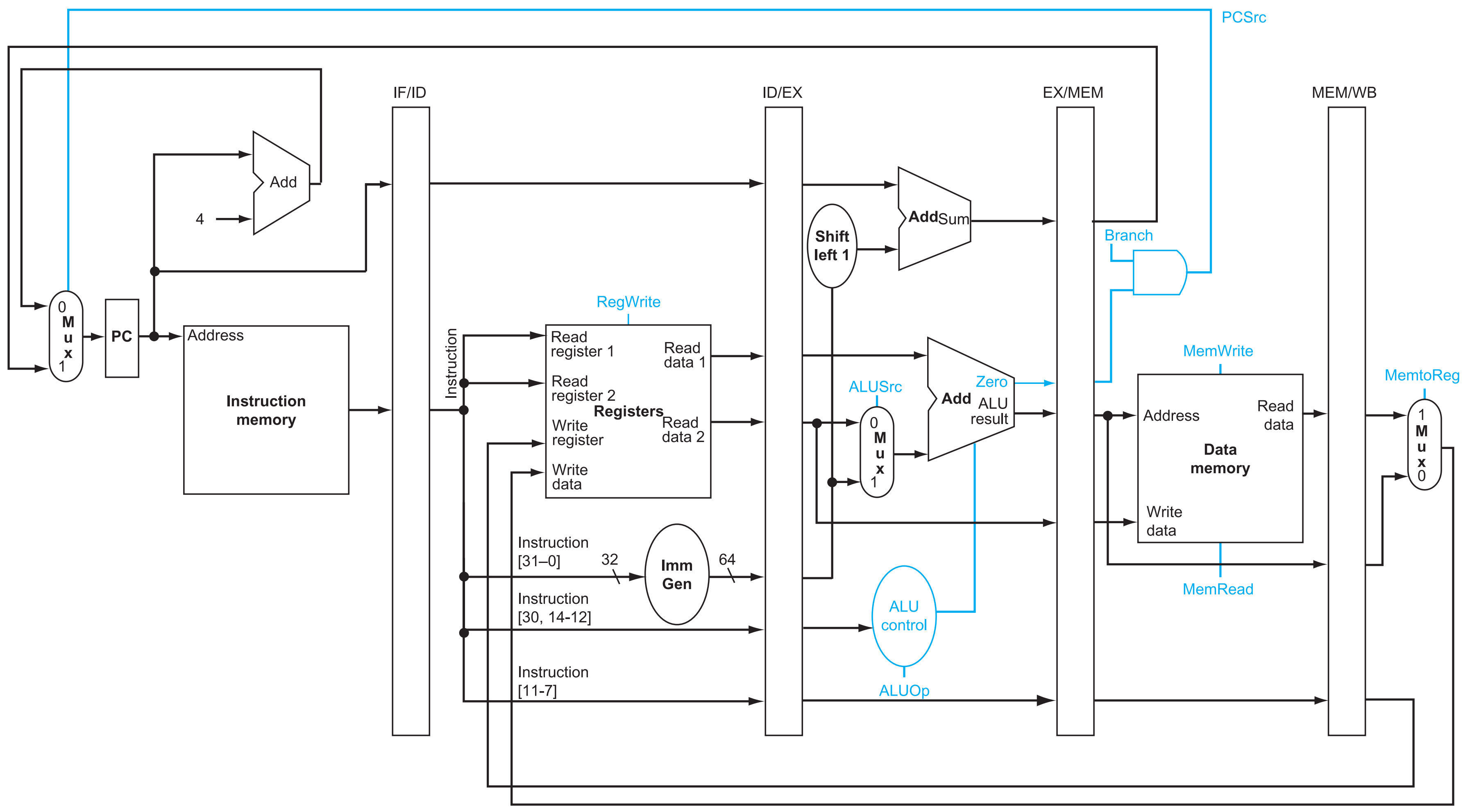
Pipelining


add x14, x5, x6
Instruction fetch
ld x13, 48(x1)
Instruction decode
add x12, x3, x4
Execution
sub x11, x2, x3
Memory
ld x10, 40(x1)
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Mux 0 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Imm Gen
32
64
Shift left 1
Add Sum
ALU
Zero
ALU result
Mux 0 1
Address
Write data
Data memory
Read data
Mux 1 0

