Introduction to Computer Architecture

From Wikipedia, the free encyclopedia

In computer engineering, computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. Some definitions of architecture define it as describing the capabilities and programming model of a computer but not a particular implementation. In other definitions computer architecture involves instruction set architecture design, microarchitecture design, logic design, and implementation.

Computer Architecture

What is Computer Architecture — Parsed

Wikipedia

In computer engineering, **computer architecture** is a set of **rules and methods** that describe the **functionality**, **organization**, and **implementation** of computer systems. Some definitions of architecture define it as describing the **capabilities** and **programming model** of a computer but not a particular implementation. In other definitions computer architecture involves **instruction set architecture** design, **microarchitecture** design, **logic** design, and implementation.

Translation:

computer architecture = { **rules and methods** | describe

Functionality — system capabilities and programming model

Organization — instruction set architecture, microarchitecture

Implementation — logic design

}

Computer Architecture

What rules and methods?

Performance

- Short run time
  - Fast programs
- Low latency
  - No waiting between programs and operations
- Low energy consumption
  - Low electric bills
  - Long battery life
  - No overheating

Market factors

- Low cost (in relation to realistic demand for devices)
- Reliable manufacture and delivery
- Profitability

Computing Platform by Application

Workstation applications

Office, basic number crunching, graphics, gaming

A few sequential loop-oriented threads

Typical CPU — Intel x86 (2 to 16 cores)

Mobile applications

Low power version of workstation

Typical CPU — ARM (1 to 4 cores)

Online Transaction Processing (OLTP)

Banking, order processing, inventory, student information system

Thousands of independent SQL transactions with memory latency

Typical CPU — SPARC (64 to 256 cores)

Supercomputer applications

Heavy number crunching, data mining

Thousands of separable sequential loop-oriented threads

Typical CPU — IBM Power (up to 512 Kcores)

Mainframe + Virtualization + Cloud

Mainframe

- 120 CPU cores + 3840 GB RAM + 8 GB/s I/O + high reliability
- Replaces 10 to 1000 servers
- Complex partitioning
  - Allocate hardware subsystems as needed
  - Multiple independent operating systems

Server Virtualization

- Hardware resources partitioned by software running over OS
- Multiple guest operating systems over OS

Cloud computing

- Provider sells standard system interface as service
  - Infrastructure as Service, Platform as Service, Software as Service
- Customer sees system specified in contract
- Provider handles operations+administration+maintenance (OAM)

Introduction to Performance

Basic Definitions

Performance (ביצועים)

Processing speed

Performance measures

Response time (זמן תגובה)

Elapsed time from start to finish of a defined task

Run Time (זמן ריצה)

Response time for a start to finish program task

Latency (חביון – זמן המתנה)

Excess response time — depends on context

Throughput (תפוקה)

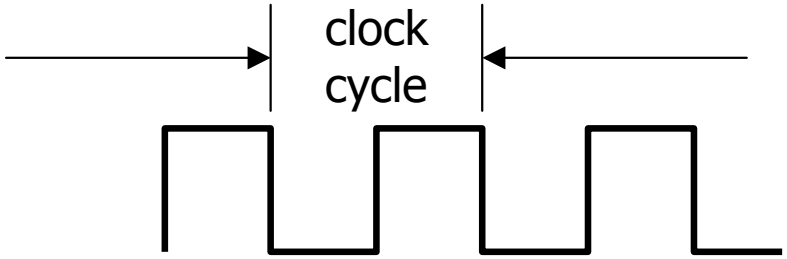
Number of defined tasks performed per unit time

Speedup (שיפור)

$$S = \frac{\text{old run time}}{\text{new run time}} \qquad S > 1 \Rightarrow \text{new run time} < \text{old run time}$$

Run Time and Clock Cycles

CPU is timed by periodic signal called a Clock



Clock Cycle (CC) measured in seconds per cycle

Clock Rate = cycles per second = Hz (Hertz)

Instruction requires 1 or more clock cycles to process

$$\text{Run time} = \text{clock cycles to run program} \times \text{seconds per clock cycles}$$

$$= \frac{\text{clock cycles to run program}}{\text{clock cycles per second}}$$

Higher clock rate $\Rightarrow$ shorter run time

Fewer clock cycles (at constant clock rate) $\Rightarrow$ shorter run time

Speedup and Clock Rate

$$S = \frac{T^{old}}{T^{new}}$$

$$= \frac{\text{program clock cycles}^{old} \times \text{seconds per clock cycle}^{old}}{\text{program clock cycles}^{new} \times \text{seconds per clock cycle}^{new}}$$

$$= \frac{\text{program clock cycles}^{old} \Big/ \text{clock rate}^{old}}{\text{program clock cycles}^{new} \Big/ \text{clock rate}^{new}}$$

$$= \frac{\text{program clock cycles}^{old}}{\text{program clock cycles}^{new}} \times \frac{\text{clock rate}^{new}}{\text{clock rate}^{old}}$$

Speedup follows from

- Higher clock rate
- Fewer program clock cycles
  - Improvements to code
  - Structural improvements in hardware

Factors Affecting Run Time

CPU hardware

Hardware → average clock cycles (CC) required per instruction

Memory (RAM + cache)

Quantity and organization affects data availability

Internal communication and I/O

Speed and organization affects data availability

Operating system efficiency

CPU devotes less time to dense OS code

OS manages tasks/threads to keep hardware busy

Compiler

Converts high level language to machine code

Optimized code runs faster

Special hardware

Dedicated processors (graphics, memory management)

Application code

Efficient algorithms, data structures, parallelization

Examples of Factors Affecting Performance

CPU Hardware Example — Multiple Core Processors

N Core Symmetric Multiprocessor (SMP)

N complete CPUs on one chip

Divide work into **THREADS** running in **PARALLEL** on N processors

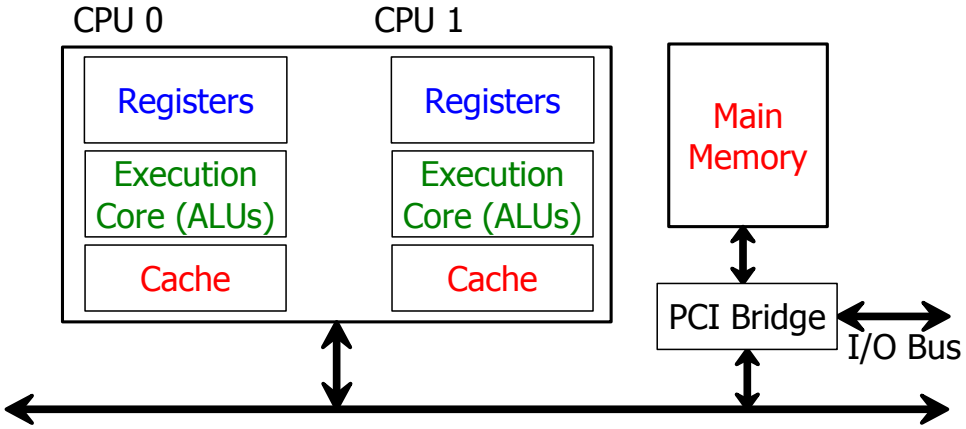
Each CPU has multiple Execution Units (EU)

ALU operates on integers

FPU operates on float / double

Vector processor operates on long registers

Dual Core Processor



OS assigns THREADS to each core

If program threads are separable

If data structures are not too entangled

CPU Hardware Example —Vector Processor

Vector Processor

SIMD — **S**ingle **I**nstruction **M**ultiple **D**ata

Performs same operation on 4, 8, or 16 bytes in parallel

No carry/borrow between bytes

Example

64-bit Source and Destination registers

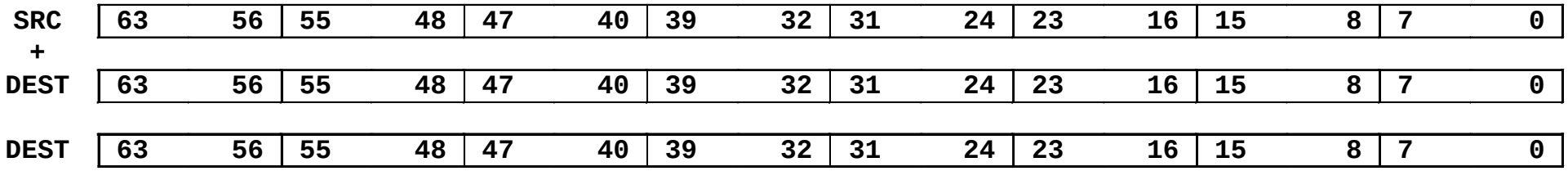
PARALLEL_ADD on 8 pairs of byte operands

$$\mathbf{SRC_{0\ldots7} + DEST_{0\ldots7} = DEST_{0\ldots7}}$$

$$\mathbf{SRC_{8\ldots15} + DEST_{8\ldots15} = DEST_{8\ldots15}}$$

$$\mathbf{\ldots}$$

$$\mathbf{SRC_{56\ldots63} + DEST_{56\ldots63} = DEST_{56\ldots63}}$$



Memory Example — Hybrid Data Structure

Graphic array

200 vertex points = 25 groups of 8 words

Hybrid Data Structure for efficient vector processing

Coordinates and colors

- Stored in separate data structures
- Structures handled in **CONCURRENT THREADS** on separate CPUs

Coordinates

```
struct { float x[8], y[8], z[8] ;
} H_xyz[25] ;
```

8-word group loaded and processed as **vector** on CPU 0

Each loop updates 8 x-coordinates, then 8 y's, then 8 z's

Colors

```
struct { float r[8], g[8], b[8] ;
} H_rgb[25] ;
```

8-word group loaded and processed as **vector** on CPU 1

Each loop updates 8 reds, then 8 greens, then 8 blues

Memory Example — Color Data Structure

Addressing in 32-bit processors

Processor sends 32-bit aligned address A (multiple of 4)

Reads 4-byte word — bytes from addresses **`A, A+1, A+2, A+3`**

Access to individual byte requires reading entire dword

24-bit True Color

3 color bytes — Red, Blue, Green

2^8 = 256 levels per color (**`0x00 – 0xFF`**)

Most 24-bit colors split between dwords

Access to pixel color ⇒ 2 memory cycles

dword			dword			dword					
R	G	B	R	G	B	R	G	B	R	G	B
1 cycle			2 cycles			2 cycles			1 cycles		

32-bit True Color

Pad 24-bit color with blank byte

Align color data on 32-bit addresses

One memory cycle per pixel

dword				dword			
R	G	B	—	R	G	B	—

Compiler Efficiency Example

C code compiled inefficiently for Intel 8086 processor

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

```
0000  MOV     WORD PTR [BP-02],0000 ; i = 0
0005  CMP     WORD PTR [BP-02],+0A
0009  JGE     0018                          ; break on i ≥ 10
000B  MOV     AX,[BP-02]                    ; AX ← i
000E  SHL     AX,1                          ; AX ← 2 * AX
0010  MOV     [BP-04],AX                    ; j ← AX
0013  INC     WORD PTR [BP-02]              ; i++
0016  JMP     0005                          ; loop
0018  RET
```

Page from Intel 8086 Manual



80186/80188 HIGH-INTEGRATION 16-BIT MICROPROCESSORS,
COPYRIGHT © INTEL CORPORATION, 1995

INSTRUCTION SET SUMMARY

Clock Cycles per Instruction →

Function	Format				80186 Clock Cycles
DATA TRANSFER					
MOV = Move:					
Register to Register/Memory	1 0 0 0 1 0 0 w	mod reg r/m			2/12
Register/memory to register	1 0 0 0 1 0 1 w	mod reg r/m			2/9
Immediate to register/memory	1 1 0 0 0 1 1 w	mod 000 r/m	data	data if w=1	12/13
Immediate to register	1 0 1 1 w reg	data	data if w=1		3/4
Memory to accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high		8
Accumulator to memory	1 0 1 0 0 0 1 w	addr-low	addr-high		9
Register/memory to segment register	1 0 0 0 1 1 1 0	mod 0 reg r/m			2/9
Segment register to register/memory	1 0 0 0 1 1 0 0	mod 0 reg r/m			2/11

Program Timing for 8086

Program contains

Setup/takedown instructions (run once)

Loop control instructions

ALU instructions

Instruction timings are given in 8086 manual (in clock cycles)

	Instruction	8086 Clock Cycles (CC)
	MOV WORD PTR [BP-02],0000	MOV imm to r/m 4/13
start:	CMP WORD PTR [BP-02],+0A	CMP r/m,imm 3/10
	JGE stop	Jcc (not taken/taken) 4/13
	MOV AX,[BP-02]	MOV r/m to reg 2/9
	SHL AX,1	Shift reg 2
	MOV [BP-04],AX	MOV reg to r/m 2/12
	INC WORD PTR [BP-02]	INC r/m 3/15
	JMP start	JMP 14
stop:	RET	RET 16

Program Run Time

	Instruction	8086 Clock Cycles (CC)
	MOV WORD PTR [BP-02],0000	13 CC (runs once)
start:	CMP WORD PTR [BP-02],+0A	10 CC on each loop
	JGE stop	4 CC on all loops but last 13 CC on last
	MOV AX,[BP-02]	9 CC on all loops but last
	SHL AX,1	2 CC on all loops but last
	MOV [BP-04],AX	12 CC on all loops but last
	INC WORD PTR [BP-02]	15 CC on all loops but last
	JMP start	14 CC on all loops but last
stop:	RET	16 CC (runs once)

N = number of loop iterations

Total clock cycles = $13 + N \times 10 + (N - 1) \times (4 + 9 + 2 + 12 + 15 + 14) + 13 + 16$

$= 66 \times N - 14$

For N = 11 (stop on i = 10), Total CC = 712

Example — More Efficient Compilation

Store Variables in Registers — Not Memory

	Instruction	8086 Clock Cycles (CC)
	MOV SI,0000	4 CC (runs once)
start:	CMP SI,+0A	3 CC on each loop
	JGE stop	4 CC on all loops but last 13 CC on last
	MOV AX,SI	2 CC on all loops but last
	SHL AX,1	2 CC on all loops but last
	MOV DI,AX	2 CC on all loops but last
	INC SI	3 CC on all loops but last
	JMP start	14 CC on all loops but last
stop:	RET	16 CC (runs once)

Total clock cycles $= 4 + N \times 3 + (N - 1) \times (4 + 2 + 2 + 2 + 3 + 14) + 13 + 16$

$= 30 \times N + 6$

For N = 11 (stop on i = 10), Total CC = 337

Using register variables requires large number of registers

$$S = \frac{712}{337} = 2.11$$

Example — Even More Efficient Compilation

Rebuild Loop

	Instruction	8086 Clock Cycles
	MOV SI,0000	MOV imm to reg 4
start:	MOV AX,SI	MOV reg to reg 2
	SHL AX,1	SHIFT reg 2
	MOV DI,AX	MOV reg to reg 2
	INC SI	INC reg 3
	CMP SI,+0A	CMP reg,imm 3/10
	JL start	Jcc (not taken/taken) 4/13
stop:	RET	RET 16

Total clock cycles = $4 + N \times (2 + 2 + 2 + 3 + 3) + (N - 1) \times 13 + 4 + 16$

$= 25 \times N + 11$

For N = 10 (stop on i = 10), Total CC = 261

$$S = \frac{712}{261} = 2.73$$

Measuring Performance

Benchmarks

Definition

Collection of programs for measurement and comparison of system performance

Requirements

Standard and scientific

- Consistent result on repeated tests
- Consistent result by anyone repeating tests

Test system in realistic way

- Reflect statistically representative use of
  - Instruction types
  - Data types
  - Loop length
  - OS and compiler conditions

Summarize data so comparisons make sense

SPEC Benchmark

Programs for system performance measurement + comparison

- Standard + repeatable
- Test system for realistic conditions
- Summary score for easy comparison
- Results posted at **http://www.spec.org/**

Specific test suites

- Cint — CPU integer instructions
- Cfp — CPU FP instructions
- Performance as file server, web server, mail server, graphics

Updated every few years to reflect realistic conditions

- Based on current statistical distributions of computing tasks
- Current CPU test version — 2017
  - Previous version — 2006

Reports speedup

- Run time compared with a standard machine

How SPEC Works

User runs n programs on test machine

Records run-time conditions

Records program run-time in seconds

$$T_i^{test} \quad , \quad i = 1, 2, \ldots, n$$

SPEC provides run-times on reference machine

Sun Fire V490

2100 MHz UltraSPARC-IV+ processor

Powerful symmetric multiprocessing (SMP) server (2006 – 2014)

$$T_i^{ref}$$

User calculates speedup for each program

$$S_i = \frac{T_i^{ref}}{T_i^{test}} \quad , \quad i = 1, 2, \ldots, n$$

User calculates geometric mean of speedups

$$S(\text{test machine on ref}) = \left[\prod_{i=1}^{n} \left(\frac{T_i^{ref}}{T_i^{test}} \right) \right]^{\frac{1}{n}}$$

$$S(\text{machine A compared to machine B}) = \frac{S(\text{machine A on ref})}{S(\text{machine B on ref})}$$

