DLX Architecture

A Model RISC Processor

DLX Model Processor

MIPS instruction set architecture (ISA)

- 32-bit and 64-bit RISC architectures
- Defines registers + instructions

Commercial MIPS cores

- Device-dependent implementation details
- Hardware organization
- Typically licensed to OEMs (Original Equipment Manufacturer)
  - Designer specifies hardware requirements compatible with ISA
  - MIPS sells license to use ISA in manufactured equipment
  - OEM manufactures MIPS processor and final product

DLX = simplified version of MIPS 32-bit ISA

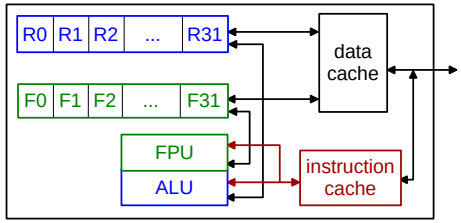
- Fewer instructions
- Simpler hardware implementation
- Appropriate for teaching

DLX Architecture — General Features

Flat memory model with 32-bit address

Data types

- Integers (32-bit)
- Floating Point
  - Single precision (32-bit)
  - Double precision (64 bits)



Register-register operation model

32 integer registers (32 bits wide)

- Named R0, R1, ... , R31
- Addressed as 00000 to 11111 in register address space
- Reg[R0] = 0 (constant)
- Other registers identical (no special purpose registers)

32 FP registers (32 bits wide)

- F0, F1, ... , F31
- Satisfy IEEE 754 standard FP format
- Store double precision FP is register pair (even , odd)

Addressing Modes

Register	`ADD R3, R4, R5`	`Reg[R3] ← Reg[R4] + Reg[R5]`
Immediate	`ADD R3, R4, #3`	`Reg[R3] ← Reg[R4] + 3`
Displacement	`LW R3, 100(R1)`	`Reg[R3] ← Mem[100+Reg[R1]]`
Register Deferred	`LW R3, 0(R1)`	`Reg[R3] ← Mem[Reg[R1]]`
Absolute	`LW R3, 100(R0)`	`Reg[R3] ← Mem[100]`

Three memory addressing modes implemented using

Displacement

`100(R1)` `Reg[R3] ← Mem[100+Reg[R1]]`

Register Deferred

`0(R1)` `Reg[R3] ← Mem[0+Reg[R1]]`

Absolute

`100(R0)` `Reg[R3] ← Mem[100+Reg[R0]]`

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[F2],Reg[F3] $\leftarrow_{64}$ Reg[F0],Reg[F1]
MOVFP2I R2, F2	FP to INT	Reg[R2] $\leftarrow_{32}$ Reg[F2]
MOVI2FP F2, R2	INT to FP	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	Divide	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	$\begin{pmatrix}\text{Reg[F0]}\\\text{Reg[F1]}\end{pmatrix} \xleftarrow{64} \begin{pmatrix}\text{Reg[F2]}\\\text{Reg[F3]}\end{pmatrix} + \begin{pmatrix}\text{Reg[F4]}\\\text{Reg[F5]}\end{pmatrix}$
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, F2, 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} ≤ offset ≤ 2^{25} - 1)
JAL offset	Jump and Link	Reg[R31] ← PC PC ← PC + offset (-2^{25} ≤ offset ≤ 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} ≤ offset ≤ 2^{15} - 1)
BEQZ R4, offset	Branch equal zero	if Reg[R4] == 0 then PC ← PC + offset (-2^{15} ≤ offset ≤ 2^{15} - 1)
BNEZ R4, offset	Branch not equal zero	if Reg[R4] != 0 then PC ← PC + offset (-2^{15} ≤ offset ≤ 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 Language

```
for ( i = 0 ; i < 256 ; i++)
   a[i] = a[i] + b[i] - c[i] + d[i]
}
```

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

```
ADDI R1, R0, #0x400 ; 256 integers = 1024 bytes = 400h bytes
LW   R2, -4(R1)     ; load word from a[]   (400 - 4 = 3FC)
LW   R3, 3FC(R1)    ; load word from b[]   (400 + 3FC = 7FC)
ADD  R4, R2, R3     ; add
LW   R2, 7FC(R1)    ; load word from c[]   (400 + 7FC = BFC)
SUB  R4, R4, R2     ; sub
LW   R2, BFC(R1)    ; load word from d[]   (400 + BFC = FFC)
ADD  R4, R4, R2     ; add
SW   -4(R1), R4     ; store sum in a[]
SUBI R1, R1, #4     ; i--
BNEZ R1, -0x28      ; if R1 <> 0 jump 10 back instructions
```

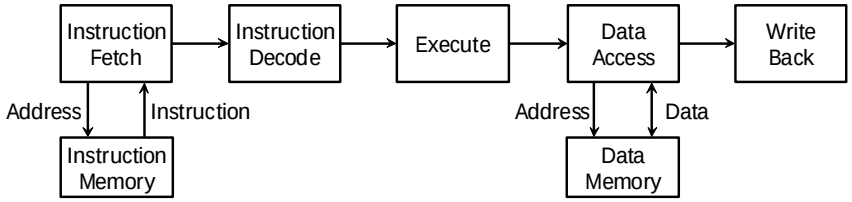

Implementation

General approach

- No central system bus
- Base hardware organization on assembly line with uniform operations
- Separate memory for instructions and data

High level design

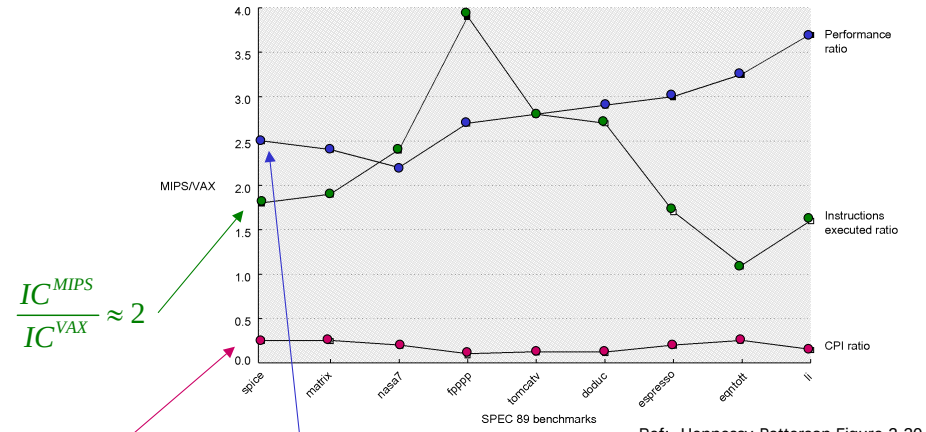
- Instructions move through 5 stages (left to right)
  - First two stages identical for all instructions — **FETCH** and **DECODE**
  - Last three stages operate according to instruction
    - **EXECUTE** (ALU instructions and address calculations)
    - **MEMORY ACCESS** (Load/Store instructions)
    - **WRITE BACK** (register update for Load and ALU instructions)



