Speeding Up DLX

DLX Execution Stages — Version 1

Clock Cycle 1

I_1 enters Instruction Fetch (IF)

Clock Cycle2

I_1 moves to Instruction Decode (ID)

Instruction Fetch (IF) holds state fixed

Clock Cycle3

I_1 moves to Execute (EX)

Instruction Fetch (IF) holds state fixed

Instruction Decode (ID) holds state fixed

Clock Cycle4

I_1 moves to Memory Access (MEM)

Instruction Fetch (IF) holds state fixed

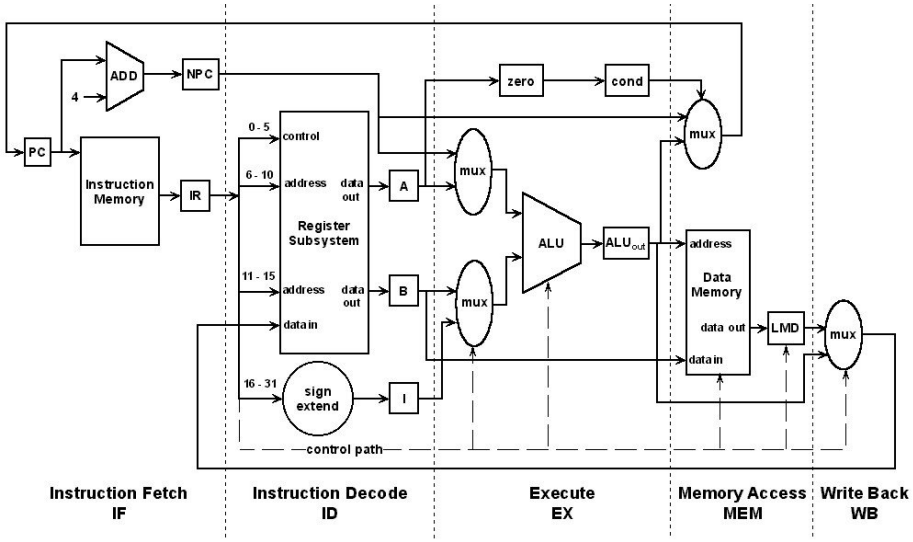
Instruction Decode (ID) holds state fixed

Execute (EX) holds state fixed

Clock Cycle5

I_1 performs Write Back (WB) using instruction (IR) stored in IF stage

PC updated and stages IF, ID, EX, MEM are reset



Room for Improvement

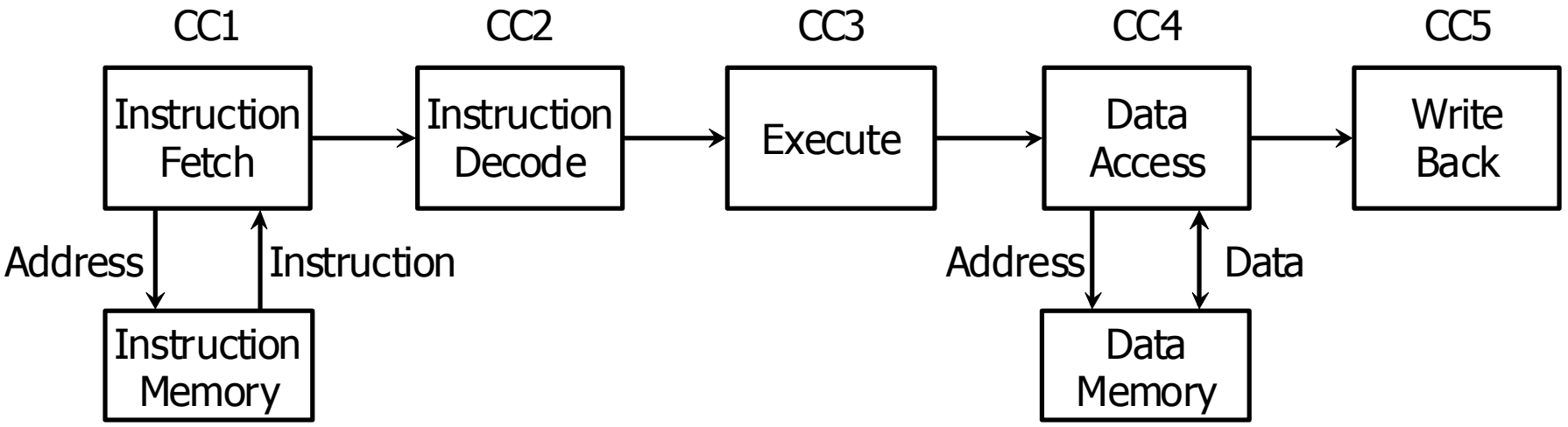
DLX based on assembly line

No central system bus

Instructions move from execution stage to execution stage

Assembly line permits pipelining

In each stage, new work begins when old work passes to next stage



DLX — Version 2

CC 1	I_1 enters Instruction Fetch (IF)
CC 2	I_1 and its execution state move to Instruction Decode (ID) I_2 enters Instruction Fetch (IF)
CC 3	I_1 and its execution state move to Execute (EX) I_2 and its execution state move to Instruction Decode (ID) I_3 enters Instruction Fetch (IF)
CC 4	I_1 and its execution state move to Memory Access (MEM) I_2 and its execution state move to Execute (EX) I_3 and its execution state move to Instruction Decode (ID) I_4 enters Instruction Fetch (IF)
CC 5	I_1 moves to Write Back (WB) I_2 and its execution state move to Memory Access (MEM) I_3 and its execution state move to Execute (EX) I_4 and its execution state move to Instruction Decode (ID) I_5 enters Instruction Fetch (IF)

Ideal Instruction Pipelining — Processor View

stage →

clock cycle ↓

	IF	ID	EX	MEM	WB
1	I_1				
2	I_2	I_1			
3	I_3	I_2	I_1		
4	I_4	I_3	I_2	I_1	
5	I_5	I_4	I_3	I_2	I_1
6	I_6	I_5	I_4	I_3	I_2
7	I_7	I_6	I_5	I_4	I_3
8	I_8	I_7	I_6	I_5	I_4

In any clock cycle (after CC 4)

5 instructions are being processed at one time

Each instruction in a different stage of execution

Ideal Instruction Pipelining — Instruction View

clock cycle →

	1	2	3	4	5	6	7	8
I_1	IF	ID	EX	MEM	WB			
I_2		IF	ID	EX	MEM	WB		
I_3			IF	ID	EX	MEM	WB	
I_4				IF	ID	EX	MEM	WB
I_5					IF	ID	EX	MEM
I_6						IF	ID	EX
I_7							IF	ID
I_8								IF

Average CPI for DLX Pipeline

From diagram

I1 finishes after N=5 clock cycles

I2 finishes after N=6 clock cycles

I3 finishes after N=7 clock cycles

Generally

IC instructions are finished after N = IC + 4 clock cycles

$$CPI = \frac{\text{clock cycles}}{\text{finished instructions}} = \frac{IC+4}{IC} = 1 + \frac{4}{IC} \xrightarrow[IC>>4]{} 1$$

On average

One instruction completes on every clock cycle

CPI is 1 clock cycle per instruction for DLX pipeline

Limitation

Dependencies between instructions cause waiting conditions

Pipelining — Functional Requirements

Each stage receives a new instruction on every clock cycle

- Cannot hold partial results for all instructions
- Must pass along all intermediate results for every instruction

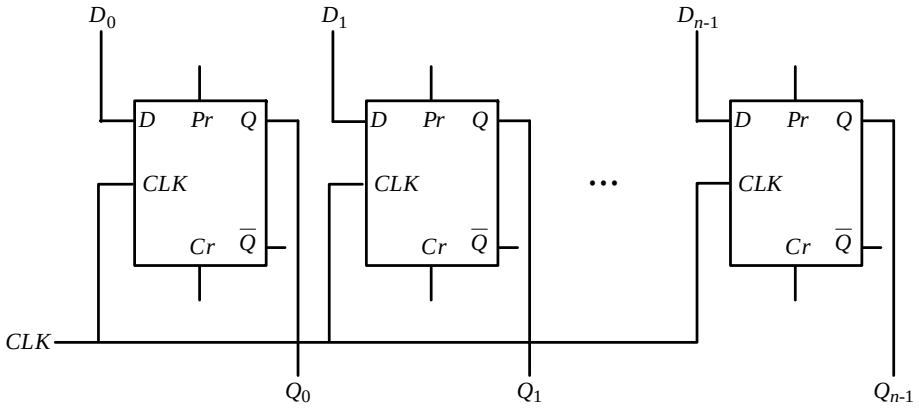
Example

- **`IF`** stage
  - Loads instruction to **`IR`**
  - Finds **`NPC`** for next instruction
  - Passes **`IR`** and **`NPC`** (intermediate results) to **`ID`** stage
- **`ID`** stage
  - Stores received **`IR`** and **`NPC`** for incoming instruction
  - Decodes **`IR`** to **`A`**, **`B`**, and **`I`**
  - Passes **`IR`**, **`NPC`**, **`A`**, **`B`**, and **`I`** to **`EX`** stage

Stage buffers

- Collection of D-flip/flops (edge-triggered latches)
- Store intermediate results of each stage at end of clock cycle

Review — Synchronous Transfer



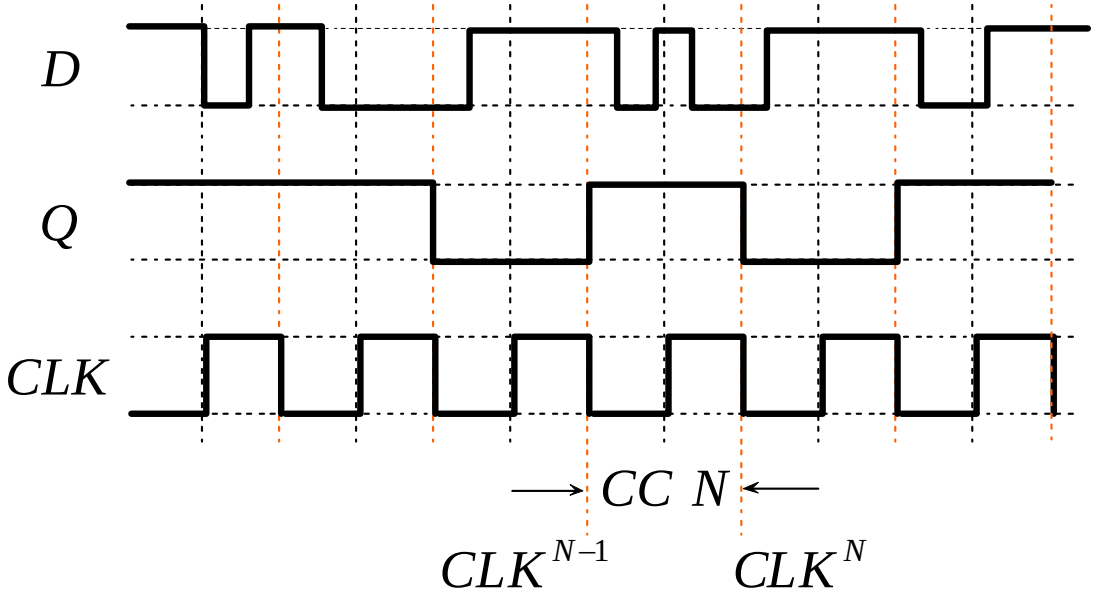
D-flip/flop (edge-triggered latch)

Input D

- Output of some digital system

Output Q

- Changes only on falling CLK edge
- Trigger — 1-to-0 CLK transition



Clock Cycle N

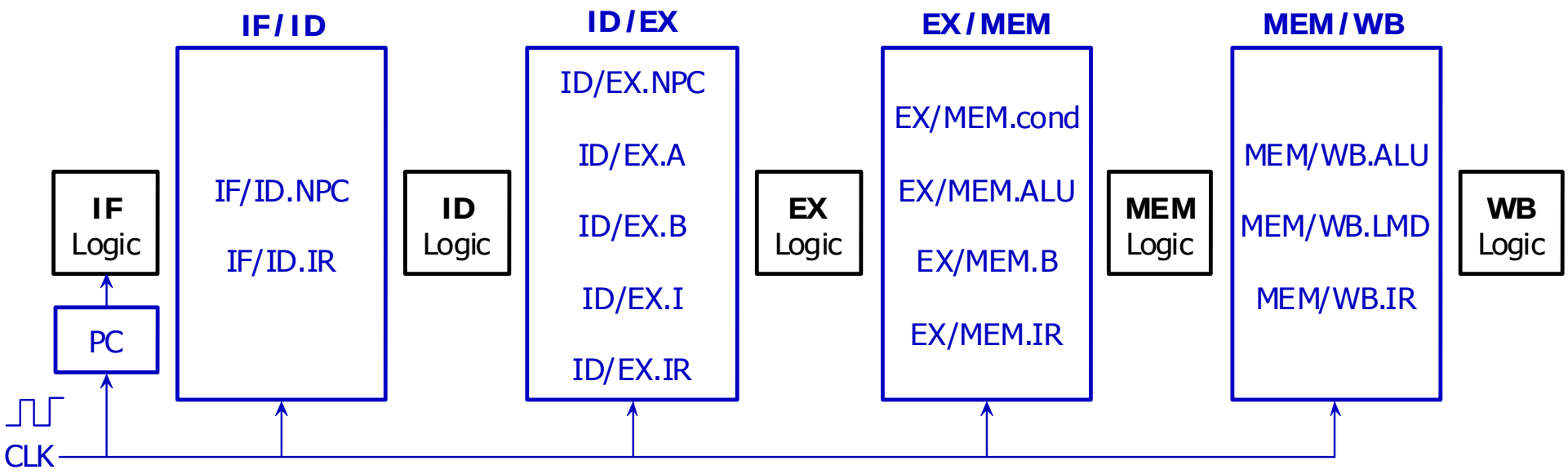
$CC\ N$ begins on CLK^{N-1}

- Input D can change
- No effect on latch

$CC\ N$ ends on CLK^{N}

- Latch samples input D
- Stores instantaneous input value
- Forwards stored value to output Q

Stage Buffers



5 execution stages built from

Combinational logic — output = function (present input)

Asynchronous memory — output = function (present input, past input)

4 stage buffers (edge-triggered latches) and PC built from

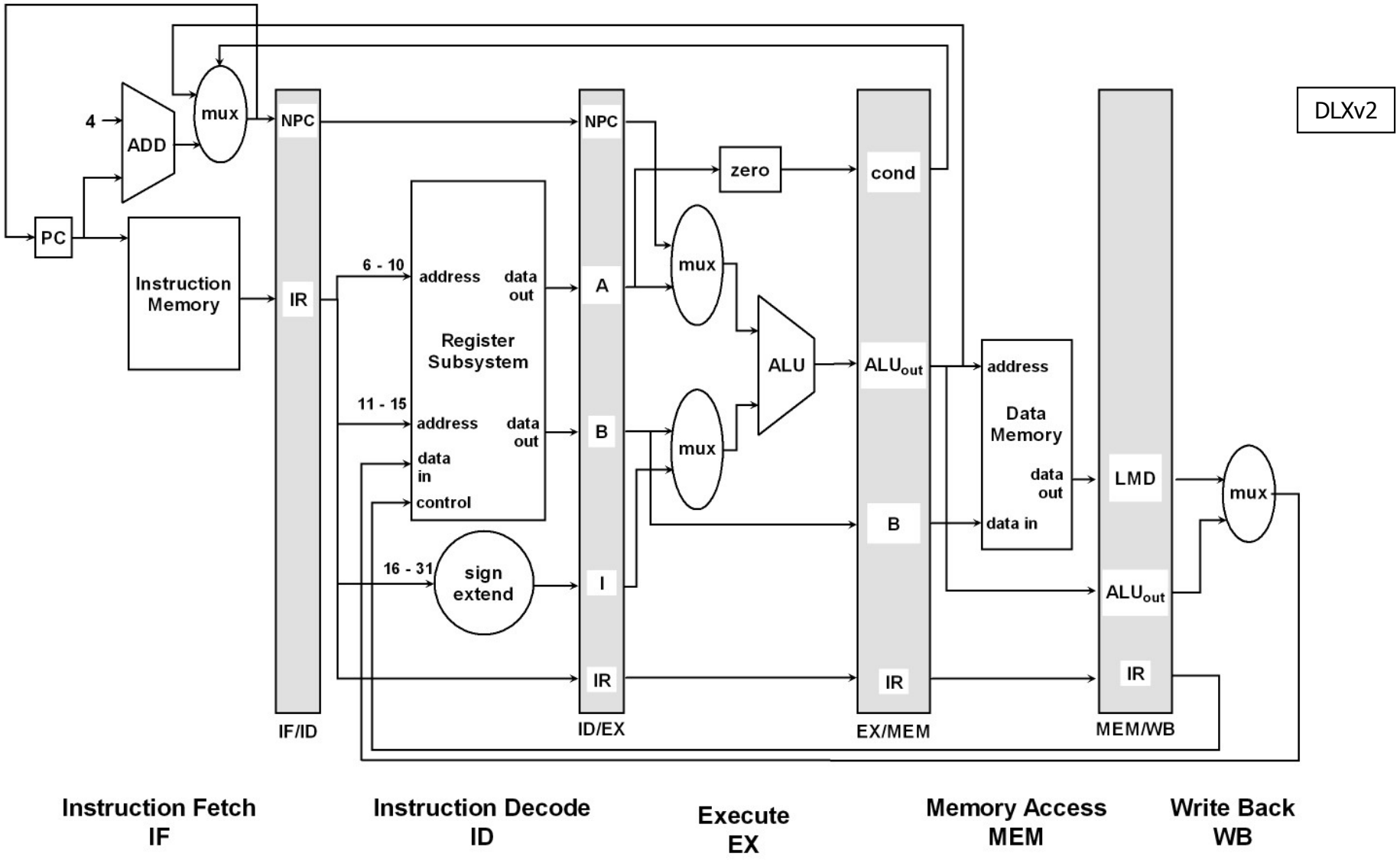
Synchronous sequential logic

output = function (present input, past input, external clock)