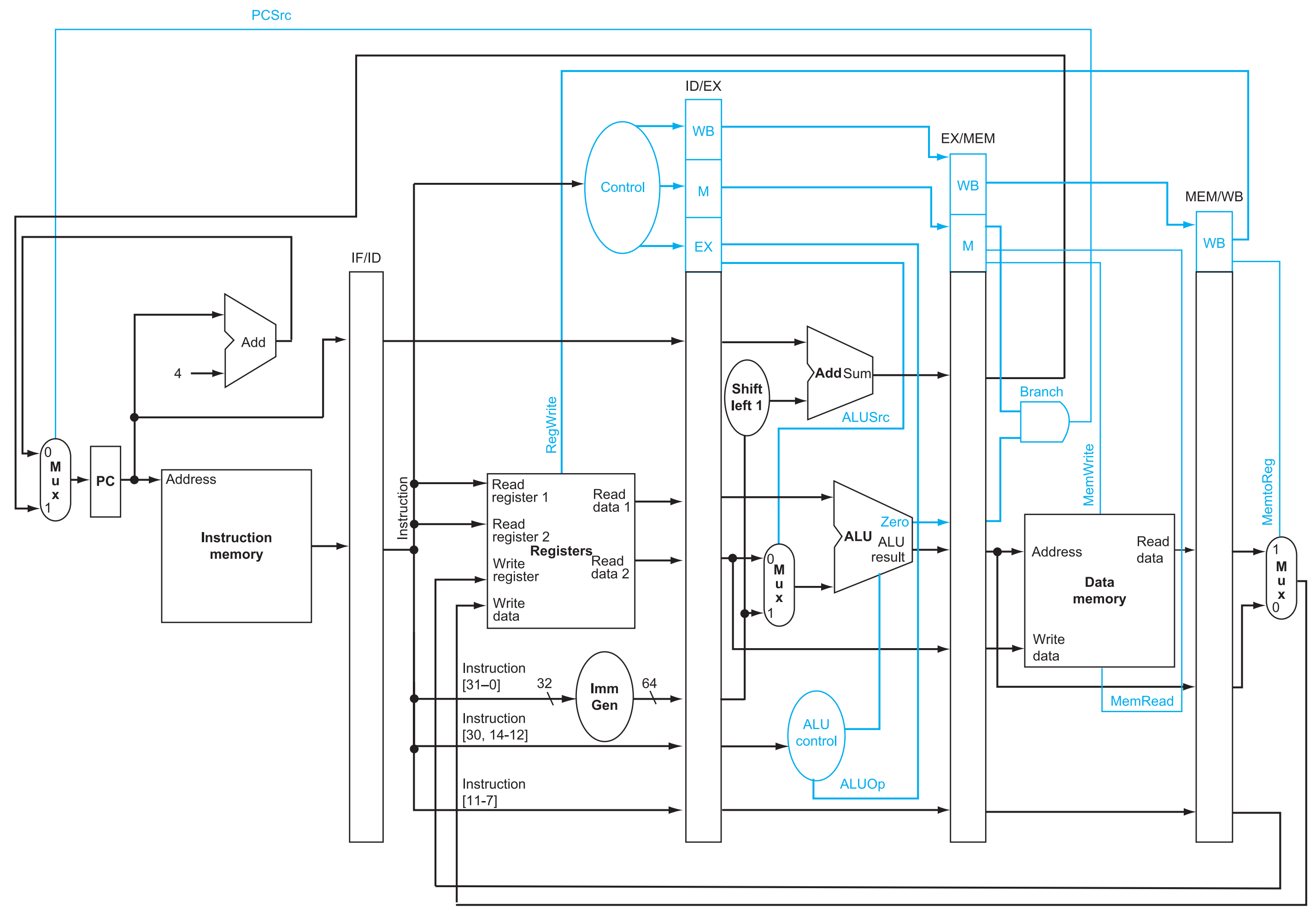
How to Wire Control Signals



How to Wire Control Signals



How to Wire Control Signals



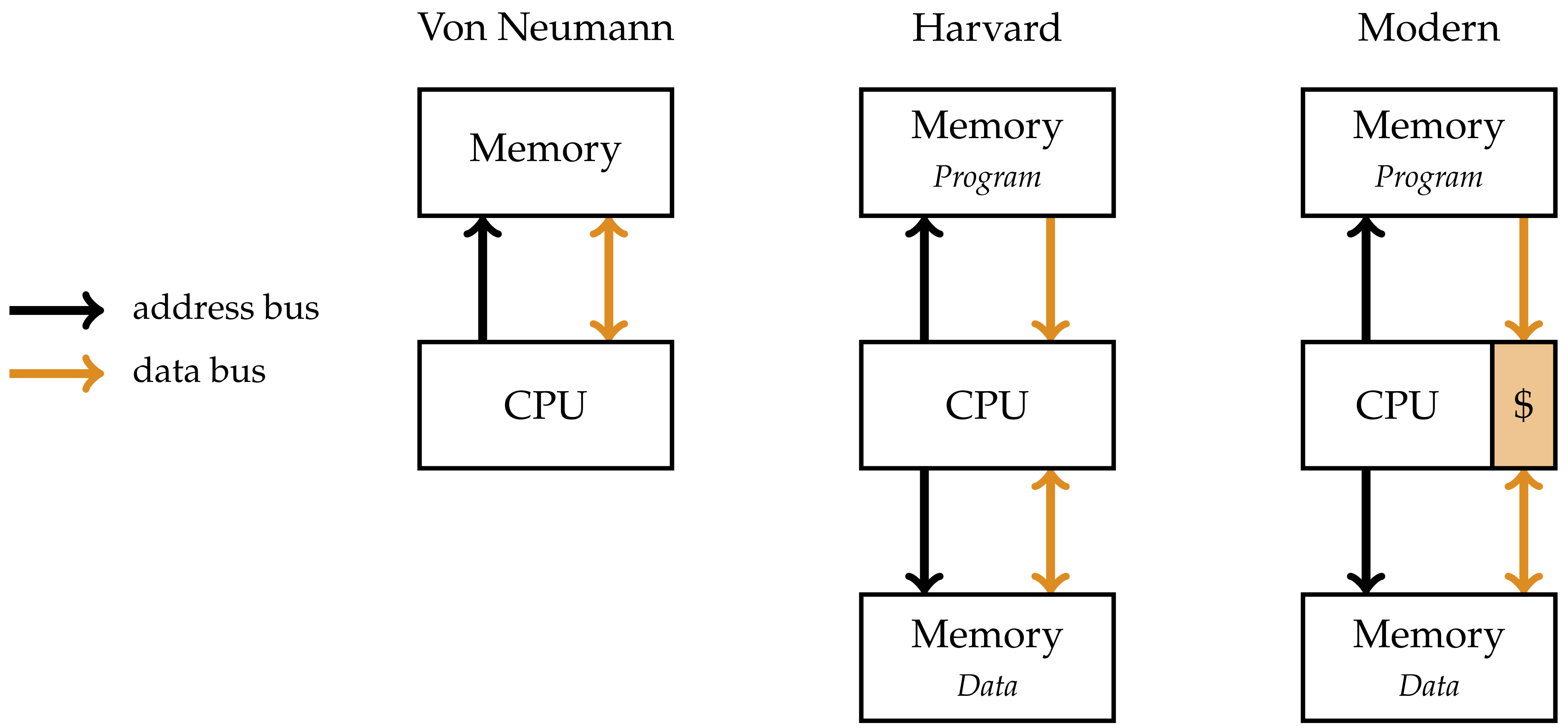
Hazards

Hazards

Hazards happen when an instruction cannot execute in the expected cycle.
There are two types of hazard

- Pipeline hazards
  - Structural hazard
  - Data hazard
- Control hazard (or Branch hazard)

Structural Hazard



Data Hazard

When a planned instruction cannot be executed at the proper clock cycle because the data needed for the execution are not yet available.

Read after Write

```
add r5, r3, r2
add r6, r5, r1
```

In this example, the second instruction READ *r*5, AFTER the first instruction WRITE it.