RISC Performance

Compare VAX with MIPS 2000 (RISC CPU) on SPEC 89 results

Same clock rate



Ref: Hennessy-Patterson Figure 2-30

$$\frac{IC^{MIPS}}{IC^{VAX}} \approx 2$$

$$\frac{CPI^{MIPS}}{CPI^{VAX}} \approx \frac{1}{6}$$

$$S = \frac{CPI^{VAX} \times IC^{VAX} \times \tau}{CPI^{MIPS} \times IC^{MIPS} \times \tau} \approx 6 \times \frac{1}{2} = 3$$

Instruction Formats

32-bit instructions (0 to 31)

Three instruction formats

J-type

- Jump (unconditional branch) instructions
- Specifies branch offset

R-type

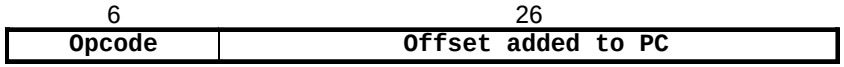
- Register-register ALU instructions
- Specifies destination register (rd), and two source registers (rs1, rs2)

I-type

- All other instructions
- Specifies destination register (rd), immediate, and source register (rs)

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			

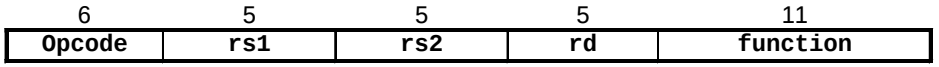
J-Type Instruction Format



Encodes:

- Jump
  PC ← PC + offset
- Jump and link
  r31 ← PC
  PC ← offset
- Trap and return from exception
  Implementation unspecified in Hennessy and Patterson
  Two possible implementations for Offset field
  1. Lower 26 bits of physical address of Interrupt Service Routine
  2. Trap number = index to Interrupt Vector Table

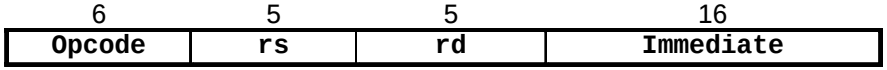
R - Type Instruction



Encodes:

- Register-register ALU operations
  rd ← rs1 function rs2
  Function encodes the ALU operation: Add, Sub, ...

I - Type Instruction



Encodes:

- Loads
  rd ← imm(rs)
- Stores
  imm(rs) ← rd
- ALU operations with immediate operand
  rd ← rs op immediate
- Conditional branch instructions
  if rs eq/ne 0 then PC ← PC + imm (rd unused)
- Jump register
  PC ← rs
- Jump and link register
  rd ← PC
  PC ← PC + immediate

Implementation Details

Execution Stages by Instruction Type

ALU	Store	Load	Branch
Fetch instruction from memory	Fetch instruction from memory	Fetch instruction from memory	Fetch instruction from memory
Decode operation and operands	Decode operation and operands	Decode operation and operands	Decode operation and operands
Calculate ALU operation	Calculate memory address	Calculate memory address	Calculate branch condition Calculate branch address
	Store data to memory Update PC	Load data from memory	Update PC
Write result to register Update PC		Write loaded data to register Update PC	

Temporary Registers for Implementation

IR
- Instruction Register
- Holds fetched instruction during execution

PC
- Program Counter
- Memory address of next instruction

NPC
- Next Program Counter
- Temporary update of PC (points to fall-through instruction)

A, B, I
- Operand buffers
- Values read from data registers according to instruction

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

Example Type-I ALU Instruction

Instruction	`addi R1, R2, #5`
Operation	`Reg[R1] ← Reg[R2] + 5`
Encoding	0-5: `addi` (op) | 6-10: `00010` (rs) | 11-15: `00001` (rd) | 16-31: `0000 0000 0000 0101` (immediate)
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
Hardware Stage 2	$A \leftarrow Reg[IR_{6-10}]$ /* A ← Reg[R2] */ $B \leftarrow Reg[IR_{11-15}]$ /* B ← Reg[R1] */ $I \leftarrow (IR_{16})^{16}$ ## IR_{16-31}
Hardware Stage 3	$ALU_{out} \leftarrow A + I$
Hardware Stage 4	
Hardware Stage 5	$Reg[IR_{11-15}] \leftarrow ALU_{out}$ /* Reg[R1] ← A + I */ `PC ← NPC`

Example Type-R ALU Instruction

Instruction	`add R1, R2, R3`
Operation	`Reg[R1] ← Reg[R2] + Reg[R3]`
Encoding	0-5: `R-R` (op) | 6-10: `00010` (rs1) | 11-15: `00011` (rs2) | 16-20: `00001` (rd) | 21-31: `add` (funct)
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
Hardware Stage 2	$A \leftarrow Reg[IR_{6-10}]$ /* A ← Reg[R2] */ $B \leftarrow Reg[IR_{11-15}]$ /* B ← Reg[R3] */ $I \leftarrow (IR_{16})^{16}$ ## IR_{16-31}
Hardware Stage 3	$ALU_{out} \leftarrow A + B$
Hardware Stage 4	
Hardware Stage 5	$Reg[IR_{16-20}] \leftarrow ALU_{out}$ /* Reg[R1] ← A + B */ `PC ← NPC`

Example Type-I Store Instruction

Instruction	`SW 32(R1), R2`
Operation	`Mem[32+Reg[R1]] ← Reg[R2]`
Encoding	0-5: SW (op) | 6-10: 00001 (rs) | 11-15: 00010 (rd) | 16-31: 0000 0000 0010 0000 (immediate)
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
Hardware Stage 2	`A ← Reg[IR6-10]` /* A ← Reg[R1] */ `B ← Reg[IR11-15]` /* B ← Reg[R2] */ `I ← (IR16)16 ## IR16-31`
Hardware Stage 3	`ALUout ← A + I`
Hardware Stage 4	`Mem[ALUout] ← B` /* Mem[A+I] ← Reg[R2] */ `PC ← NPC`
Hardware Stage 5	