Store and forward input on falling edge of CLK

Described as data structure using C notation

DLX Drawing — version 2



Formal Specification of Version 2

Instruction Fetch (IF)

$$PC \leftarrow NPC$$

New **PC** for new instruction fetch in every clock cycle

$$IF/ID.IR \leftarrow Mem[PC]$$

$$IF/ID.NPC \leftarrow \begin{cases} PC + 4 \text{ (no branch)} \\ ALU_{OUT} \text{ (branch taken - special case)} \end{cases}$$

Instruction Decode (ID)

Type	0-5	6-10	11-15	16-31	
R	op	rs1	rs2	rd	function
I	op	rs	rd	immediate	

$$ID/EX.NPC \leftarrow IF/ID.NPC$$

$$ID/EX.A \leftarrow Reg[IF/ID.IR_{6\text{-}10}]$$

$$ID/EX.B \leftarrow Reg[IF/ID.IR_{11\text{-}15}]$$

$$ID/EX.I \leftarrow (IR_{16})^{16} \,\#\#\, IF/ID.IR_{16\text{-}31}$$

$$ID/EX.IR \leftarrow IF/ID.IR$$

Stage Buffers (←)	"See" inputs during clock cycle Sample and store inputs on falling CLK at end of clock cycle

Formal Specification of Version 2

Execute (EX)

$$EX/MEM.cond \leftarrow (ID/EX.A == 0)$$

$$EX/MEM.ALU_{OUT} \leftarrow \begin{cases} ID/EX.A \text{ function } ID/EX.B & (R\text{-}ALU) \\ ID/EX.A \text{ op } ID/EX.I & (I\text{-}ALU, \text{Memory}) \\ ID/EX.NPC + ID/EX.I & (\text{Branch}) \end{cases}$$

$$EX/MEM.B \leftarrow ID/EX.B$$

$$EX/MEM.IR \leftarrow ID/EX.IR$$

Type	0-5	6-10	11-15	16-31	
R	op	rs1	rs2	rd	function
I	op	rs	rd	immediate	

Memory (MEM)

$$MEM/WB.ALU_{OUT} \leftarrow EX/MEM.ALU_{OUT}$$

$$MEM/WB.LMD \leftarrow Mem[EX/MEM.ALU_{OUT}] \quad (Load)$$

$$Mem[EX/MEM.ALU_{OUT}] \leftarrow EX/MEM.B \quad (Store)$$

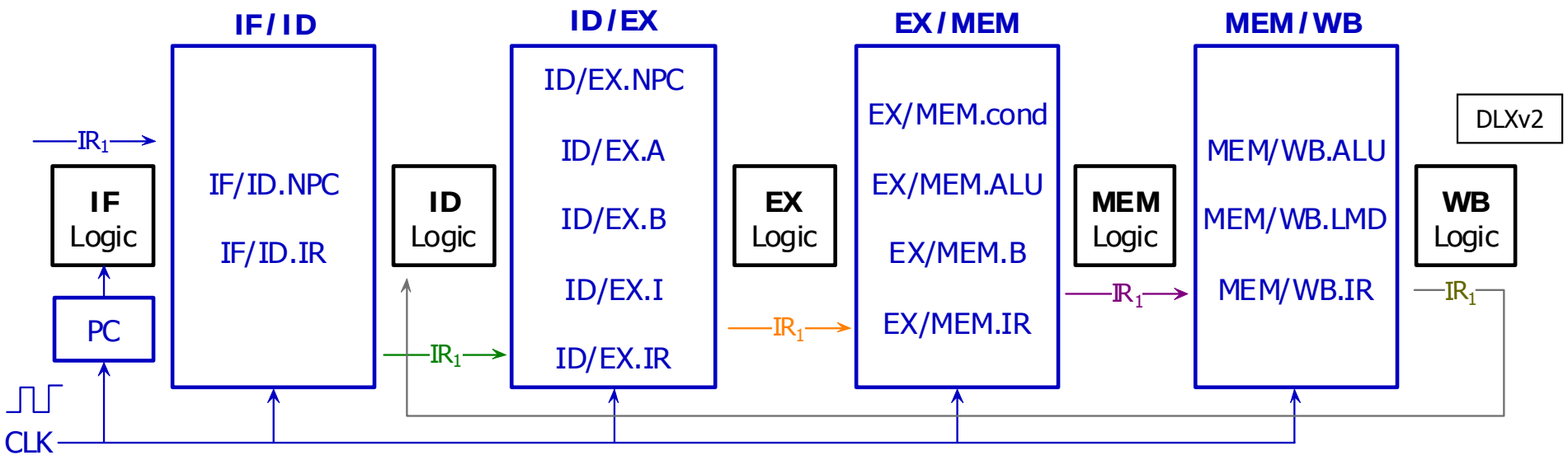
$$MEM/WB.IR \leftarrow EX/MEM.IR$$

Write Back (WB)

$$Reg[MEM/WB.IR_{11\text{-}15}] \leftarrow \begin{cases} MEM/WB.ALU_{OUT} & (I\text{-}ALU) \\ MEM/WB.LMD & (Load) \end{cases}$$

$$Reg[MEM/WB.IR_{16\text{-}20}] \leftarrow MEM/WB.ALU_{OUT} \quad (R\text{-}ALU)$$

Instruction Transfer Timing

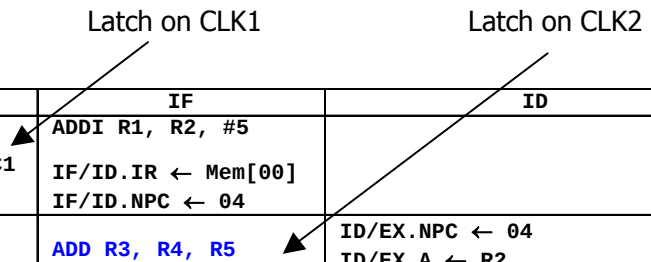


CLK 0	CC 1 begins	Memory ← PC(I_1)	IF/ID.IR "sees" Mem[PC(I_1)]
CLK 1	CC 2 begins	IF/ID.IR ← Mem[PC(I_1)] Memory ← PC(I_2)	ID/EX.IR "sees" Mem[PC(I_1)] IF/ID.IR "sees" Mem[PC(I_2)]
CLK 2	CC 3 begins	ID/EX.IR ← Mem[PC(I_1)] IF/ID.IR ← Mem[PC(I_2)] Memory ← PC(I_3)	EX/MEM.IR "sees" Mem[PC(I_1)] ID/EX.IR "sees" Mem[PC(I_2)] IF/ID.IR "sees" Mem[PC(I_3)]
CLK 3	CC 4 begins	EX/MEM.IR ← Mem[PC(I_1)] …	MEM/WB.IR "sees" Mem[PC(I_1)] …
CLK 4	CC 5 begins	MEM/WB.IR ← Mem[PC(I_1)]	Mem[PC(I_1)] controls Write Back

Simple 5-Instruction Program for DLX

Instruction Number	Address	Instruction
I_1	00	ADDI R1, R2, #5
I_2	04	ADD R3, R4, R5
I_3	08	SW 32(R6), R7
I_4	0C	LW R8, 32(R9)
I_5	10	AND R10, R12, R13

Program Execution Table



	IF	ID	EX	MEM	WB
CC1	ADDI R1, R2, #5 IF/ID.IR ← Mem[00] IF/ID.NPC ← 04				
CC2	ADD R3, R4, R5 IF/ID.IR ← Mem[04] IF/ID.NPC ← 08	ID/EX.NPC ← 04 ID/EX.A ← R2 ID/EX.B ← R1 ID/EX.I ← 5 ID/EX.IR ← ADDI R1, R2, #5			
CC3	SW 32(R6), R7 IF/ID.IR ← Mem[08] IF/ID.NPC ← 0C	ID/EX.NPC ← 08 ID/EX.A ← R4 ID/EX.B ← R5 ID/EX.I ← ??? ID/EX.IR ← ADD R3, R4, R5	EX/MEM.cond ← (R2 == 0) EX/MEM.ALU ← R2 + 5 EX/MEM.B ← R1 EX/MEM.IR ← ADDI R1, R2, #5		
CC4	LW R8, 32(R9) IF/ID.IR ← Mem[0C] IF/ID.NPC ← 10	ID/EX.NPC ← 0C ID/EX.A ← R6 ID/EX.B ← R7 ID/EX.I ← 32 ID/EX.IR ← SW 32(R6), R7	EX/MEM.cond ← (R4 == 0) EX/MEM.ALU ← R4 + R5 EX/MEM.B ← R5 EX/MEM.IR ← ADD R3, R4, R5	MEM/WB.ALU ← R2 + 5 MEM/WB.IR ← ADDI R1, R2, #5	
CC5	AND R10, R12, R13 IF/ID.IR ← Mem[10] IF/ID.NPC ← 14	ID/EX.NPC ← 10 ID/EX.A ← R9 ID/EX.B ← R8 ID/EX.I ← 32 ID/EX.IR ← LW R8, 32(R9)	EX/MEM.cond ← (R6 == 0) EX/MEM.ALU ← R6 + 32 EX/MEM.B ← R7 EX/MEM.IR ← SW 32(R6), R7	MEM/WB.ALU ← R4 + R5 MEM/WB.IR ← ADD R3, R4, R5	R1 ← R2 + 5
CC6		ID/EX.NPC ← 14 ID/EX.A ← R12 ID/EX.B ← R13 ID/EX.I ← ??? ID/EX.IR ← AND R10, R12, R13	EX/MEM.cond ← (R9 == 0) EX/MEM.ALU ← R9 + 32 EX/MEM.B ← R8 EX/MEM.IR ← LW R8, 32(R9)	Mem[R6 + 32] ← R7 MEM/WB.ALU ← R6 + 32 MEM/WB.IR ← SW 32(R6), R7	R3 ← R4 + R5
CC7			EX/MEM.cond ← (R12 == 0) EX/MEM.ALU ← R12 AND R2 EX/MEM.B ← R13 EX/MEM.IR ← AND R10, R12, R13	MEM/WB.LMD ← Mem[R9 + 32] MEM/WB.ALU ← R9 + 32 MEM/WB.IR ← LW R8, 32(R9)	
CC8				MEM/WB.ALU ← R12 AND R2 MEM/WB.IR ← AND R10, R12, R13	R8 ← Mem[R9 + 32]
CC9					R10 ← R12 AND R2

DLXv2

First Clock Cycles

	IF	ID	EX
CC1	ADDI R1, R2, #5 IF/ID.IR ← Mem[00] IF/ID.NPC ← 04		
CC2	ADD R3, R4, R5 IF/ID.IR ← Mem[04] IF/ID.NPC ← 08	ID/EX.NPC ← 04 ID/EX.A ← R2 ID/EX.B ← R1 ID/EX.I ← 5 ID/EX.IR ← ADDI R1, R2, #5	
CC3	SW 32(R6), R7 IF/ID.IR ← Mem[08] IF/ID.NPC ← 0C	ID/EX.NPC ← 08 ID/EX.A ← R4 ID/EX.B ← R5 ID/EX.I ← ??? ID/EX.IR ← ADD R3, R4, R5	EX/MEM.cond ← (R2 == 0) EX/MEM.ALU ← R2 + 5 EX/MEM.B ← R1 EX/MEM.IR ← ADDI R1, R2, #5
CC4	LW R8, 32(R9) IF/ID.IR ← Mem[0C] IF/ID.NPC ← 10	ID/EX.NPC ← 0C ID/EX.A ← R6 ID/EX.B ← R7 ID/EX.I ← 32 ID/EX.IR ← SW 32(R6), R7	EX/MEM.cond ← (R4 == 0) EX/MEM.ALU ← R4 + R5 EX/MEM.B ← R5 EX/MEM.IR ← ADD R3, R4, R5