Typical Reference Run Times

Cint2017 Programs

Program	Language	KLOC	Application	Ref Run Time
600.perlbench_s	C	362	Perl interpreter	1773
602.gcc_s	C	1,304	GNU C compiler	3982
605.mcf_s	C	3	Route planning	4709
620.omnetpp_s	C++	134	Discrete Event simulation - computer network	1630
623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT	1413
625.x264_s	C	96	Video compression	1770
631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)	1434
641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)	1706
648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)	2948
657.xz_s	C	33	General data compression	6188

KLOC = 1000 lines of code

Typical SPEC Report — 1

SPEC(R) CPU2017 Integer Speed Result
ASUSTeK Computer Inc.
ASUS RS700-E9(Z11PP-D24) Server System
(2.70 GHz, Intel Xeon Gold 6150)

CPU2017 License: 9016 — **Test date: Dec-2017**
Test sponsor: ASUSTeK Computer Inc. — **Hardware availability: Jul-2017**
Tested by: ASUSTeK Computer Inc. — **Software availability: Sep-2017**

Base Benchmarks	Base Thrds	Base Run Time	Base Ratio	Peak Thrds	Peak Run Time	Peak Ratio
600.perlbench_s	72	286	6.22	72	239	7.42
602.gcc_s	72	423	9.42	72	413	9.65
605.mcf_s	72	426	11.1	72	421	11.2
620.omnetpp_s	72	257	6.35	72	248	6.58
623.xalancbmk_s	72	150	9.46	72	140	10.1
625.x264_s	72	150	11.8	72	150	11.8
631.deepsjeng_s	72	280	5.11	72	282	5.08
641.leela_s	72	393	4.34	72	392	4.36
648.exchange2_s	72	220	13.4	72	220	13.4
657.xz_s	72	280	22.1	72	277	22.3
SPECspeed2017_int_base			8.87			
SPECspeed2017_int_peak						9.16

Base = standard configuration **Peak = specialist configuration**

Typical SPEC Report — 2

```
                          HARDWARE
                         --------
      CPU Name: Intel Xeon Gold 6150
      Max MHz.: 3700
       Nominal: 2700
       Enabled: 36 cores, 2 chips
     Orderable: 1, 2 chip(s)
      Cache L1: 32 KB I + 32 KB D on chip per core
            L2: 1 MB I+D on chip per core
            L3: 24.75 MB I+D on chip per chip
         Other: None
        Memory: 768 GB (24 x 32 GB 2Rx4 PC4-2666V-R)
       Storage: 1 x 240 GB SATA SSD
         Other: None
                          SOFTWARE
                          --------
            OS: Red Hat Enterprise Linux Server release 7.3 (x86_64)
                Kernel 3.10.0-514.el7.x86_64
      Compiler: C/C++: Version 18.0.0.128 of Intel C/C++ Compiler;
                Fortran: Version 18.0.0.128 of Intel Fortran Compiler
      Parallel: Yes
      Firmware: Version 0601 released Oct-2017
    File System: xfs
   System State: Run level 3 (multi-user)
  Base Pointers: 64-bit
  Peak Pointers: 32/64-bit
         Other: jemalloc: jemalloc memory allocator library V5.0.1
```

Some Cint2017 Results

Processor	Clock (GHz)	Total Chips	Total Cores	Total Threads	Cint 2017 Base
Intel Xeon E-2278G (Lenovo)	3.4	1	8	16	13.2
Intel Xeon E-2278G (Asus)	3.4	1	8	8	12.4
Intel Xeon E-2278G (Fujitsu)	3.4	1	8	16	12.1
Intel Xeon Gold 6250 (Lenovo)	3.9	2	16	32	12.6
Intel Xeon Gold 6256 (Lenovo)	3.6	2	24	48	12.6
Intel Core i9-9900K	3.6	1	6	6	11.2
Intel Core i7-9700K	3.6	1	8	8	10.6

Some Comments on Cint2017 Results

Auto parallel

High level Cint code not explicitly threaded for parallel processing

Auto parallel compiler creates parallel threads using heuristics

Provides limited speed up (or even degradation)

Intel Xeon E-2278G with 3.4 GHz clock

Fastest CPU in Cint2017 tests

16 threads slightly faster than 8 threads

- Communication between more threads can slow processing

16 threads from different manufacturer slightly slower

Intel Xeon Gold 6256 with 3.6 GHz clock

Cint with 48 threads slightly slower than E-2278G

With 3.6 GHz clock, expect 3.6 GHz / 3.4 GHz = 1.06 faster

48 threads slower than 16 threads

Some Cfp2017 Results

Processor	Clock (GHz)	Total Chips	Total Cores	Total Threads	Cfp 2017 Base	Cint 2017 Base
Intel Xeon Platinum 8280L	2.7	8	224	224	244	10.5
Intel Xeon Platinum 8280L	2.7	4	112	112	232	10.5
Intel Xeon Platinum 8156	3.6	2	8	8	53.8	8.26
Intel Core i9-9900K	3.6	1	6	6	33.2	11.2
Intel Core i7-9700K	3.6	1	8	8	32.3	10.6

Actual Sources of Performance Improvement

1978 Clock speed of 8086 is 4 MHz

2008 Xeon (clock speed of 4 GHz) is 100,000 times faster

- Clock speedup = 4 GHz / 4 MHz = 1000
- Structural speedup = 100,000 / 1000 = 100
  - Reducing waiting time between operations
  - Performing operations in parallel

No more clock speedup

- Pentium 4 clock rate (4 GHz) = 4 x Pentium III clock (1 GHz)
- Clock speedup 1 GHz $\rightarrow$ 4 GHz required structural slowdown
  - Pentium 4 at 1 GHz slower than Pentium III at 1 GHz
  - Run Pentium III at 4 GHz $\Rightarrow$ melt CPU
- Clock speed $\rightarrow$ physical limit of about 10 GHz
  - Signal takes clock cycle to cross Pentium 4 at speed of light
- Future speedup comes from structural improvements
  - More cores
  - Better architectures

Instruction Set Architecture

Choosing Ingredients for a Computer Design

סקירת הפרק

מהו מעבד

- מבנה von Neumann
- שלבי תכנון מעבד

קבוצת הפקודות

- מבנה פקודה
- אופרנדים (נתונים)
- שמירת נתונים וסוגי זכרון
- פעולות

שיקולים לתכנון קבוצת הפקודות

- שפה מורכבת (CISC)

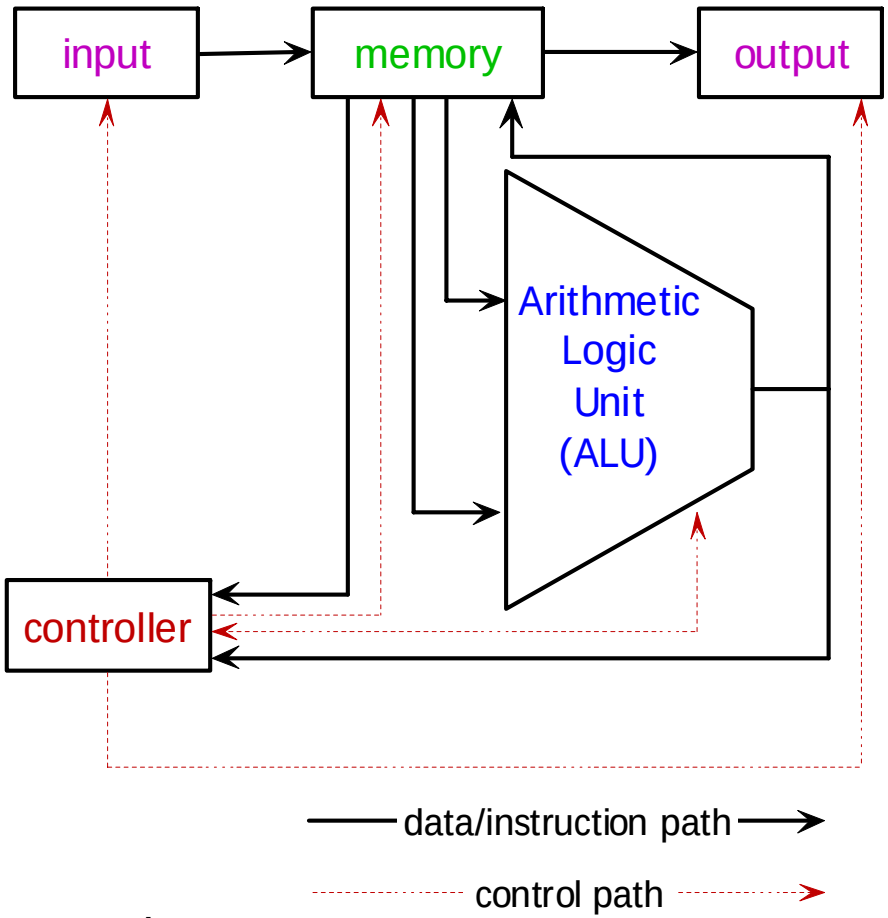
מימוש פקודות בחומרה

- microcode

Von Neumann Architecture

Stored-Program Digital Computer

- Digital computation in **ALU**
- **Programmable** via set of standard instructions
- Internal **storage** of data
- Internal **storage** of program
- Automatic **Input/Output**
- Automatic **sequencing** of instruction execution by decoder/controller



Von Neumann Architecture

Data and instructions stored in a single memory unit

Harvard Architecture

Data and instructions stored in a separate memory units

Stages in Computer Design

Instruction Set Architecture (ISA)

1. Look at universe of problems to be solved
2. Define atomic operations at level of system programmer
   - Small and orthogonal operations (each performs different task)
   - Can be combined to perform any operation
3. Specify instruction set for machine language
   - Choose a minimum set of basic operations
   - Not too many ways to solve same problem

Implementation

1. Design machine as implementation of ISA
2. Evaluate theoretical performance
3. Identify performance problem areas
4. Improve processor efficiency

Instruction Features

Instruction

Description of an **Operation** performed on **Operands**

Operations

Actions performed on data

Operands

Sources — data inputs to operation

Destinations — data outputs from operation

Specified by

Addressing Mode — location of data in machine

Data Type —Integer, Long, Floating Point, Decimal, String, Constant, etc.

Instruction Set Architecture

General Instruction is instance of data structure



Machine Language is range of data structure Instruction for

Operation ∈ {legal actions}

Operand ∈ {legal Addressing Modes}

Describe sources and destinations

Typical machine instruction

```
ADD destination, source_1, source_2
destination ← source_1 + source_2
```

Instruction Definitions

Operations and operands

unary — one source operand

binary — two source operands

n-ary — n source operands

Address specifier

Describes address format

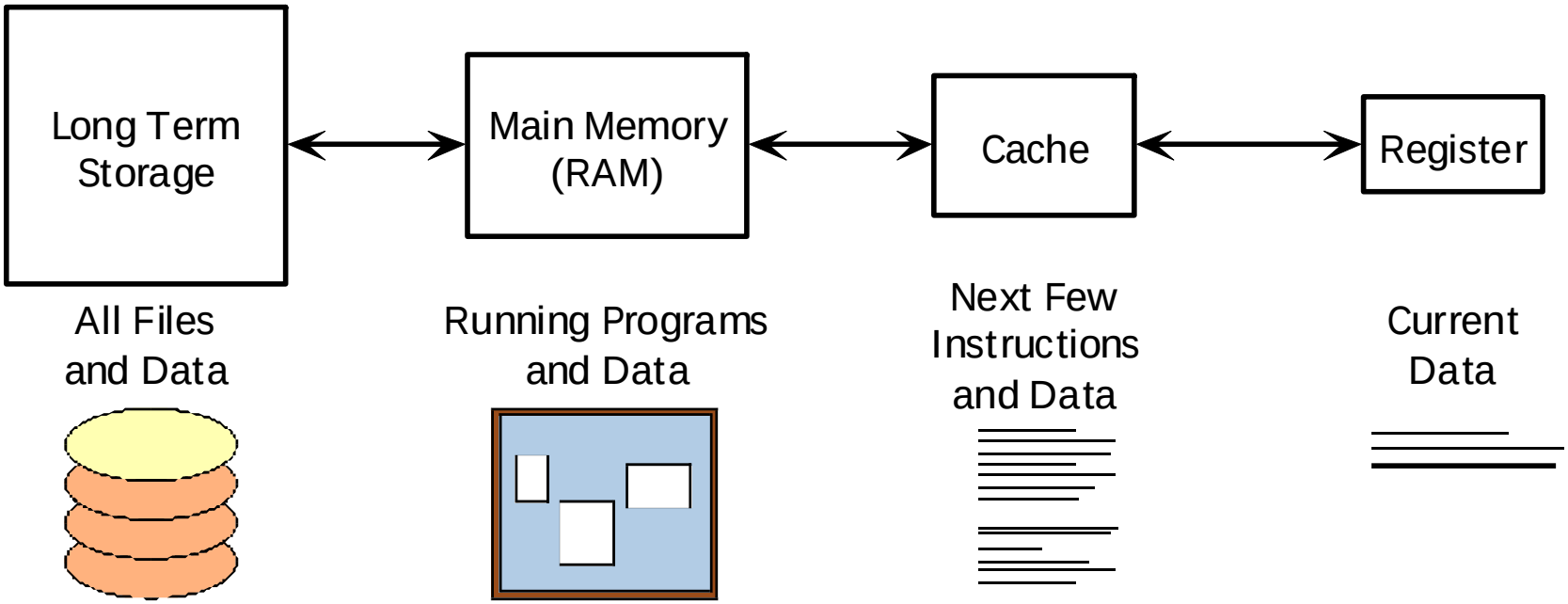
- Addressing mode
- Operation model

Data width

	Intel	Non-Intel
2 bytes	word	half-word
4 bytes	dword	word
8 bytes	quadword	doubleword

Memory Hierarchy

Memory locations outside CPU and RAM	Memory location outside CPU	Memory location in or near CPU	Memory location inside CPU
Stores data and instructions of "all" programs	Stores "all" data and instructions of running programs	Fast access to important data and instructions from RAM	Fast access to small amount of information
Organized by OS	Organized by addresses	Copy of RAM section	Organized by CPU



Register Naming

Registers are part of CPU design

- Information stored in registers called architectural state
- Describes machine status and program status

General Purpose (GP) registers

- Hold data for instructions
  - Width of data is width of standard integer in CPU
- Referenced by names or numbers
  - Intel x86: **`EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP`**
  - General: **`R0, R1, … , R127`**

Special Purpose registers

- Machine status registers
- Operating system registers

Flat Memory Organization

N-bit address space

Physical Address = $A_{N-1}\, A_{N-2} \ldots A_1\, A_0$

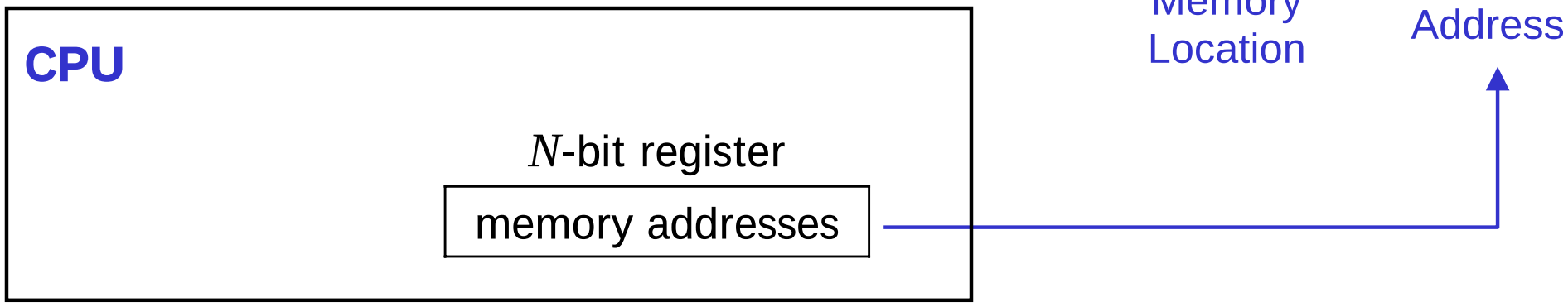
Can form 2^N addresses, from 0 to $(2^N - 1)$