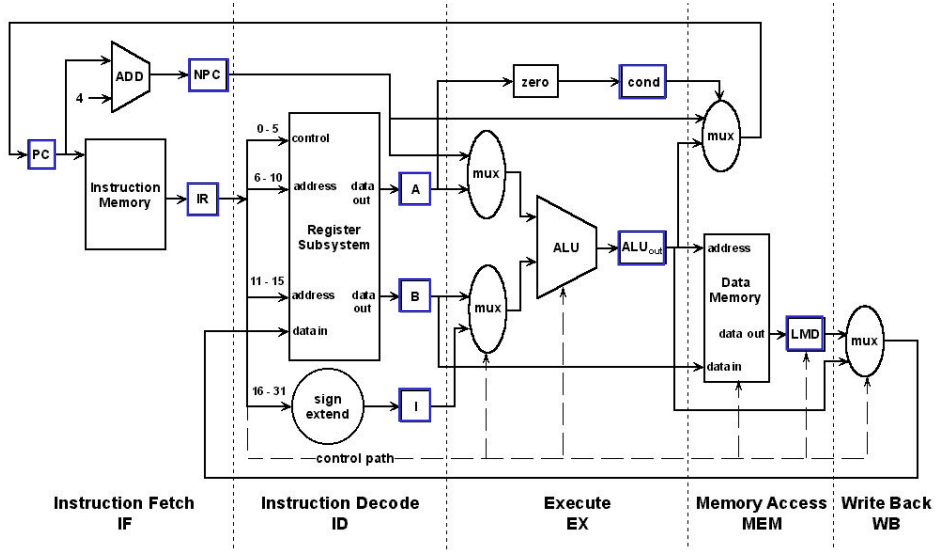
Example Type-I Load Instruction

Instruction	`LW R2, 32(R1)`
Operation	`Reg[R2] ← Mem[32+Reg[R1]]`
Encoding	0-5: LW (op) | 6-10: 00001 (rs) | 11-15: 00010 (rd) | 16-31: 0000 0000 0010 0000 (immediate)
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
Hardware Stage 2	`A ← Reg[IR6-10]` /* A ← Reg[R1] */ `B ← Reg[IR11-15]` /* B ← Reg[R2] */ `I ← (IR16)16 ## IR16-31`
Hardware Stage 3	`ALUout ← A + I`
Hardware Stage 4	`LMD ← Mem[ALUout]` /* LMD ← Mem[A+I] */
Hardware Stage 5	`Reg[IR11-15] ← LMD` /* Reg[R2] ← Mem[A+I] */ `PC ← NPC`

Example Type-I Conditional Branch Instruction

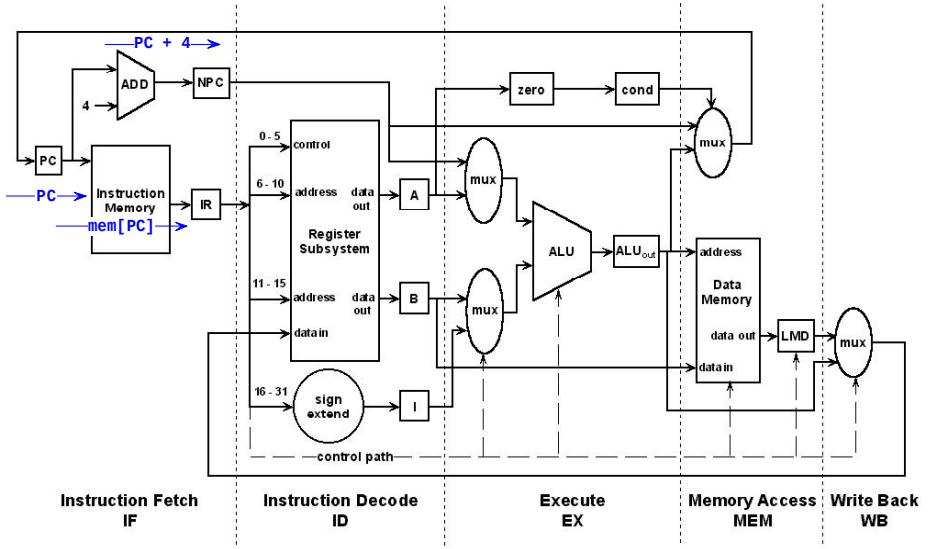
Instruction	`beqz R1, 1024`
Operation	`if (Reg[R1] == 0) PC ← NPC + 1024` `else PC ← NPC`
Encoding	0-5: beqz (op) | 6-10: 00001 (rs) | 11-15: 00000 (rd) | 16-31: 0000 0100 0000 0000 (immediate)
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
Hardware Stage 2	`A ← Reg[IR6-10]` /* A ← Reg[R1] */ `B ← Reg[IR11-15]` /* B ← Reg[R0] */ `I ← (IR16)16 ## IR16-31`
Hardware Stage 3	`ALUout ← NPC + I` `if (A == 0) cond = 1 else cond = 0`
Hardware Stage 4	`if (cond == 1) PC ← ALUout else PC ← NPC`
Hardware Stage 5	

DLX Hardware Drawing — Version 1



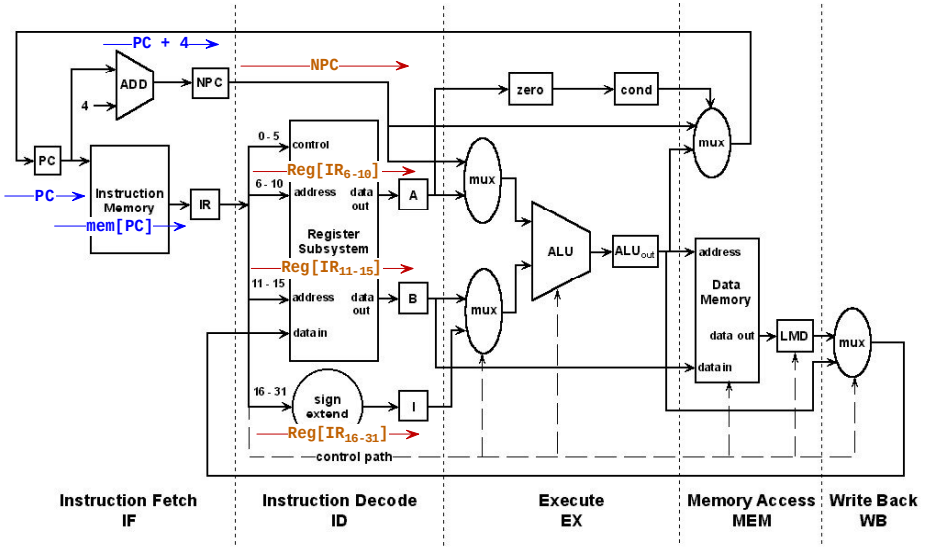
mux (multiplexer) — chooses 1 output from N inputs