DLXv2

After CLK0

Memory ← PC =00 ⇒ IF/ID.IR "sees" Mem[00] and IF/ID.NPC "sees" 04 as inputs

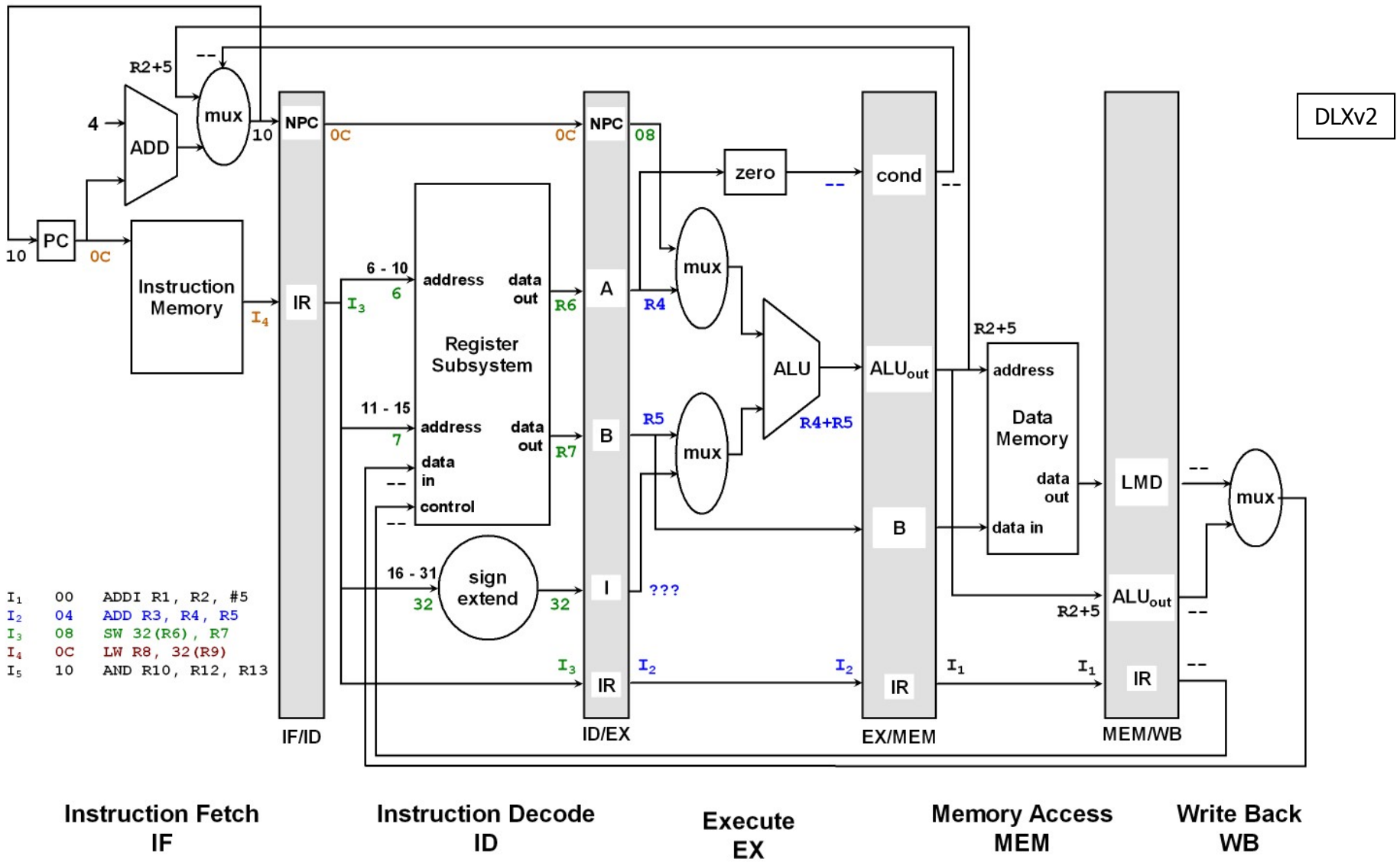
After CLK 1

Memory ← PC =04 ⇒ IF/ID.IR "sees" Mem[04] and IF/ID.NPC "sees" 08 as inputs

IF/ID.IR latches Mem[00] and ID/EX.IR "sees" IF/ID.IR (ADDI R1, R2, #5) as input

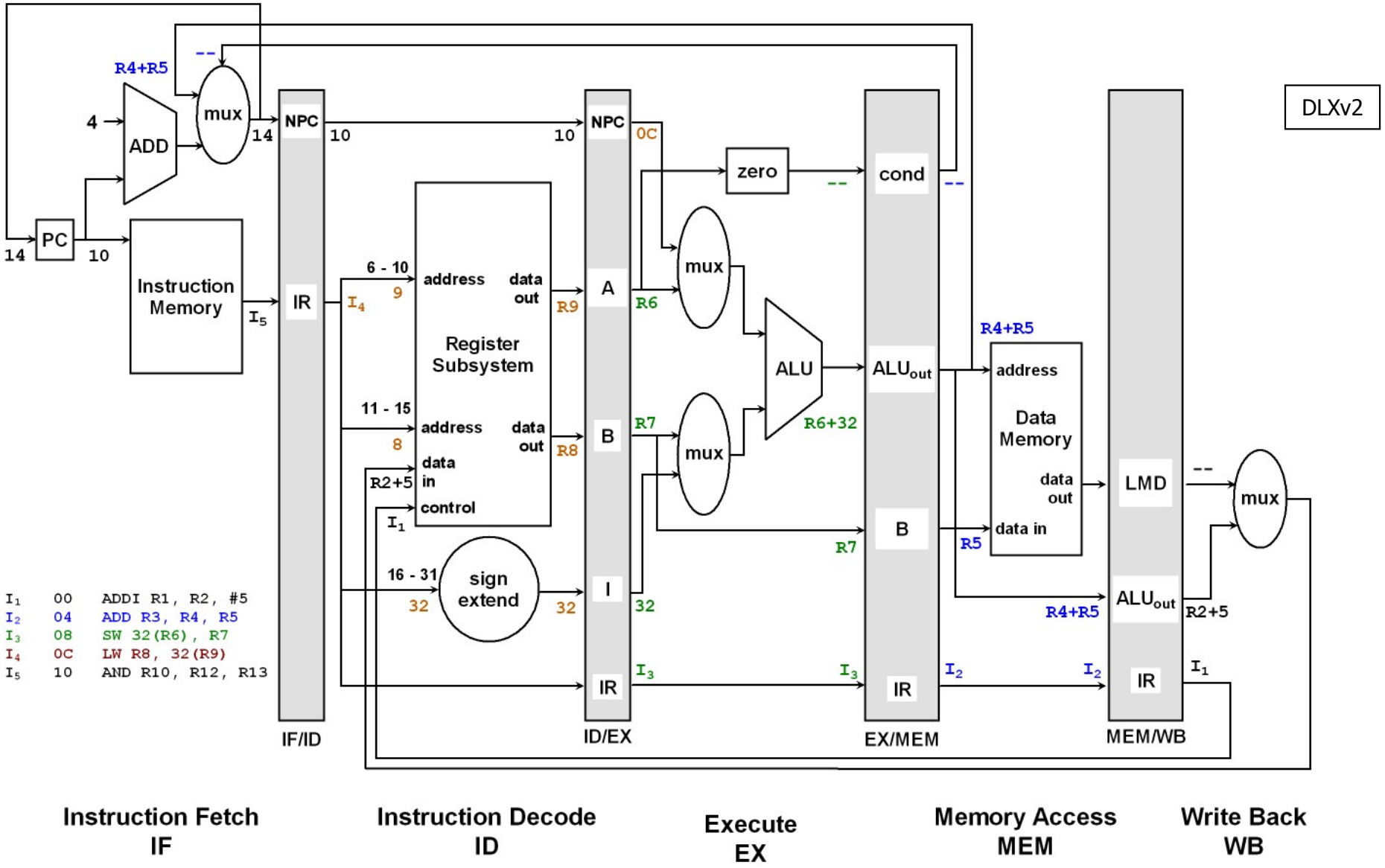
Registers "see" IF/ID.IR and ID/EX.A,B,I "see" R2, R1, 5 as inputs

Processor State Just Before CLK 4



Input and Output Data at Stage Buffers in CC 4

Processor State Just After CLK 4



Input and Output Data at Stage Buffers in CC 5

New Technology,
New Headaches

Analysis of Pipeline Hazards

Instruction Dependencies: Definitions

Instruction dependencies

Result of one instruction needed to execute later instruction

Hazard

Processor runs smoothly but provides wrong answers

Pipeline hazard

Several instructions in various stages of execution

Pipeline uses a resource value before update by earlier instruction

Example

PC ← **NPC** on each clock cycle

Branch instruction requires **PC** ← **NPC+I**

Correct evaluation of **NPC+I** not available on next clock cycle

Hazard Types

Structural Hazard — conflict over access to resource

Data Hazard — instruction result not ready when needed

Control Hazard — branch address not ready when needed

Dealing with Hazards

Avoid error

Pause pipeline and wait for resource to be available

Called **wait state** or **pipeline stall**

Degrades processor performance

Adds stall clock cycles to instruction execution

$$CPI = \frac{\text{processing clock cycles (ideal)} + \text{stalled clock cycles}}{\text{completed instructions}}$$

$$= \frac{N^{ideal} + N^{stall}}{IC} = CPI^{ideal} + CPI^{stall} \xrightarrow[IC \rightarrow \text{ large on DLX}]{} 1 + CPI^{stall}$$

$$\text{performance degradation} = 1 - \frac{CPI^{ideal}}{CPI^{ideal} + CPI^{stall}} = \frac{CPI^{stall}}{1 + CPI^{stall}}$$

Eliminate cause of stall

Improve implementation based on analysis of stalls

Main activity of hardware architects

Structural Hazards

Conflict over access to resource

No structural hazards in DLX

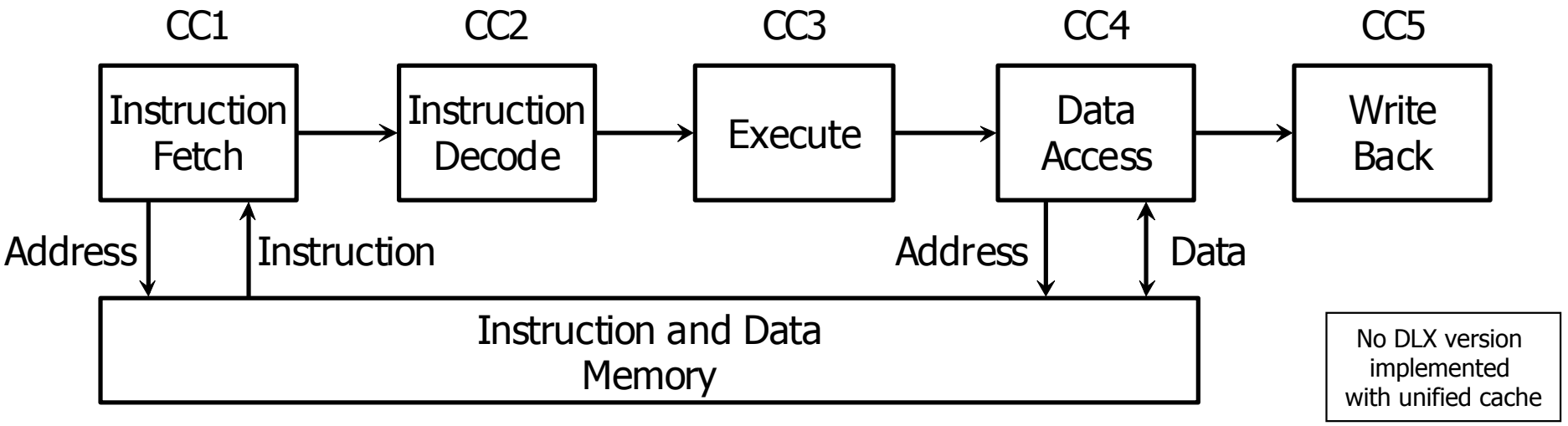
Typical structural hazard — unified cache hazard

Instructions and data in same memory device

Cannot access data and fetch instruction on same clock cycle

Instruction fetch waits 1 clock cycle for every data memory access

Loads and Stores



Stall on Cache Hazard

	IF	ID	EX	MEM	WB
CC1	I_1				
CC2	LW	I_1			
CC3	I_2	LW	I_1		
CC4	I_3	I_2	LW	I_1	
CC5	ϕ	I_3	I_2	LW	I_1
CC6	I_4	ϕ	I_3	I_2	LW
CC7		I_4	ϕ	I_3	I_2
CC8			I_4	ϕ	I_3
				I_4	ϕ
					I_4

On CC5 Load Word (LW) instruction blocks Instruction Fetch (IF)

No instruction is fetched on CC5

No instruction (NOP) is forwarded to ID on CC6

NOP = bubble = Φ forwarded to EX on CC7, etc

No DLX version implemented with unified cache

Effect of Cache Hazard on CPI

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{instructions}} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instructions}} = \sum_{\text{i = type}} \left(\frac{\text{stall cycles}}{\text{stall}} \right)_i \times \left(\frac{\text{stalls}}{\text{instructions}} \right)_i$$

$$= \sum_{i,j} \left(\frac{\text{stall cycles}}{\text{stall}} \right)_i \times \frac{\text{stalls of type i}}{\text{instructions of type j}} \times \frac{\text{instructions of type j}}{\text{instructions}}$$

$$= \sum_{i} \left(\frac{\text{stall cycles}}{\text{data stall}} \right)_i \times \left(\frac{\text{data stalls}}{\text{instructions}} \right)_i \times \frac{IC_i}{IC}$$

(instruction j only causes stall type j)

$$CPI^{stall}_{cache} = \frac{\text{1 stall cycle}}{\text{stall}} \times \frac{\text{1 stall}}{\text{data memory load}} \times \frac{IC^{load}}{IC} + \frac{\text{1 cycle}}{\text{stall}} \times \frac{\text{1 stall}}{\text{data memory store}} \times \frac{IC^{store}}{IC}$$

$$= \frac{\text{1 stall cycle}}{\text{stall}} \times \frac{\text{1 stall}}{\text{data memory access}} \times \left(\frac{IC^{load}}{IC} + \frac{IC^{store}}{IC} \right)$$

$$= \frac{\text{1 stall cycle}}{\text{stall}} \times \frac{\text{1 stall}}{\text{data memory access}} \times \left(\frac{\text{0.25 loads}}{\text{instruction}} + \frac{\text{0.15 stores}}{\text{instruction}} \right)$$

$$= 0.40 \frac{\text{stall cycles}}{\text{instruction}} \Rightarrow CPI = CPI^{ideal} + CPI^{stall} = 1.40$$

Data Hazards

Instruction result not ready when needed

Operations performed in the wrong order

Classification named for correct order of operations

Read After Write (RAW)

Correct	I2 reads register after I1 writes to it
Hazard	I2 reads register before I1 writes to it
	I2 uses incorrect value

Write After Write (WAW)

Correct	I2 writes to register after I1 writes to it
Hazard	I2 writes to register before I1 writes to it
	Incorrect value stays in register

Write After Read (WAR)

Correct	I2 writes to register after I1 reads it
Hazard	I2 writes to register before reads I1 it
	I1 uses incorrect value

Read After Read (RAR)

No hazard — reads do not affect registers

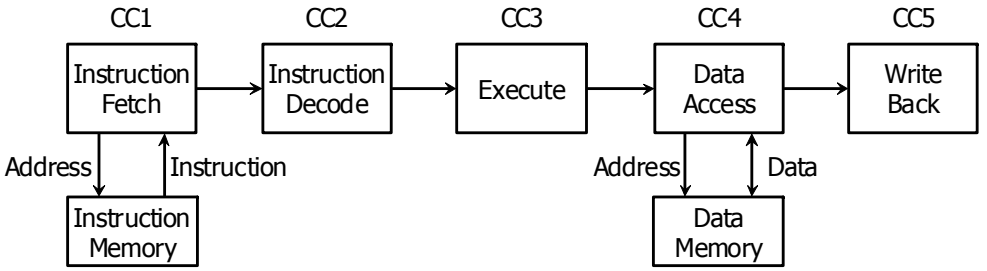
Data Hazards in DLXv2

RAW hazards

DLX registers updated in stage 5

Next instruction may read register in stage 2

Possible hazard to be avoided



WAW hazards cannot occur

DLX writes in uniform order

- Memory updated in MEM
- Registers updated in WB

All updates performed in order of execution

- I_2 cannot perform WB or MEM before I_1 performs WB or MEM

WAR hazards cannot occur

Loads performed in MEM and register reads in ID

Stores performed in MEM and registers updated in WB

I_2 cannot perform WB or MEM before I_1 performs ID or MEM

Register-Register RAW Dependencies in DLXv2

Program with register-register dependencies

```
I1   ADD  R1,R2,R3
I2   SUB  R4,R5,R1
I3   AND  R6,R7,R1
I4   OR   R8,R9,R1
```

I_1 has R1 as **destination**

I_2 — I_4 have R1 as **source**

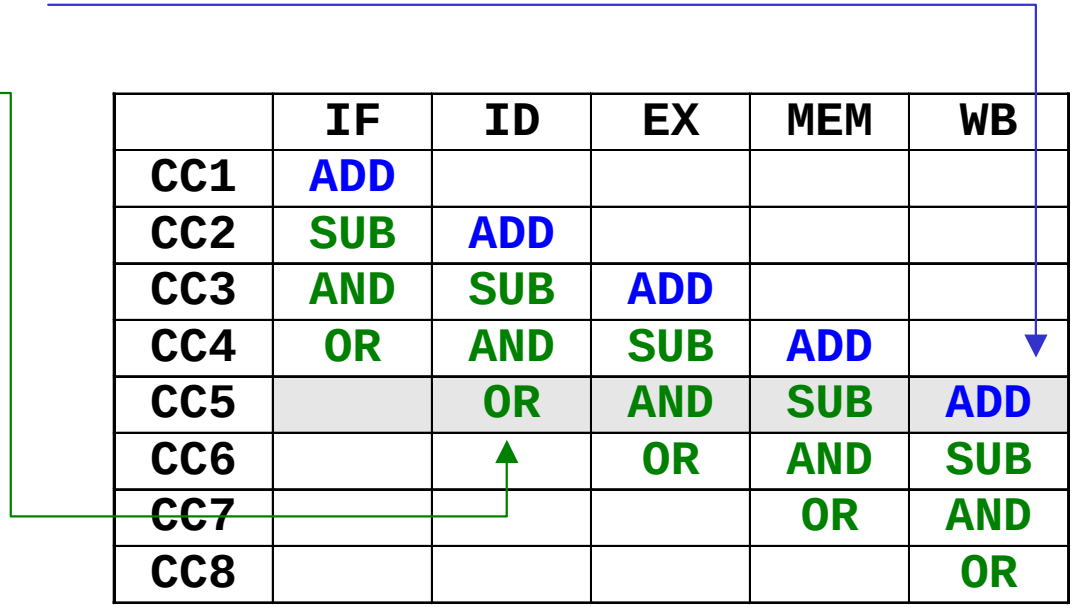
Bad timing (uncorrected execution)