Every byte in RAM has N-bit address

Processor refers to memory locations by physical RAM addresses

Processor stores memory addresses in N-bit address registers

Memory Location	Address
Data Byte	11111…111
Data Byte	11111…110
Data Byte	11111…101
Data Byte	11111…100
…	…
Data Byte	00000…111
Data Byte	00000…110
Data Byte	00000…101
Data Byte	00000…100
Data Byte	00000…011
Data Byte	00000…010
Data Byte	00000…001
Data Byte	00000…000



Word Organization in Memory

Word order

Little endian

- Least significant byte stored at lower address
- Word is stored "little-end-first"
- Example: 4-byte word **`69 b3 36 7d`** stored as

stored byte					**`69`**	**`b3`**	**`36`**	**`7d`**
address	**`07`**	**`06`**	**`05`**	**`04`**	**`03`**	**`02`**	**`01`**	**`00`**

Big endian

- Most significant byte stored at lower address
- Word is stored "big-end-first"
- Example: 4-byte word **`69 b3 36 7d`** stored as

stored byte					**`7d`**	**`36`**	**`b3`**	**`69`**
address	**`07`**	**`06`**	**`05`**	**`04`**	**`03`**	**`02`**	**`01`**	**`00`**

Alignment

Requirement that address of **`s`**-byte data unit be multiple of **`s`**

Formally — **`address A % s = 0`**

- 8086 requires segments to be aligned on 16-byte boundaries
- IA-32 requires pages to be aligned on 4 KB boundaries

Specifying Operands

Immediate

Constant = IMM = numerical value coded into instruction

Register operands

register name = a CPU storage location

REGS[register name] = data stored in register

REGS[R3] = data stored in register **R3 = 11223340**

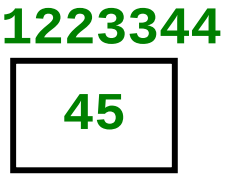


Memory operands

address = a memory storage location

MEM[address] = data stored in memory

MEM[11223344] = data stored at address **11223344** = **45**



Effective Address (EA) — pointer arithmetic

REGS[R3] ← &(variable)

MEM[REGS[R3]+4] = *(&(variable)+4) = *(REGS[R3]+4)

= *(11223340+4) = 45

Structured Operation Models

Defines basic arithmetic procedure and ALU organization

Stack

Push	Pointer ← Pointer – d Stack[Pointer] ← memory/register
Pop	memory/register ← Stack[Pointer] Pointer ← Pointer + d
Binary Op	Stack[Pointer + d] ← Stack[Pointer + d] Op Stack[Pointer] Pointer ← Pointer + d

```
Z = X + Y →     push X
                push Y
                ADD
                pop Z
```

Stack ALU used in Java bytecode

Accumulator

All operations use accumulator **A**

```
Z = X + Y →     load X
                add Y
                store Z
```

Accumulator ALU used in hand calculator

General Register Operation Models

Register-Memory Model

Operands can be stored in any **REGISTER** or **MEMORY** location

```
Z = X + Y →     load R1, X
                add R1, R1, Y
                store Z, R1
```

Register- Register Model

MEMORY operands must be loaded to a **REGISTER**

Also called **LOAD-STORE MODEL**

```
Z = X + Y →     load R1, X
                load R2, Y
                add R1, R1, R2
                store Z, R1
```

Easier to implement

Statistically, most loaded operands are used more than once

Typical Addressing Modes

Mode	Syntax	Memory Access	Use
Register	`R3`	`Regs[R3]`	Register data
Immediate	`#3`	`3`	Constant
Direct (absolute)	`(1001)`	`Mem[1001]`	Static data
Register deferred	`(R1)`	`Mem[Regs[R1]]`	Pointer
Displacement	`100(R1)`	`Mem[100+Regs[R1]]`	Local variable
Indexed	`(R1 + R2)`	`Mem[Regs[R1]+Regs[R2]]`	Array addressing
Memory indirect	`@(R3)`	`Mem[Mem[Regs[R3]]]`	Pointer to pointer
Auto Increment	`(R2)+`	`Mem[Regs[R2]]` `Regs[R2] ← Regs[R2]+d`	Stack access
Auto Decrement	`-(R2)`	`Regs[R2] ← Regs[R2]-d` `Mem[Regs[R2]]`	Stack access
Scaled	`100(R2)[R3]`	`Mem[100+Regs[R2]+Regs[R3]*d]`	Indexing arrays
PC-relative	`(PC)`	`Mem[PC+value]`	Store data relative to program counter (instruction address)
PC-relative deferred	`1001(PC)`	`Mem[PC+Mem[1001]]`	

Typical Operations

Data transfer

Load (r ← m), store (m ← r), move (r/m ← r/m), convert data types

Arithmetic/Logical (ALU)

Integer arithmetic (+ − × ÷ compare shift) and logical (AND, OR, NOR, XOR)

Decimal

Integer arithmetic on decimal numbers

Floating point (FPU)

Floating point arithmetic (+ − × ÷ sqrt trig exp ...)

String

String move, string compare, string search

Control

Conditional and unconditional branch, call/return, trap

Operating System

System calls, virtual memory management instructions

Graphics

Pixel operations, compression/decompression operations

Classic Computer Organization

Considerations in Classic Computer Design

Expensive memory

RAM ~ $5000/MB wholesale in 1977

Poor compilers

Non-optimizing

Bad error messaging

Fast code written or optimized in assembly language

Semantic Gap Argument

Belief among theoreticians in 1960s and 1970

Computer language should imitate natural language

- Large vocabulary
- High redundancy

Implications for Machine Language

Machine Language should be high level

Language defines many instructions

Each instruction performs a lot of work

Language defines many addressing modes

Advantages

Assembly language programming is easier

Each stored instruction in memory more powerful

More power per instruction requires less memory

Classic Machine Design

CISC (Complex Instruction Set Computer)

300+ instruction types

15+ addressing modes

10+ data types

Automated procedure handling

Complex machine implementations

CISC

CISC was conventional wisdom in 1960s and 1970s

Mainframes

Large and expensive computers

Owned by big businesses and governments

Manufacturers: IBM, Control Data, Burrows, Honeywell

From 1960s to 1980s, mainframes were CISC machines

Minicomputers

Smaller computers for smaller organizations

Manufacturers: Digital (PDP/VAX), Data General (Eclipse)

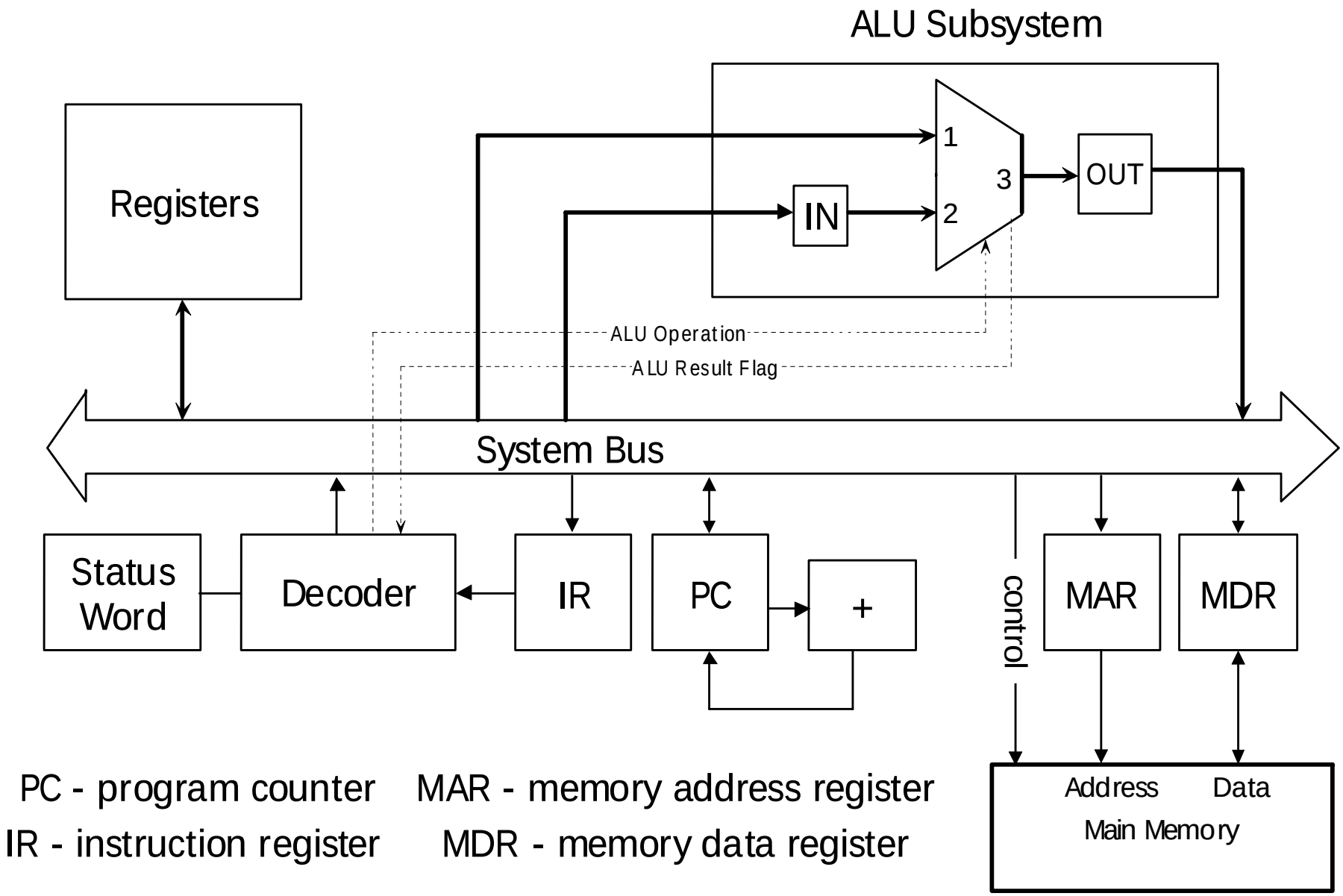
Promoted academic computer science, smaller operating systems (Unix), computer networking