Type-I ALU Instruction — 1



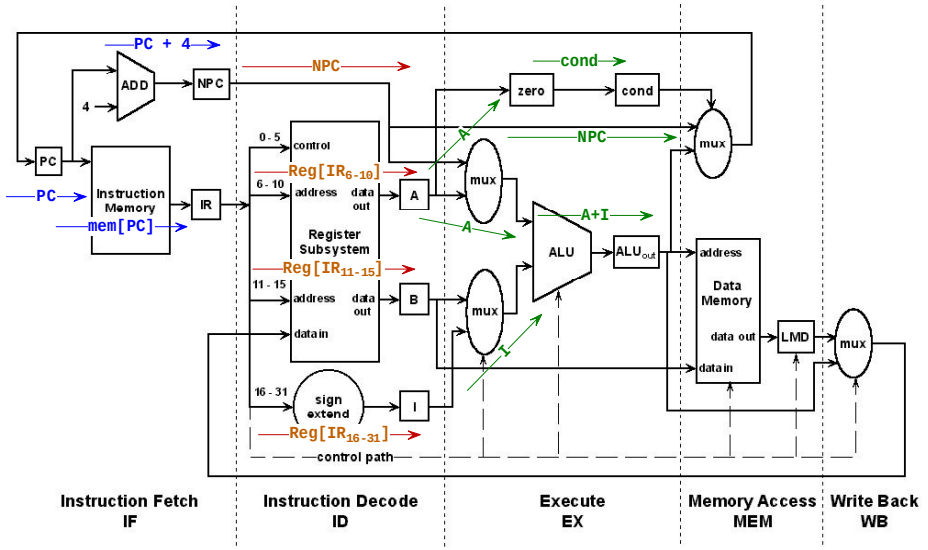
`addi r1, r2, #5` `regs[r1] ← regs[r2] + 5`

Type-I ALU Instruction — 2



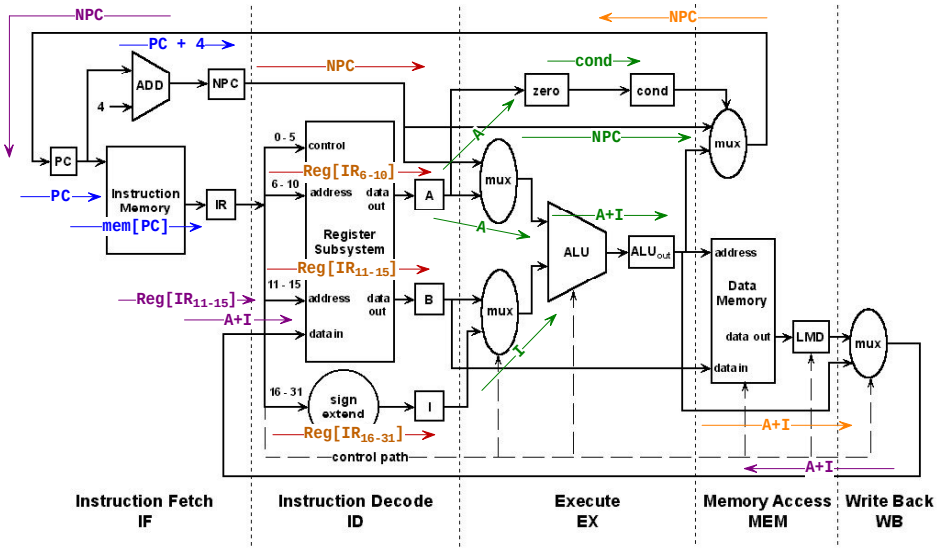
`addi r1, r2, #5` `regs[r1] ← regs[r2] + 5`

Type-I ALU Instruction — 3



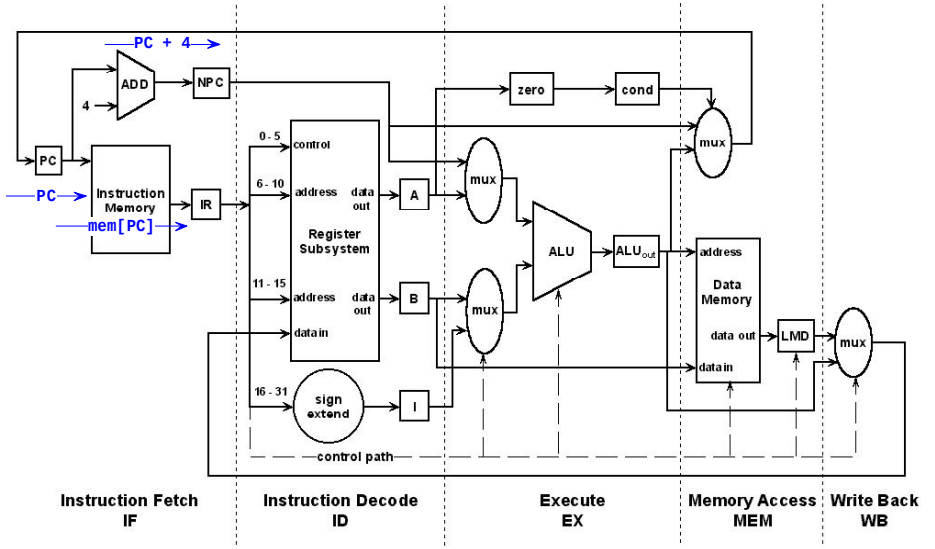
`addi r1, r2, #5` `regs[r1] ← regs[r2] + 5`

Type-I ALU Instruction — 4



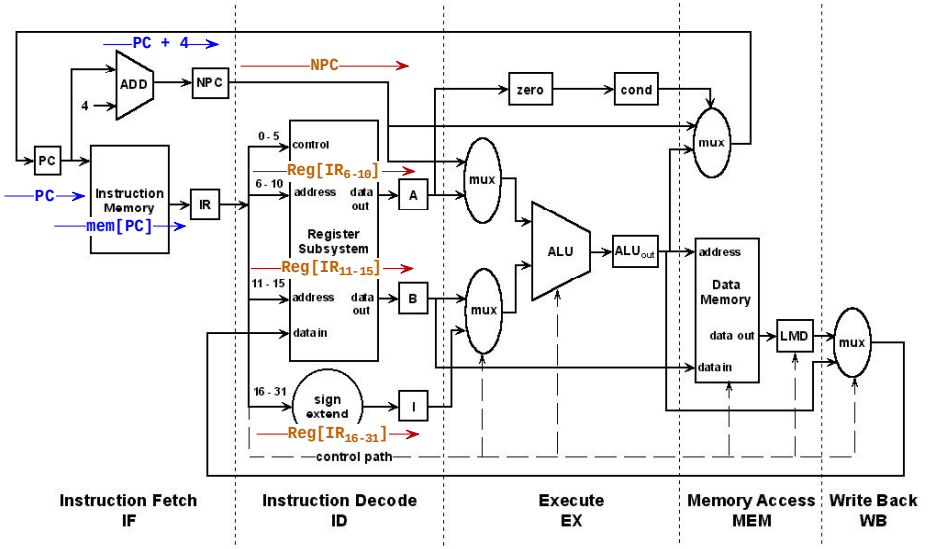
`addi r1, r2, #5` `regs[r1] ← regs[r2] + 5`

