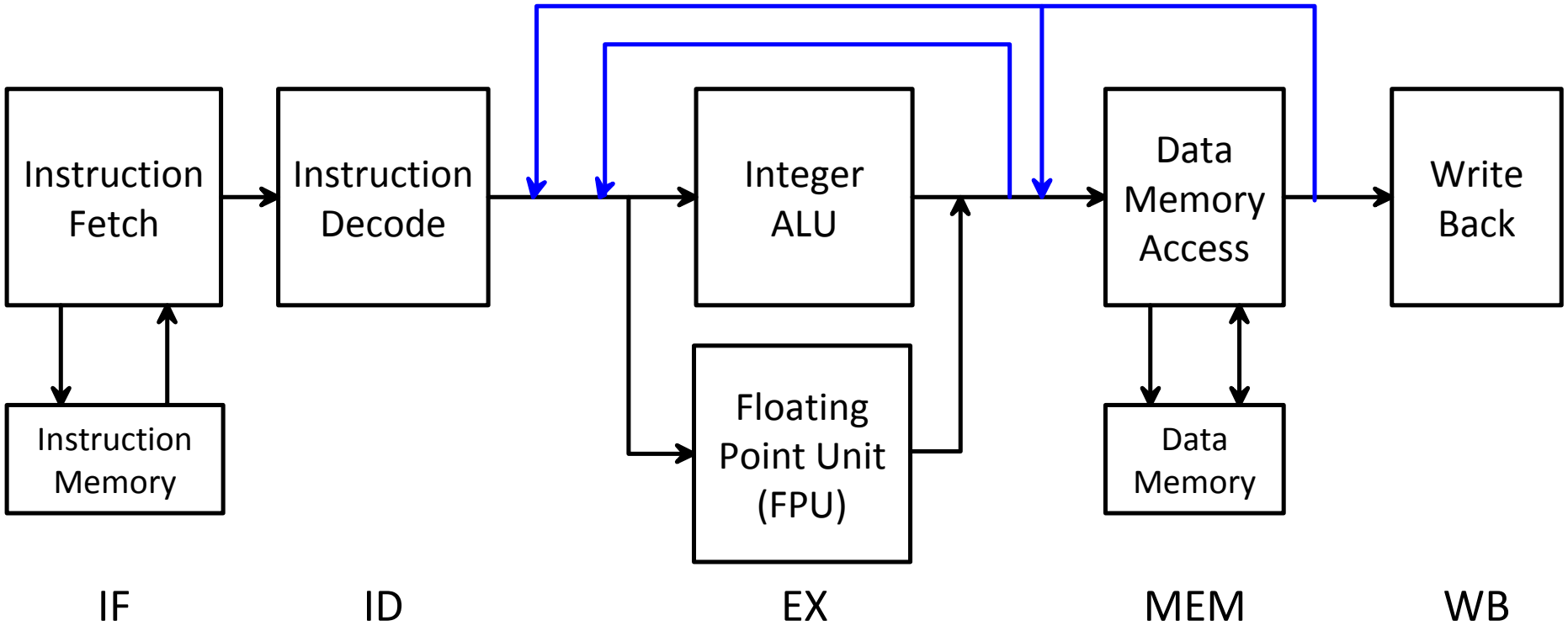
DLX: A Simplified RISC Model

DLX Pipeline



Hardware definition based on MIPS 2000 commercial microprocessor

32-bit machine — address, integer, register width, instruction length

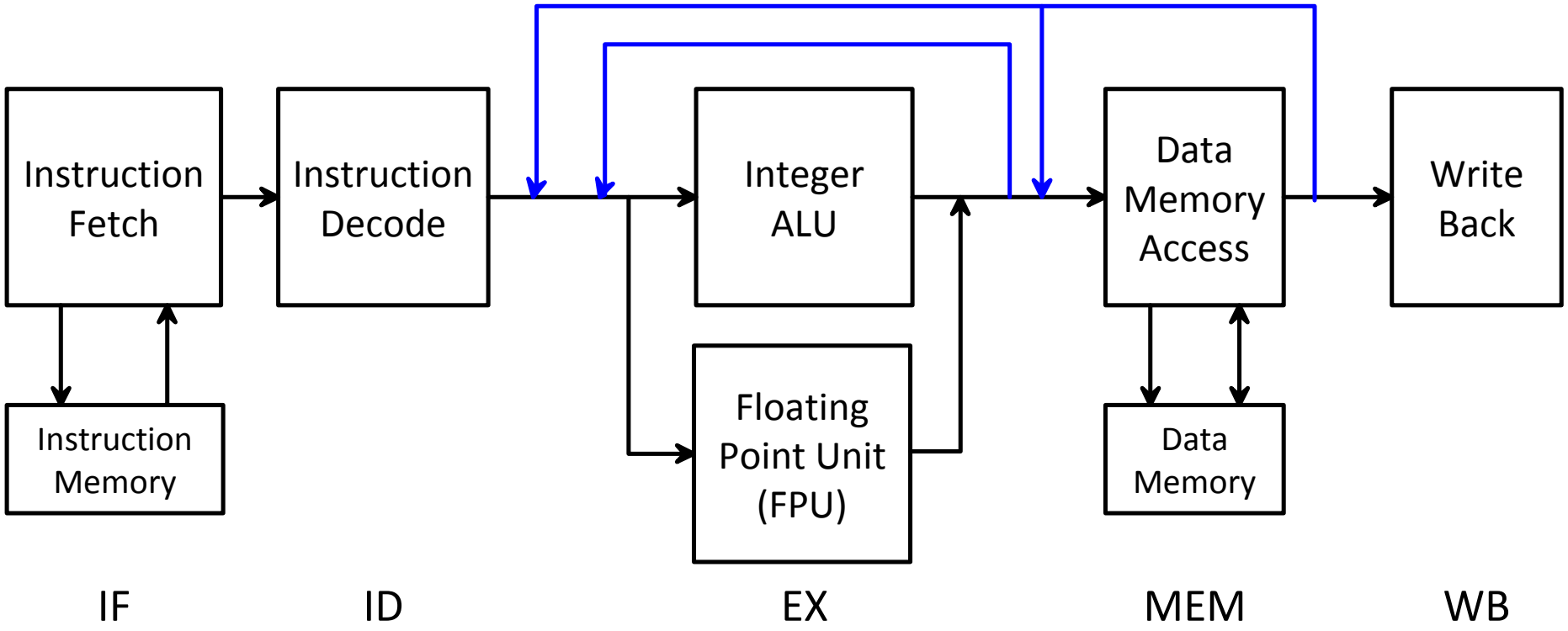
32 integer registers R0, R1, ... , R31

Regs[R0] = 0 (read-only)