Microcomputers

Intel designed the 8086 (1979) to work like a tiny VAX

The PC is the only CISC computer still manufactured

Physical Implementation



Registers

General Registers

R_0 ... R_{n-1}

Register width is standard integer in ISA

PC

Program Counter

Holds address of next instruction to execute

IR

Instruction Register

Holds binary code of instruction being executed

MAR

Memory Address Register

Holds physical address for RAM access

MDR

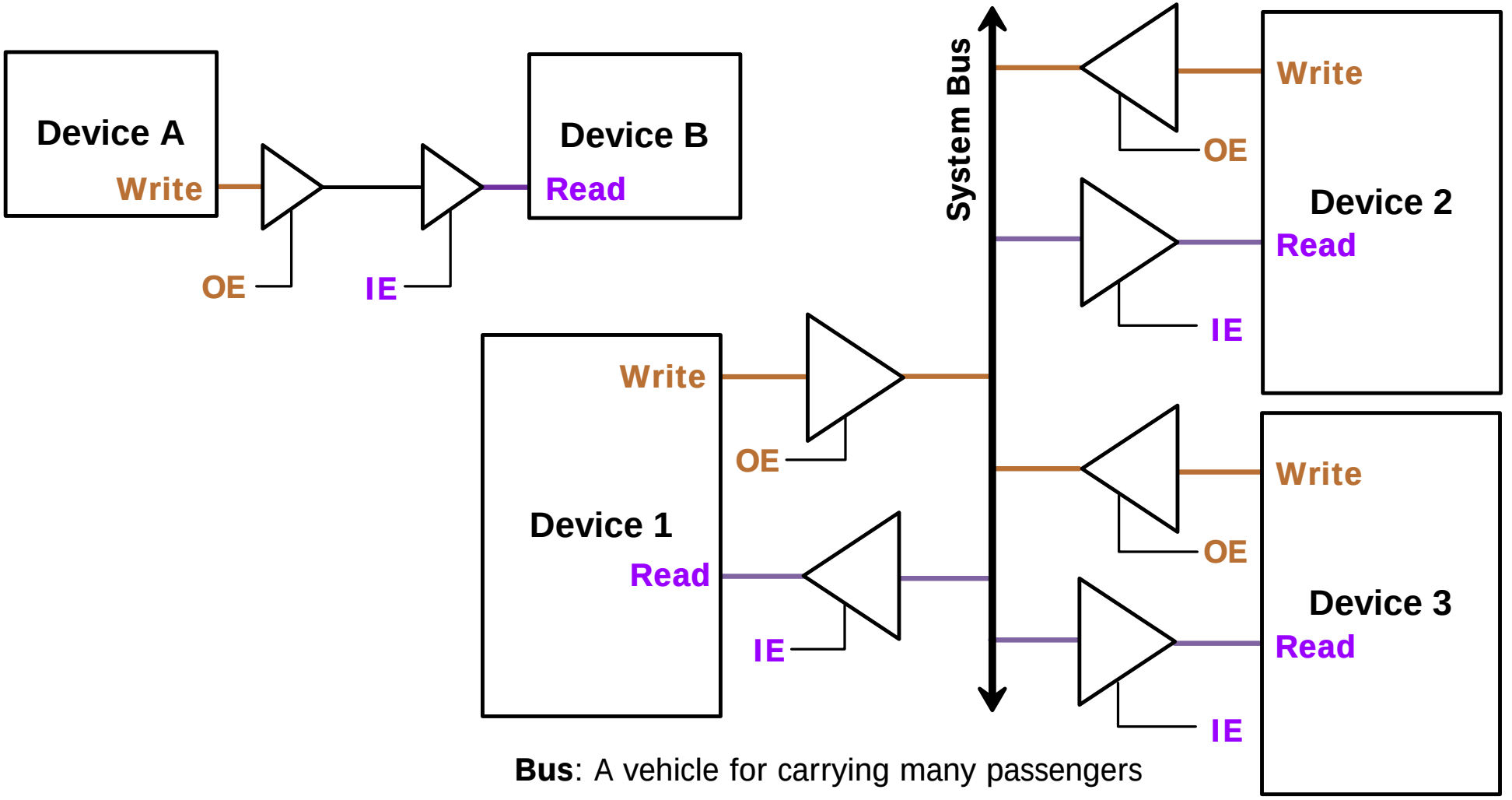
Memory Data Register

Holds data during Read/Write memory operations

Device Communication

A device WRITES with OE = 1 and READS with IE = 1

Von Neumann controller distributes OE and IE signals to devices



Bus: A vehicle for carrying many passengers

Atomic Operations — Instruction Fetch

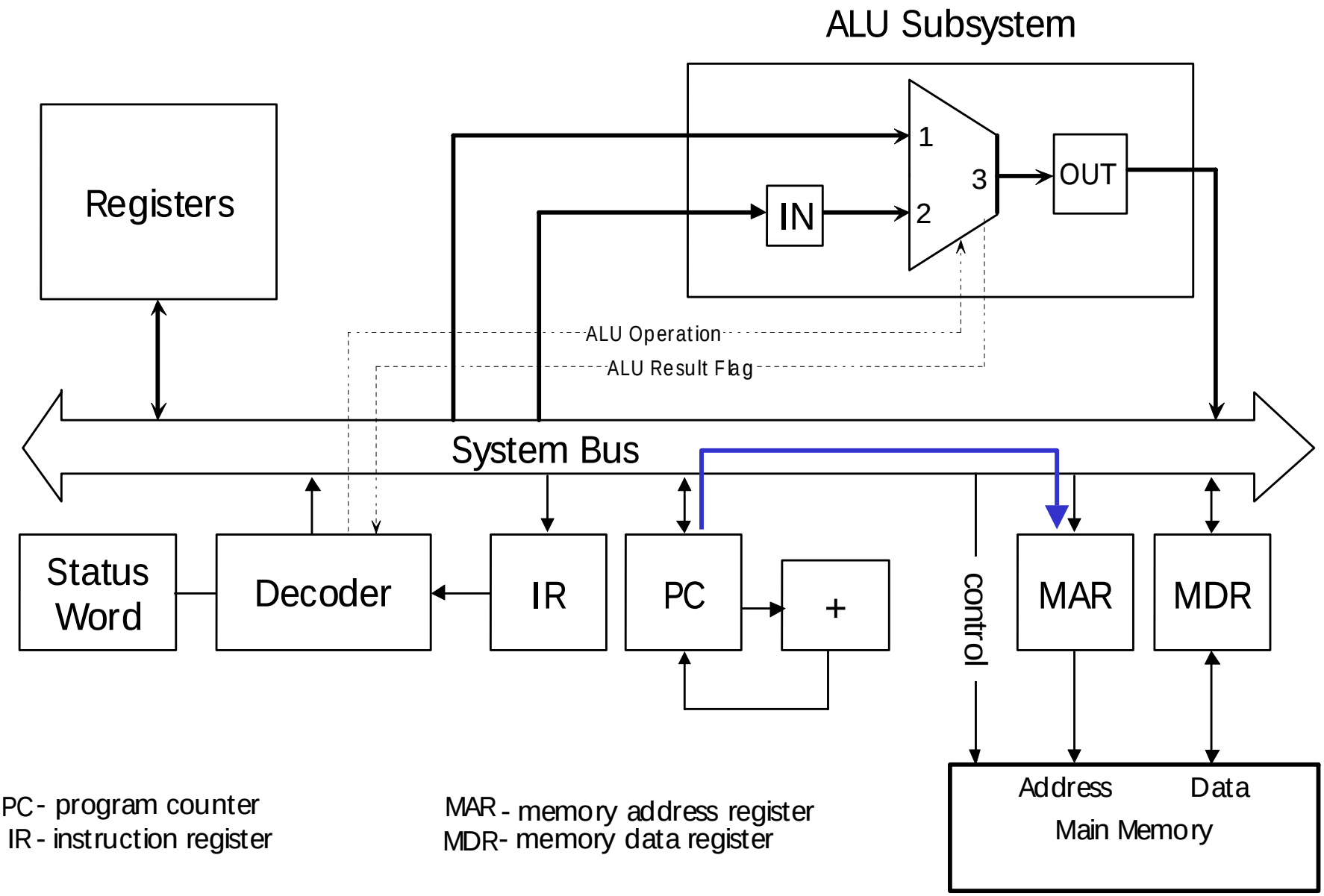
(1) MAR ← PC

(2) READ

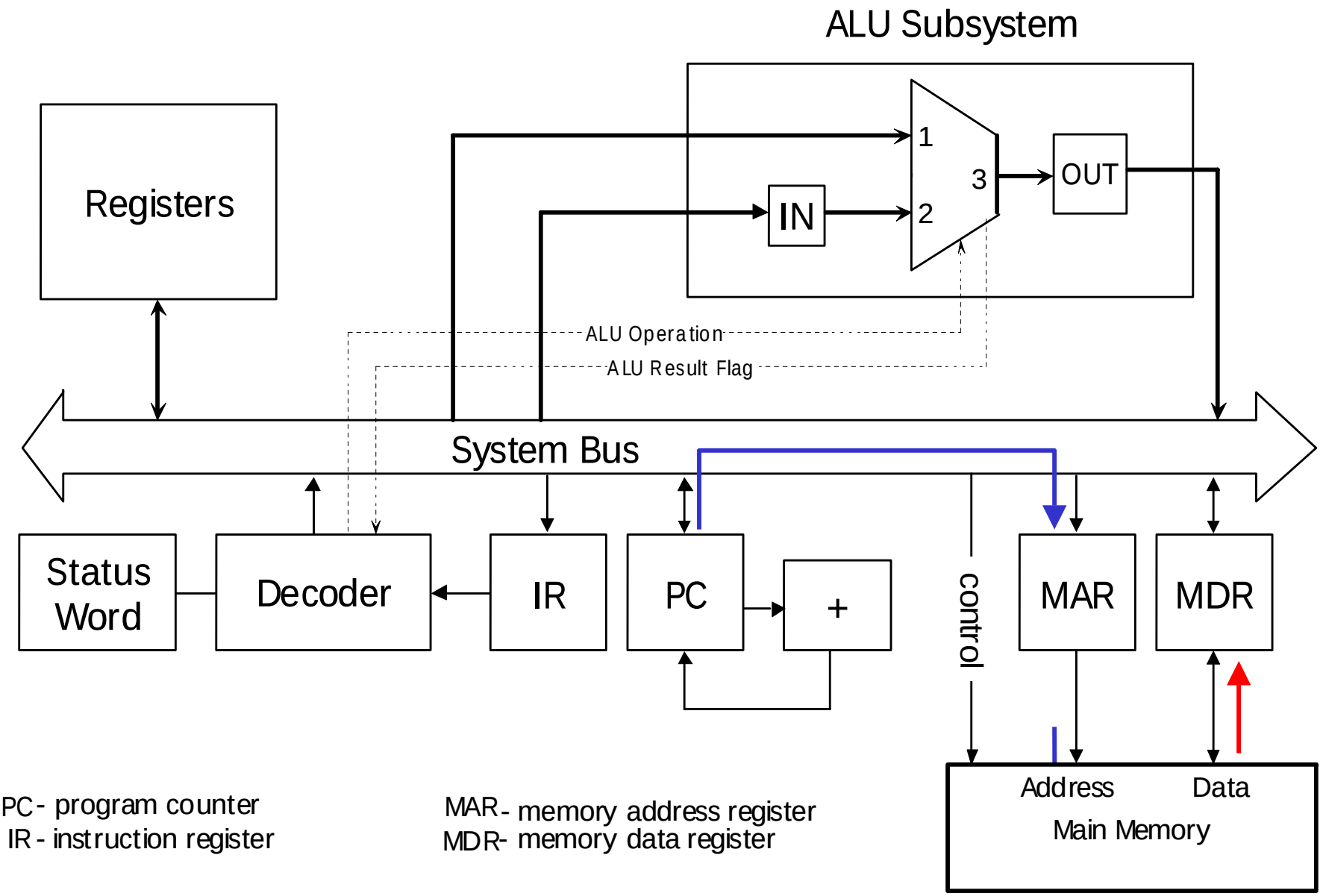
(3) IR ← MDR

(4) PC ← PC + length(instruction)

(1) MAR ← PC


ALU Subsystem
Registers
1
2
3
IN
OUT
ALU Operation
ALU Result Flag
System Bus
Status Word
Decoder
IR
PC
+
control
MAR
MDR
Address
Data
Main Memory
PC - program counter
IR - instruction register
MAR - memory address register
MDR - memory data register

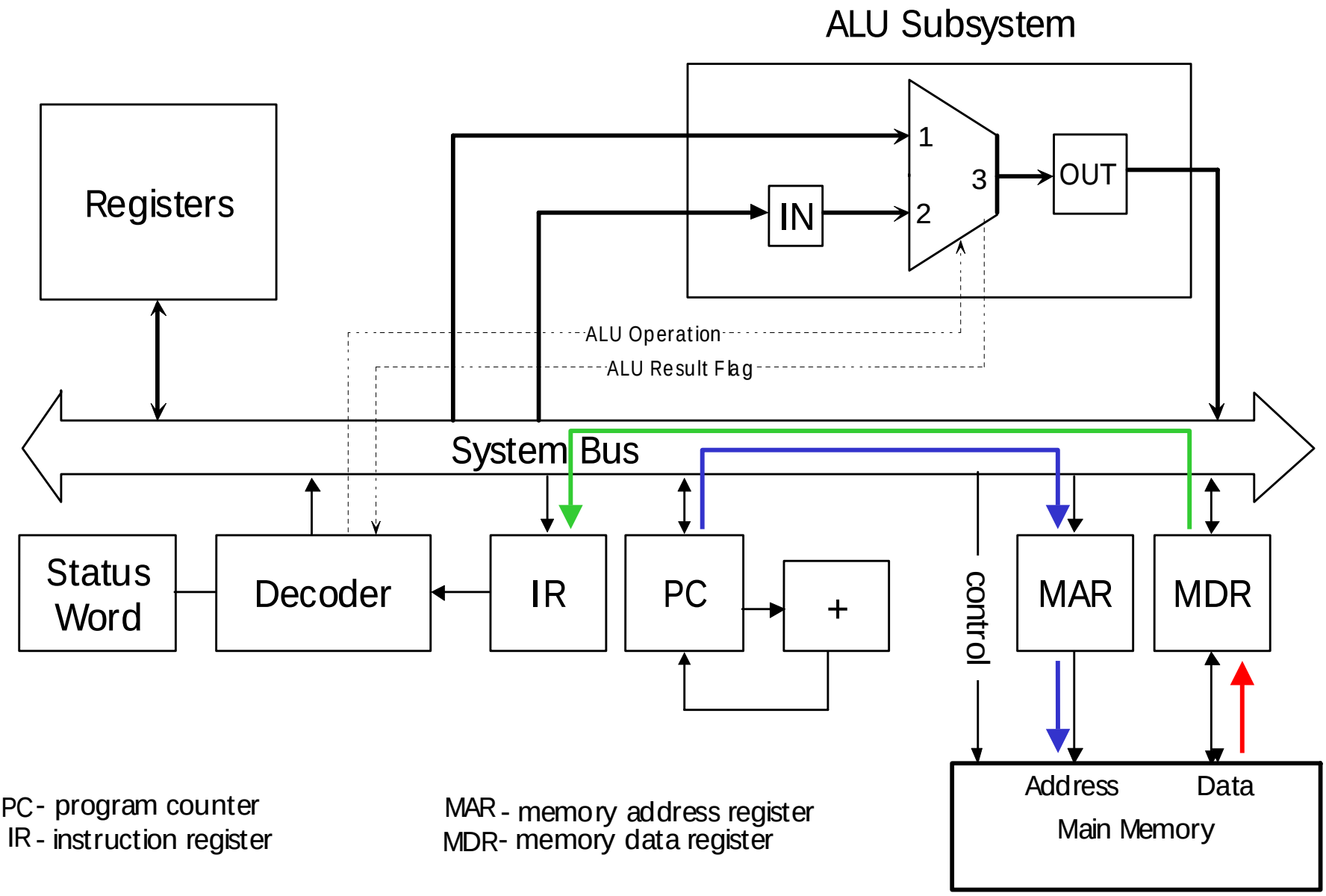
(2) READ



PC - program counter
IR - instruction register

MAR - memory address register
MDR - memory data register

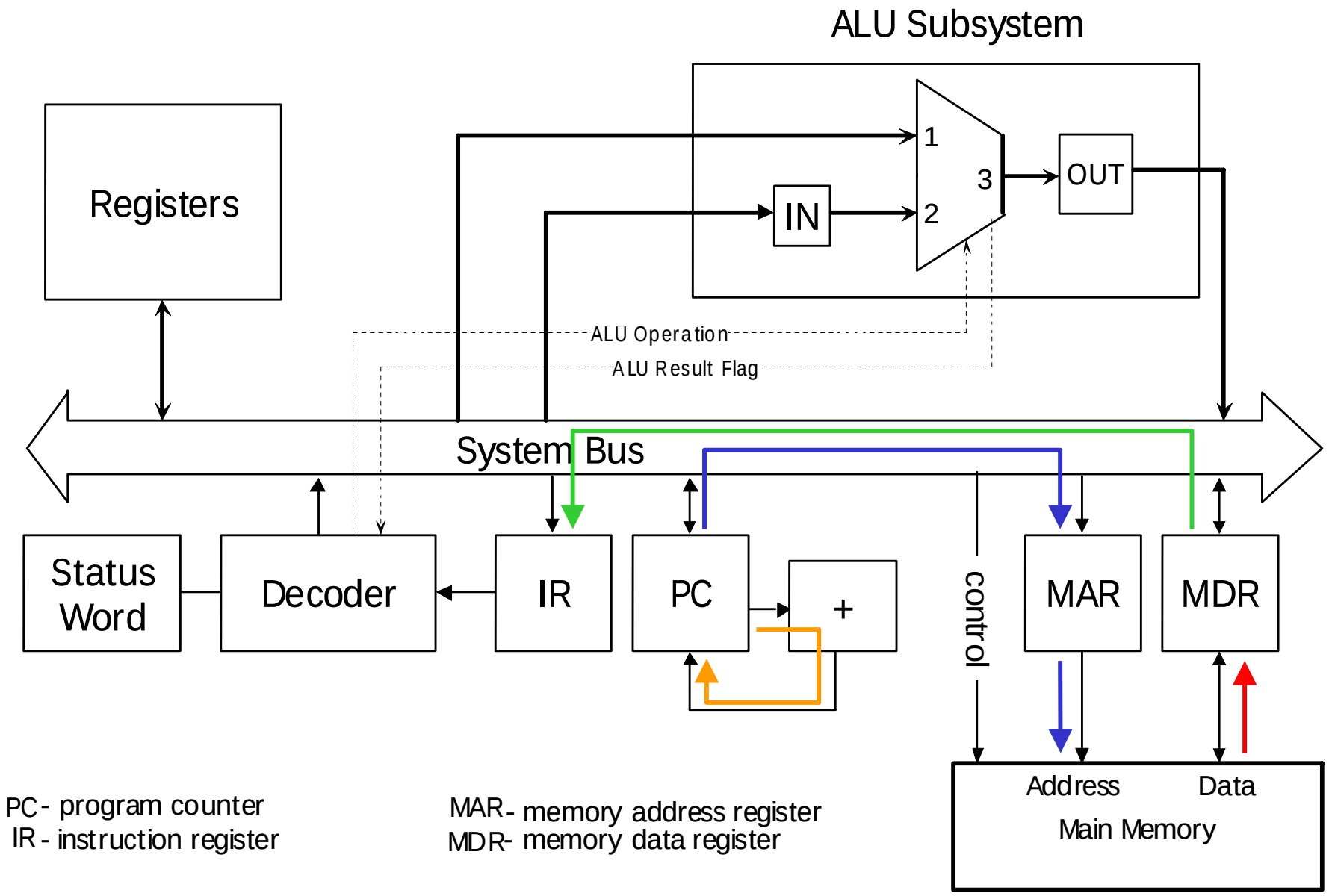
(3) IR ← MDR



PC - program counter
IR - instruction register

MAR - memory address register
MDR - memory data register

(4) PC ← PC + length(instruction)