Type-R ALU Instruction — 1



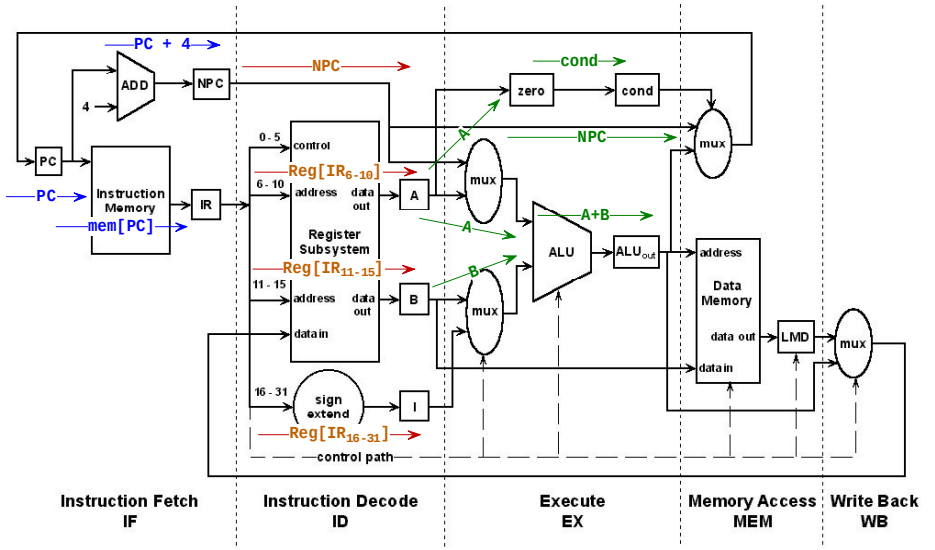
`add r1, r2, r3` `regs[r1] ← regs[r2] + regs[r3]`

Type-R ALU Instruction — 2



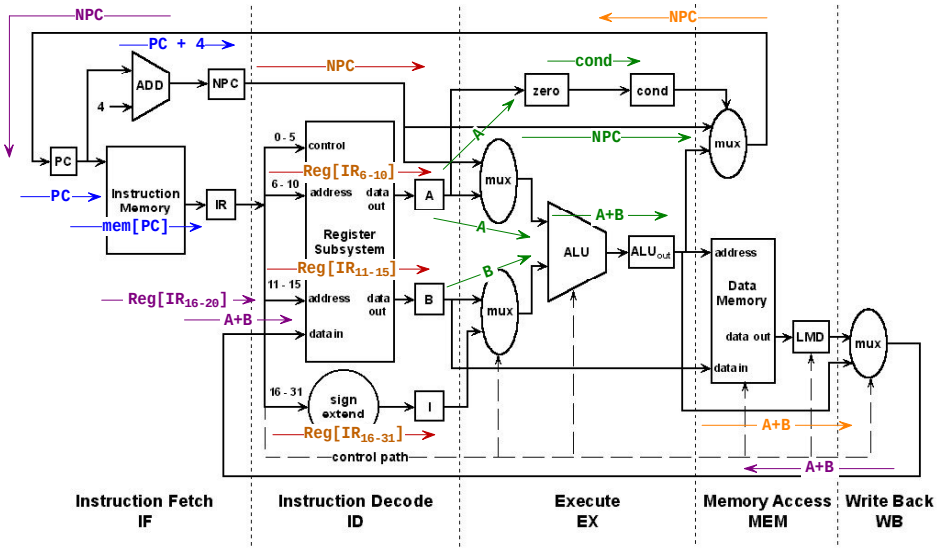
`add r1, r2, r3` `regs[r1] ← regs[r2] + regs[r3]`

Type-R ALU Instruction — 3



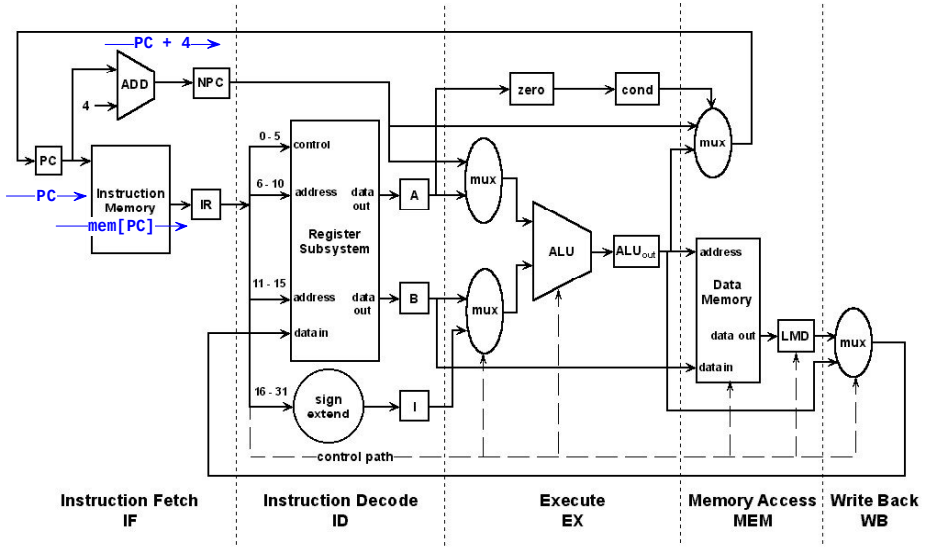
`add r1, r2, r3` `regs[r1] ← regs[r2] + regs[r3]`

Type-R ALU Instruction — 4



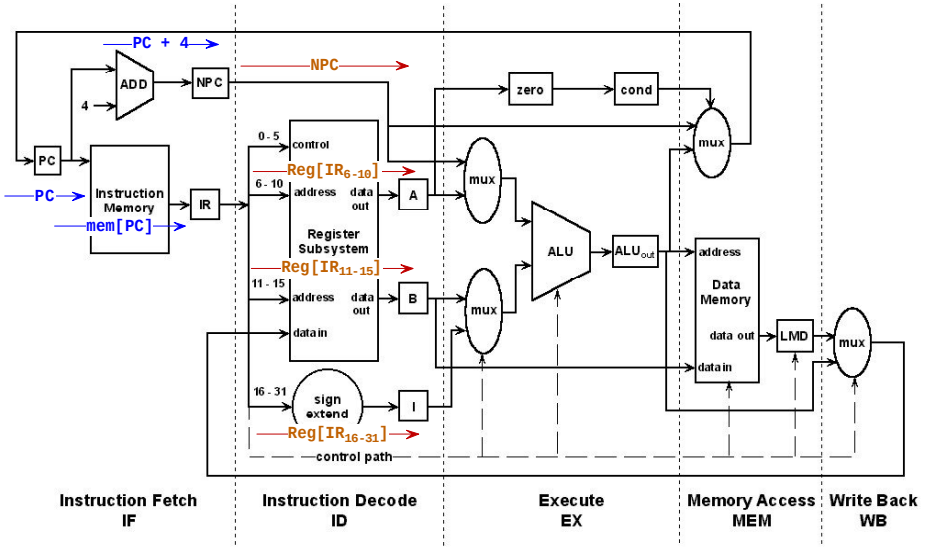
`add r1, r2, r3` `regs[r1] ← regs[r2] + regs[r3]`