32 FP registers F0, F1, ... , F31

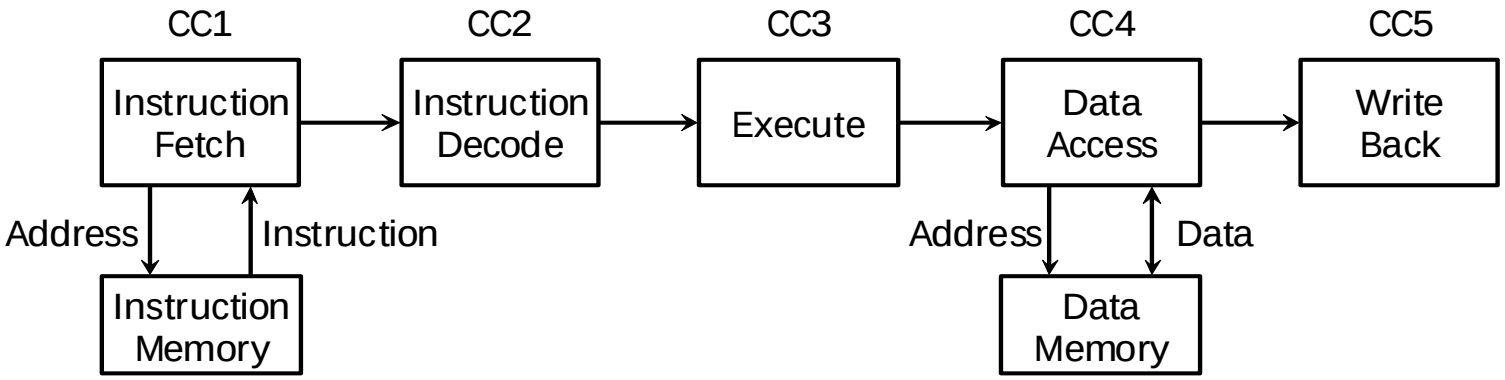
Reference: Hennessey and Patterson, 2nd ed, chapter 2

DLX Pipeline Stages



IF	ID	EX	MEM	WB
Fetch next instruction Update program counter	Prepare source operands Evaluate branches (condition + target address)	Perform ALU/FPU operations Calculate data memory addresses	Data memory access (load / store)	Update registers with ALU / load results

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

DLX Instruction Formats

Three 32-bit instruction formats

J-type — absolute branch (jump) instructions

`PC ← PC + OFFSET`

R-type — register-register ALU instructions

`rd ← ALU_function (rs1, rs2)`

I-type — all other instructions

Load: **`rd ← imm(rs)`**

Store: **`imm(rs) ← rd`**

ALU: **`rd ← ALU_operation (rs, imm)`**

Branch: **`if (rs == 0)`** {**`PC ← PC + imm`**}

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

Data Transfer Instructions

LW R1, 30(R2)	Load Word	$Reg[R1] \leftarrow_{32} Mem[30 + Reg[R2]]$
SW 30(R2), R1	Store Word	$Mem[30 + Reg[R2]] \leftarrow_{32} Reg[R1]$
LB R1, 30(R2)	Load Byte	$Reg[R1] \leftarrow_{32} (Mem[30 + Reg[R2]]_0)^{24}$ ## $Mem[30 + Reg[R2]]$
SB 30(R2), R1	Store Byte	$Mem[30 + Reg[R2]] \leftarrow_{8} Reg[R1]_{24..31}$
LBU R1, 30(R2)	Load Byte unsigned	$Reg[R1] \leftarrow_{32} 0^{24}$ ## $Mem[30 + Reg[R2]]$
LH R1, 30(R2)	Load Half Word	$Reg[R1] \leftarrow_{32} (Mem[30 + Reg[R2]]_0)^{16}$ ## $Mem[30 + Reg[R2]]$
LF F1, 30(R2)	Load Float	$Reg[F1] \leftarrow_{32} Mem[30 + Reg[R2]]$
SF 30(R2), F1	Store Float	$Mem[30 + Reg[R2]] \leftarrow_{32} Reg[F1]$
MOVF F3, F1	Move Float	$Reg[F3] \leftarrow_{32} Reg[F1]$
MOVD F2, F0	Move Double	$Reg[F3],Reg[F2] \leftarrow_{64} Reg[F1],Reg[F0]$
MOVFP2I R2, F2	FP to INT	$Reg[R2] \leftarrow_{32} Reg[F2]$
MOVI2FP F2, R2	FP to INT	$Reg[F2] \leftarrow_{32} Reg[R2]$

Arithmetic/Logic Instructions

ADD R1, R2, R3	Add	Reg[R1] ← Reg[R2] + Reg[R3]
ADDI R1, R2, #3	Add Immediate	Reg[R1] ← Reg[R2] + 3
SUB R1, R2, R3	Sub	Reg[R1] ← Reg[R2] - Reg[R3]
SUBI R1, R2, #3	Sub Immediate	Reg[R1] ← Reg[R2] - 3
MULT R1, R2, R3	Multiply	Reg[R1] ← Reg[R2] * Reg[R3]
DIV R1, R2, R3	Div	Reg[R1] ← Reg[R2] ÷ Reg[R3]
AND R1, R2, R3	And	Reg[R1] ← Reg[R2] AND Reg[R3]
ANDI R1, R2, #3	And Immediate	Reg[R1] ← Reg[R2] AND 3
OR R1, R2, R3	Or	Reg[R1] ← Reg[R2] OR Reg[R3]
ORI R1, R2, #3	Or Immediate	Reg[R1] ← Reg[R2] OR 3
XOR R1, R2, R3	Exclusive Or	Reg[R1] ← Reg[R2] XOR Reg[R3]
XORI R1, R2, #3	Exclusive Or Immediate	Reg[R1] ← Reg[R2] XOR 3
LHI R1, #42	Load High	Reg[R1] ← 42 ## 0^{16}
SLT R1, R2, R3	Set Less Than	if Reg[R2] < Reg[R3] then Reg[R1] ← 1 else Reg[R1] ← 0
SGT R1, R2, R3	Set Greater Than	if Reg[R2] > Reg[R3] then Reg[R1] ← 1 else Reg[R1] ← 0
SLE R1, R2, R3	Set Less Than or Equal	if Reg[R2] ≤ Reg[R3] then Reg[R1] ← 1 else Reg[R1] ← 0
SGE R1, R2, R3	Set Greater Than or Equal	if Reg[R2] ≥ Reg[R3] then Reg[R1] ← 1 else Reg[R1] ← 0
SEQ R1, R2, R3	Set Equal	if Reg[R2] = Reg[R3] then Reg[R1] ← 1 else Reg[R1] ← 0
SNE R1, R2, R3	Set Not Equal	if Reg[R2] ≠ Reg[R3] then Reg[R1] ← 1 else Reg[R1] ← 0