Data Hazard

Case 1

```
i0: add r5, r3, r2
i1: xor r6, r5, r1
i2: sub r9, r3, r5
i3: or  r2, r7, r5
i4: sll r4, r5, r5
```

IF	ID	EX	ME	WB
*i*2	*i*1	*i*0		
*i*2	*i*1	*nop*	*i*0	
*i*2	*i*1	*nop*	*nop*	*i*0
*i*2	*i*1	*nop*	*nop*	*nop*
*i*3	*i*2	*i*1	*nop*	*nop*
*i*4	*i*3	*i*2	*i*1	*nop*

"*nop*" to stall *i*1 until *i*0 writes successfully.

Data Hazard

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
*i*2	*i*1	*i*0		
*i*3	*i*2	*i*1	*i*0	
*i*3	*i*2	*nop*	*i*1	*i*0
*i*3	*i*2	*nop*	*nop*	*i*1
*i*3	*i*2	*nop*	*nop*	*nop*
*i*4	*i*3	*i*2	*nop*	*nop*

"*nop*" to stall *i*2 until *i*1 writes successfully.

Control Hazard

Case 3

```
i0: add r4, r3, r2
i1: beq r5, r6, _L0
i2: sub r9, r5, r1
...
i8: _L0
    sll r4, r5, r1
i9: xor r6, r8, r2
```

`IF`	`ID`	`EX`	`ME`	`WB`
$i2$	$i1$	$i0$		
$i3$	$i2$	$i1$	$i0$	
...	$i3$	$i2$	$i1$	$i0$

If instruction $i1$ is NOT TAKEN, there is nothing to worry about.

Data Hazard

Case 3

```
i0: add r4, r3, r2
i1: beq r5, r6, _L0
i2: sub r9, r5, r1
...
i8: _L0
    sll r4, r5, r1
i9: xor r6, r8, r2
```

IF	ID	EX	ME	WB
*i*2	*i*1	*i*0		
*i*3	*i*2	*i*1	*i*0	
*i*8	*nop*	*nop*	*i*1	*i*0
*i*9	*i*8	*nop*	*nop*	*i*1
	*i*9	*i*8	*nop*	*nop*
		*i*9	*i*8	*nop*

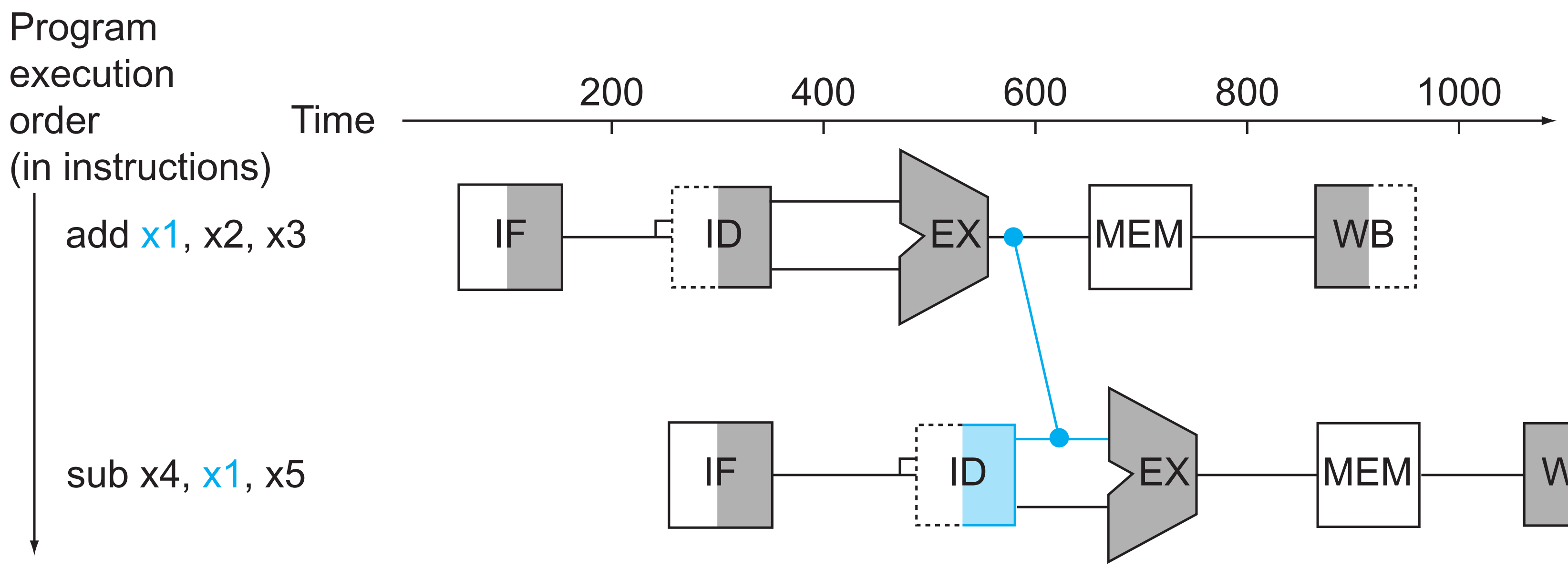
But in the other case, *i*2 and *i*3 must be flushed.

How to Stall and Flush

Enable and Reset of Flipflop.