I_1 updates R1 in WB during CC5

I_2 reads R1 in ID during CC3

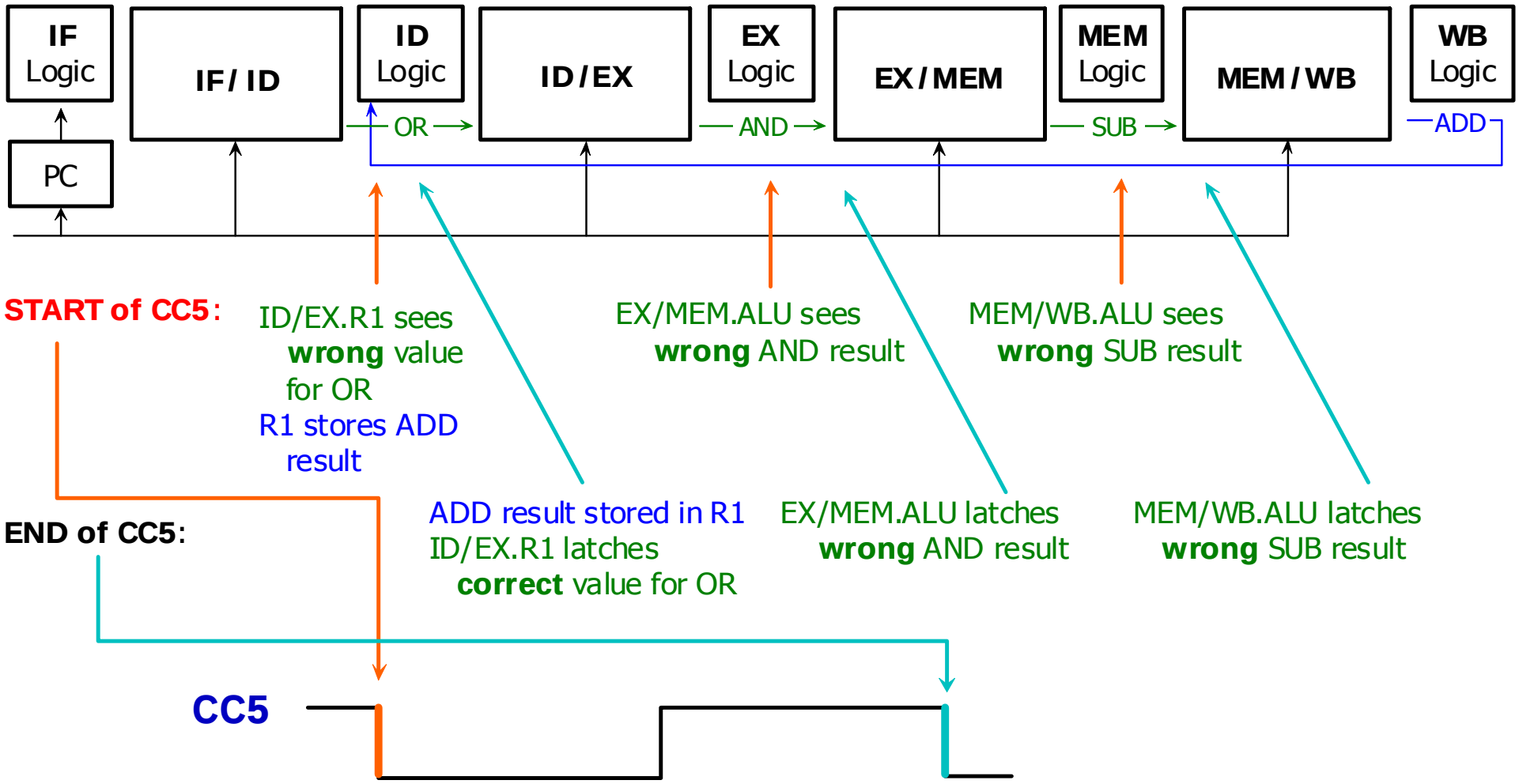
I_3 reads R1 in ID during CC4

I_4 reads R1 in ID during CC5



	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

Detailed View of CC5 (Uncorrected) in DLXv2



SUB and AND instructions suffer RAW hazard — read wrong value of R1

OR instruction reads correct value of R1

Pipeline Stall to Avoid RAW Hazard in DLXv2

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	AND	SUB	ϕ	ADD	
CC5	AND	SUB	ϕ	ϕ	ADD
CC6	OR	AND	SUB	ϕ	ϕ
CC7		OR	AND	SUB	ϕ
CC8			OR	AND	SUB
				OR	AND
					OR

The DLX control system must be able to identify all hazards and insert stall cycles when necessary.

Wait states during CC3 and CC4

ID/EX freezes internal state on **SUB**

IF/ID freezes internal state on **AND** (cannot enter **ID** until **SUB** finishes and moves to **EX**)

ID performs **NOP** (no operation) to avoid reading old value of R1

ID/EX passes ϕ (**NOP**) to **EX**

Continuation — no hazard in CC5

WB operation performed at start of clock cycle

Latching of register values in **ID** performed at end of clock cycle

Pipeline Stall in Instruction View in DLXv2

Clock Cycle →

	1	2	3	4	5	6	7	8
ADD R1,R2,R3	IF	ID	EX	MEM	WB			
SUB R4,R5,R1		IF	ID	ID	ID	EX	MEM	WB
AND R6,R7,R1			IF	IF	IF	ID	EX	MEM
OR R8,R9,R1						IF	ID	EX

Wait states — ID/EX freezes state and passes NOP (no operation) to EX

Performance degradation too large

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

$$= \frac{\text{2 stall cycle}}{\text{stall}} \times \frac{\text{0.5 register dependencies}}{\text{ALU instruction}} \times \frac{\text{0.4 ALU}}{\text{instruction}}$$

$$\frac{IC^{ALU}}{IC} = 40\%$$

$$= 2 \times 0.5 \times 0.4 \frac{\text{cycles}}{\text{instruction}} \quad \Rightarrow \quad CPI = 1.4 \Rightarrow \text{(29\% degradation)}$$

Forwarding or Bypass (DLX Version 3)

ADD writes **ALU result** to **R1** in **CC5**

SUB needs **R1** for **ALU** operation in **CC4**

AND needs **R1** for **ALU** operation in **CC5**

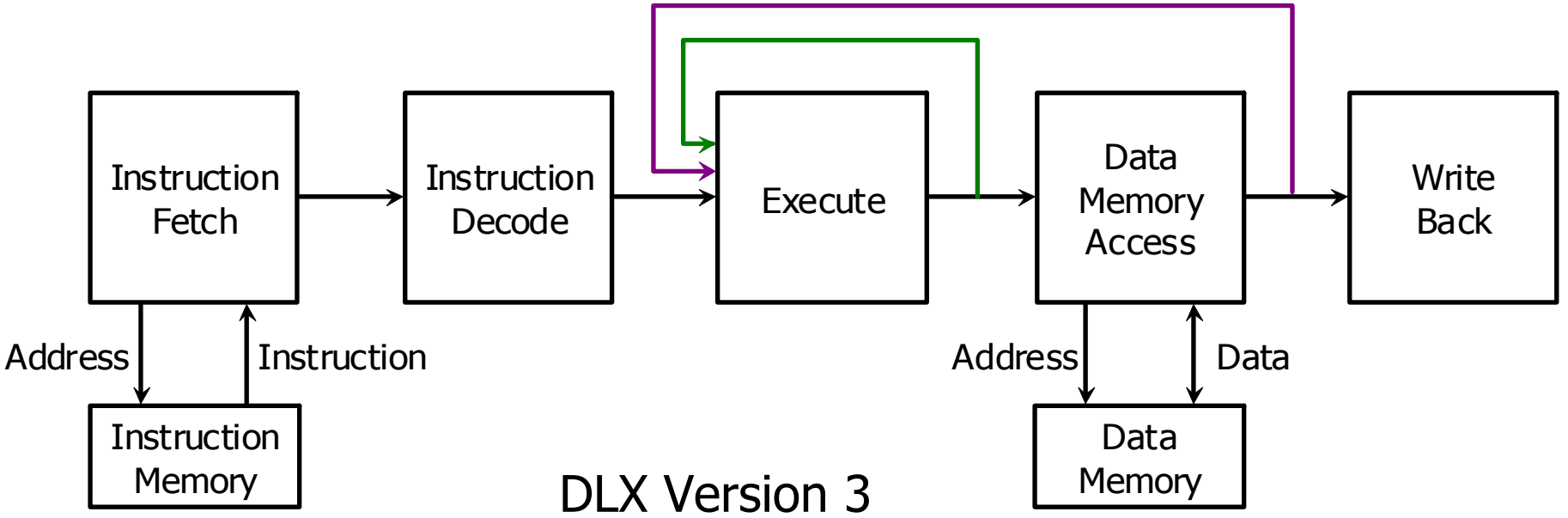
	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

Trick to prevent stall

ADD calculates **ALU result** in **CC3**

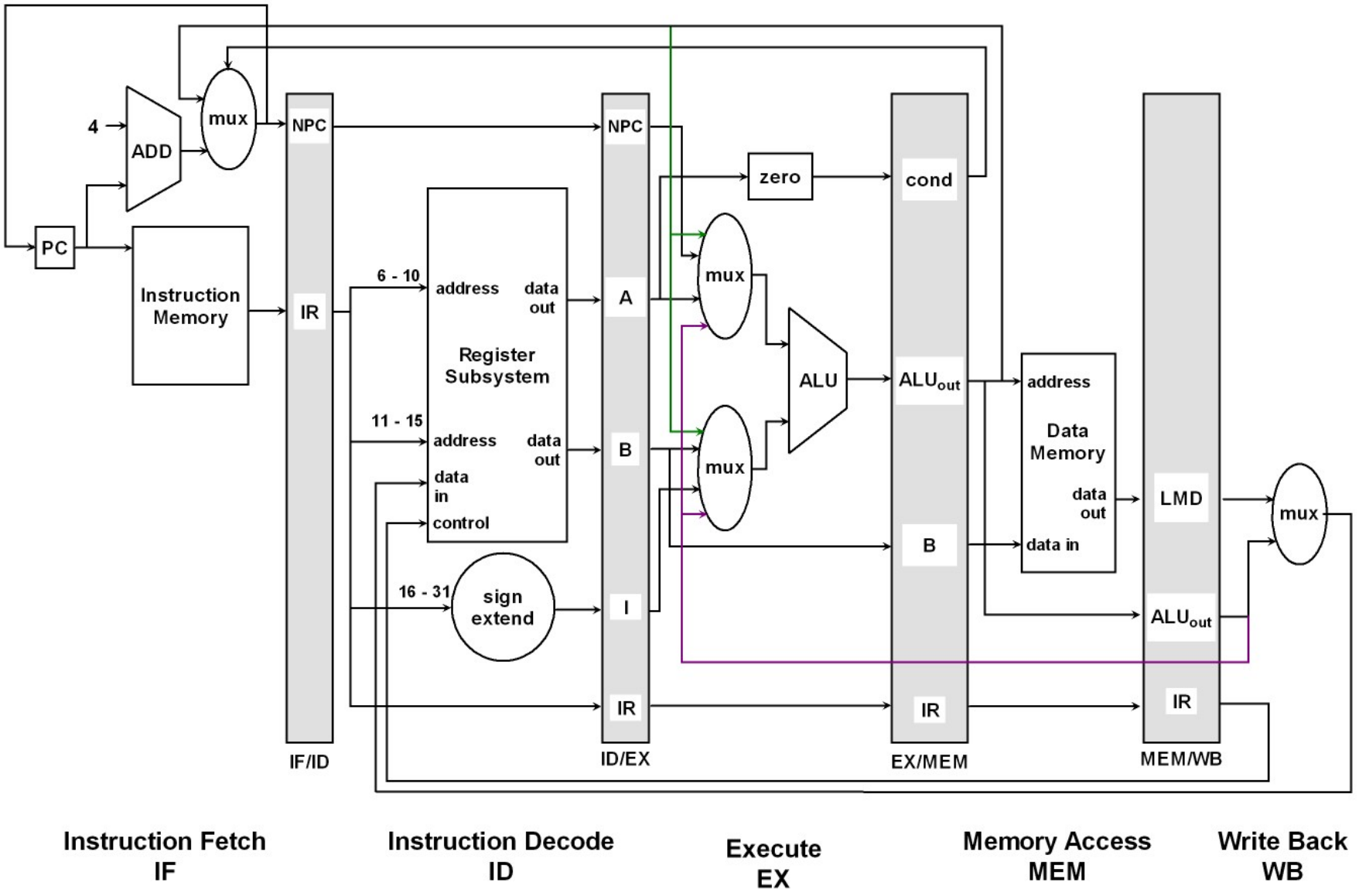
Allow **SUB** and **AND** to read incorrect value in **ID**

Provide correct value from **EX/MEM.ALU** and **MEM/WB.ALU** directly to **EX**



DLX Version 3

DLX Pipelined Implementation in DLXv3



MUXes in EX choose from NPC, A, B, I, EX/MEM.ALU, MEM/WB.ALU

Forwarding in Instruction View in DLXv3

Clock Cycle →

	1	2	3	4	5	6
ADD R1,R2,R3	IF	ID	EX	MEM	WB	
SUB R4,R5,R1		IF	ID	EX	MEM	WB
AND R6,R7,R1			IF	ID	EX	MEM
OR R8,R9,R1				IF	ID	EX

Processor moves state of ADD instruction from buffer to buffer

SUB needs **ALU result** in **CC4**

ADD provides **ALU result** from **EX/MEM.ALU**

AND needs **ALU result** in **CC5**

ADD provides **ALU result** from **MEM/WB.ALU**

No stall cycles for Register-Register RAW hazard

$$CPI^{stall} = 0$$

Register-Load RAW Dependencies in DLXv3

Program with register-load dependencies

```
I1   LW   R1,32(R2)
I2   SUB  R4,R5,R1
I3   AND  R6,R7,R1
I4   OR   R8,R9,R1
```

I_1 has **R1** as **destination**

I_2 — I_4 have **R1** as **source**

Bad timing (uncorrected execution)

I_1 updates **R1** in **WB** during **CC5**

I_2 reads **R1** in **ID** during **CC3**

I_3 reads **R1** in **ID** during **CC4**

I_4 reads **R1** in **ID** during **CC5**

	IF	ID	EX	MEM	WB
CC1	LW				
CC2	SUB	LW			
CC3	AND	SUB	LW		
CC4	OR	AND	SUB	LW	
CC5		OR	AND	SUB	LW
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

Memory Forwarding or Bypass (Version 4)

LW writes **loaded data** to **R1** in **CC5**

SUB needs **R1** for **ALU** operation in **CC4**

AND needs **R1** for **ALU** operation in **CC5**

	IF	ID	EX	MEM	WB
CC1	LW				
CC2	SUB	LW			
CC3	AND	SUB	LW		
CC4	OR	SUB	ϕ	LW	
CC5		AND	SUB	ϕ	LW
CC6		OR	AND	SUB	ϕ
CC7			OR	AND	SUB
CC8				OR	AND
CC9					OR