Floating Point Instructions

ADDF F1, F2, F3	Add Float	Reg[F1] ← Reg[F2] + Reg[F3]
ADDD F0, F2, F4	Add Double	$\begin{pmatrix}\text{Reg[F1]}\\\text{Reg[F0]}\end{pmatrix} \xleftarrow{64} \begin{pmatrix}\text{Reg[F3]}\\\text{Reg[F2]}\end{pmatrix} + \begin{pmatrix}\text{Reg[F5]}\\\text{Reg[F4]}\end{pmatrix}$
SUBF F1, F2, F3	Sub Float	NOTE: Floating point numbers are represented as single or double precision numbers according to IEEE 754. The ALU functions for FP are not simple binary operations on the bits in the register.
SUBD F0, F2, F4	Sub Double	
MULTF F1, F2, F3	Multiply Float	
MULTD F0, F2, F4	Multiply Double	
DIV F1, F2, F3	Divide Float	
DIVD F0, F0, F4	Divide Double	
LTF F2, F3	Set Less Than	if Reg[F2] < Reg[F3] then StatFP $\leftarrow_1$ 1 else StatFP $\leftarrow_1$ 0
GTF F2, F3	Set Greater Than	if Reg[F2] > Reg[F3] then StatFP $\leftarrow_1$ 1 else StatFP $\leftarrow_1$ 0
LEF F2, F3	Set Less Than or Equal	if Reg[F2] ≤ Reg[F3] then StatFP $\leftarrow_1$ 1 else StatFP $\leftarrow_1$ 0
GEF F2, F3	Set Greater Than or Equal	if Reg[F2] ≥ Reg[F3] then StatFP $\leftarrow_1$ 1 else StatFP $\leftarrow_1$ 0
EQF F2, F3	Set Equal	if Reg[F2] = Reg[F3] then StatFP $\leftarrow_1$ 1 else StatFP $\leftarrow_1$ 0
NEF F2, F3	Set Not Equal	if Reg[F2] ≠ Reg[F3] then StatFP $\leftarrow_1$ 1 else StatFP $\leftarrow_1$ 0
LTD, GTD, LED, GED, EQD, NED		Double precision comparisons

Control Instructions

J offset	**Jump**	**PC ← PC + offset** **($-2^{25} \le$ offset $\le 2^{25} - 1$)**
JAL offset	**Jump and Link**	**Reg[R31] ← PC** **PC ← PC + offset** **($-2^{25} \le$ offset $\le 2^{25} - 1$)**
JR R3	**Jump Register**	**PC ← Reg[R3]**
JALR R2, offset	**Jump and Link Register**	**Reg[R2] ← PC** **PC ← PC + offset** **($-2^{15} \le$ offset $\le 2^{15} - 1$)**
BEQZ R4, offset	**Branch equal zero**	**if Reg[R4] == 0 then PC ← PC + offset** **($-2^{15} \le$ offset $\le 2^{15} - 1$)**
BNEZ R4, offset	**Branch not equal zero**	**if Reg[R4] != 0 then PC ← PC + offset** **($-2^{15} \le$ offset $\le 2^{15} - 1$)**
TRAP N	**Software interrupt**	**Details not specified in Hennessy and Patterson**

Note: Register **NPC** is updated (**NPC ← PC + 4**) when branch instruction is loaded
Register **PC** is updated (**PC ← NPC** or **PC ← NPC + offset**) at end of instruction execution

Programming in DLX Assembly

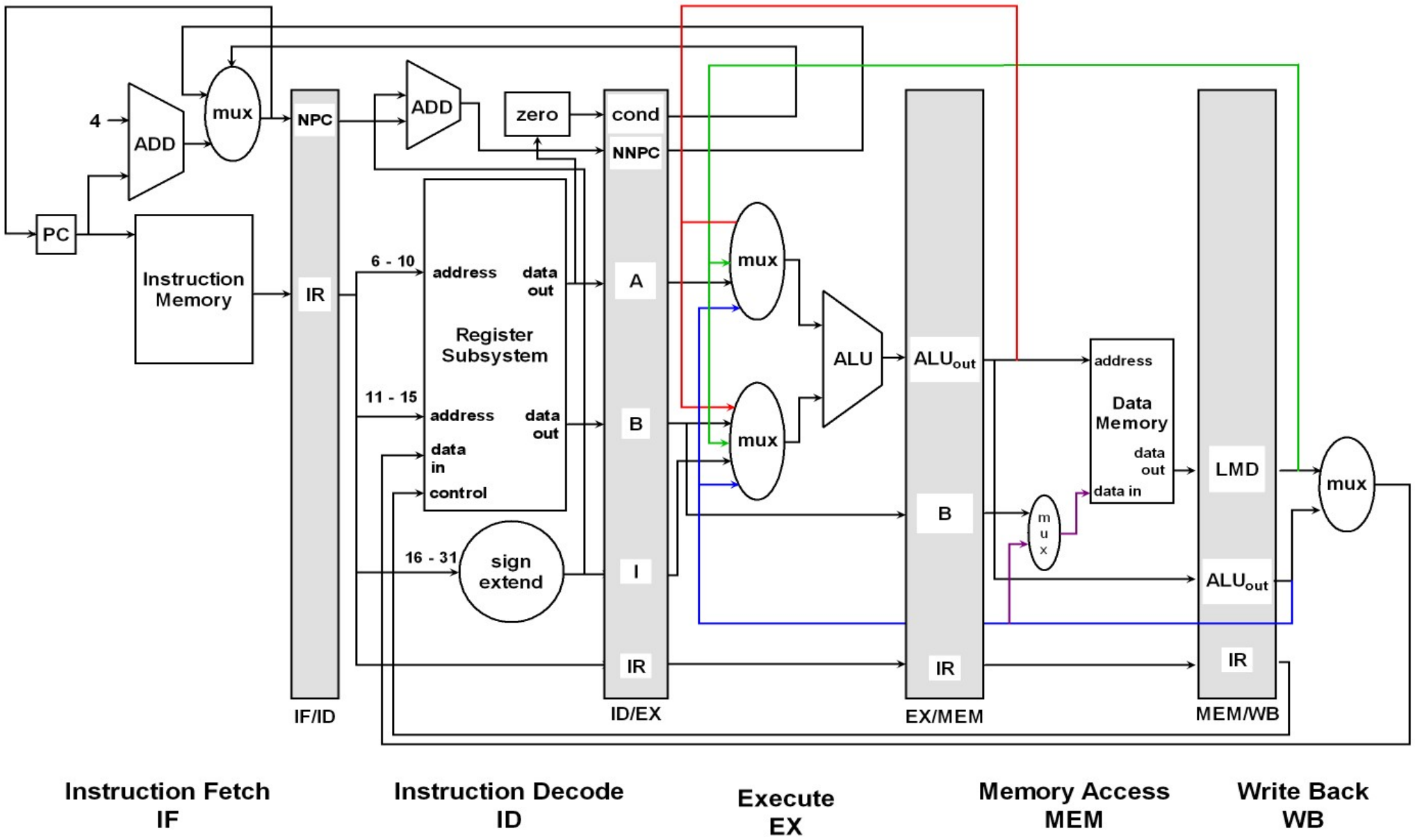
C program

```
main()
{
    int i,j;
    for (i = 0; i < 10; i++){
        j = 2 * i;
    }
}
```

DLX version

```
        ADDI R1, R0, #0     ; i = R1 <-- 0
        ADDI R10, R0, #0A   ; R10 <-- 10
start:  SGE R11, R1, R10    ; R11 <-- 1 iff R1 >= R10 = 10
        BNEZ R11, stop      ; jump to stop if R1 >= 10
        ADD R2, R1, R1      ; R2 <-- R1 * 2
        ADDI R1, R1, #1     ; R1++
        J start             ; jump to start
stop:   SW -2(R13), R2      ; store j <-- R2
                            ; R13 = base pointer for variables
        JR R31              ; return to calling function
```