Type-I Store Instruction — 1



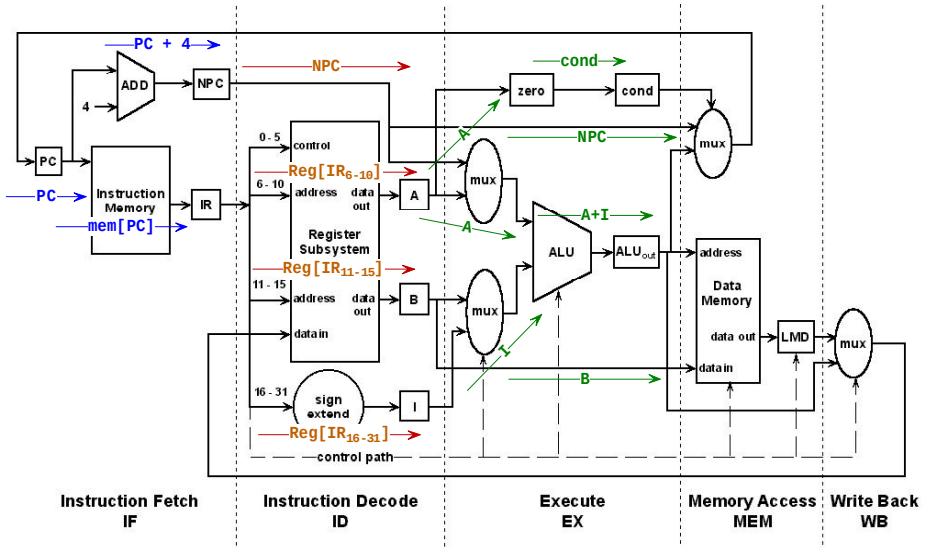
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Store Instruction — 2



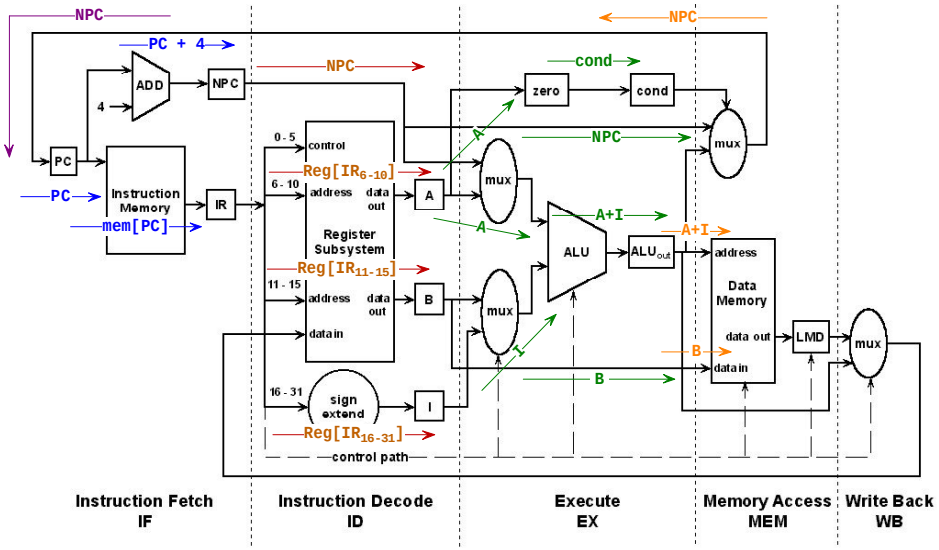
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Store Instruction — 3



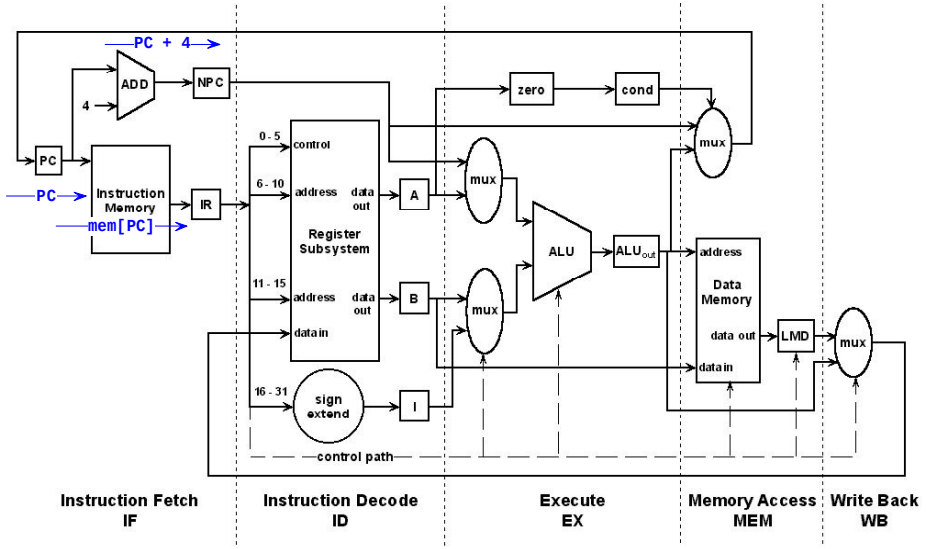
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Store Instruction — 4



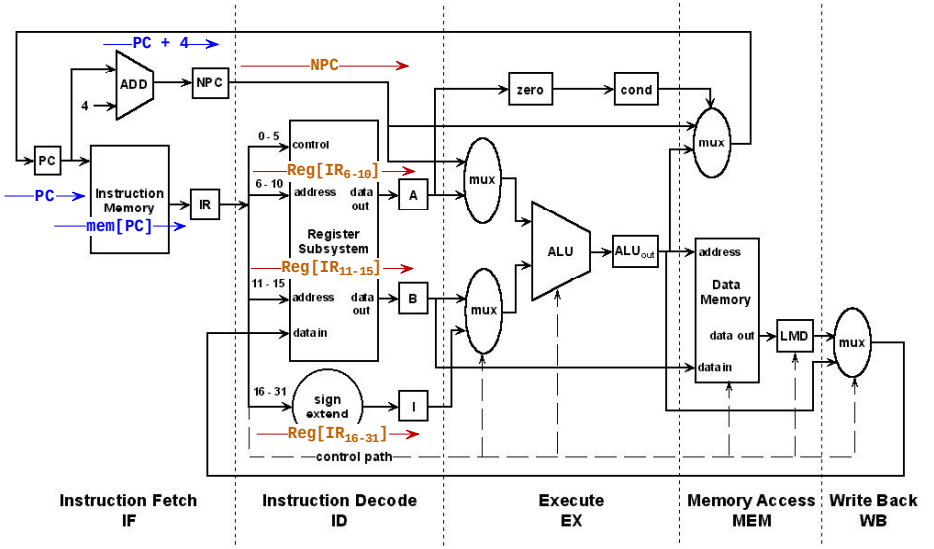
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Load Instruction — 1