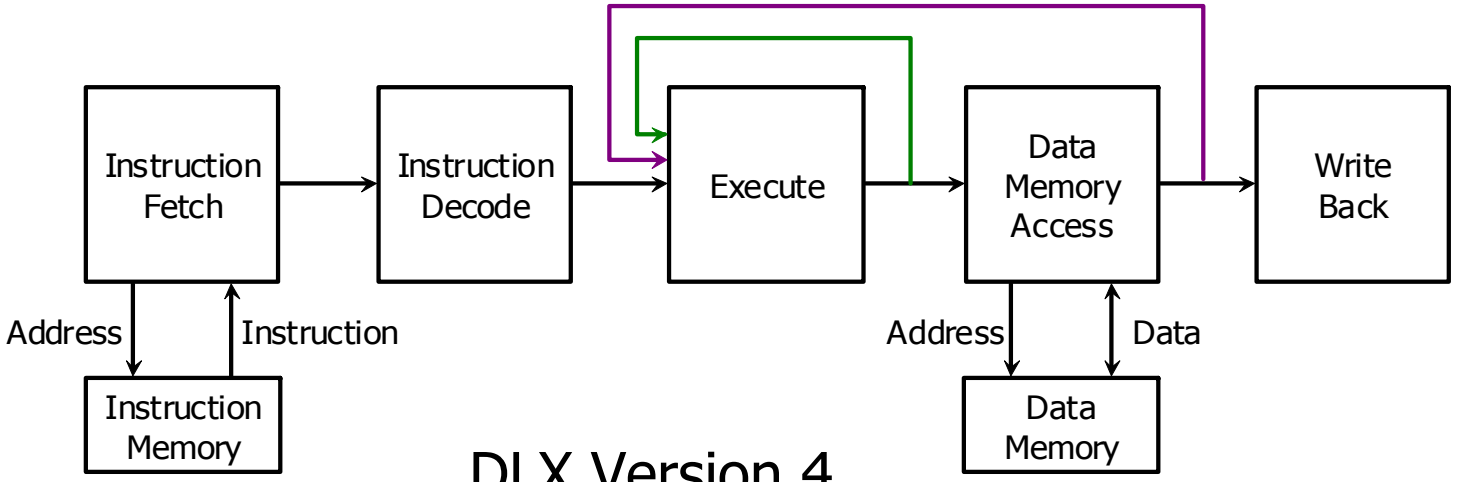
Trick to minimize stall

LW loads **loaded data** in **CC4**

Allow **SUB** to read incorrect value in **ID**

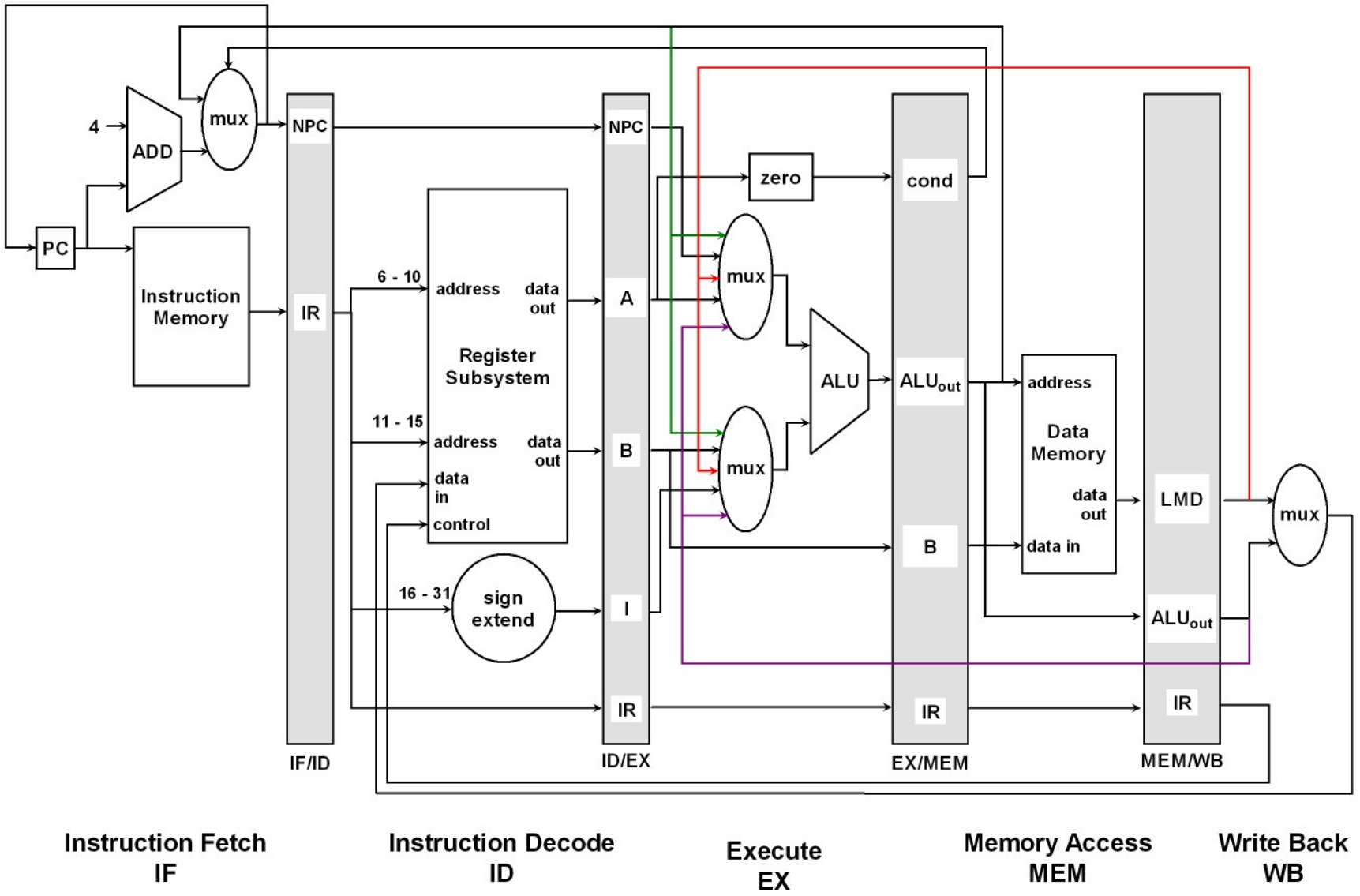
Stall **SUB** for 1 clock cycle in ID (load performed later than ALU operation)

Provide correct value from **MEM/WB.LMD** directly to **EX**



DLX Version 4

DLX Pipelined Implementation in DLXv4



MUXes in EX choose from NPC, A, B, I, EX/MEM.ALU, MEM/WB.ALU, MEM/WB.ALU

Forwarding in Instruction View in DLXv4

Clock Cycle →

	1	2	3	4	5	6	7
LW R1,32(R2)	IF	ID	EX	MEM	WB		
SUB R4,R5,R1		IF	ID	ID	EX	MEM	WB
AND R6,R7,R1			IF	IF	ID	EX	MEM
OR R8,R9,R1					IF	ID	EX

Loaded data used immediately in ALU operation in about 50 % of loads

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

$$= \frac{\text{1 stall cycle}}{\text{stall}} \times \frac{\text{0.5 ALU uses loaded data}}{\text{Load instruction}} \times \frac{IC^{load}}{IC}$$

$$= 0.50 \times 0.25 \frac{\text{cycles}}{\text{instruction}} = 0.125 \frac{\text{cycles}}{\text{instruction}}$$

$$CPI = 1.125 \Rightarrow (11\% \text{ degradation})$$

Register-Store RAW Dependencies in DLXv4

Program with register-store dependency

```
I1   SUB  R1,R5,R4
I2   SW   32(R2),R1
```

I_1 has R1 as **destination**

I_2 has R1 as **source**

Bad timing (uncorrected execution) in DLXv4

I_1 updates R1 in WB during CC5

I_2 reads R1 in ID during CC3

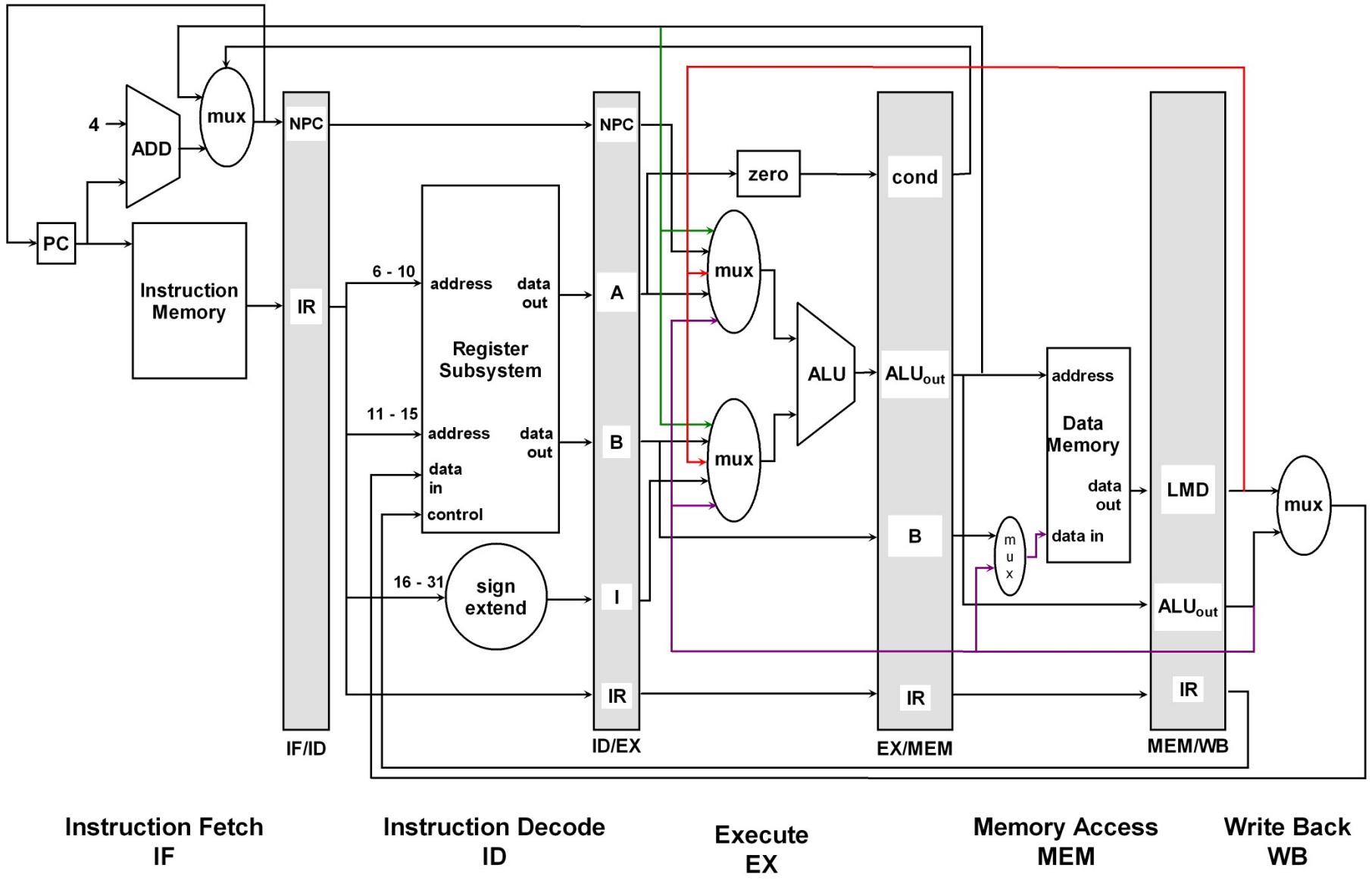
	IF	ID	EX	MEM	WB
CC1	SUB				
CC2	SW	SUB			
CC3		SW	SUB		
CC4			SW	SUB	
CC5				SW	SUB
CC6					SW

Trick to prevent stall (Version 5)

SW reads incorrect value in ID

Provide correct value from MEM/WB.ALU directly to data memory

DLX Pipelined Implementation — Version 5



New MUX in MEM chooses B or MEM / WB.ALU

Compiler Scheduling to Prevent RAW Hazards

C program code

```
I = I + 123;
J = J - 567;
```

	1	2	3	4	5	6	7	8	9	10	11	12
LW	F	D	X	M	W							
ADD		F	D	D	X	M	W					
SW			F	F	D	X	M	W				
LW					F	D	X	M	W			
SUB						F	D	D	X	M	W	
SW							F	F	D	X	M	W

First pass compilation

```
LW   R2, I
ADD R2,R2, #123
SW   I, R2
LW   R3, J
SUB R3, R3, #567
SW   J, R3
```

	1	2	3	4	5	6	7	8	9	10	11	12
LW	F	D	X	M	W							
LW		F	D	X	M	W						
ADD			F	D	X	M	W					
SW				F	D	X	M	W				
SUB					F	D	X	M	W			
SW						F	D	X	M	W		

Second pass compilation

```
LW   R2, I
LW   R3, J
ADD R2,R2, #123
SW   I, R2
SUB R3, R3, #567
SW   J, R3
```

DLXv5

DLX Control Hazard

On each clock cycle

PC ← NPC

New **PC** for new instruction fetch in every clock cycle

Control hazard

Incorrect address on branch instructions

Stages of branch execution

CLK	Clock Cycle	Latched state	Action during CC
0	1	Memory ← PC(I_1)	IF/ID.IR "sees" instruction and PC(I_1)
1	2	IF/ID.IR ← branch	Decode of branch instruction, NPC, I
2	3	ID/EX.NPC,I ← NPC,I	Calculate address NPC+I and cond
3	4	EX/MEM.ALU,cond ← ALU, cond	PC "sees" correct address via MUX using cond to choose NPC or NPC+I
4	5	PC ← branch address	IF/ID.IR "sees" correct instruction

Pipeline Flush for Control Hazard in DLXv5

Pipeline flush

Empty and restart pipeline

Simplest solution to implement

		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_2	Fall-Through		IF	ϕ	ϕ	IF	ID	EX	MEM	WB
I_3				ϕ	ϕ					
...	…	…	…	…	…	…	…	…	…	…
I_T	Target					IF	ID	EX	MEM	WB

Decode branch and flush pipeline

PC "sees" correct address

Fall-Through (NPC)

Target (NPC+I)

Correct instruction is fetched

Performance Degradation for Pipeline Flush

		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_2	Fall-Through		IF	ϕ	ϕ	IF	ID	EX	MEM	WB
I_3				ϕ	ϕ					
...	…	…	…	…	…	…	…	…	…	…
I_T	Target					IF	ID	EX	MEM	WB



Stalled (wasted) cycles

DLXv5

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

$$= \frac{\text{3 stall cycle}}{\text{stall}} \times \frac{\text{1 branch stall}}{\text{branch instruction}} \times \frac{IC^{branch}}{IC}$$

$$= 3 \times 0.20 \frac{\text{cycles}}{\text{instruction}} = 0.60 \frac{\text{cycles}}{\text{instruction}}$$