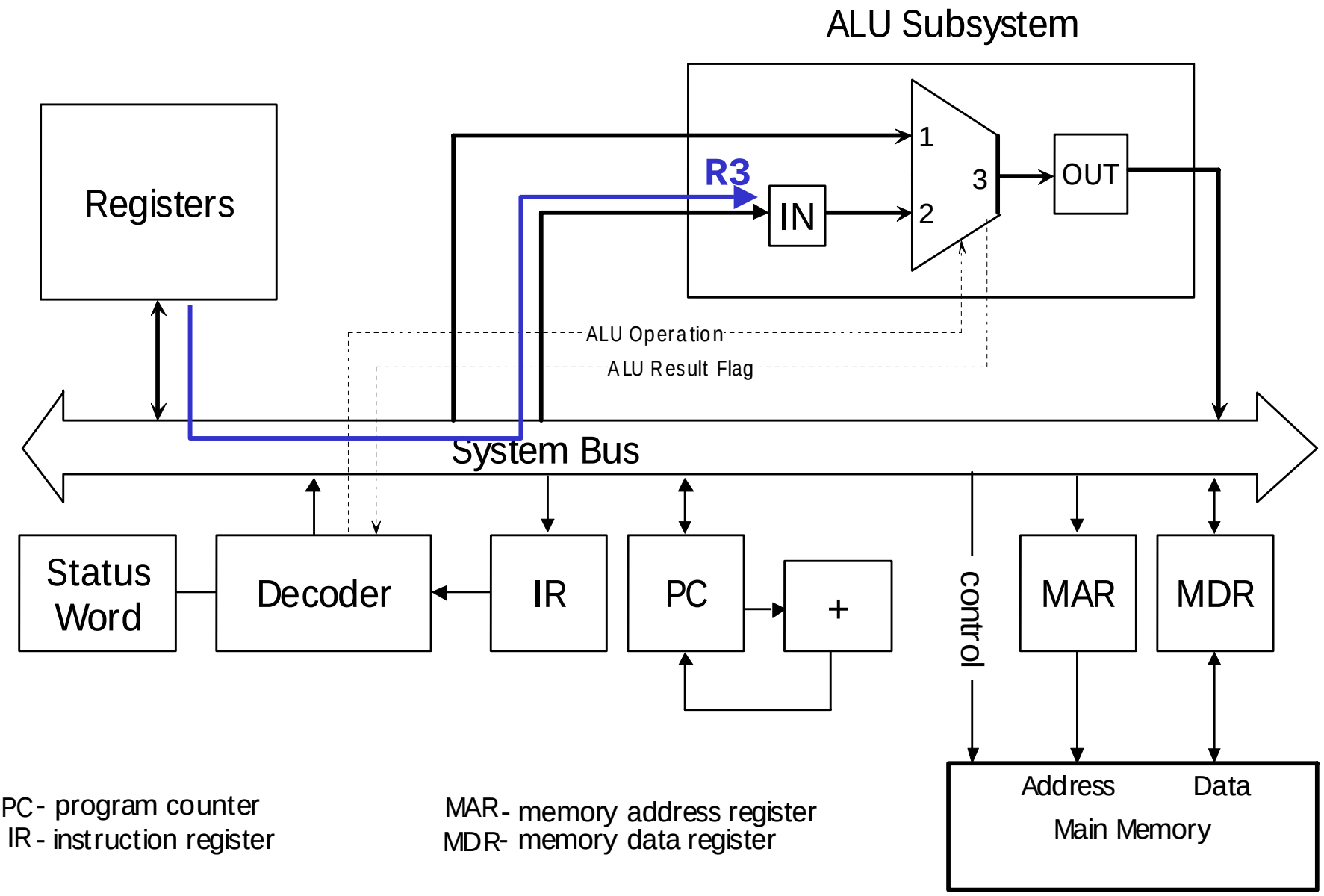


Atomic Operations

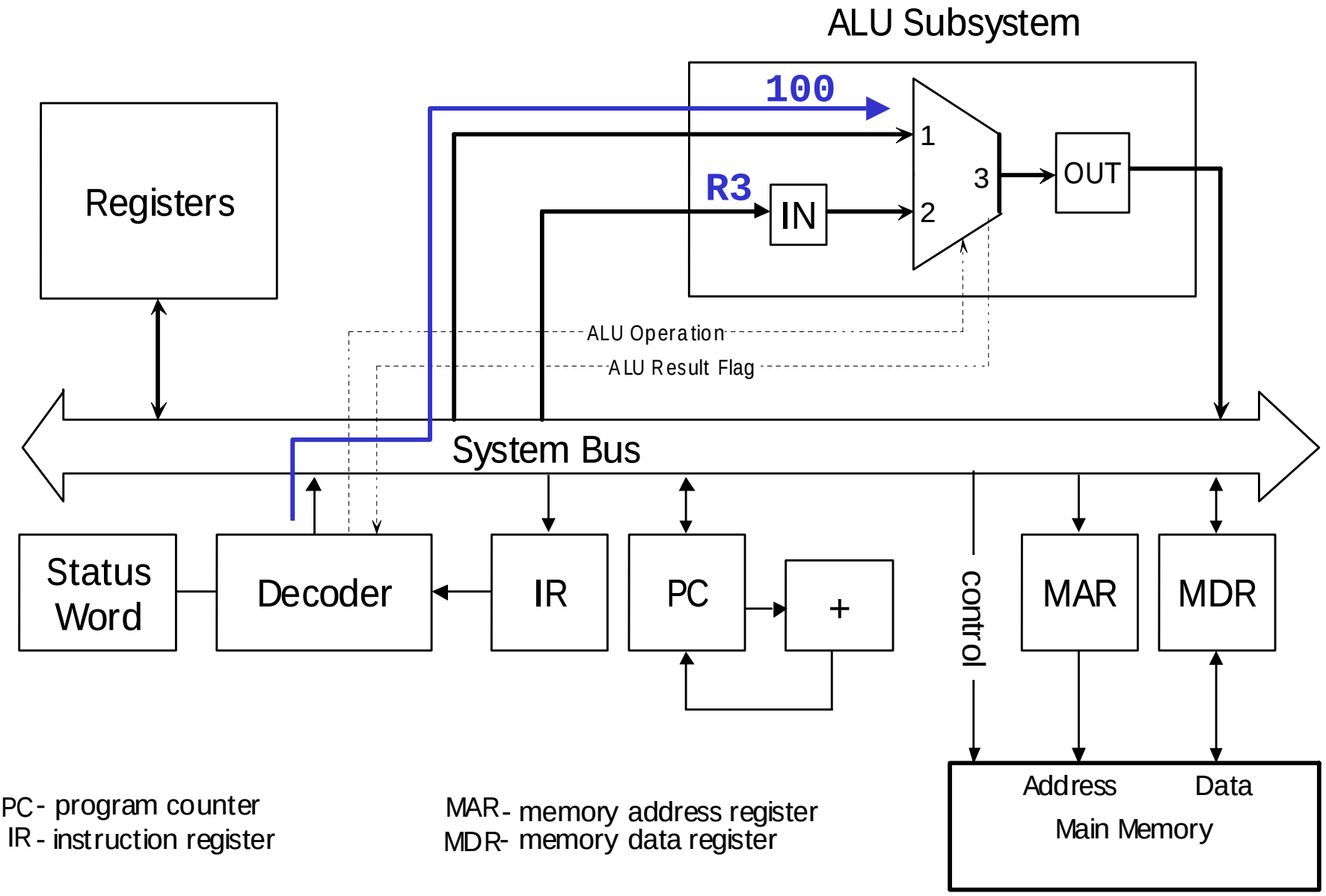
Instruction: `SUB R1, R2, 100(R3)`

```
ALU_IN ← R3
ALU ← 100
ADD
MAR ← OUT
READ
ALU_IN ← MDR
ALU ← R2
SUB
R1 ← OUT
```