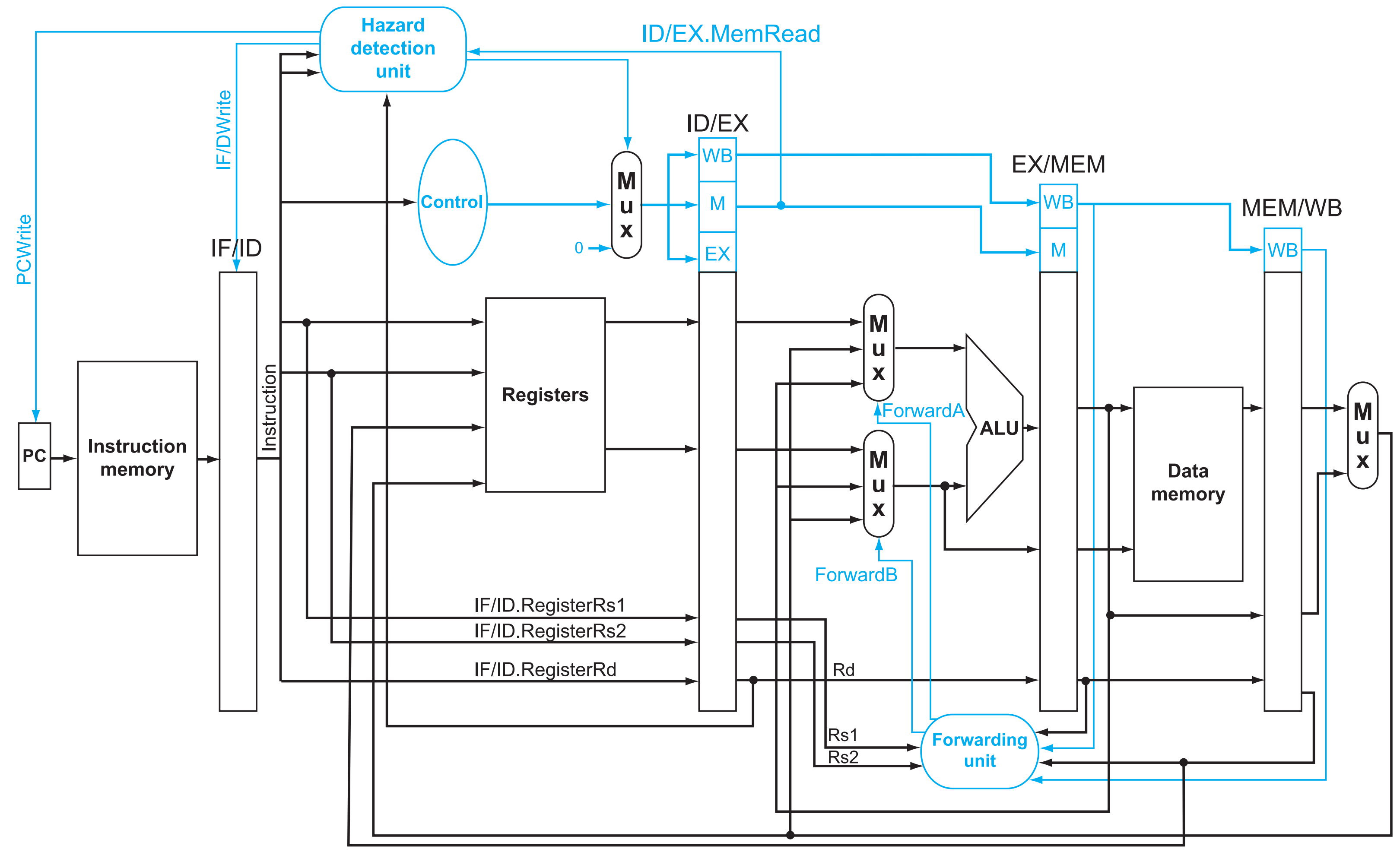
Forwarding

When the second instruction is in the pipeline, register $x1$ isn't updated, yet its value is in the pipeline already.



Forwarding — bypassing — is a method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

Forwarding


Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
M u x
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
PC
Instruction memory
Instruction
Registers
M u x
ForwardA
ALU
M u x
ForwardB
Data memory
M u x
IF/ID.RegisterRs1
IF/ID.RegisterRs2
IF/ID.RegisterRd
Rd
Rs1
Rs2
Forwarding unit

Forwarding

Case 1

```
i0: add r5, r3, r2
i1: xor r6, r5, r1
i2: sub r9, r3, r5
i3: or  r2, r7, r5
i4: sll r4, r5, r5
```

IF	ID	EX	ME	WB

Forwarding

```
// 00: no forward
// 01: forward from WB
// 10: forward from MEM
ForwardA <- 0;
if (MEM.RdWren and (MEM.RdAddr != 0) and (MEM.RdAddr == EX.Rs1Addr))
  ForwardA <- 10;
if (WB.RdWren and (WB.RdAddr != 0) and (WB.RdAddr == EX.Rs1Addr))
  ForwardA <- 01;
```

What if both conditions are true?

Forwarding

```
// 00: no forward
// 01: forward from WB
// 10: forward from MEM
ForwardA <- 0;
if (MEM.RdWren and (MEM.RdAddr != 0) and (MEM.RdAddr == EX.Rs1Addr))
  ForwardA <- 10;
else if (WB.RdWren and (WB.RdAddr != 0) and (WB.RdAddr == EX.Rs1Addr))
  ForwardA <- 01;
```

Forwarding

Case 1

```
i0: add r5, r3, r2
i1: xor r6, r5, r1
i2: sub r9, r3, r5
i3: or  r2, r7, r5
i4: sll r4, r5, r5
```

IF	ID	EX	ME	WB
$i2$	$i1$	$i0$		
$i3$	$i2$	$i1$	$i0$	
$i4$	$i3$	$i2$	$i1$	$i0$
	$i4$	$i3$	$i2$	$i1$
		$i4$	$i3$	$i2$
			$i4$	$i3$

Forwarding resolves hazards of case 1.

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB

Forwarding

```
MEM.enable <- 1;
if (MEM.RdWren and (MEM.RdAddr != 0) and MEM.isload and ((MEM.RdAddr == EX.Rs1Addr) or
↪  (MEM.RdAddr == EX.Rs2Addr))
  MEM.enable <- 0;
```

Is this enough?

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
*i*2	*i*1	*i*0		
*i*3	*i*2	*i*1	*i*0	
*i*4	*i*3	*i*2	*i*1	*i*0
	*i*4	*i*3	*i*1	*i*1

Where did instruction *i*2 go?

Forwarding

```
x.enable <- 1;
if (MEM.RdWren and (MEM.RdAddr != 0) and MEM.isload and ((MEM.RdAddr == EX.Rs1Addr) or
↪  (MEM.RdAddr == EX.Rs2Addr))
  MEM.enable <- 0;
  ID.enable  <- 0;
  EX.enable  <- 0;
```

Is this enough?