DLX Implementation (Integer Pipeline)



Temporary Registers in DLX Implementation

5 stage buffers — IF/ID, ID/EX, EX/MEM, MEM/WB, PC

Store and forward instruction states between 5 stages

Update on falling edge of system clock

PC

Program Counter

Memory address of next instruction

IR

Instruction Register

Holds fetched instruction during execution

NPC

Next Program Counter

Temporary update of PC (points to fall-through instruction)

A, B, I

Operand buffers

Values read from data registers

ALU_{out}

ALU output

Result of ALU operation

LMD

Load Memory Data

Data loaded from memory

Cond

Condition flag

Result of test for conditional branch

DLX 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)

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

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

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

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

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

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

DLX Formal Specification (Integer Pipeline) — 2

Execute (EX)

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

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)

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

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

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

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

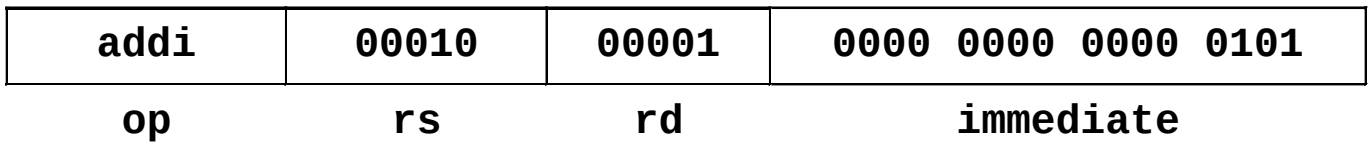
$$MEM/WB.IR \leftarrow EX/MEM.IR$$

Write Back (WB)

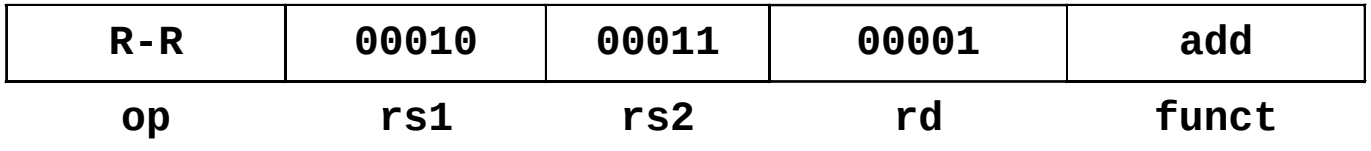
$$Reg[MEM/WB.IR_{11\text{-}15}] \leftarrow \begin{cases} MEM/WB.ALU_{OUT} & (ALU\text{-}I) \\ MEM/WB.LMD & (Load) \end{cases}$$

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

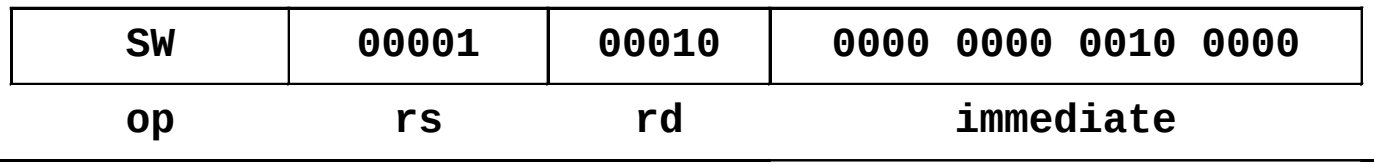
Example Type-I ALU Instruction

Instruction	`addi R1, R2, #5`
Operation	$Reg[R1] \leftarrow Reg[R2] + 5$
Encoding	
Hardware Stage 1	$IR \leftarrow Mem[PC]$ $NPC \leftarrow PC + 4$
Hardware Stage 2	$A \leftarrow Reg[IR_{6\text{-}10}]$ /* A ← Reg[R2] */ $B \leftarrow Reg[IR_{11\text{-}15}]$ /* B ← Reg[R1] */ $I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$ if (A == 0) cond = 1 else cond = 0 $NNPC \leftarrow NPC + I$
Hardware Stage 3	$ALU_{out} \leftarrow A + I$
Hardware Stage 4	
Hardware Stage 5	$Reg[IR_{11\text{-}15}] \leftarrow ALU_{out}$ /* Reg[R1] ← A + I */ $PC \leftarrow NPC$

Example Type-R ALU Instruction

Instruction	`add R1, R2, R3`
Operation	$Reg[R1] \leftarrow Reg[R2] + Reg[R3]$
Encoding	
Hardware Stage 1	$IR \leftarrow Mem[PC]$ $NPC \leftarrow PC + 4$
Hardware Stage 2	$A \leftarrow Reg[IR_{6\text{-}10}]$ /* A ← Reg[R2] */ $B \leftarrow Reg[IR_{11\text{-}15}]$ /* B ← Reg[R3] */ $I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$ `if (A == 0) cond = 1 else cond = 0` $NNPC \leftarrow NPC + I$
Hardware Stage 3	$ALU_{out} \leftarrow A + B$
Hardware Stage 4	
Hardware Stage 5	$Reg[IR_{16\text{-}20}] \leftarrow ALU_{out}$ /* Reg[R1] ← A + B */ $PC \leftarrow NPC$

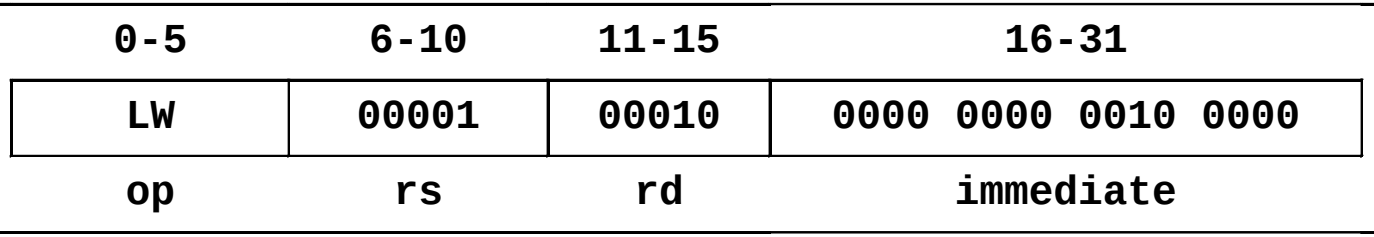
Example Type-I Store Instruction

Instruction	`SW 32(R1), R2`
Operation	`Mem[32+Reg[R1]] ← Reg[R2]`
Encoding	
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`