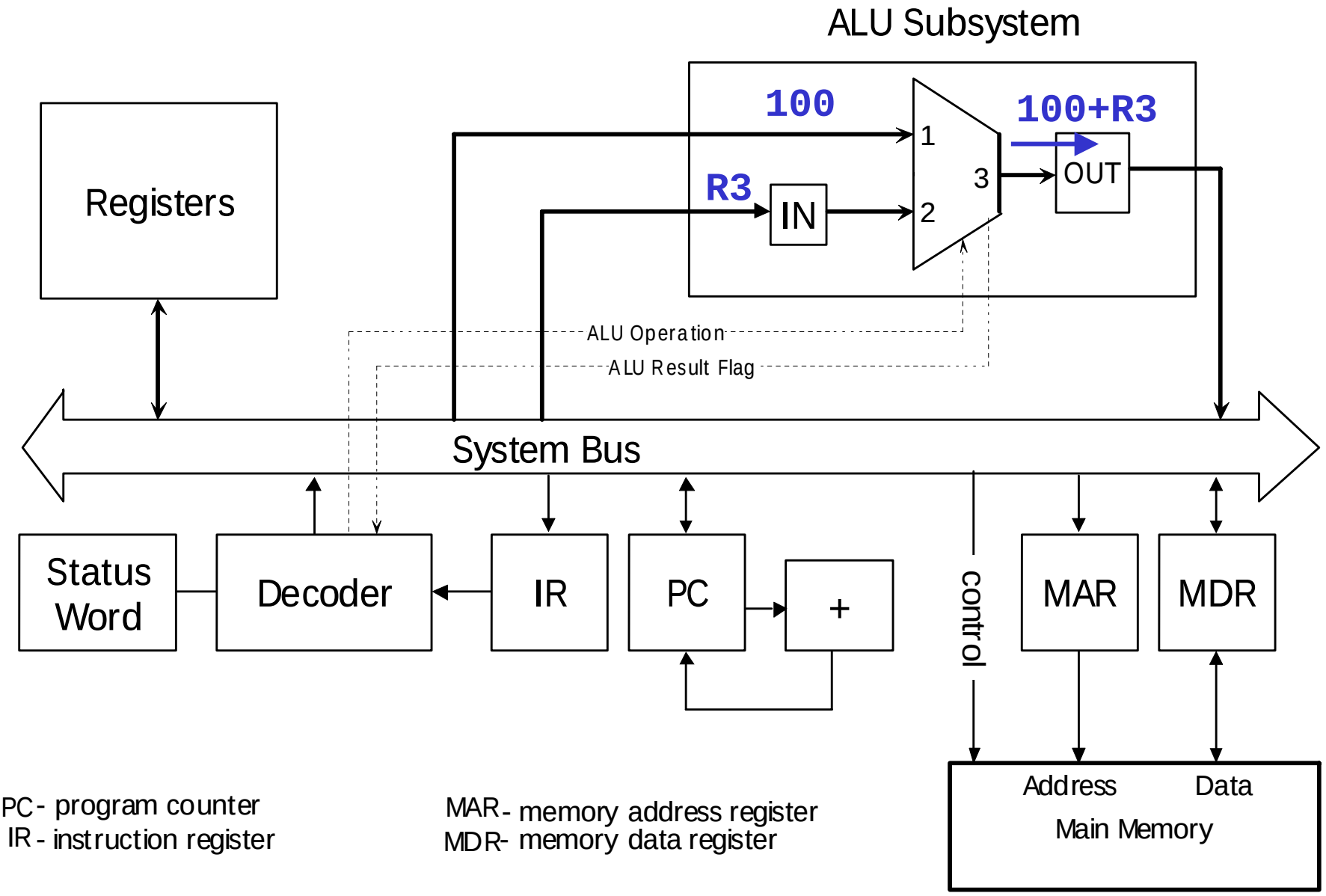
SUB R1, R2, 100(R3): ALU_IN ← R3



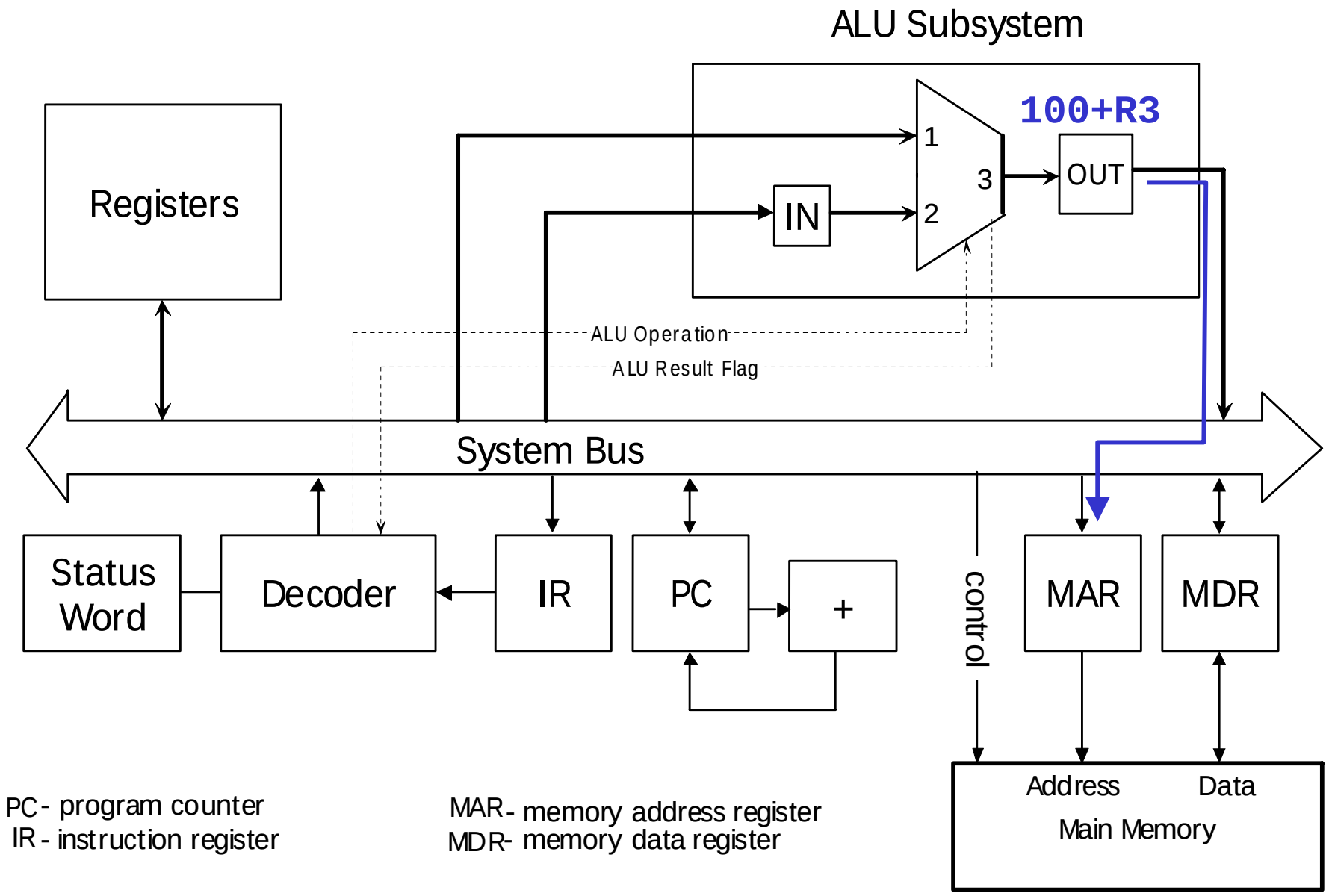
SUB R1, R2, 100(R3): ALU ← 100



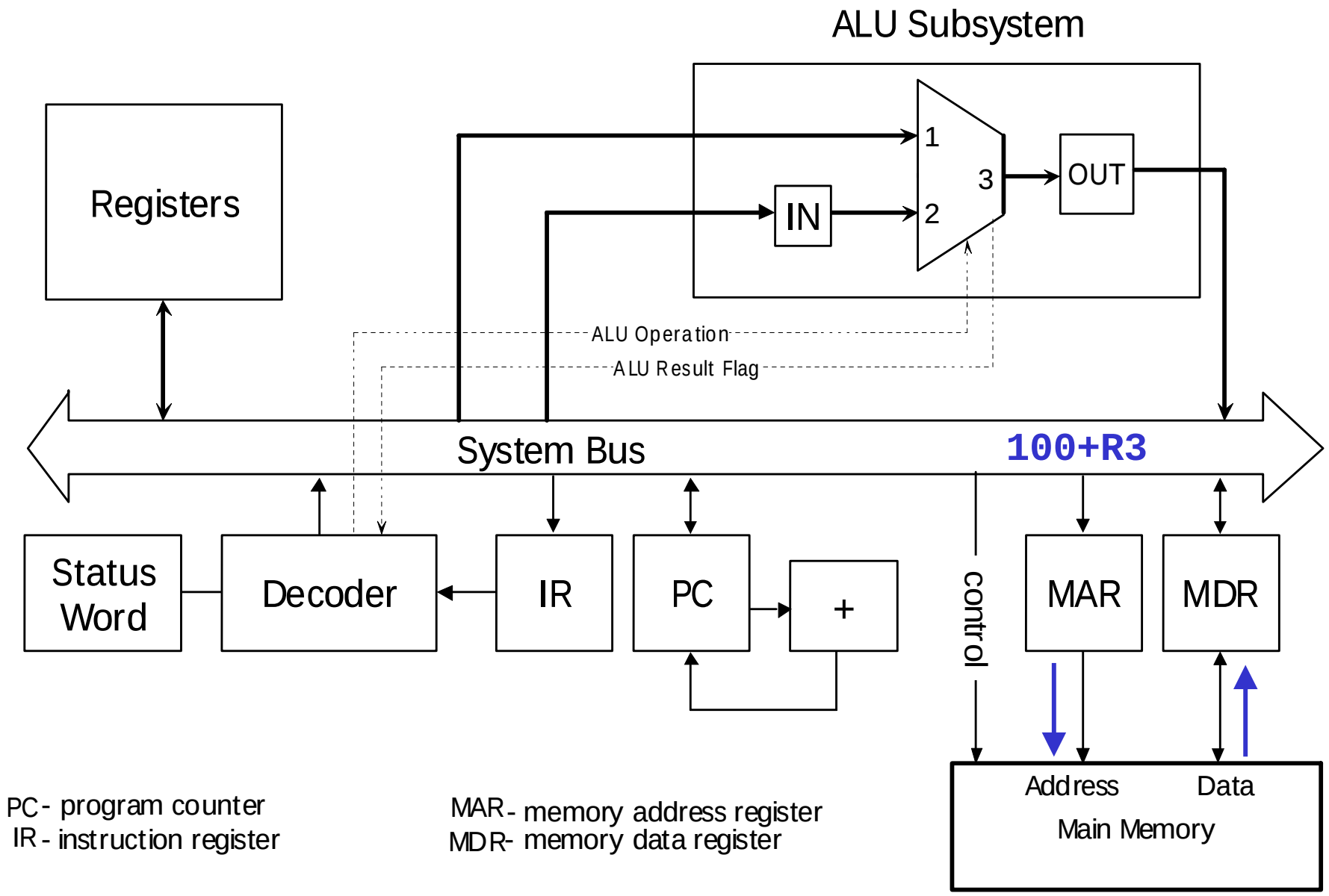
SUB R1, R2, 100(R3): ADD



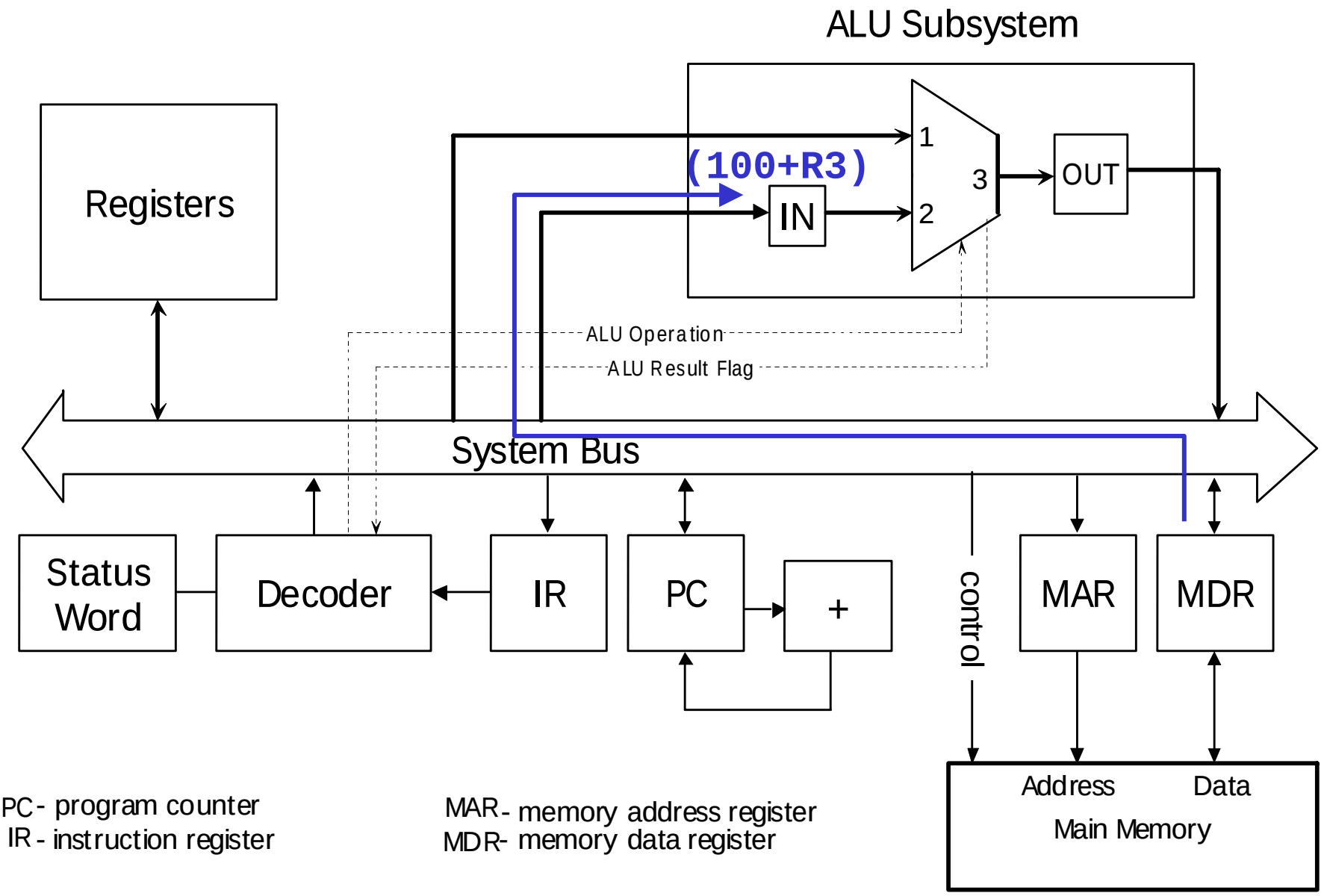
SUB R1, R2, 100(R3): MAR ← OUT



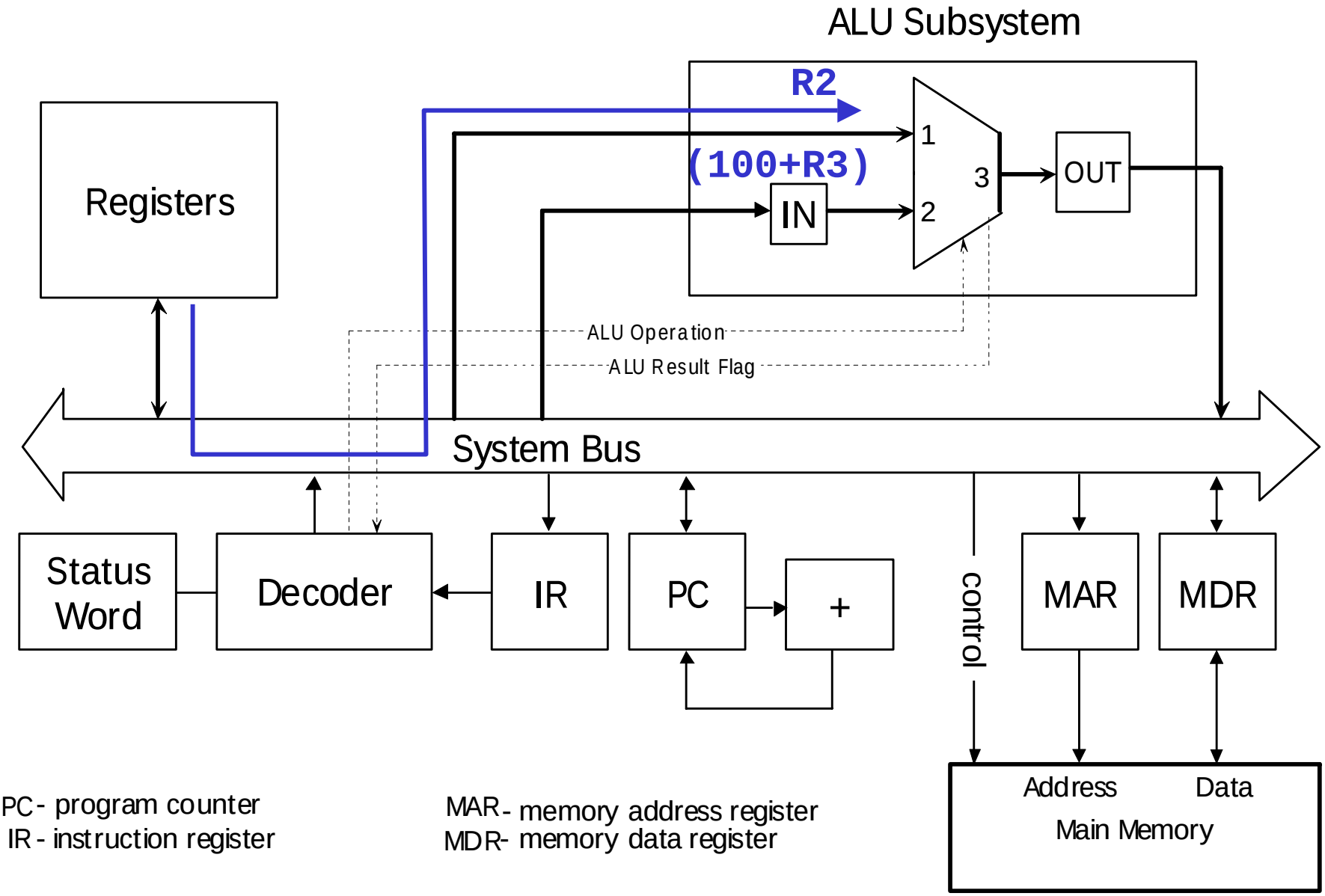
SUB R1, R2, 100(R3): READ



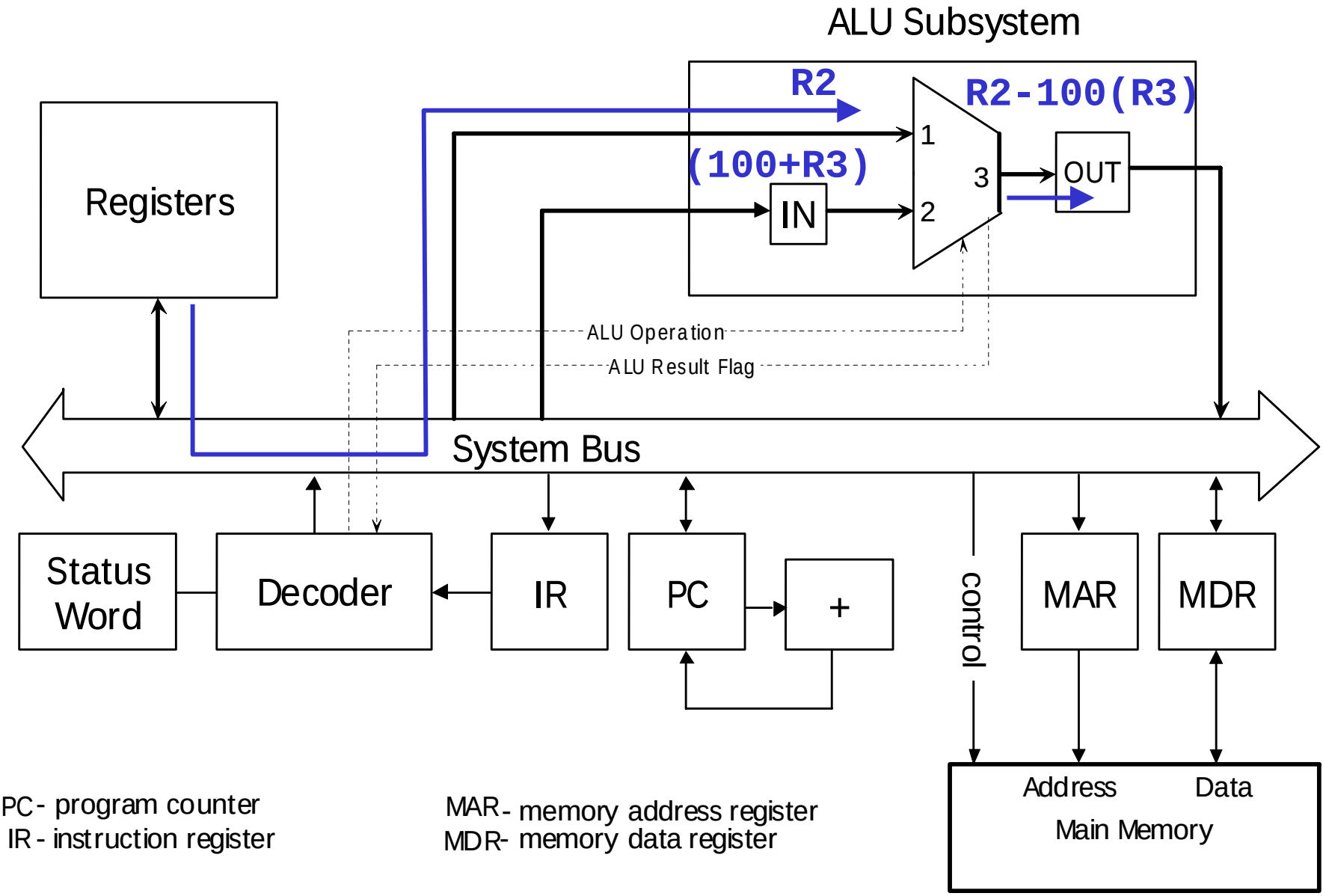
SUB R1, R2, 100(R3): ALU_IN ← MDR



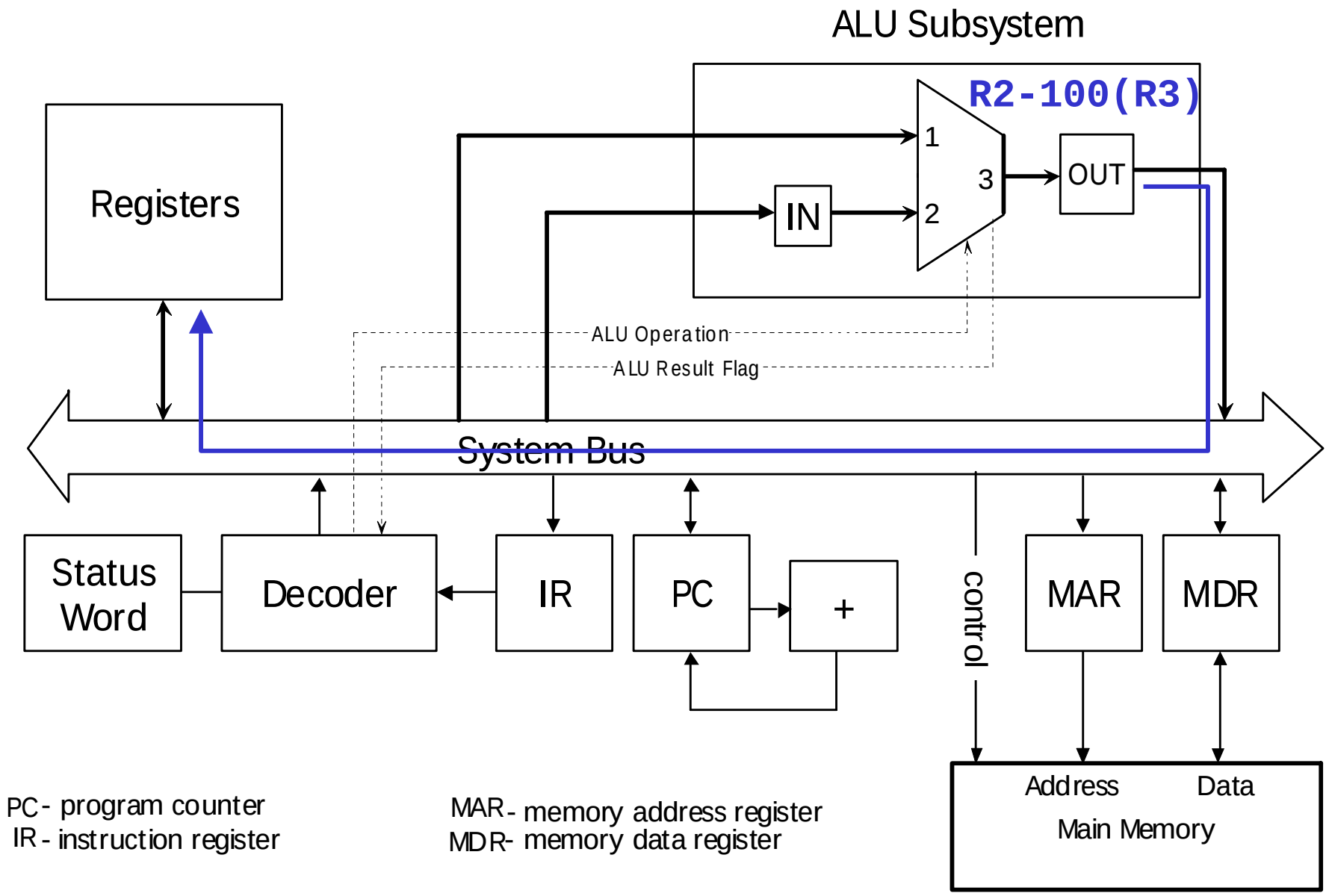
SUB R1, R2, 100(R3): ALU ← R2



SUB R1, R2, 100(R3): SUB



SUB R1, R2, 100(R3): R1 ← OUT



Decoding Machine Instructions

Machine Language Instruction

```
SUB R1, R2, 100(R3)
```

Microcode Instruction Sequence (Microprogram)

```
ALU_IN ← R3
ALU ← 100
ADD
MAR ← OUT
READ
ALU_IN ← MDR
ALU ← R2
SUB
R1 ← OUT
```

Microcode

Microcode

One line of microprogram

Implementation-level atomic operation

Atomic $\Rightarrow$ operation must complete before servicing interrupt

Decoder

"Interprets" machine language instruction into microprogram

Decoder ROM stores microprogram for every legal instruction

New instruction $\Rightarrow$ add microprogram to decoder

Microprogram is sequenced by decoder

State machine for each instruction

Each state provides control signals to every subsystem

Each line of microcode is executed in the correct order

Based on work of Maurice V. Wilkes (1951)

Clock Cycles Per Instruction

Clock Cycle (CC)

Determined by length of longest microcode operations

One line of microcode finishes before next line begins

- Most microcode lines finish in one clock cycle
- Memory access may take several clock cycles

Clock Cycles Per Instruction

Machine language instruction implemented as lines of microcode

Clock Cycles Per Instruction = number of microcode lines

- Memory accesses may take extra clock cycles

Clock cycles for program = number of microcode lines in program

$$CC^{\text{program}} = \sum_{i=\text{instruction types}} \left(\text{Instructions of type } i\right) \times \left(CC^{\text{Instruction of type } i}\right)$$

Instruction type — same basic microcode structure

CISC Creates Anti-CISC Revolution

1974 — 1977

- Data General introduces Eclipse 32-bit CISC minicomputer
- Digital (DEC) introduces VAX 32-bit CISC minicomputer
- First serious inexpensive competition to mainframe computers