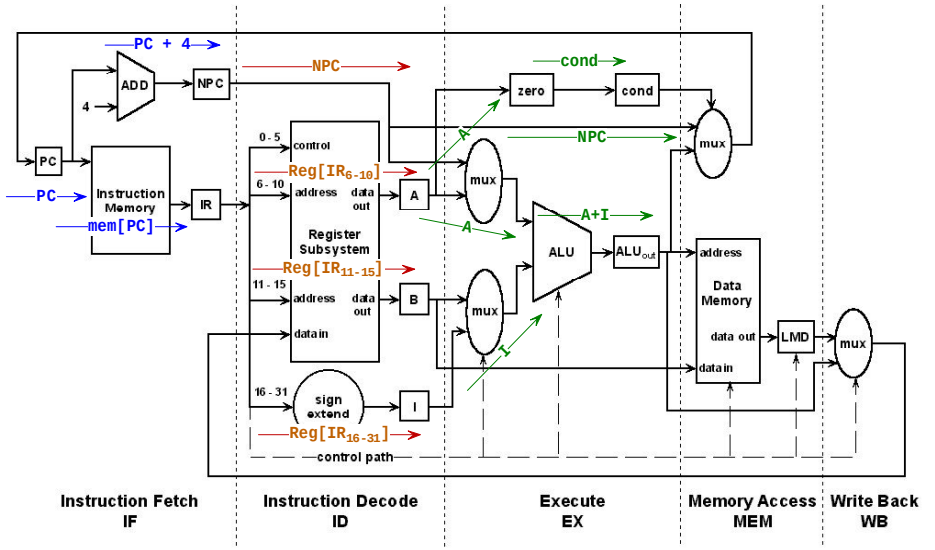
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 2



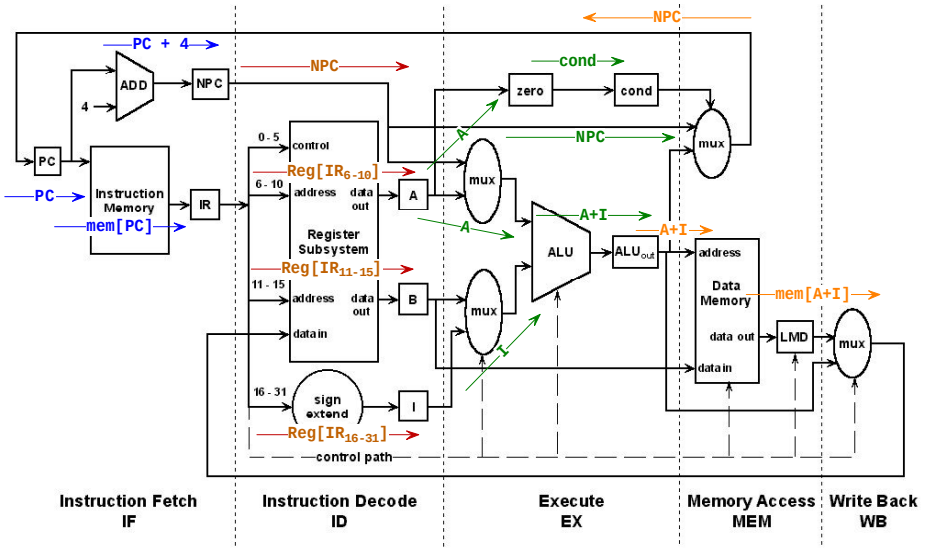
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 3



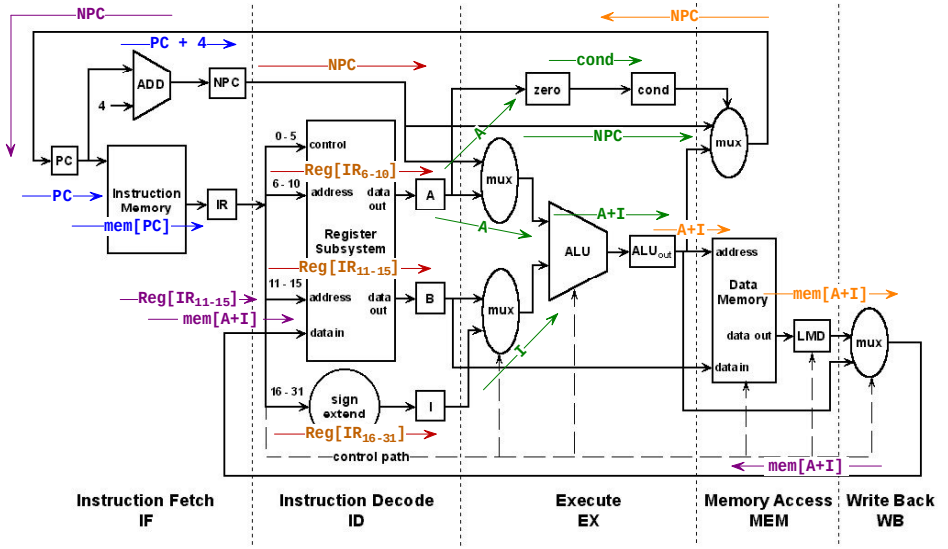
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 4



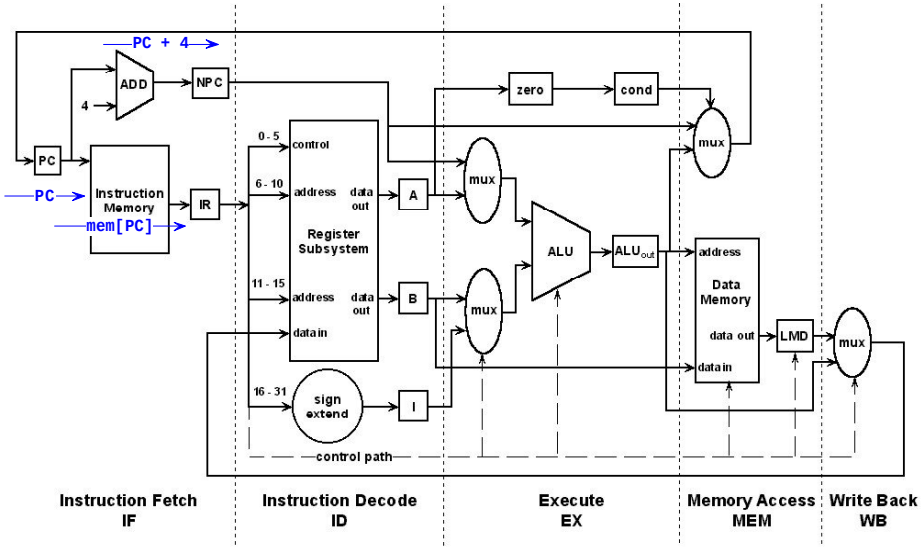
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 5