$$CPI = 1.60 \Rightarrow \text{(38\% degradation)}$$

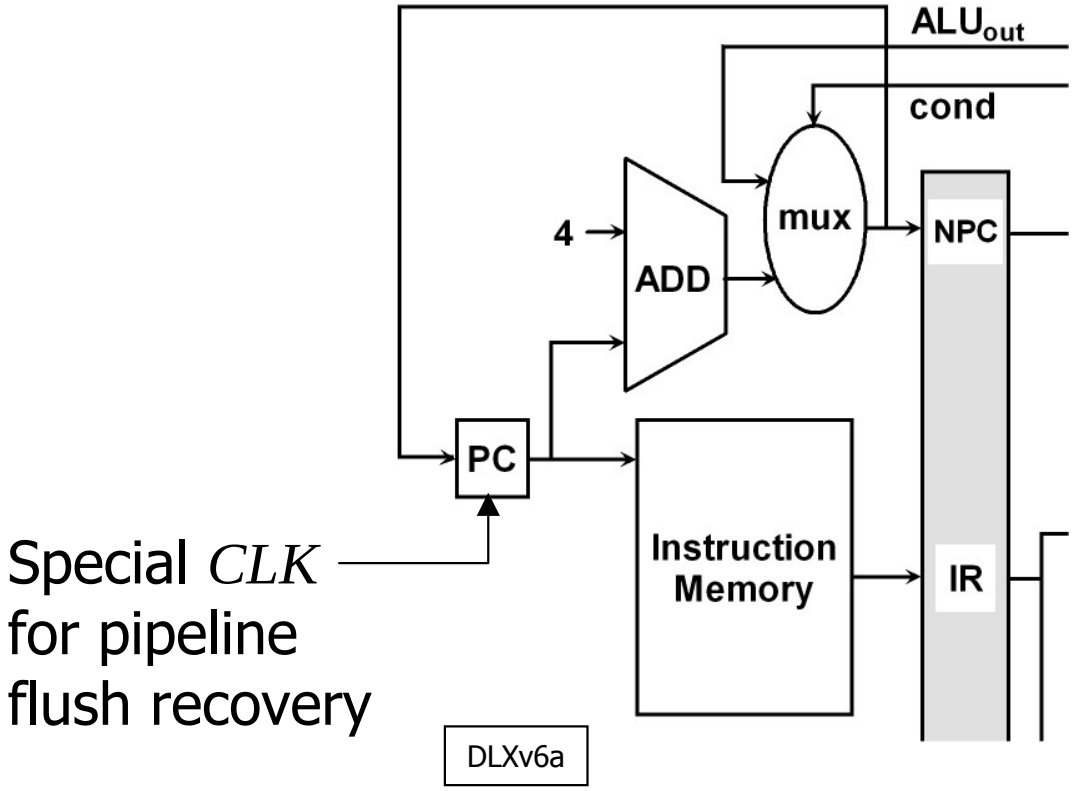
Improving Branch Performance — 1

Enhancement 1

Earlier instruction fetch after pipeline flush

Version 5 PC "sees" correct address in CC 4 but fetches in CC5

Version 6a PC latches correct address when ready — in CC 4



		1	2	3	4
I_1	BEQZ	IF	ID	EX	MEM
I_2	F-T		IF	ϕ	IF
I_3				ϕ	
…	…	…	…	…	…
I_T	Targ				IF

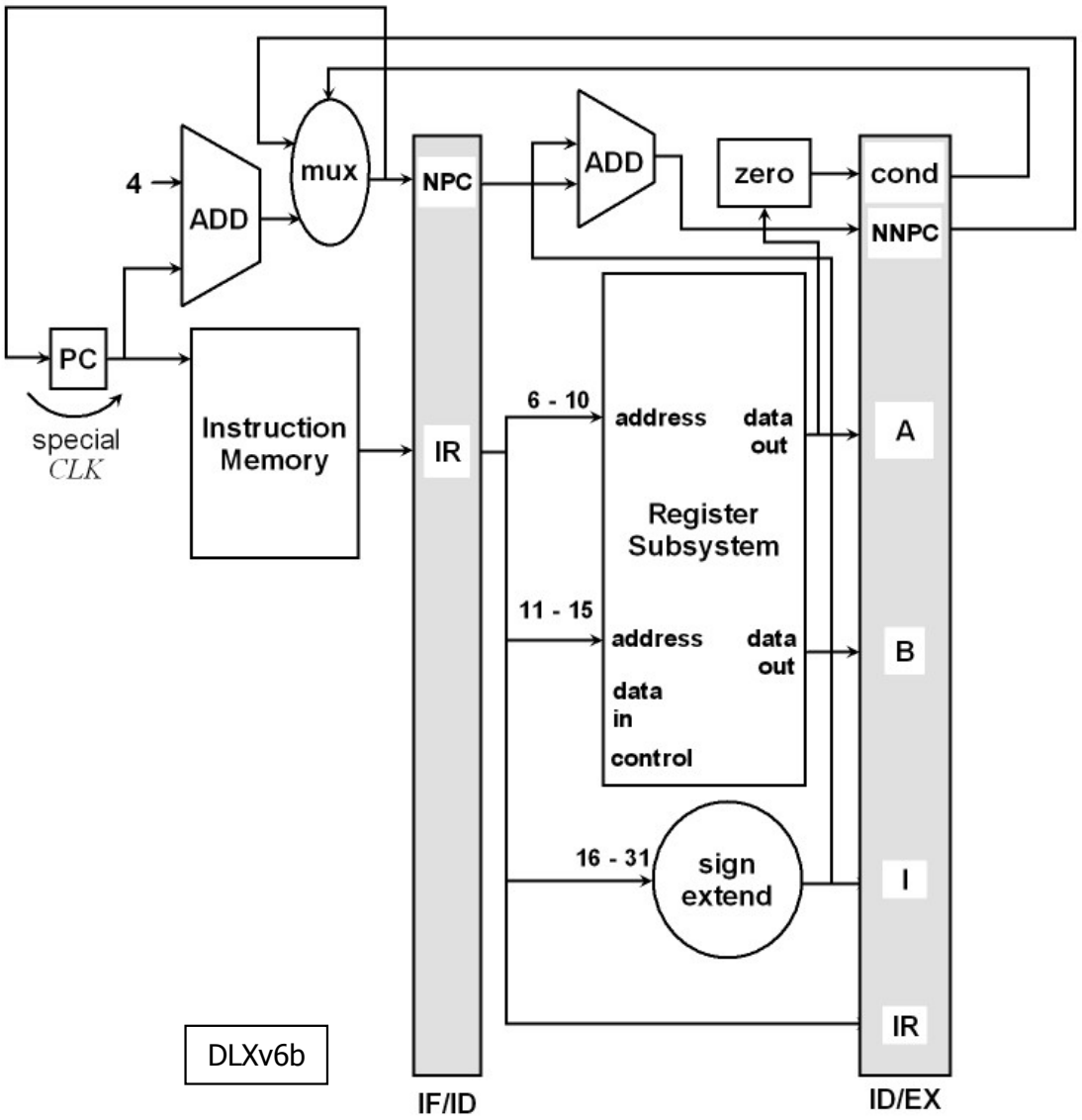
$$CPI^{stall} = 2 \times 0.20 \frac{\text{cycles}}{\text{instruction}}$$

$$= 0.40 \frac{\text{cycles}}{\text{instruction}}$$

$$CPI = 1.40 \Rightarrow \text{(29\% degradation)}$$

Improving Branch Performance — 2

Enhancement 2 — dedicated ALU for branch address in ID stage



		1	2	3
I_1	BEQZ	IF	ID	EX
I_2	F-T		IF	IF
I_3				
…	…	…	…	…
I_T	Targ			IF

Version 6b

Branch address available in CC3

PC updates in CC3

$$CPI^{stall} = 1 \times 0.20 \frac{\text{cycles}}{\text{instruction}}$$

$$= 0.20 \frac{\text{cycles}}{\text{instruction}}$$

$$CPI = 1.20 \Rightarrow \text{(17\% degradation)}$$

Improving Branch Performance — 3

Enhancement 3

Versions 5 – 6b Flush entire pipeline
Restart with correct branch address

Version 6c Flush entire pipeline on **branch taken**
Continue instruction in IF on **branch not taken**

Branch address and cond ready

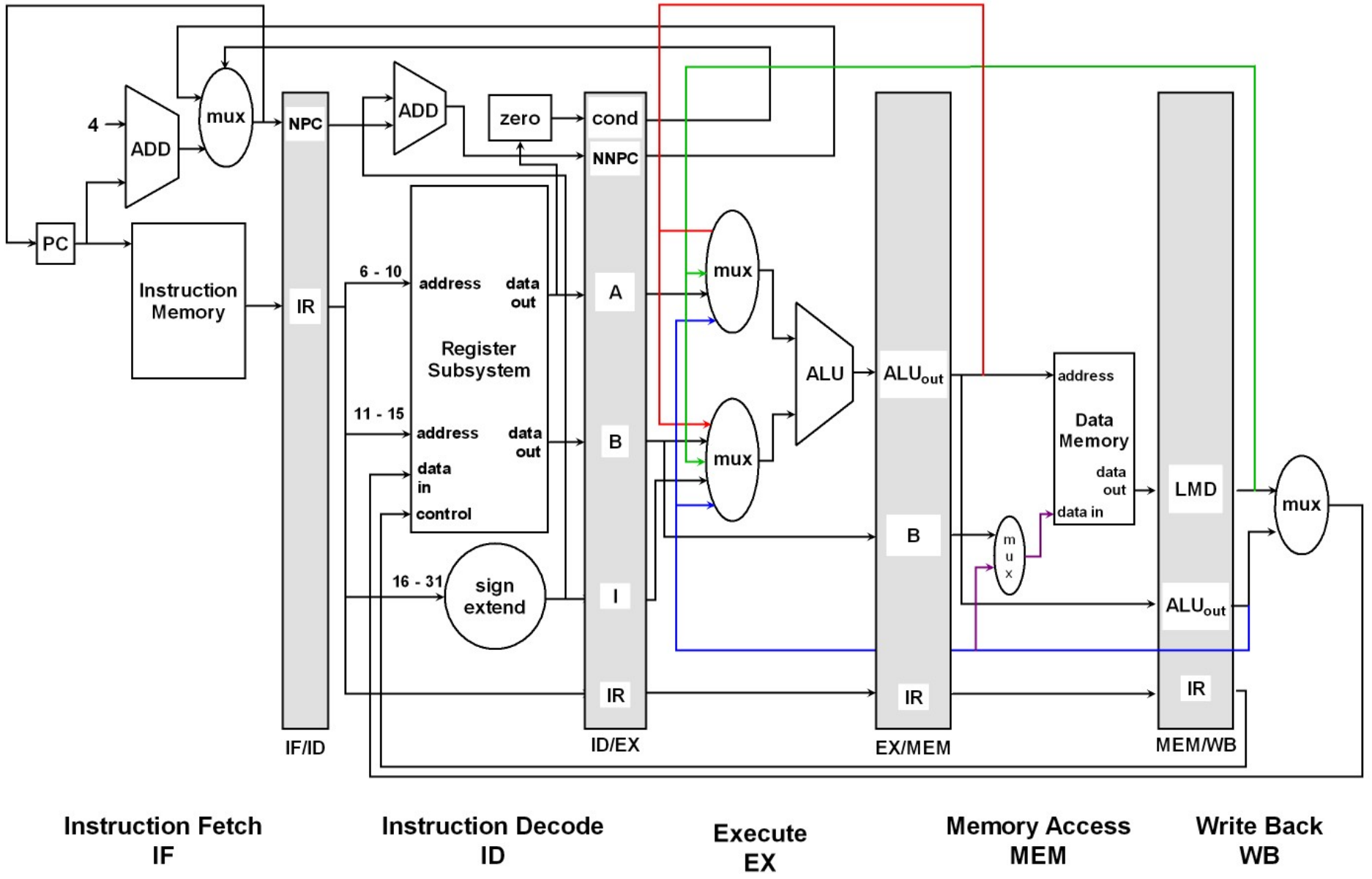
		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_2	Fall-Through		IF	ID	EX	MEM	WB			
I_3				IF						
. . .	…	…	…	…	…	…	…	…	…	…
I_T	Target			IF	ID	EX	MEM	WB		

Branch taken (cond = 1 ⇒ PC ← NPC + I)

Branch not taken (cond = 0 ⇒ PC ← NPC)

DLXv6c

DLX Version 6c



Version 6c Branch Processing — 1

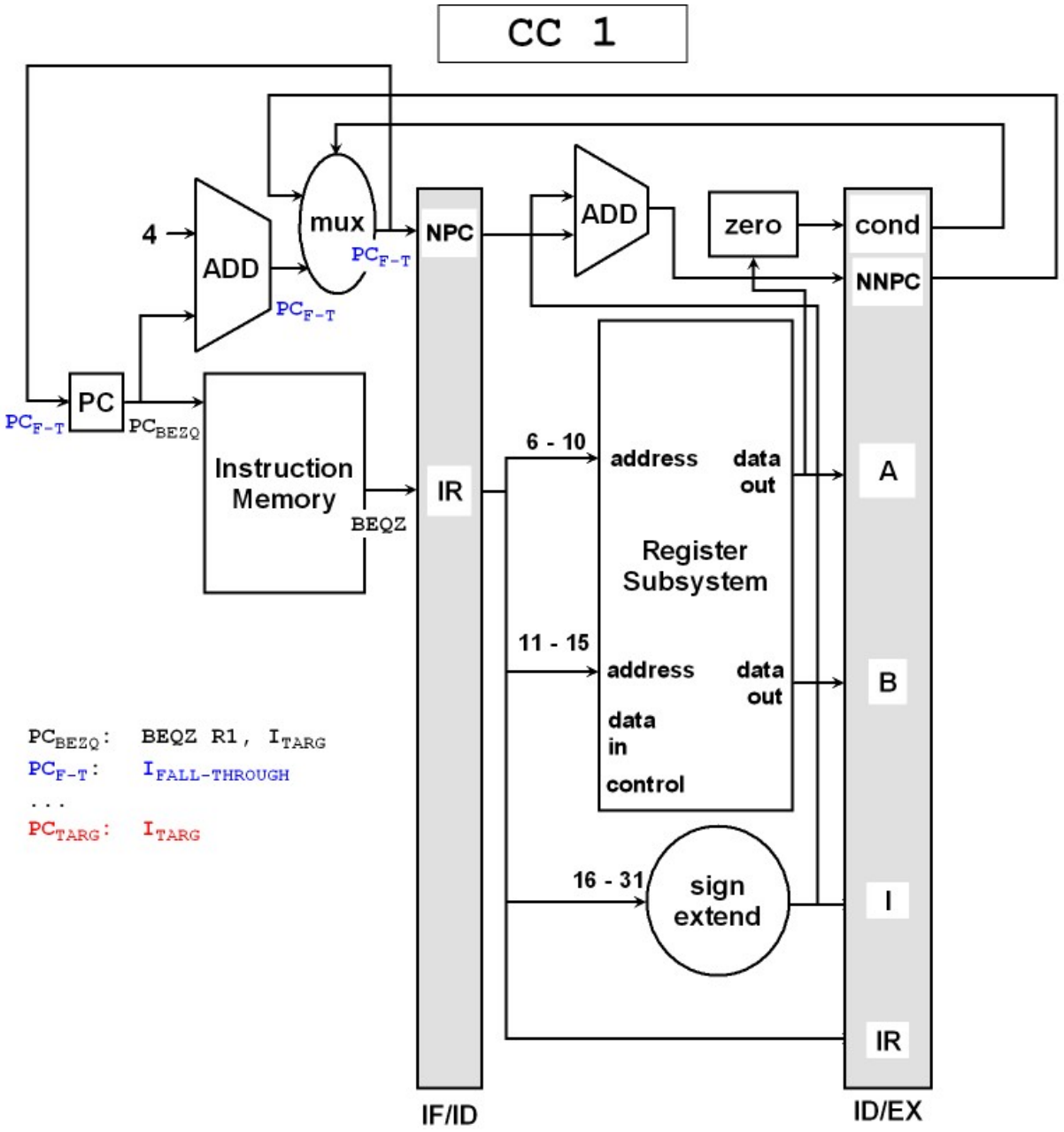
CC1

BEQZ fetched to **IF**

PC "sees"

$PC_{F-T} = NPC = PC+4$

Points to $I_{FALL-THROUGH}$



Version 6c Branch Processing — 2

CC2

IF fetches $I_{FALL-THROUGH}$

BEQZ advances to **ID**

Calculates

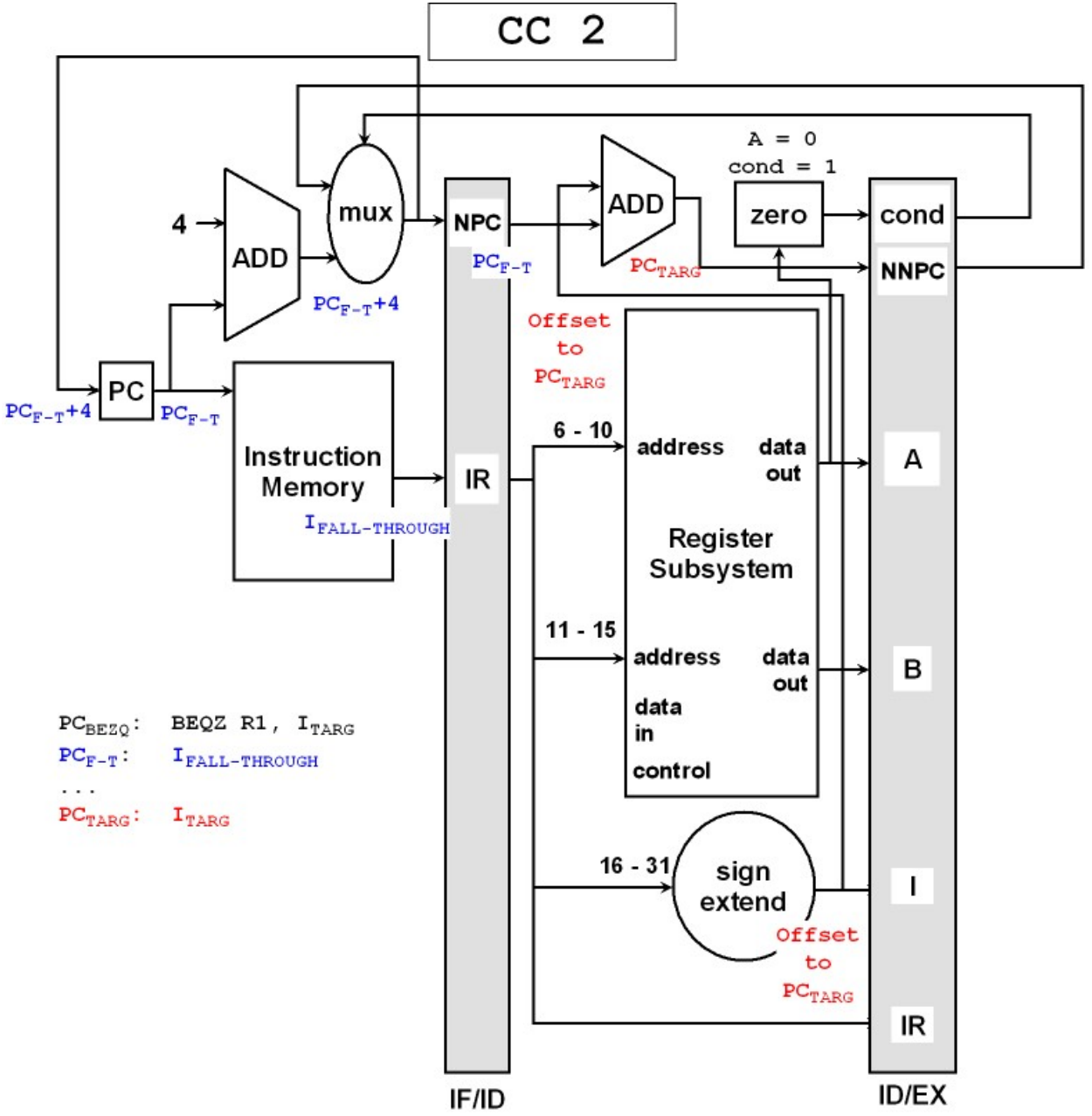
I_{TARG} = NPC+I

cond

PC "sees"