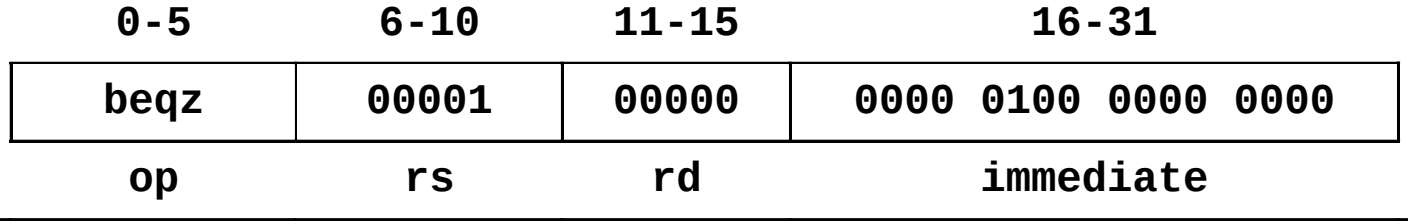
Hardware Stage 2	$A \leftarrow Reg[IR_{6\text{-}10}]$ /* A ← Reg[R1] */
	$B \leftarrow Reg[IR_{11\text{-}15}]$ /* B ← Reg[R2] */
	$I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$
	`if (A == 0) cond = 1 else cond = 0`
	`NNPC ← NPC + I`

Hardware Stage 3	$ALU_{out} \leftarrow A + I$
Hardware Stage 4	$Mem[ALU_{out}] \leftarrow B$ /* Mem[A+I] ← Reg[R2] */ `PC ← NPC`
Hardware Stage 5	

Example Type-I Load Instruction

Instruction	`LW R2, 32(R1)`
Operation	$Reg[R2] \leftarrow Mem[32+Reg[R1]]$
Encoding	
Hardware Stage 1	$IR \leftarrow Mem[PC]$ $NPC \leftarrow PC + 4$
Hardware Stage 2	$A \leftarrow Reg[IR_{6-10}]$ /* $A \leftarrow Reg[R1]$ */ $B \leftarrow Reg[IR_{11-15}]$ /* $B \leftarrow Reg[R2]$ */ $I \leftarrow (IR_{16})^{16}$ ## IR_{16-31} `if (A == 0) cond = 1 else cond = 0` $NNPC \leftarrow NPC + I$
Hardware Stage 3	$ALU_{out} \leftarrow A + I$
Hardware Stage 4	$LMD \leftarrow Mem[ALU_{out}]$ /* $LMD \leftarrow Mem[A+I]$ */
Hardware Stage 5	$Reg[IR_{11-15}] \leftarrow LMD$ /* $Reg[R2] \leftarrow Mem[A+I]$ */ $PC \leftarrow NPC$

Example Type-I Conditional Branch Instruction

Instruction	`beqz R1, 1024`
Operation	`if (Reg[R1] == 0) PC ← NPC + 1024` `else PC ← NPC`
Encoding	
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`

Hardware Stage 2	$A \leftarrow Reg[IR_{6\text{-}10}]$ /* A ← Reg[R1] */
	$B \leftarrow Reg[IR_{11\text{-}15}]$ /* B ← Reg[R0] */
	$I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$
	`if (A == 0) cond = 1 else cond = 0`
	`NNPC ← NPC + I`

Hardware Stage 3	`if (cond == 1) PC ←` ALU_{out} `else PC ← NPC`
Hardware Stage 4	
Hardware Stage 5	

DLX Integer Pipeline Statistics

Instruction distribution

Compile SPEC CINT → DLX instruction set

Sort object code into 4 groups

ALU	**40 %**
Load	**25 %**
Store	**15 %**
Branch	**20 %**

Register dependencies

ALU instruction I_N

Destination operand ← ALU operation(source operands)

In 50% of ALU instructions

1 source operand = destination operand of instruction I_{N-1}

I_{N-1} = ALU or load

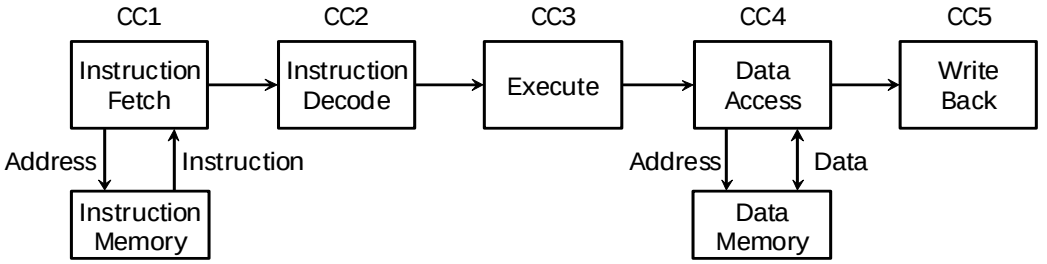
Data Hazards in DLX Integer Pipeline

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

ALU-ALU RAW Dependencies

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**

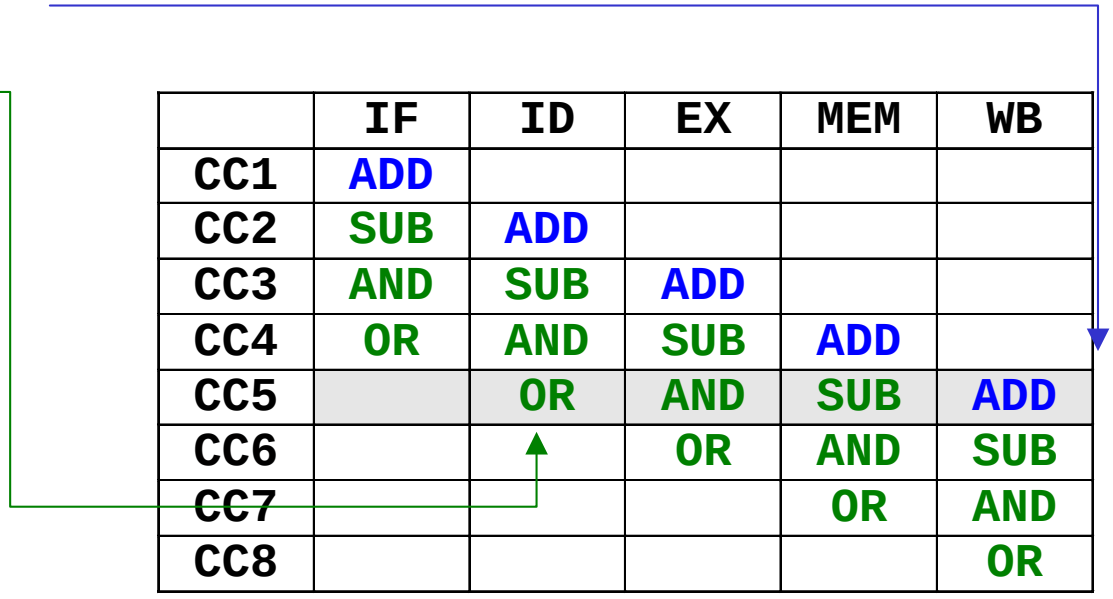
Bad timing (uncorrected)