1977 — 1990

- Serious computers became available to small organizations
- UNIX developed as minicomputer operating system
- TCP/IP developed to support networks of minicomputers
- Computer Science emerged as separate academic discipline
- Students needed topics for projects, theses, dissertations

1980 — 1990

- Research results on minicomputer performance
  - CISC uses machine resources inefficiently
  - Most machine instructions are rarely used in programs
  - CISC machines run slowly to support unnecessary features

Quantitative Performance Theory

Amdahl Equation for Multiprocessors

Symmetric Multiprocessor (SMP)

N equivalent microprocessors

Communication network between processors

Operating system runs on one+ processor

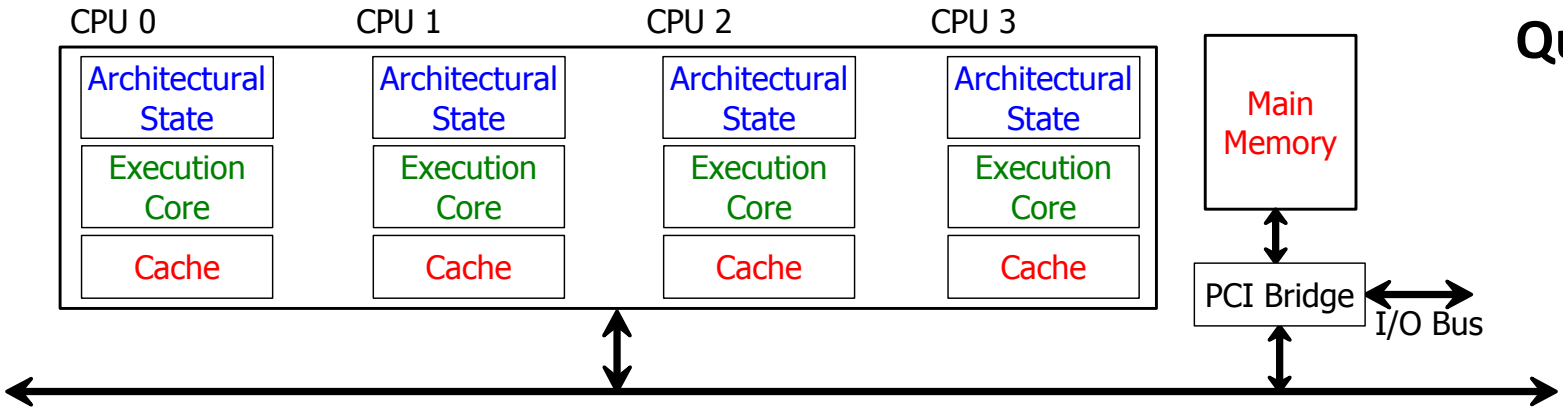
OS assigns tasks to processors by some scheduling system

Amdahl equation for SMP

$$S = \frac{1}{(1 - F_P) + \frac{F_P}{N}}$$

F_P = fraction of work that can be enhanced (parallelized)

N = speedup for part to be enhanced (number of processors)



Quad Core CPU

Example of Amdahl Equation

For multiprocessor system

$$S = \frac{1}{(1-F_P) + \frac{F_P}{N}}$$

F_P = fraction of work that can be enhanced (parallelized)

N = speedup for part to be enhanced (number of processors)

Typical small Dell file server

N = 8 Xeon processors

F_p = 80% of work can be parallelized

$$S = \frac{1}{(1-0.80) + \frac{0.80}{8}} = \frac{1}{0.20 + 0.10} \cong 3.33$$

If number of processors were unlimited

$$S = \frac{1}{(1-F_P) + \frac{F_P}{N}} \xrightarrow{N \to \infty} \frac{1}{1-F_P} = \frac{1}{1-0.80} = 5$$

Maximum speedup is 5
Future enhancements require more parallelization F_p

Basic Performance Measures

Run Time (זמן ריצה)

Elapsed time T from start to finish of a defined program task

Latency (זמן המתנה)

Excess response time — depends on context

Throughput (תפוקה)

Number of defined tasks performed per unit time

$$\text{Throughput} = \frac{1}{\text{T} + \text{ latency between tasks}}$$

Enhancement (שינוי מבנה)

Change to system $\Rightarrow$ new run time T '

Speedup (שיפור)

$$S = \frac{T}{T'} \qquad S > 1 \Rightarrow T' < T$$

Processor Performance Enhancements

Hardware Enhancements

Clock rate

Instruction implementation

Memory organization

Number of processing elements (CPUs, ALUs, registers)

Software Enhancements

Run time optimizations

Compiler

Operating system

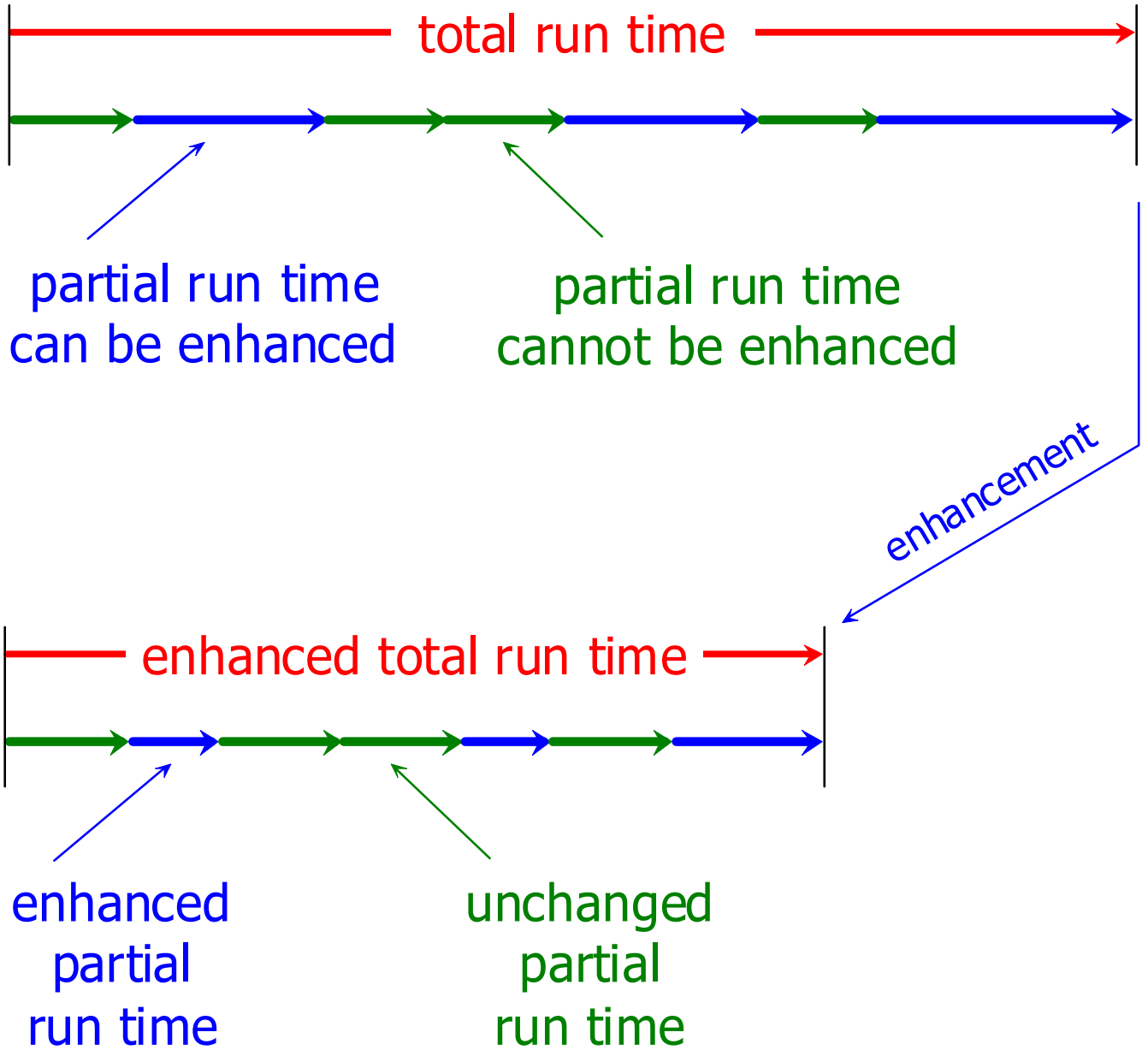
Enhanced Run Time

Run time = sum of partial run times

Enhancement $\Rightarrow$ partial run times are longer, shorter, or unchanged

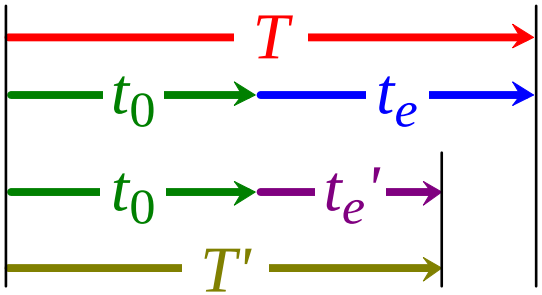
$S > 1 \Rightarrow$ Sum of new partial run times < sum of old partial run times

Run Time Enhancements



Amdahl Equation

Definitions



T total run time of a task

T' total run time of a task after enhancement

t_e partial run time that can be enhanced

t_e' partial run time that can be enhanced, after enhancement

t_0 partial run time that cannot be enhanced

F_e fraction of run time that can be enhanced = t_e / T

S_e Speedup of portion of run time that can be enhanced = t_e / t_e'

$$S = \frac{T}{T'} = \frac{T}{t_0 + t'_e} = \frac{1}{\frac{t_0}{T} + \frac{t'_e}{T}} = \frac{1}{\frac{T - t_e}{T} + \frac{t'_e}{t_e}\frac{t_e}{T}} = \frac{1}{1 - F_e + \frac{1}{S_e}F_e}$$

Amdahl equation expresses speedup in terms of relative quantities
Actual run-times not needed if **RELATIVE ENHANCEMENTS** are known

Example of Amdahl Equation

Program partial run times

Integer Instructions	t_{INT}	**100 ms**
Float Instructions	t_{FP}	**300 ms**
Total Run Time	T	**400 ms**

Enhance partial run time of Float Instructions

Integer Instructions	t_{INT}	**100 ms**
Float Instructions	t_{FP}'	**200 ms**
Total Run Time	T'	**300 ms**

Speedup from actual run times

$$S = \frac{T}{T'} = \frac{400\ \text{ms}}{300\ \text{ms}} = \frac{4}{3}$$

$$F_e = \frac{t_{FP}}{T} = \frac{300\ \text{ms}}{400\ \text{ms}} = \frac{3}{4} = 75\%$$

$$S_e = \frac{t_{FP}}{t_{FP}'} = \frac{300\ \text{ms}}{200\ \text{ms}} = \frac{3}{2} = 1.50$$

Speedup from relative enhancements

$$S = \frac{1}{1 - F_e + \frac{F_e}{S_e}} = \frac{1}{1 - \frac{3}{4} + \frac{3/4}{3/2}} = \frac{1}{\frac{1}{4} + \frac{1}{2}} = \frac{4}{3}$$

Application of Amdahl Equation

On some CPU

Float (FP) instructions account for 50% of total run time

Square root (FP) accounts for 20% of total run time

Choose between two alternative enhancements

Speedup of $S_e = 2$ for all FP instructions

Speedup of $S_e = 10$ for square root instruction

Enhancement 1

$$S_1 = \frac{1}{1 - F_e + \frac{1}{S_e} F_e} = \frac{1}{1 - 0.50 + \frac{0.50}{2}} = \frac{1}{0.50 + 0.25} \cong 1.33 \Rightarrow \boxed{33\% \text{ speedup}}$$

Enhancement 2

$$S_2 = \frac{1}{1 - F_e + \frac{1}{S_e} F_e} = \frac{1}{1 - 0.20 + \frac{0.20}{10}} = \frac{1}{0.80 + 0.02} \cong 1.22 \Rightarrow \boxed{22\% \text{ speedup}}$$

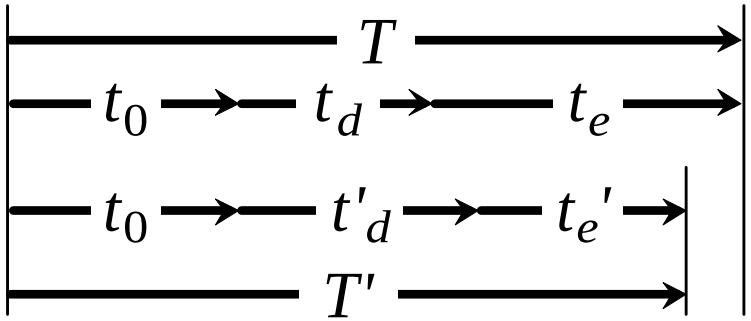
Generalized Amdahl Equation

Result of reasonable architectural change

Enhancements to most features

Degradations to some features

Overall enhancement



New definitions

t_d portion of run time that is degraded

t_d' portion of run time that is degraded, after degradation

F_d fraction of run time that is degraded = t_d / T

S_d Speedup of portion of run time that is degraded = t_d / t_d'

$$S = \frac{T}{T'} = \frac{T}{t_0 + t'_e + t'_d} = \frac{1}{\frac{T - t_e - t_d}{T} + \frac{t'_e}{t_e}\frac{t_e}{T} + \frac{t'_d}{t_d}\frac{t_d}{T}} = \frac{1}{1 - (F_e + F_d) + \frac{F_e}{S_e} + \frac{F_d}{S_d}}$$

Amdahl's "Law"

To make good architectural improvements

Focus on enhancements that positively affect most features

Ignore degradations that negatively affect few features

Example — simple "RISC" processor

94% of run time is 5 times faster than CISC processor

1% of run time is 10 times slower than CISC processor

5% of run time is same as for CISC processor

$$S = \frac{1}{1-\left(F_e + F_d\right) + \frac{F_e}{S_e} + \frac{F_d}{S_d}}$$

$$= \frac{1}{1-\left(0.94 + 0.01\right) + \frac{0.94}{5} + \frac{0.01}{1/10}}$$

$$= \frac{1}{0.05 + 0.19 + 0.10} \cong \boxed{2.94}$$

This RISC processor is (overall) about 3 times faster than CISC

Even though some operations are slower

Detailed Analysis of CPU Run Times

Amdahl equation requires relative run time data

Run time data requires measurements on running programs

Measurements on running programs require CPU implementation

CPU analysis predicts run time without building CPU

Assumptions:

Instructions can be grouped together according to resource usage

Example — **`ADD R1, R2, R3`** and **`SUB R1, R2, R3`**

All instructions in a group run in same number of clock cycles

Every clock cycle measures same unit of time

Instruction run time = clock cycle time × number of clock cycles

Group run time = instruction run time × instructions in group

Total run time = sum of instruction group run times

Definitions

T = total run time of program

t_i = total run time of instructions in group i

IC_i = number of instructions in group i (Instruction Count)

CPI_i = number of clock cycles to run 1 instruction in group i (Cycles Per Instruction)

N_i = number of clock cycles to run all instructions in group i

τ = seconds per clock cycle

R = clock rate = clock frequency = clock cycles per second = Hertz (Hz) = $1/\tau$

IC = total number of instructions in program

N = total number of clock cycles to run program

CPI = average number of clock cycles per instruction for the program

$quantity'$ = new value of $quantity$ after architectural change

CPU Equation

Clock cycles to run all instructions of type i

$$N_i = \text{instructions of type } i \times \frac{\text{clock cycles}}{\text{instruction of type } i} = IC_i \times CPI_i$$

Total clock cycles to run all instructions in program

$$N = \sum_{\text{all groups } i} N_i = \sum_i IC_i \times CPI_i$$

Average number of clock cycles per instruction for program

$$CPI = \frac{\text{total number of clock cycles to run program}}{\text{total number of instructions in program}} = \frac{N}{IC}$$

$$CPI = \frac{1}{IC}\sum_i N_i = \frac{1}{IC}\sum_i IC_i \times CPI_i = \sum_i CPI_i \times \frac{IC_i}{IC}$$

weighted average

$$\sum_i \frac{IC_i}{IC} = 1$$

Ratio $\frac{IC_i}{IC}$ is proportion (percent) of instructions in group i

Example of CPU Equation

Program distribution

Instruction Type i	IC_i	CPI_i
Integer	5,000	4
Branch	1,000	12
Load / Store	4,000	8

$$IC = IC_{\text{int}} + IC_{\text{branch}} + IC_{\text{load/store}}$$
$$= 5,000 + 1,000 + 4,000$$
$$= 10,000 \text{ instructions}$$