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
$i2$	$i1$	$i0$		
$i3$	$i2$	$i1$	$i0$	
$i4$	$i3$	$i2$	$i1$	$i0$
$i4$	$i3$	$i2$	$i1$	$i1$

Why did instruction $i1$ stay there?

Forwarding

```
x.enable <- 1;
if (MEM.RdWren and (MEM.RdAddr != 0) and MEM.isload and ((MEM.RdAddr == EX.Rs1Addr) or
↪  (MEM.RdAddr == EX.Rs2Addr))
  MEM.enable <- 0;
  MEM.reset  <- 0;
  ID.enable  <- 0;
  EX.enable  <- 0;
```

Forwarding

Case 2

```
i0: add r4, r3, r2
i1: lw  r5, 0x40(r1)
i2: sub r9, r5, r1
i3: or  r2, r7, r5
i4: sll r4, r5, r1
```

IF	ID	EX	ME	WB
*i*2	*i*1	*i*0		
*i*3	*i*2	*i*1	*i*0	
*i*4	*i*3	*i*2	*i*1	*i*0
*i*4	*i*3	*i*2	*nop*	*i*1
	*i*4	*i*3	*i*2	*nop*
		*i*4	*i*3	*i*2

Forwarding can't completely resolve hazards of case 2. Why?

Branch Prediction

Branch Prediction

Branch Prediction is a technique that predicts the next instruction when the processor encounters a control transfer instruction, which is branch instruction or jump instruction.

$\rightarrow$ ***The lower the miss rate, the lower the consumed power***

Let’s do a little bit math

There are three types of branches, and each has its own “taken” probability.

- Unconditional branch (jumps): $P_{jump} = 100\%$
- Forward conditional branch (`if/else`): $P_{fw} = 50\%$
- Backward conditional branch (`do/while`): $P_{bw} = 90\%$

Because $P_{jump} = 100\%$, let’s consider the other two only.

Given: The number of forward branches and that of backward branches are equal. The probability of a banch to be taken:

$$P = \frac{P_{fw} + P_{bw}}{2} = \frac{50\% + 90\%}{2} = 70\% \tag{1}$$

IPC

IPC — instructions per cycles — indicates the performace of a processor. If a pipelined processor has too many *"nop"*, IPC will decrease.

Given: The number of branch instructions is 20% (P_{br}), and the processor takes TWO-CYCLE penalty (delay) for a branch instruction (Δ).

$$IPC = \frac{N_{instr}}{N_{cycle}} = \frac{1}{P_{br}\Delta + 1} = \frac{1}{20\% \times 2 + 1} = 71\%$$

Static Prediction

However, because we do NOT need to delay if a branch instruction is not taken, (1) shows that $P_{mis} = 70\%$, so:

$$IPC = \frac{N_{instr}}{N_{cycle}} = \frac{1}{P_{br}P_{mis}\Delta + 1} = \frac{1}{20\% \times 70\% \times 2 + 1} = 78\%$$

Static Prediction

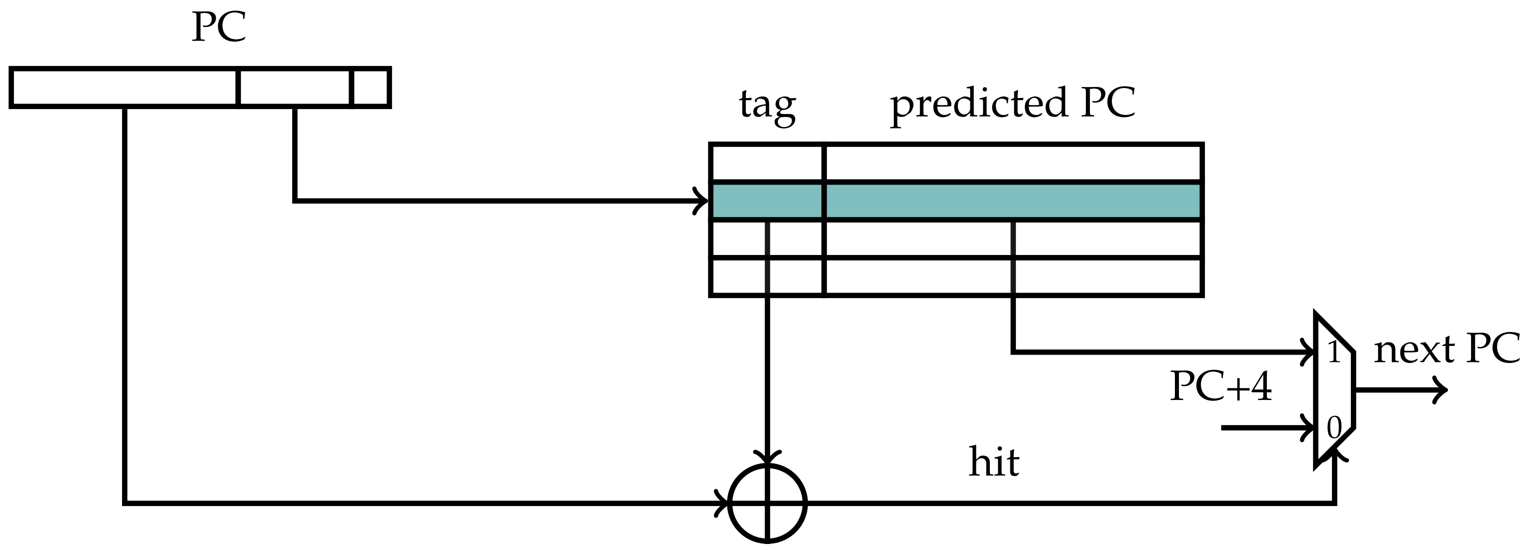
But if we know the next PC of a branch instruction and allow the processor to always go to that PC, $P_{mis} = 30\%$, so:

$$IPC = \frac{N_{instr}}{N_{cycle}} = \frac{1}{P_{br}P_{mis}\Delta + 1} = \frac{1}{20\% \times 30\% \times 2 + 1} = 89\%$$

$\rightarrow$ The performance is clearly enhanced.