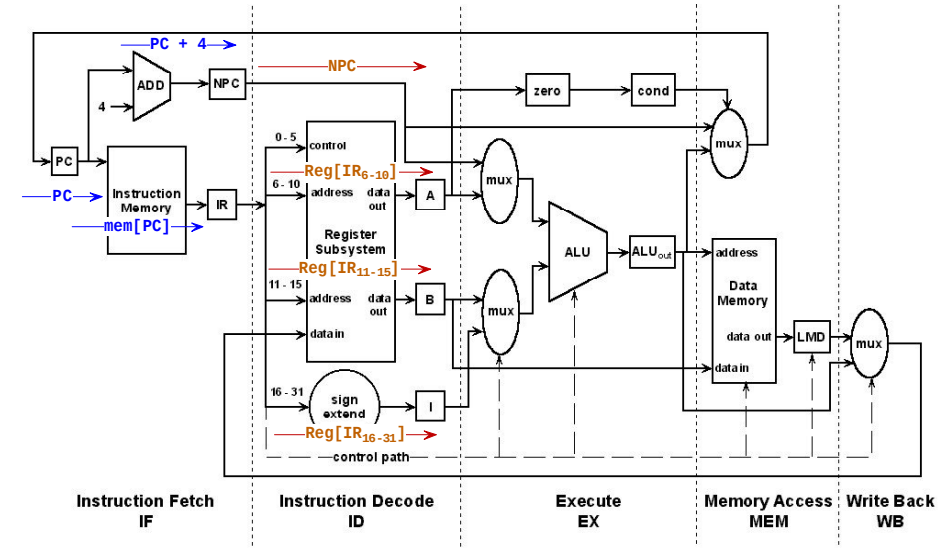
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Branch Instruction — 1



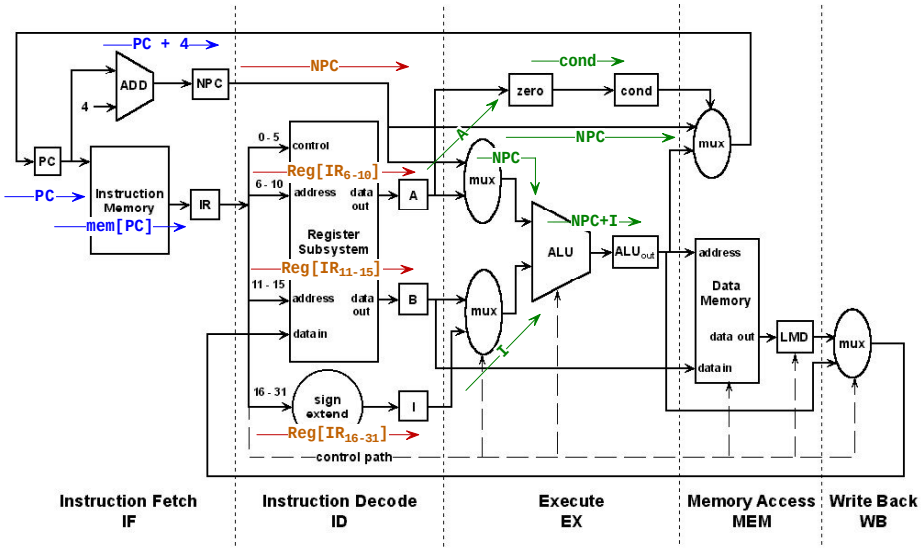
`beqz r1, 1024` `if (regs[r1] == 0) PC ← NPC + I else PC ← NPC`

Type-I Branch Instruction — 2



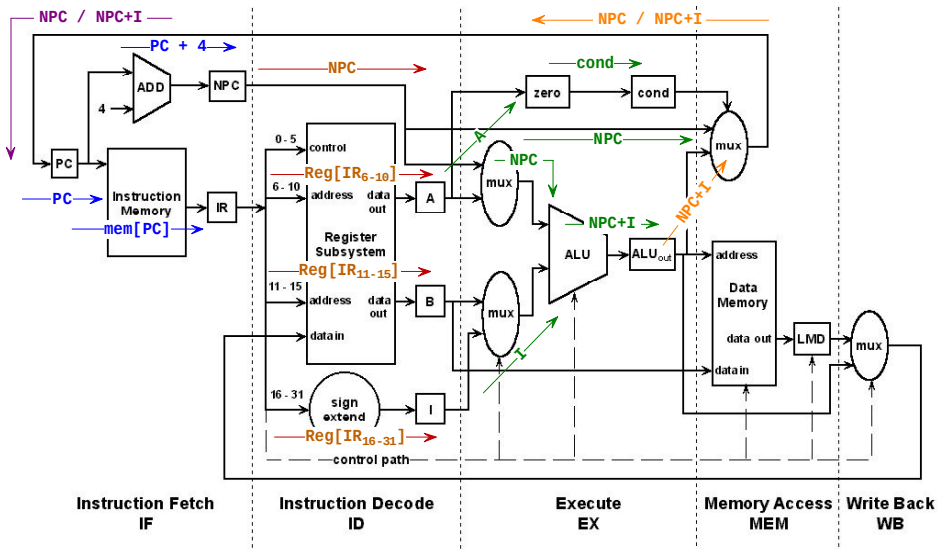
`beqz r1, 1024` `if (regs[r1] == 0) PC ← NPC + I else PC ← NPC`

Type-I Branch Instruction — 3



`beqz r1, 1024` `if (regs[r1] == 0) PC ← NPC + I else PC ← NPC`

Type-I Branch Instruction — 4




```
beqz r1, 1024      if (regs[r1] == 0) PC ← NPC + I else PC ← NPC
```


Performance

Instruction distribution for version 1 based on compilation of SPEC 92

Type i	IC_i / IC	CPI_i
ALU	40%	4
Load	25%	5
Store	15%	4
Branch	20%	4

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC}$$
$$= 4 \times 0.40 + 5 \times 0.25 + 4 \times 0.15 + 4 \times 0.25$$
$$= 4.25$$