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

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

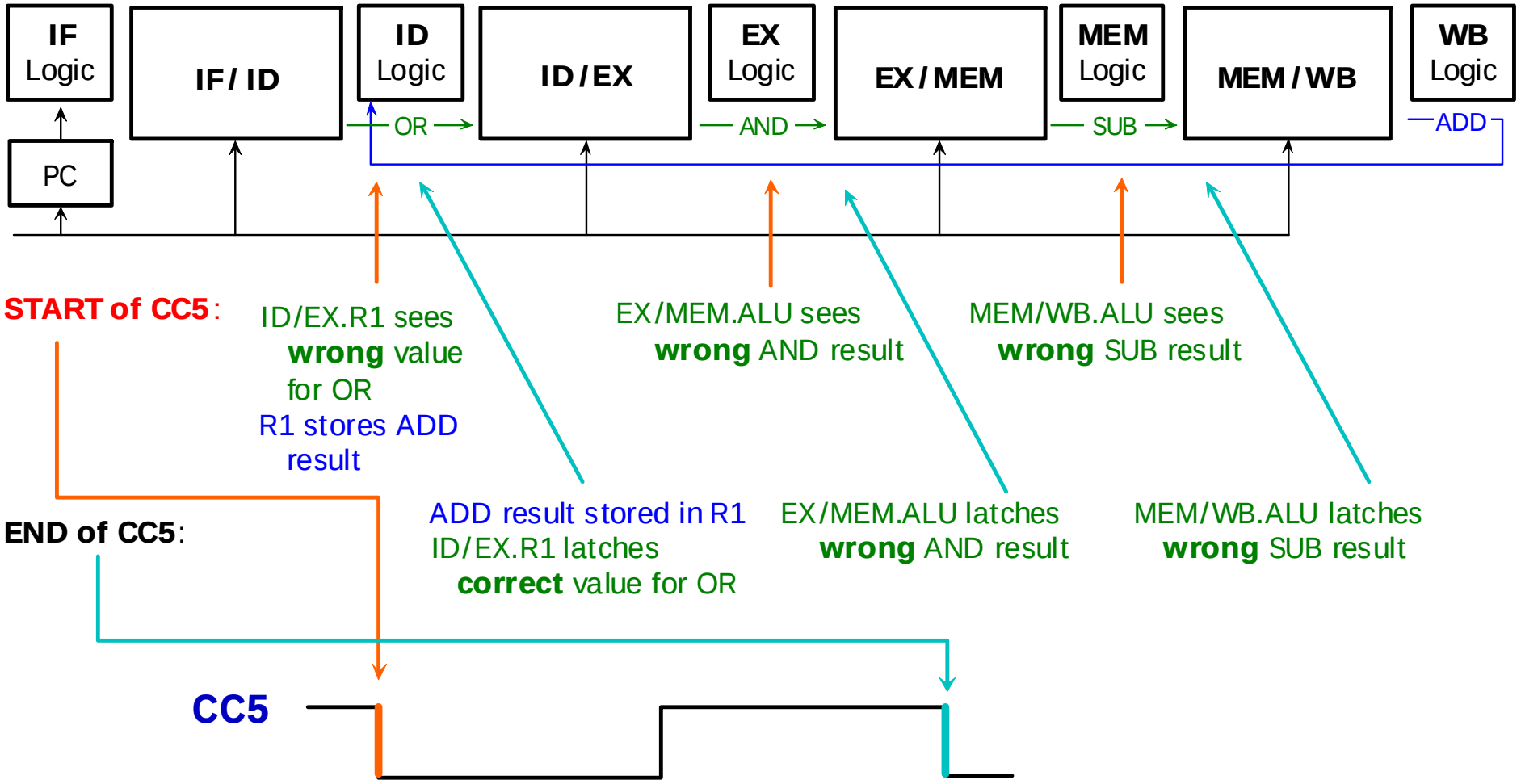
I_3 reads **R1** in **ID** during **CC4**

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)



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

	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	SUB	ϕ	ADD		
CC4	SUB	ϕ	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

Wait states

IF/ID freezes internal state on **SUB** for **CC3** and **CC4**

IF/ID passes ϕ (**NOP** — no operation) 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

Clock Cycle →

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

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

Performance degradation too large

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

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

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

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

Forwarding or Bypass

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

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

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

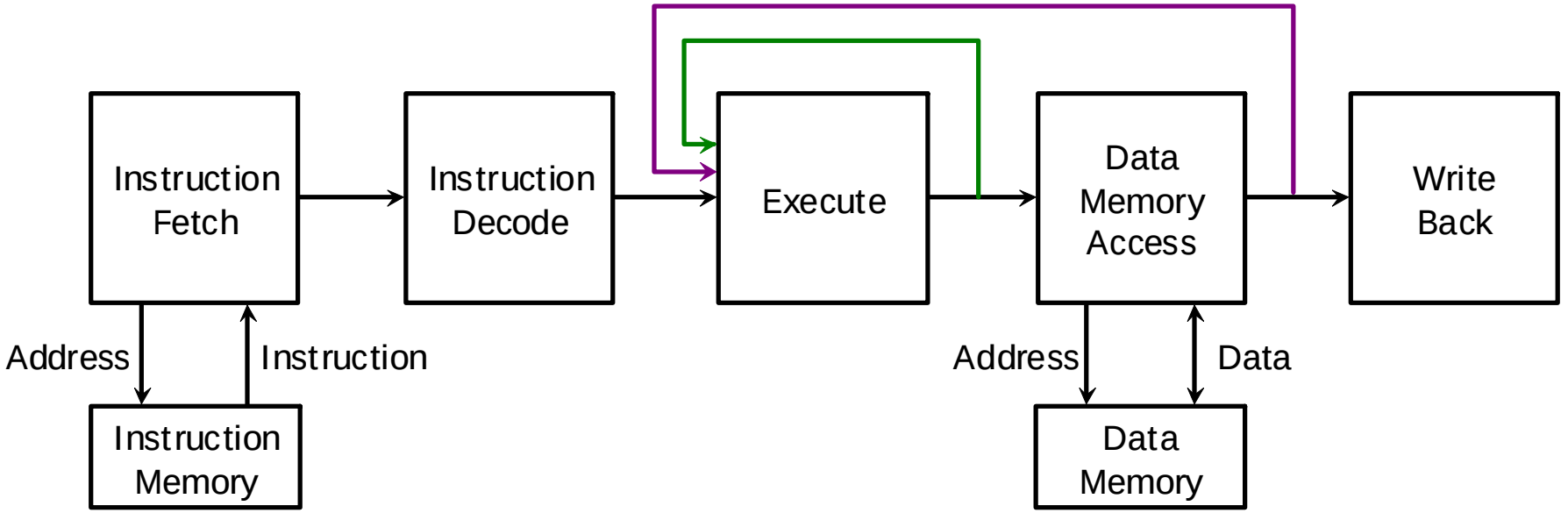
	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