Branch Target Buffer

BTB, Branch Target Buffer, will save predicted PCs of branch instructions, using PC as an index or address. Because the last two bits are 00, and if all the other 30 bits are used, the buffer size will be $2^{30} = 4GiB$. It is big, and not all instructions are branches; it is wasteful. Thus, to ensure the correctness of a smaller buffer, only low bits are used as the index, and the rest are called tags.



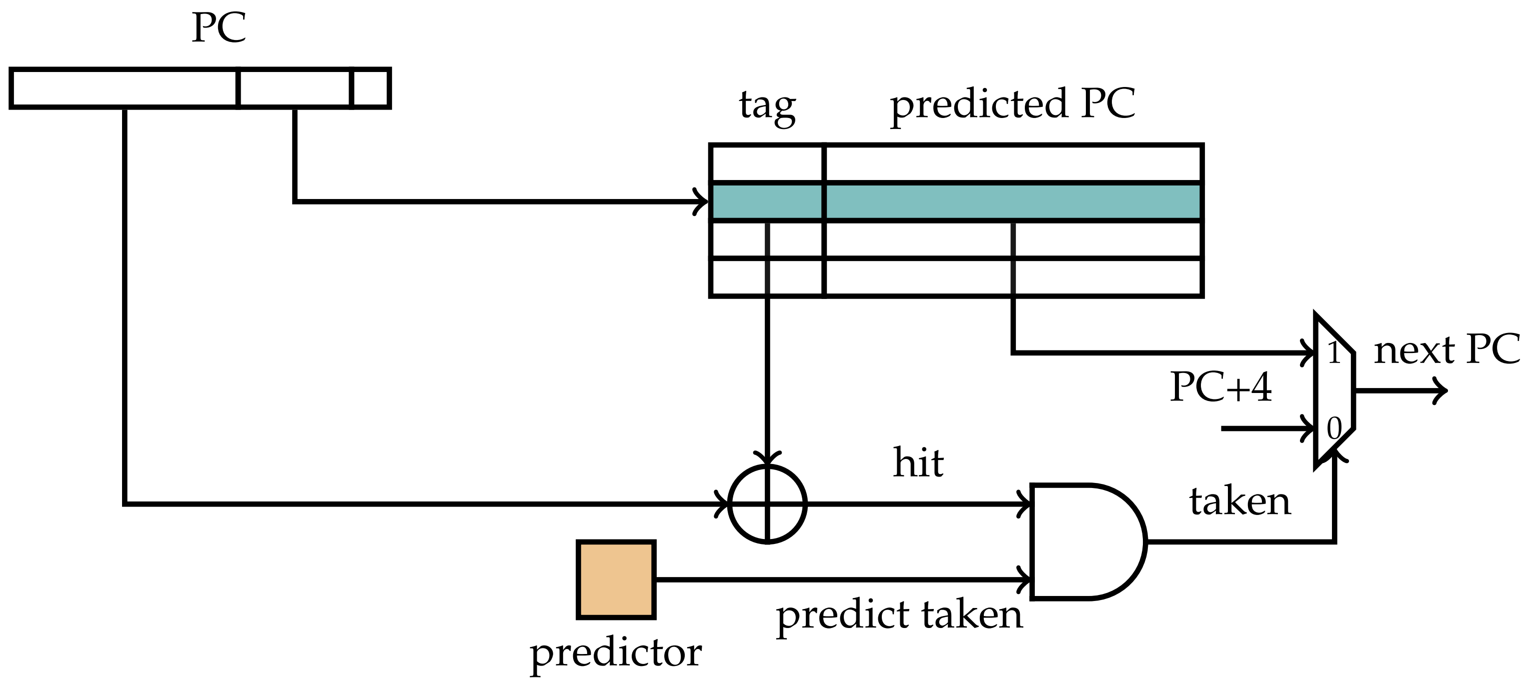
Branch Target Buffer

```
x1CA800 addi x11, x0, 30
x1CA804 addi x18, x0, 1
x1CA808 add x13, x0, x0
_COMPARE:
x1CA80C and x12, x11, x18
x1CA810 beq x12, x0, _ADD_EVEN
x1CA814 j _DECREASE
_ADD_EVEN:
x1CA818 add x13, x11, x13
_DECREASE:
x1CA81C sub x11, x11, x18
x1CA820 bne x11, x0, _COMPARE
_EXIT:
x1CA824 ...
...
_ANOTHER:
x2FB810 bgt x13, x12, _EXIT
```

	tag	predicted pc
...	...	...
0x203	0x00000	0x00000000
0x204	0x001CA	0x001CA818
0x205	0x001CA	0x001CA81C
0x206	0x00000	0x00000000
0x207	0x00000	0x00000000
0x208	0x001CA	0x001CA80C
0x209	0x00000	0x00000000
...	...	...

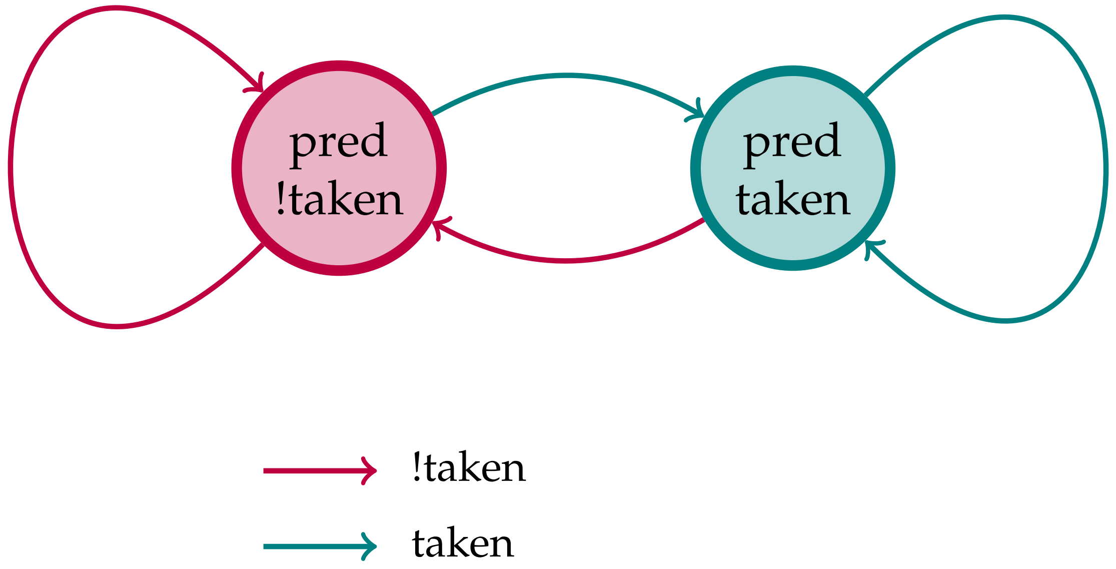
Dynamic Prediction

To increase performance, the processor needs to decide when to “taken” or not.