NPC = PC_{F-T}+4

Points to $I_{FALL-THROUGH+1}$



Version 6c Branch Processing — 3

CC3

IF fetches $I_{FALL-THROUGH+1}$

BEQZ advances to **EX**

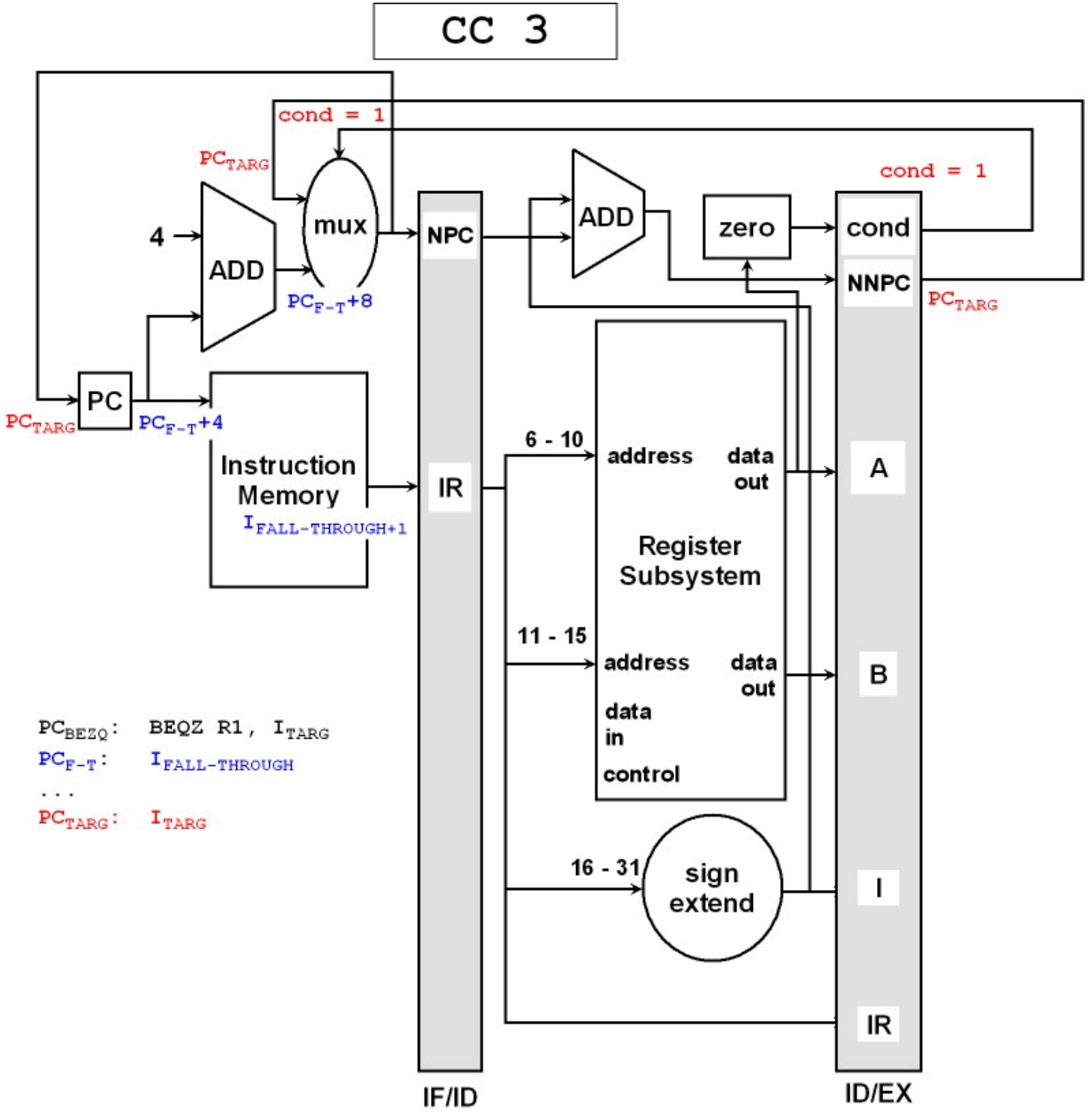
ID/EX latches

NPC+I

cond

PC "sees" PC_{TARG} = **PC+I**

Points to I_{TARG}



Version 6c Branch Processing — 4

CC3

PC

Receives special *CLK*

Latches $PC_{TARG} = PC+I$

ID fetches I_{TARG}

PC "sees"

$PC_{TARG+1} = PC_{TARG+1}+4$

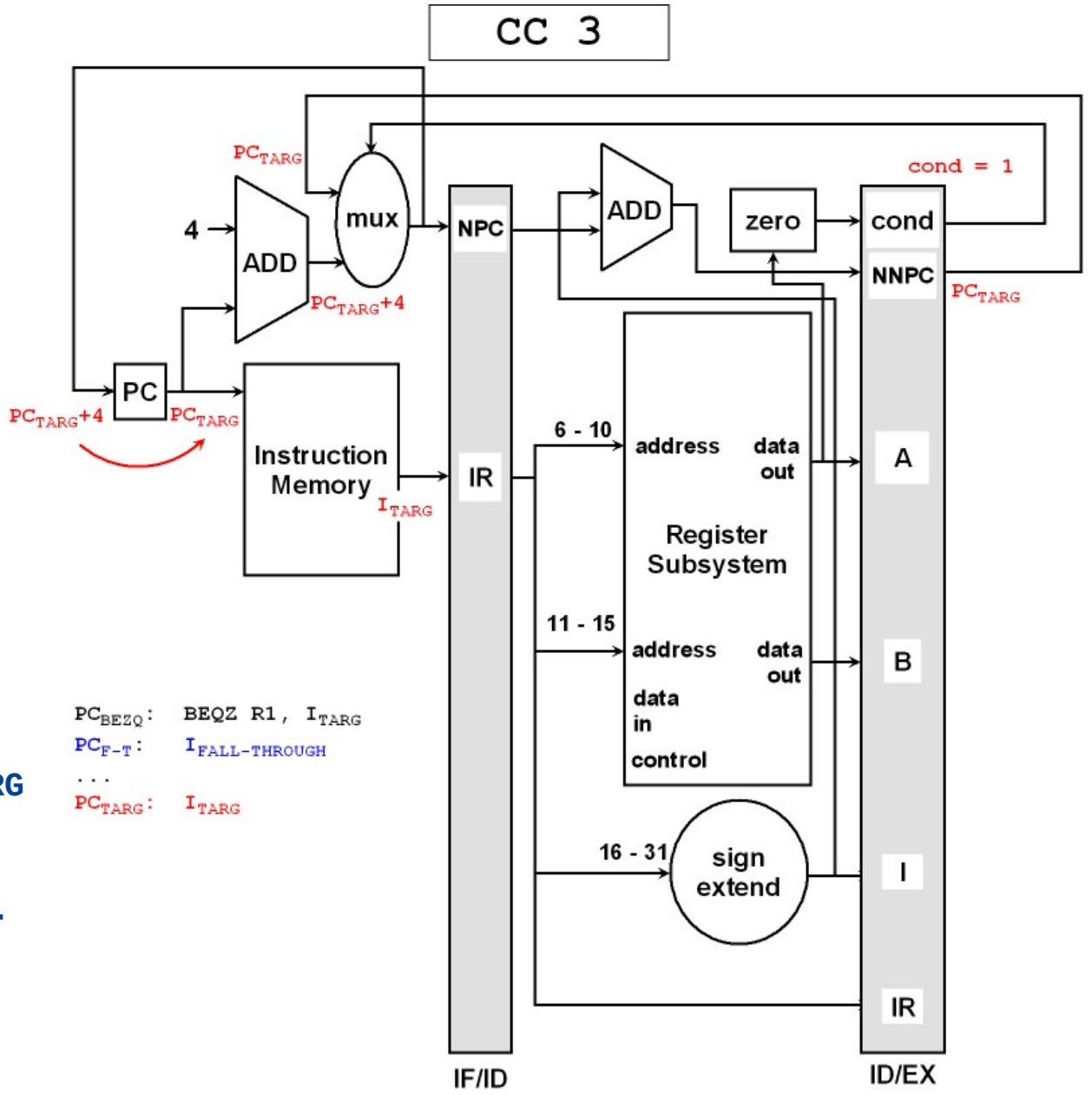
Points to I_{TARG+1}

On **CC4**

IF/ID.IR latches I_{TARG}

PC latches

$PC_{TARG+1} = PC_{TARG}+4$



Branch Performance of Version 6c

Method called Predict-Not-Taken

Branch taken — Flush entire pipeline

Branch not taken — Continue instruction in IF

Better performance on not taken (no pipeline stall)

Ideal method if most branches are not taken

Statistics from SPEC CINT

Not taken	33%
Taken	67%

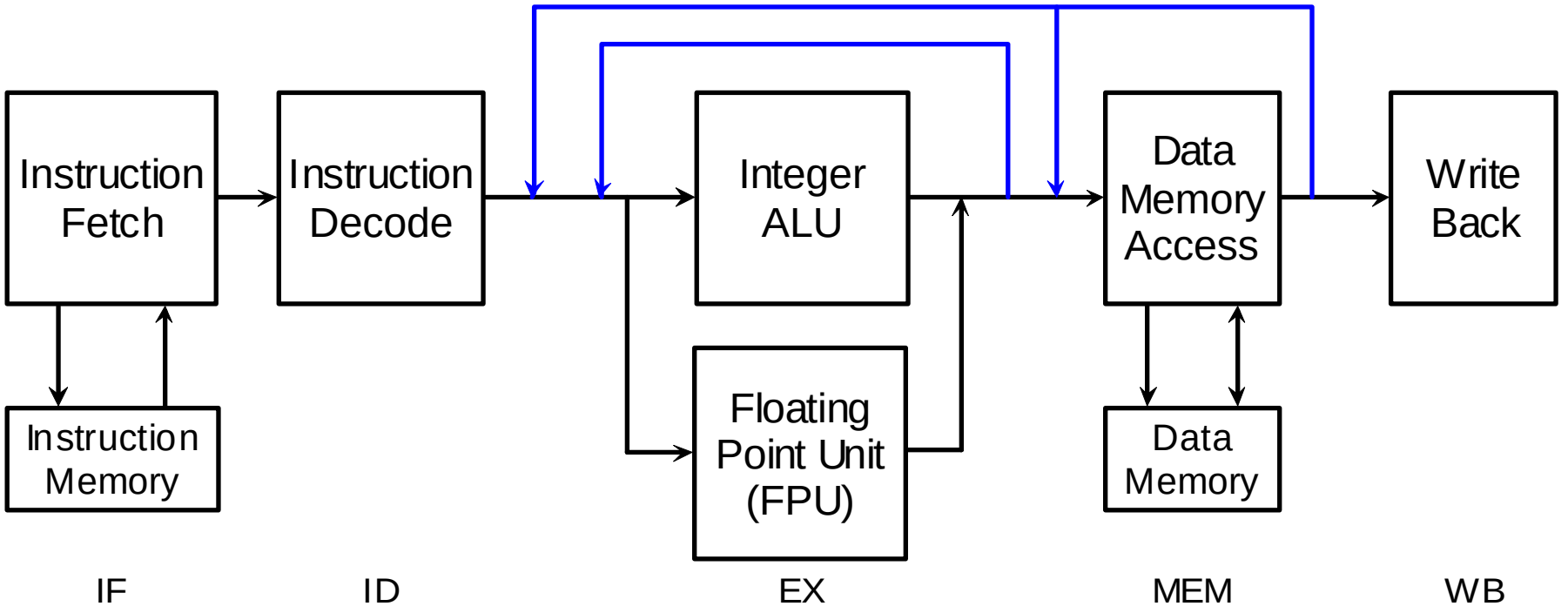
$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

$$= \frac{\text{stall cycles}}{\text{taken branch}} \times \frac{\text{taken branch}}{\text{branch instruction}} \times \frac{IC^{branch}}{IC}$$

$$= 1 \times 0.67 \times 0.20 \frac{\text{cycles}}{\text{instruction}} = 0.13 \frac{\text{cycles}}{\text{instruction}}$$

$CPI = 1.13 \Rightarrow$ (12% degradation)

DLXv6c Pipeline



Forwarding

DLXv6c

ALU result to ALU source

Memory load to ALU source (with 1 CC stall)

ALU result to memory store

Other dependencies

Require stall until Write-Back of intermediate result

DLXv6c Formal Specification (Integer Pipeline) — 1

Instruction Fetch (IF)

$$PC \leftarrow \begin{cases} PC + 4, & cond = 0 \\ ID/EX.NNPC, & cond = 1 \end{cases}$$

$$IF/ID.NPC \leftarrow \begin{cases} PC + 4, & cond = 0 \\ ID/EX.NNPC, & cond = 1 \end{cases}$$

$$IF/ID.IR \leftarrow Mem[PC]$$

Stage Buffers (←)
Sample and store inputs on falling CLK
"See" new inputs during clock cycle
(between falling CLKs)

Type	0-5	6-10	11-15	16-31	
R	op	rs1	rs2	rd	function
I	op	rs	rd	immediate	

Instruction Decode (ID)

$$ID/EX.A \leftarrow Reg[IF/ID.IR_{6\text{-}10}]$$

$$ID/EX.B \leftarrow Reg[IF/ID.IR_{11\text{-}15}]$$

$$ID/EX.I \leftarrow (IR_{16})^{16} \,\#\#\, IF/ID.IR_{16\text{-}31}$$

$$ID/EX.IR \leftarrow IF/ID.IR$$

$$ID/EX.NNPC \leftarrow IF/ID.NPC + (IR_{16})^{16} \,\#\#\, IF/ID.IR_{16\text{-}31}$$

$$ID/EX.cond \leftarrow (Reg[IF/ID.IR_{6\text{-}10}] == 0)$$

DLXv6c Formal Specification (Integer Pipeline) — 2

Execute (EX)

$$EX/MEM.ALU_{OUT} \leftarrow \begin{cases} ID/EX.A \text{ function } ID/EX.B & (R\text{-}ALU) \\ ID/EX.A \text{ op } ID/EX.I & (I\text{-}ALU, \text{Memory}) \end{cases}$$

Forwarding: $EX/MEM.ALU_{OUT}$ or $MEM/WB.ALU_{OUT}$ or $MEM/WB.LMD$ substituted for A or B

$$EX/MEM.B \leftarrow ID/EX.B$$

$$EX/MEM.IR \leftarrow ID/EX.IR$$

Type	0-5	6-10	11-15	16-31	
R	op	rs1	rs2	rd	function
I	op	rs	rd	immediate	

Memory (MEM)

$$MEM/WB.ALU_{OUT} \leftarrow EX/MEM.ALU_{OUT}$$

$$MEM/WB.LMD \leftarrow Mem[EX/MEM.ALU_{OUT}] \quad (Load)$$

$$Mem[EX/MEM.ALU_{OUT}] \leftarrow EX/MEM.B \quad (Store)$$

Fowarding: $MEM/WB.ALU_{OUT}$ substituted for B

$$MEM/WB.IR \leftarrow EX/MEM.IR$$

Write Back (WB)

$$Reg[MEM/WB.IR_{11\text{-}15}] \leftarrow \begin{cases} MEM/WB.ALU_{OUT} & (I\text{-}ALU) \\ MEM/WB.LMD & (Load) \end{cases}$$

$$Reg[MEM/WB.IR_{16\text{-}20}] \leftarrow MEM/WB.ALU_{OUT} \quad (R\text{-}ALU)$$