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**



Forwarding in Instruction View

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

$$CPI^{stall} = 0$$

Load-ALU RAW Dependencies

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**

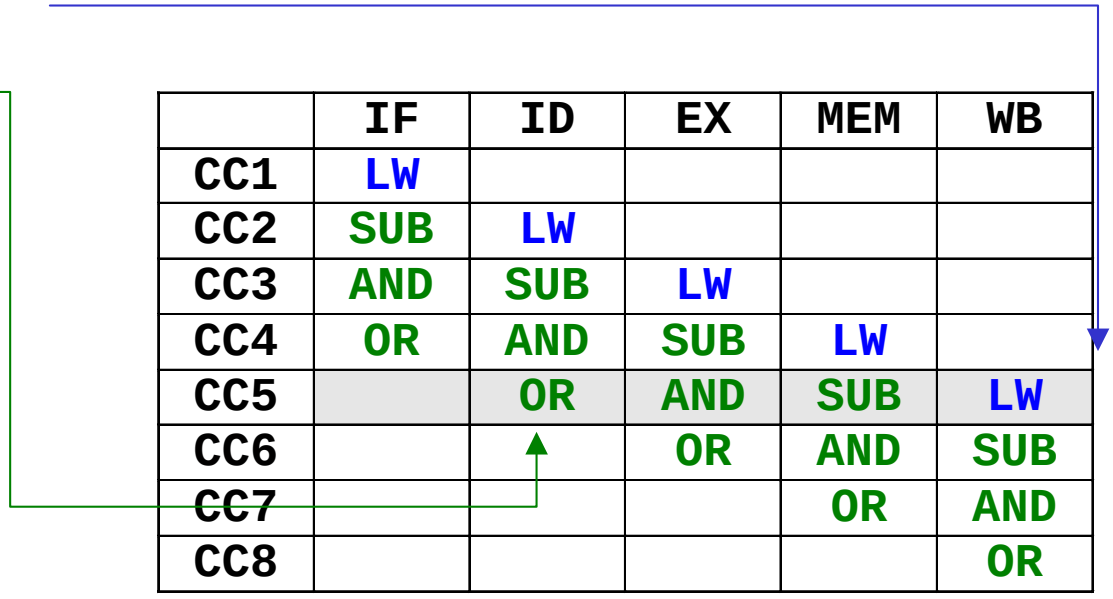
Bad timing

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

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

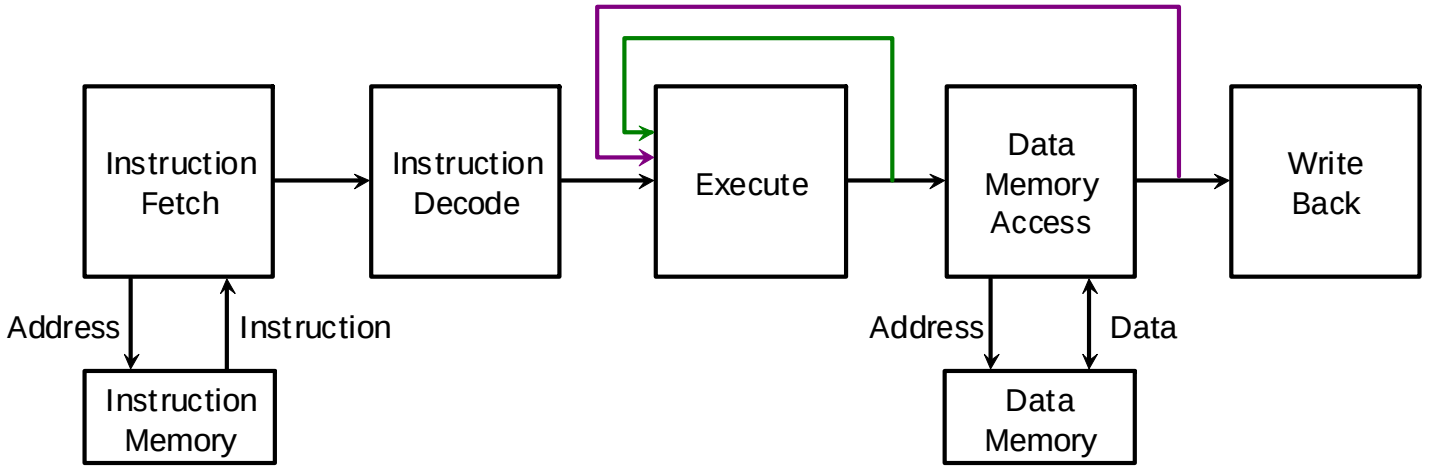
Trick to minimize stall

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**



Forwarding in Instruction View

Clock Cycle →

	1	2	3	4	5	6	7
LW R1,R2,R3	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

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

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

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

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

Register-Store RAW Dependencies

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

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

SW reads incorrect value in **ID**

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

Clock Cycle →

	1	2	3	4	5	6
SUB R1,R5,R4	IF	ID	EX	MEM	WB	
SW 32(R2),R1		IF	ID	EX	MEM	WB

DLX Control Hazard

Predict-Not Taken Policy

Flush stage IF 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_{FT}	Fall-Through		IF	ID	EX	MEM	WB			
I_{FT+1}				IF						
...	…	…	…	…	…	…	…	…	…	…
I_T	Target			IF	ID	EX	MEM	WB		
I_{T+1}					IF	ID	EX	MEM	WB	

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

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

DLX Control Performance

Predict-Not-Taken

Branch taken — Flush instruction in IF

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

Branch	20% of instructions
Not taken	33% of branch
Taken	67% of branch

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

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

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

$$CPI = 1.13 \Rightarrow \text{(12\% degradation)}$$

Other Stalls

Some instruction dependencies are not repaired by forwarding

Default handling — stall dependent instruction until source ready

ALU – Branch Stall

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

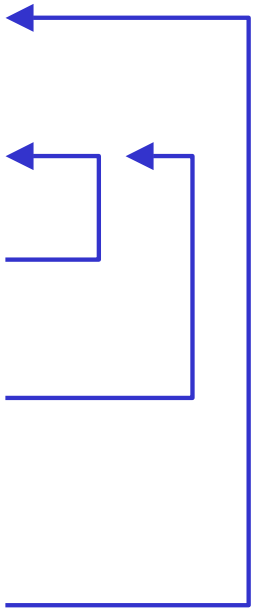
ALU – Store

	1	2	3	4	5	6	7	8
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
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

Rescheduling

ADDI R1, R0, #400	
LW R2, -4(R1)	
LW R3, 3FC(R1)	
ADD R4, R2, R3	1 stall cycle
LW R2, 7FC(R1)	
SUB R4, R4, R2	1 stall cycle
LW R2, BFC(R1)	
ADD R4, R4, R2	1 stall cycle
SW -4(R1), R4	
SUBI R1, R1, #4	
BNEZ R1, FFD8	2 stall cycles
XOR R1, R1, R1	



1	ADDI R1, R0, #400	WB – 5
2	SUBI R1, R1, #4	WB – 6
3	LW R2, 0(R1)	WB – 7
4	LW R3, 400(R1)	WB – 8
5	LW R5, 800(R1)	WB – 9
6	LW R6, C00(R1)	WB – 10
7	ADD R4, R2, R3	ID – 8
8	SUB R4, R4, R5	ID – 9
9	ADD R4, R4, R6	ID – 10
10	SW 0(R1), R4	ID – 11
11	BNEZ R1, FFD8	ID – 12
12	XOR R1, R1, R1	