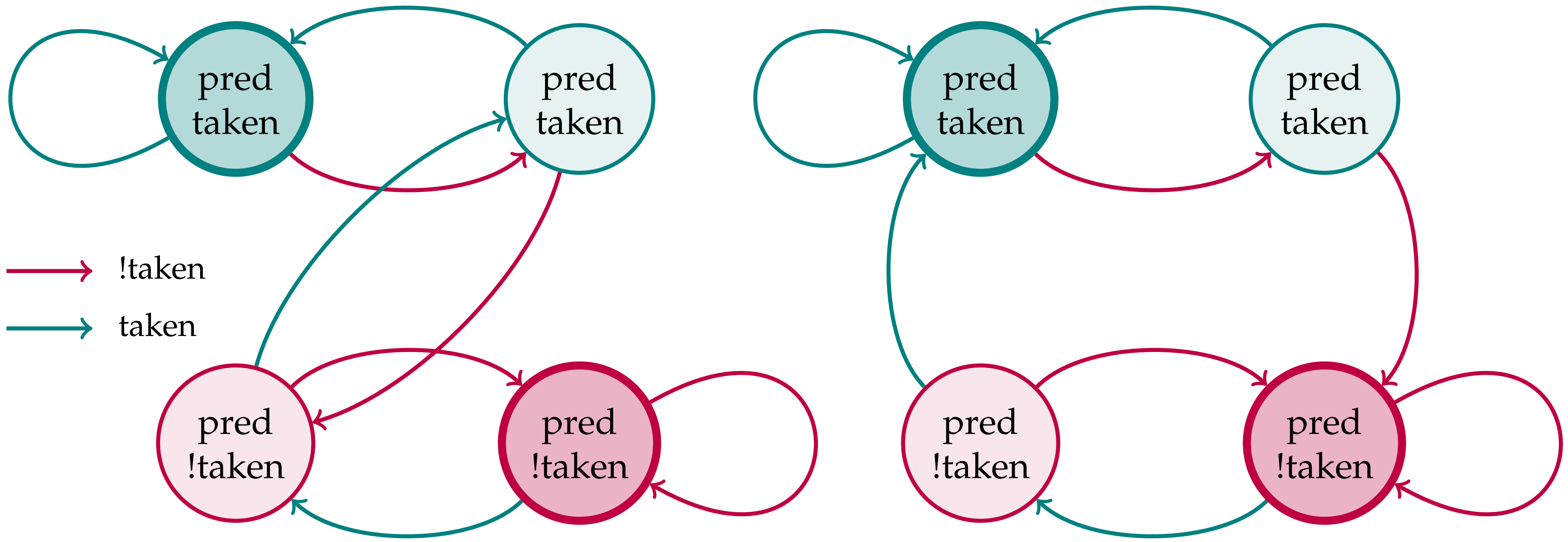
One Bit Scheme Predictor

Let's set one bit to predict taken/not taken of branch instructions, this bit simply changes its prediction when it predicts wrong. The accuracy might be 85%.