Forwarding ALU – ALU

	1	2	3	4	5	6	7	8	9
ADD R1, R2, R3	IF	ID	EX	MEM	WB				
ADD R4, R1, R5		IF	ID	EX	MEM	WB			
ADD R6, R4, R1			IF	ID	EX	MEM	WB		
ADD R7, R2, R1				IF	ID	EX	MEM	WB	

Forwarding Load – ALU

	1	2	3	4	5	6	7	8	9
LW R1, 8(R2)	IF	ID	EX	MEM	WB				
ADD R3, R1, R2		IF	ID	ID	EX	MEM	WB		
ADD R4, R3, R1			IF	IF	ID	EX	MEM	WB	

	1	2	3	4	5	6	7	8
LW R1, 8(R2)	IF	ID	EX	MEM	WB			
ADD R4, R4, R1		IF	ID	ID	EX	MEM	WB	
ADD R4, R4, R3			IF	IF	ID	EX	MEM	WB

	1	2	3	4	5	6	7	8
LW R1, 8(R2)	IF	ID	EX	MEM	WB			
ADD R4, R4, R3		IF	ID	EX	MEM	WB		
ADD R4, R4, R1			IF	ID	EX	MEM	WB	

Forwarding ALU - Store

	1	2	3	4	5	6	7	8	9
ADD R1, R3, R2	IF	ID	EX	MEM	WB				
SW 8(R2), R1		IF	ID	EX	MEM	WB			

	1	2	3	4	5	6	7	8	9
ADD R1, R3, R2	IF	ID	EX	MEM	WB				
ADD R4, R5, R6		IF	ID	EX	MEM	WB			
SW 8(R2), R1			IF	ID	ID	EX	MEM	WB	
SW 10(R4), R1				IF	IF	ID	EX	MEM	WB

ALU - Branch

	1	2	3	4	5	6	7	8	9
ADD R1, R3, R2	IF	ID	EX	MEM	WB				
BEQZ R1, targ		IF	ID	ID	ID	EX	MEM	WB	

	1	2	3	4	5	6	7	8	9
ADD R1, R3, R2	IF	ID	EX	MEM	WB				
ADD R4, R5, R6		IF	ID	EX	MEM	WB			
ADD R7, R8, R9			IF	ID	EX	MEM	WB		
BEQZ R1, targ				IF	ID	EX	MEM	WB	

Improvement by Re-Scheduling in DLXv6c

```
a[i] = a[i] + b[i] – c[i] + d[i]
```

```
a[] = 000 – 3FF
b[] = 400 – 7FF
c[] = 800 – BFF
d[] = C00 – FFF
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ADDI R1, R0, #400	F	D	X	M	W															
LW R2, -4(R1)		F	D	X	M	W						Forward R1								
LW R3, 3FC(R1)			F	D	X	M	W													
ADD R4, R2, R3				F	D	D	X	M	W				Forward R3							
LW R2, 7FC(R1)					F	F	D	X	M	W										
SUB R4, R4, R2							F	D	D	X	M	W		Forward R2						
LW R2, BFC(R1)								F	F	D	X	M	W							
ADD R4, R4, R2										F	D	D	X	M	W		Forward R2			
SW -4(R1), R4											F	F	D	X	M	W				
SUBI R1, R1, #4													F	D	X	M	W			
BNEZ R1, -40														F	D	D	D	X	M	W

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ADDI R1, R0, #400	F	D	X	M	W										
SUBI R1, R1, #4		F	D	X	M	W					Forward R1				
LW R2, 0(R1)			F	D	X	M	W								
LW R3, 400(R1)				F	D	X	M	W							
LW R5, 800(R1)					F	D	X	M	W						
LW R6, C00(R1)						F	D	X	M	W					
ADD R4, R2, R3							F	D	X	M	W				
SUB R4, R4, R5			Forward R4					F	D	X	M	W			
ADD R4, R4, R6									F	D	X	M	W		
SW 0(R1), R4										F	D	X	M	W	
BNEZ R1, FFD8											F	D	X	M	W

Improvement by Parallel Threads in DLXv6c

Source code

```
for ( i = 0 ; i < 100; i++ ){
      c[i] = a[i] + b[i];
      d[i] = a[i] - b[i];
}
```

Sequential code

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	ADDI R1, R0, #0	F	D	X	M	W													
	ADDI R2, R0, #400		F	D	X	M	W												
L1:	LW R3, 20000(R1)			F	D	X	M	W											
	LW R4, 20400(R1)				F	D	X	M	W										
	ADD R5, R3, R4					F	D	D	X	M	W								
	SW 20800(R1), R5						F	F	D	X	M	W							
	SUB R6, R3, R4								F	D	X	M	W						
	SW 21200(R1), R6									F	D	X	M	W					
	ADDI R1, R1, #4										F	D	X	M	W				
	SLT R7, R1, R2											F	D	X	M	W			
	BNEZ L1, R7												F	D	D	D	X	M	W
	J end																		

Stalls: LW $\rightarrow$ ADD = 1, SRT $\rightarrow$ BNEZ = 2, BNEZ $\rightarrow$ L1 = 1 (except on last)

$CPI_{stall} = 4/9 \longrightarrow CPI = 1 + 4/9 = 13/9$

Total CC for loop = 100 iterations $\times$ 9 instructions $\times$ 13/9 CC = 1300 CC

Total CC = 4 (CC1 – CC4) + 2 (ADDI, ADDI) + 1300 – 1 + 1 = 1306 CC

Parallel Threads By Data Decomposition

Split data between 2 threads

Thread 0

```
for ( i = 0 ; i < 50; i++ ){
        c[i] = a[i] + b[i];
        d[i] = a[i] - b[i];
}
```

Thread 1

```
for ( i = 50 ; i < 100; i++ ){
        c[i] = a[i] + b[i];
        d[i] = a[i] - b[i];
}
```

Thread 0

```
        ADDI R1, R0, #0
        ADDI R2, R0, #200
L1:     LW R3, 20000(R1)
        LW R4, 20400(R1)
        ADD R5, R3, R4
        SW 20800(R1), R5
        SUB R6, R3, R4
        SW 21200(R1), R6
        ADDI R1, R1, #4
        SLT R7, R1, R2
        BNEZ L1, R7
        J end
```

9 instructions per loop

Thread 1

```
        ADDI R1, R0, #200
        ADDI R2, R0, #400
L1:     LW R3, 20000(R1)
        LW R4, 20400(R1)
        ADD R5, R3, R4
        SW 20800(R1), R5
        SUB R6, R3, R4
        SW 21200(R1), R6
        ADDI R1, R1, #4
        SLT R7, R1, R2
        BNEZ L1, R7
        J end
```

Same stalls and same CPI as sequential code

Total CC for loop = 50 iterations $\times$ 9 instructions $\times$ 13/9 CC = 650

Total CC = 4 + 2 + 650 – 1 + 1 = 656 $\longrightarrow$ S = 1306 / 656 = 1.99

Parallel Threads By Functional Decomposition

Split functions between 2 threads

Thread 0

```
for ( i = 0 ; i < 100; i++ ){
        c[i] = a[i] + b[i];
}
```

Thread 1

```
for ( i = 0 ; i < 100; i++ ){
        d[i] = a[i] - b[i];
}
```

Thread 0

```
      ADDI R1, R0, #0
      ADDI R2, R0, #400
L1:   LW R3, 20000(R1)
      LW R4, 20400(R1)
      ADD R5, R3, R4
      SW 20800(R1), R5
      ADDI R1, R1, #4
      SLT R7, R1, R2
      BNEZ L1, R7
      J end
```

7 instructions per loop

Thread 1

```
      ADDI R1, R0, #0
      ADDI R2, R0, #400
L1:   LW R3, 20000(R1)
      LW R4, 20400(R1)
      SUB R6, R3, R4
      SW 21200(R1), R6
      ADDI R1, R1, #4
      SLT R7, R1, R2
      BNEZ L1, R7
      J end
```

Same stalls as sequential code

CPI = 1 + 4/7 = 11/7

Total CC for loop = 100 iterations × 7 instructions × 11/7 CC = 1100

Total CC = 4 + 2 + 1100 – 1 + 1 = 1106 ⟶ S = 1306 / 1106 = 1.18

General Branch Prediction

Branch statistics from SPEC CINT

- Branch not taken 33%
- Branch taken 67%
- Most branch instructions
  - Used to build loops
  - Run more than once

Branch prediction

- Advanced technique
- Not implemented in DLX model
- Used in modern RISC processors and Intel x86 since Pentium

Branch predictor

- Records statistics on branch instructions
  - Source address, target address, taken/not-taken
- Predicts branch behavior based on previous behavior

Branch Prediction for DLX Pipeline

1. Branch predictor in IF stage

Identifies branch instruction

According to source address

Predicts branch from branch history

Taken

Predicts branch target address

Not-taken

Uses fall-through address

2. Validate branch instruction in ID stage

Usual Calculation:

Target address

Condition flag — taken or not-taken

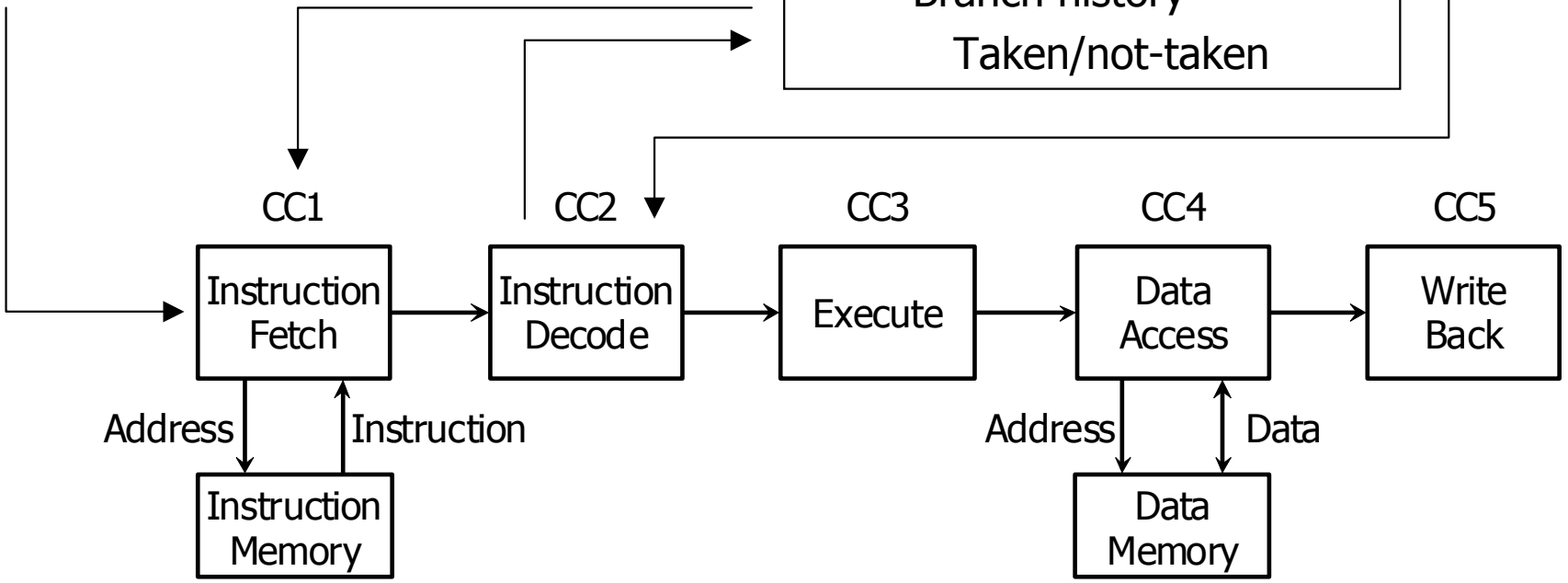
3. After validation

Update branch predictor

Target address

Branch history

Taken/not-taken



Branch Prediction Performance

Branch taken — first execution

		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_2	Fall-Through		IF	ID	EX	MEM	WB			
I_3				IF						
...	…	…	…	…	…	…	…	…	…	…
I_T	Target			IF	ID	EX	MEM	WB		

Misprediction

Branch taken — second execution

		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_T	Target		IF	ID	EX	MEM	WB			
I_{T+1}	Target+1			IF	ID	EX	MEM	WB		
I_{T+2}	Target+2				IF	ID	EX	MEM	WB	

Correct prediction

Branch Prediction Performance for Simple Loop

Simple static loop

```
        ADDI R1, R0, #N    ; N iterations
L1:     ALU Block
        SUBI R1, R1, #1    ; B lines of code
        BNEZ R1, L1
        I fall-through
```

$$CPI_{branch}^{stall} = \frac{2}{N \times B + 2} \xrightarrow[N \times B \to \text{large}]{} 0$$

	ADDI R1, R0, # N	IF	ID	EX	MEM	WB		
L1:	ALU Block		IF	ID	EX	MEM	WB	
	< B-2 lines of ALU code >							
	BNEZ R1, L1	IF	ID	EX	MEM	WB		
	$I_{fall-through}$		IF	ID	ϕ	ϕ	ϕ	
L1:	ALU Block			IF	ID	EX	MEM	WB
	< B-2 lines of ALU code >							
	BNEZ R1, L1	IF	ID	EX	MEM	WB		
L1:	ALU Block		IF	ID	EX	MEM	WB	
	...							
	< B-2 lines of ALU code >							
	BNEZ R1, L1	IF	ID	EX	MEM	WB		
L1:	ALU Block		IF	ID	ϕ	ϕ	ϕ	
	$I_{fall-through}$			IF	ID	EX	MEM	WB

R1 = N - 1

R1 = N - 2

R1 = 0

More Compiler Optimizations — 1

Common sub-expression elimination

Compiler encounters instructions

```
B = 10*(A/3);
C = (A/3)/4;
```

Calculates `(A/3)` into register

Uses register in later calculations

First-pass compilation

```
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#10
MULT R1,R1,R2
SW B,R1
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#4
DIV R1,R1,R2
SW C,R1
```


Second-pass compilation

```
LW R1,A
ADDI R2,R0,#3
DIV R1,R1,R2
ADDI R2,R0,#10
MULT R3,R1,R2
SW B,R3
ADDI R2,R0,#4
DIV R3,R1,R2
SW C,R3
```

More Compiler Optimizations — 2

Loop unrolling

Instead of loop compiler replicates instructions

Eliminates overhead of testing loop control variable

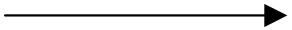
Inlining

Procedure call replaced by code of procedure or macro

First-pass compilation

```
00   ADDI R2,R0,#0x05
04   ADDI R1,R0,#0x08
08   LW R3,0x1000(R1)
0C   JAL 10
10   SW 2000(R1),R3
14   SUBI R1,R1,#0x04
18   BNEZ R1,-0x14
1C   ADDI R2,R0,#3
20   ADD R3,R3,R2
24   JR R31
```

Second-pass compilation



```
00   ADDI R2,R0,#0x05
04   LW R3,0x1008(R0)
08   ADD R3,R3,R2
0C   SW 2008(R0),R3
10   LW R3,0x1004(R0)
14   ADD R3,R3,R2
18   SW 2004(R0),R3
1C   ADDI R2,R0,#3
```

More Hardware Optimizations

Superscaling

Run 2 or more pipelines in parallel

Instructions without dependencies execute in parallel

Used in most RISC processors and Pentium 1 – 4, Centrino, Core

Dynamic Scheduling

Processor performs dynamic instruction scheduling

Same result as compiler scheduling

Very efficient when combined with superscaling

Used in IBM mainframes since 1967

Used in Pentium II – 4, Centrino, and Core processors

Register Aliasing

Tasks require logical registers (R0, R1, … as defined in ISA)

Physical registers allocated per task from large register pool

Multiple tasks use same logical register in parallel

Instruction Predication

Usual test-and-set instructions (SLT, SGT, SEQ, …) set predication flags

Instruction can be run or cancelled according to a predicate flag