Change to improve performance

Re-order instruction execution without affecting dependencies

Register renaming — remove false dependencies

Adjust address offsets

Improvement by Re-Scheduling

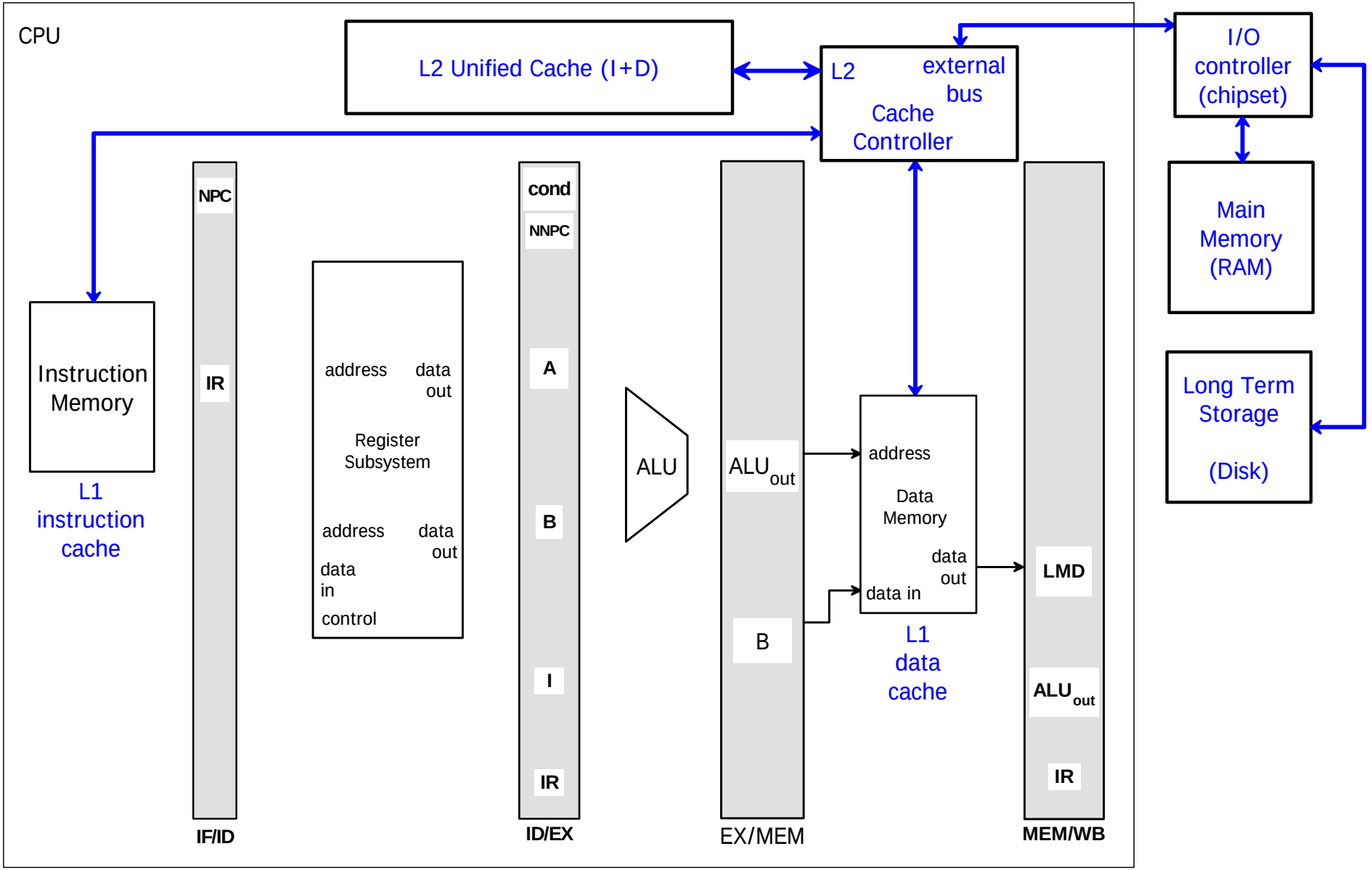
```
a[i] = a[i] + b[i] – c[i] + d[i]
```

```
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

DLX Memory Hierarchy



MIPS Architecture

RISC Instruction Set Architecture (ISA)

Defines registers + instructions

MIPS cores

Define device-dependent implementation details

Pipeline organization, I/O organization, control registers, …

MIPS32

32-bit RISC ISA

DLX = simplified MIPS32

MIPS64

64-bit RISC ISA = 64-bit integers / 64-bit addresses

Binary compatible with MIPS32

Applications

Typically licensed to OEMs

Design implemented in embedded systems

MIPS-based PCs used in China

MIPS32 ISA — 1

Registers

32-bit integer registers

$0, $1, ... , $31

Regs[$0] = 0 (read-only)

32-bit FP registers

F0, F1, ... , F31

Special registers

HI, LO

64-bit result of integer multiply

Quotient + remainder result of integer divide

Instruction formats

Type	32 26	25 21	20 16	15 0		
	6	5	5	5	5	6
R	opcode	s	t	d	a	function
I	opcode	s	t	immediate		
J	opcode	target				

MIPS32 ISA — 2

Coprocessors

- Logical extensions of basic MIPS ISA
- Accessed via coprocessor read / write instructions

CP0

- System Control Coprocessor — on CPU
- Supports virtual memory system and exception handling
  - Translates virtual addresses into physical addresses
  - Controls cache subsystem
  - Handles switches between kernel / supervisor/ user states
  - Manages exceptions / diagnostic control / error recovery

CP1

- Interface to FPU

CP2

- Available for device-specific implementations

CP3

- Interface to FPU on MIPS64 and newer MIPS32

MIPS32 Instruction Set — 3

A few MIPS instructions not in DLX

Coprocessor Load / Store	`LWCz t, imm(reg)`	Load Word to Coprocessor_z, z = 1 or 2
	`SWCz imm(reg), t`	Store Word from Coprocessor_z, z = 1 or 2
Test+Set	`SLTI t, s, imm`	Set on Less Than Immediate
Shift	`ROTR`	Rotate Word Right
	`SLL / SRA`	Shift Word Left Logical / Arithmetic
Multiply	`MUL d, s, t`	Multiply to general register
	`MULT s, t`	Multiply to HI/LO (64-bit output)
	`MADD s, t`	Multiply and add result to HI/LO
Extract	`EXT t, s, pos, size`	`t ← substr(s,pos=a,size=d)`
Branch	`BGTZ / BGEZ`	Branch greater / greater or equal zero
	`BLTZ / BLEZ`	Branch less / less or equal zero
	`BNE`	Branch not equal
Synchronize	`SYNC`	Critical section for shared memory
System	`SYSCALL`	System Call
Trap	`TEQ / TGE / TNE`	Trap if equal / greater or equal / not equal
Cache	`PREF`	Prefetch

MIPS64 ISA

Registers

64-bit integer registers

`$0, $1, ... , $31`

`Regs[$0] = 0 (read-only)`

32-bit FP registers on 32-bit FPU

64-bit FP registers on 64-bit FPU

`F0, F1, ... , F31`

Special registers

`HI, LO`

128-bit result of integer multiply

Quotient + remainder result of integer divide

Instruction formats

32-bit instruction length — binary compatible with MIPS32

MIPS32/64 instructions act on lower 32-bits in registers

MIPS64_double instructions act on full 64-bits in registers

Memory address = 64-bit pointer (register) + 16-bit immediate