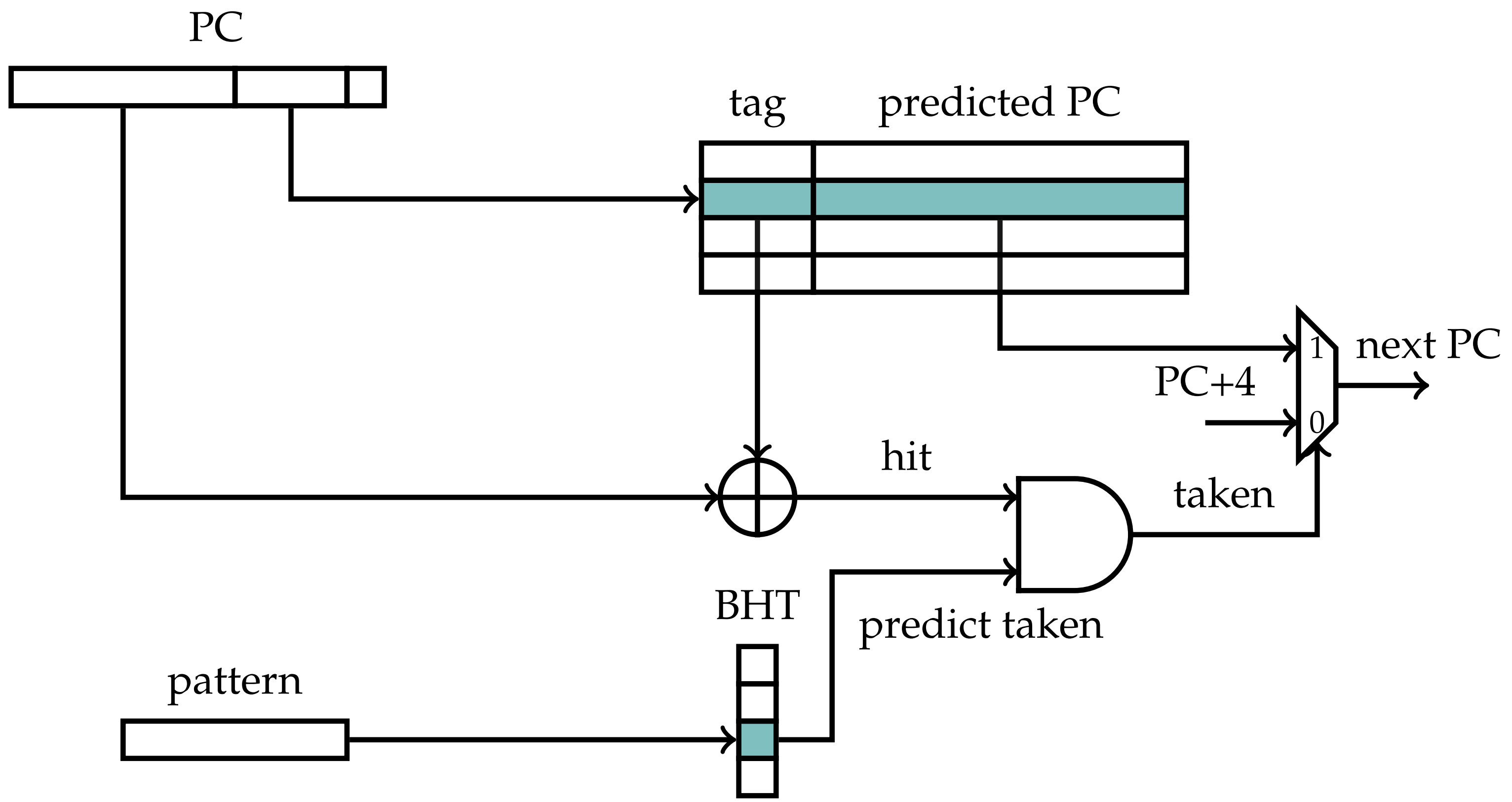
Two Bit Scheme Predictor

However, if the outcome is changing every time: TK–NT–TK–NT–TK–. . . , one-bit scheme becomes useless, and thus two-bit schemes become superior. The accuracy might be 90%.



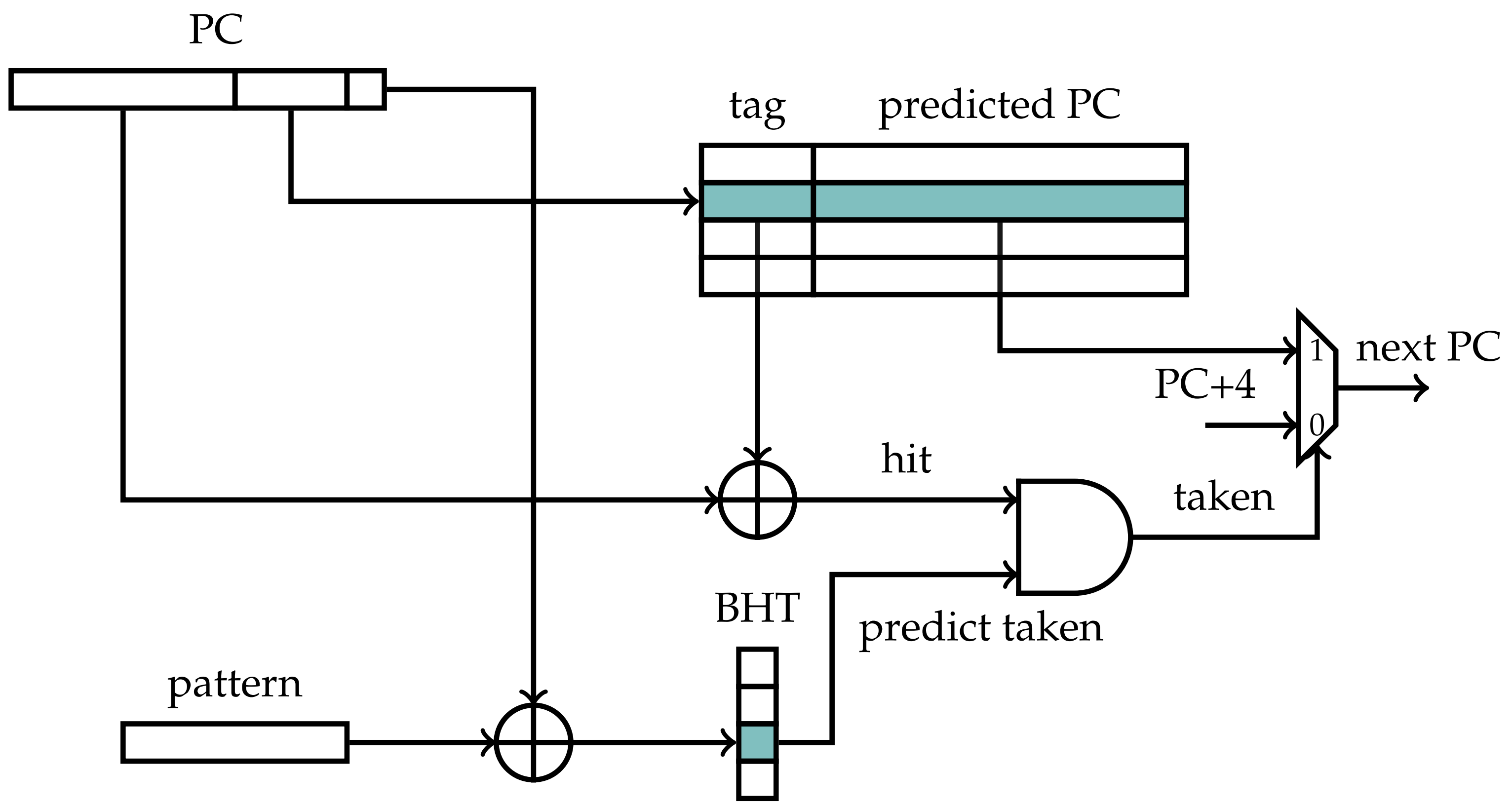
G-share

Because the previous scheme only uses the last prediction result to predict future branches, it is not sufficient. G-share or two-bit adaptive global utilizes a pattern — a register storing branch history — to predict. Instead of one predictor FSM, a table called BHT (Branch History Table) contains a collection of N predictor FSMs, with pattern as index.



G-share

However, pattern should be used with PC to make pattern PC-relative, and thus BHT is utilized efficiently. In this way, the accuracy would be 93%.



Questions?