Instruction Type i	IC_i / IC	$N_i = IC_i \times CPI_i$
Integer	5000/10000 = 50%	4×5000 = 20,000 cycles
Branch	1000/10000 = 10%	12×1000 = 12,000 cycles
Load / Store	4000/10000 = 40%	8×4000 = 32,000 cycles

$$N = 20,000 \text{ cycles} + 12,000 \text{ cycles} + 32,000 \text{ cycles} = 64,000 \text{ cycles}$$

$$CPI = N / IC = 64,000 \text{ cycles} / 10,000 \text{ instuctions} = 6.4 \text{ cycles per instruction}$$

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.50 + 12 \times 0.10 + 8 \times 0.40 = 6.4 \text{ cycles per instruction}$$

CPU Run Time

Run time of one instruction of type i

$$\frac{\text{clock cycles}}{\text{instruction of type } i} \times \frac{\text{seconds}}{\text{clock cycle}} = CPI_i \times \tau$$

Run time for all instructions of type i

$$t_i = \text{instructions of type } i \times \frac{\text{clock cycles}}{\text{instruction of type } i} \times \frac{\text{seconds}}{\text{clock cycle}}$$

$$= IC_i \times CPI_i \times \tau$$

Total run time for program

$$T = \sum_{\text{all groups } i} t_i = \sum_i CPI_i \times IC_i \times \tau = \left(\sum_i CPI_i \times \frac{IC_i}{IC} \right) \times IC \times \tau$$

So

$$T = CPI \times IC \times \tau$$

$$= \text{clock cycles per instruction} \times \text{number of instructions} \times \text{clock cycle}$$

CPU Run Time — Example

For a certain CPU

Instructions in a typical programs can be grouped as

- 50% integer ALU instructions that run in 8 clock cycles
- 10% float ALU instructions that run in 20 clock cycles
- 20% load instructions that run in 10 clock cycles
- 10% store instructions that run in 15 clock cycles
- 10% branch instructions that run in 10 clock cycles

The clock speed is 100 MHz

A typical program runs 1,000,000 instructions

- Running 500,000 ALU instructions, 100,000 FP instructions, 100,000 loads, …

The average number of cycles per instruction is

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 8 \times 0.5 + 20 \times 0.1 + 10 \times 0.2 + 15 \times 0.1 + 10 \times 0.1 = 10.5$$

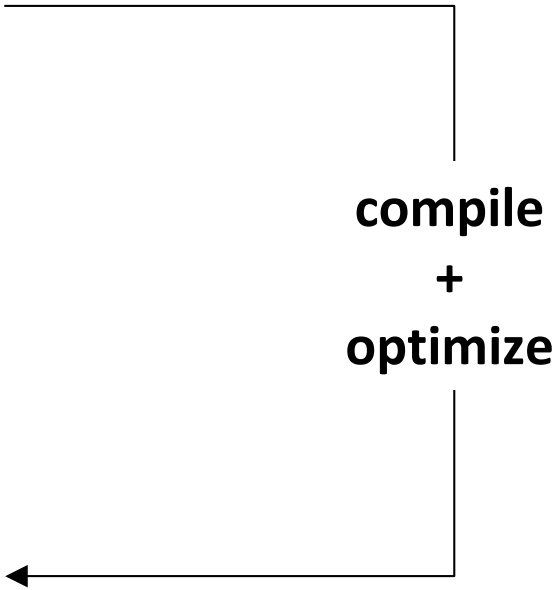
The typical program runs in

$$T = CPI \times IC \times \tau = \frac{CPI \times IC}{R} = \frac{10.5 \text{ cycles/instruction} \times 10^6 \text{ instructions}}{10^8 \text{ Hz}} = 0.105 \text{ seconds}$$

C Code to Runtime Example — 1

High level code

```
int x = 0 , n = 0 , a[5] ;
while ( n < 5 ){
    x = x + a[n] ;
    n++ ;
}
```



Assembly program

```
1000      MOV R1, 0
1002      MOV R2, 2000
1004      ADD R1, R1, (R2)+
1008      CMP R2, 2020
1012      JL 1004
```

load	1	
load	1	13%
ALU_{AI}	5	29%
ALU	5	29%
JMP	5	29%
IC =	17	100%

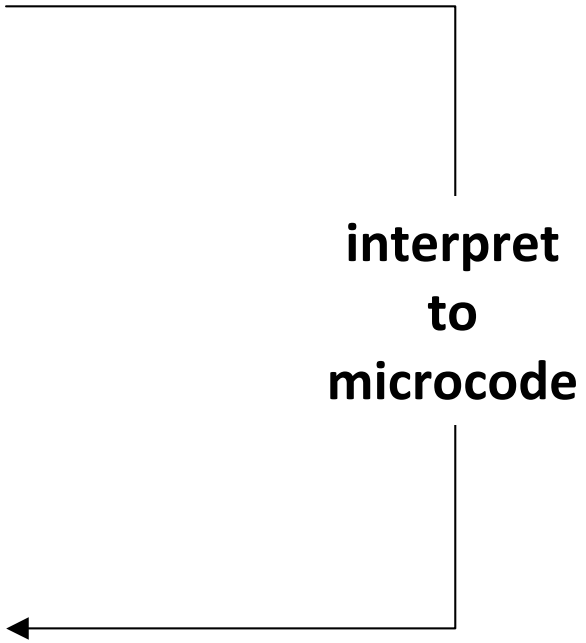
C Code to Runtime Example — 2

Assembly code

```
ADD R1, R1, (R2)+
```

Microprogram

```
ALU_IN, MAR ← R2
ALU ← 4
ADD
R2 ← ALU_OUT
READ
ALU_IN ← MDR
ALU ← R1
ADD
R1 ← ALU_OUT
```



$$CPI_{ALU\text{-}autoinc} = 9$$

C Code to Runtime Example — 3

Assembly program

```
1000      MOV R1, 0                          IC_i    CPI_i
1002      MOV R2, 2000          load         13%     2
1004      ADD R1, R1, (R2)+     ALU_AI       29%     9
1008      CMP R2, 2020          ALU          29%     3
1012      JL 2004               JMP          29%     12
                                IC =   17    100%
```

Average CPI

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 2 \times 0.13 + 9 \times 0.29 + 3 \times 0.29 + 12 \times 0.29 = 7.2$$

Total clock cycles

$$N = CPI \times IC = 7.2 \times 17 = 122$$

Run time with 1 GHz clock rate

$$T = N \times \tau = 122 \times 10^{-9} \text{seconds} = 1.22 \times 10^{-7} \text{seconds} = 0.122 \text{ microseconds}$$

Applying CPU Equation

1. Run time before enhancement

$$T = CPI \times IC \times \tau$$

2. Characterize enhancement

Calculate IC'

Calculate CPI'

Calculate τ'

3. Run time after enhancement

$$T' = CPI' \times IC' \times \tau'$$

4. Speedup

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'}$$

CPU Equation — Example 2

For a certain CPU

25% of all instructions in programs are float ($IC_{FP} / IC = 0.25$)

FP group includes ADD, SUB, MULT, DIV, SQRT

Average FP instruction runs in 4 clock cycles ($CPI_{FP} = 4$)

2% of all instructions in program are square root ($IC_{SQRT} / IC = 0.02$)

SQRT (FP) instruction runs in 20 cycles ($CPI_{SQRT} = 20$)

Average CPI for all other instructions in program is 4/3 clock cycles

($IC_{other} / IC = 1 - 0.25 = 0.75$ $\quad$ $CPI_{other} = 4/3$)

Average cycles per instruction

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.25 + \frac{4}{3} \times (1 - 0.25) = 2.00$$

Example 2 — 2

Two possible enhancements

1. Improve performance of all FP instructions

   Enhance average CPI_{FP} = 4 cycles to CPI_{FP}' = 2 cycles

   No change in program $\Rightarrow IC_i' = IC_i$ for all instruction types

   No change to clock rate $\Rightarrow \tau' = \tau$

2. Improve performance of SQRT (FP) instruction

   Enhance CPI_{SQRT} = 20 cycles to CPI_{SQRT}' = 2 cycles

   No change in program $\Rightarrow IC_i' = IC_i$ for all instruction types

   No change to clock rate $\Rightarrow \tau' = \tau$

To evaluate enhancements, must find CPI'

Example 2 — 3

Enhancement 1

Improve average FP from CPI_{FP} = 4 cycles to CPI_{FP}' = 2 cycles

$$CPI' = \sum_i CPI'_i \times \frac{IC_i'}{IC'}$$

$$= CPI'_{FP} \times \frac{IC_{FP}'}{IC'} + CPI'_{\text{other}} \times \frac{IC_{\text{other}}'}{IC'}$$

$$= 2 \times 0.25 + \frac{4}{3} \times (1 - 0.25)$$

$$= 1.50$$

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{CPI \times IC \times \tau}{CPI' \times IC \times \tau} = \frac{CPI}{CPI'} = \frac{2.00}{1.50} \cong 1.33$$

Example 2 — 4

Enhancement 2

Improve square root (FP) from $CPI_{SQRT} = 20$ cycles to $CPI_{SQRT}' = 2$ cycles

Must separate into 3 instruction groups

FP/SQRT = FP group without SQRT = ADD, SUB, MULT, DIV

SQRT

All other instructions

$$IC_{FP} / IC = 25\%, \;\; IC_{SQRT} / IC = 2\% \;\Rightarrow\; IC_{FP/SQRT} / IC = 25\% - 2\% = 23\%$$

$$CPI' = \sum_i CPI'_i \times \frac{IC_i'}{IC'}$$

$$= CPI'_{FP/SQRT} \times \frac{IC_{FP/SQRT}'}{IC'} + CPI'_{SQRT} \times \frac{IC_{SQRT}'}{IC'} + CPI'_{\text{other}} \times \frac{IC_{\text{other}}'}{IC'}$$

$$= CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT}}{IC} + CPI'_{SQRT} \times \frac{IC_{SQRT}}{IC} + CPI_{\text{other}} \times \frac{IC_{\text{other}}}{IC}$$

First calculate $CPI_{FP/SQRT}$ from CPI_{FP}, CPI_{SQRT}, IC_{FP} / IC, IC_{SQRT} / IC

Example 2 — 5

$$CPI_{FP} = \text{Average } CPI \text{ for } FP \text{ group with } SQRT = 4$$

$$= \frac{\text{total } FP \text{ cycles}}{FP \text{ instructions}} = \frac{N_{FP}}{IC_{FP}} = \sum_{k \in FP} \frac{N_k}{IC_{FP}}$$

$$= \sum_{k \in FP} \frac{CPI_k \times IC_k}{IC_{FP}} = \sum_{k \in FP} CPI_k \times \frac{IC_k}{IC_{FP}}$$

$$CPI_{FP/SQRT} = \text{Average } CPI \text{ for } FP \text{ group without } SQRT$$

$$= \frac{\text{total } FP / SQRT \text{ cycles}}{FP / SQRT \text{ instructions}} = \frac{N_{FP/SQRT}}{IC_{FP/SQRT}} = \sum_{k \in FP/SQRT} \frac{N_k}{IC_{FP/SQRT}}$$

$$= \sum_{k \in FP/SQRT} \frac{CPI_k \times IC_k}{IC_{FP/SQRT}} = \sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP/SQRT}}$$

Example 2 — 6

$$CPI_{FP} = \sum_{k \in FP} CPI_k \times \frac{IC_k}{IC_{FP}}$$

$$= \sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP}} + CPI_{SQRT} \times \frac{IC_{SQRT}}{IC_{FP}}$$

$$= \left[\sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP}} \right] \times \frac{IC_{FP/SQRT} / IC}{IC_{FP/SQRT} / IC} + \left[CPI_{SQRT} \times \frac{IC_{SQRT}}{IC_{FP}} \right] \times \frac{1/IC}{1/IC}$$

$$= \left[\sum_{k \in FP/SQRT} CPI_k \times \frac{IC_k}{IC_{FP/SQRT}} \right] \frac{IC_{FP/SQRT} / IC}{IC_{FP} / IC} + CPI_{SQRT} \times \frac{IC_{SQRT} / IC}{IC_{FP} / IC}$$

$$= CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT} / IC}{IC_{FP} / IC} + CPI_{SQRT} \times \frac{IC_{SQRT} / IC}{IC_{FP} / IC}$$

$$4 = CPI_{FP/SQRT} \times \frac{0.25 - 0.02}{0.25} + 20 \times \frac{0.02}{0.25} \Rightarrow \boxed{CPI_{FP/SQRT} = 2.61}$$

Example 2 — 7

Speedup for Enhancement 2

$$CPI' = \sum_i CPI'_i \times \frac{IC_i'}{IC'}$$

$$= CPI_{FP/SQRT} \times \frac{IC_{FP/SQRT}}{IC} + CPI'_{SQRT} \times \frac{IC_{SQRT}}{IC} + CPI_{\text{other}} \times \frac{IC_{\text{other}}}{IC}$$

$$= 2.61 \times 0.23 + 2 \times 0.02 + \frac{4}{3} \times (1 - 0.25)$$

$$= 1.64$$

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{CPI \times IC \times \tau}{CPI' \times IC \times \tau} = \frac{CPI}{CPI'} = \frac{2.00}{1.64} \cong 1.22$$

Example 2 — 8

Trick — technique to avoid calculating $CPI_{FP/\text{SQRT}}$

$$CPI' = CPI - (CPI - CPI')$$

$$= CPI - \left(\sum_i CPI_i \times \frac{IC_i}{IC} - \sum_i CPI'_i \times \frac{IC'_i}{IC'} \right)$$

$$= CPI - \sum_i \left(CPI_i \times \frac{IC_i}{IC} - CPI'_i \times \frac{IC'_i}{IC'} \right)$$

If $IC'_i = IC_i$ then combine terms as

$$CPI' = CPI - \sum_i \left[(CPI_i - CPI'_i) \times \frac{IC_i}{IC} \right]$$

$$= CPI - \left[(CPI_{SQRT} - CPI'_{SQRT}) \times \frac{IC_{SQRT}}{IC} \right]$$

$$= 2.00 - \left[(20 - 2) \times 0.02 \right]$$

$$= 1.64$$

Example 2 — 9

5. Speedups

Enhancement to square root — S = 1.22

Enhancement to all FP — S = 1.33

Results identical to analysis by Amdahl equation

Can derive inputs to Amdahl equation from CPU analysis

1.

$$F_e^{FP} = \frac{t_{FP}}{T} = \frac{CPI_{FP} \times IC_{FP} \times \tau}{CPI \times IC \times \tau} = \frac{4 \times (0.25 \times IC) \times \tau}{2 \times IC \times \tau} = 50\%$$

$$S_e^{FP} = \frac{CPI_{FP} \times IC_{FP} \times \tau}{CPI'_{FP} \times IC'_{FP} \times \tau'} = \frac{CPI_{FP}}{CPI'_{FP}} = \frac{4}{2} = 2$$

2.

$$F_e^{SQRT} = \frac{t_{SQRT}}{T} = \frac{CPI_{SQRT} \times IC_{SQRT} \times \tau}{CPI \times IC \times \tau} = \frac{20 \times (0.02 \times IC) \times \tau}{2 \times IC \times \tau} = 20\%$$

$$S_e^{SQRT} = \frac{CPI_{SQRT} \times IC_{SQRT} \times \tau}{CPI'_{SQRT} \times IC'_{SQRT} \times \tau'} = \frac{CPI_{SQRT}}{CPI'_{SQRT}} = \frac{20}{2} = 10$$

Changing Instruction Mix — 1

Program distribution

Instruction Type i	IC_i	IC_i / IC	CPI_i
Integer	5,000	5000/10000 = 50%	4
Branch	1,000	1000/10000 = 10%	12
Load / Store	4,000	4000/10000 = 40%	8
Total	10,000	10000/10000 = 100%	

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.50 + 12 \times 0.10 + 8 \times 0.40 = 6.4 \text{ cycles per instruction}$$

New program distribution

Instruction Type i	IC_i	IC_i / IC	CPI_i
Integer	3,000	3000/8000 = 37.5%	4
Branch	1,000	1000/8000 = 12.5%	12
Load / Store	4,000	4000/8000 = 50%	8
Total	8,000	8000/8000 = 100%	

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC} = 4 \times 0.375 + 12 \times 0.125 + 8 \times 0.50 = 7.0 \text{ cycles per instruction}$$

Changing Instruction Mix — 2

Speedup

$$S = \frac{T}{T'}$$

$$= \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'}$$

$$= \frac{6.4 \times 10000 \times \tau}{7.0 \times 8000 \times \tau}$$

$$= 1.14$$

The Instructions Per Second Myth

Measures often used to describe computer power

MIPS = million instructions per second

FLOPS = floating point operations per second

Neither gives fair comparison

$$\text{MIPS} = \frac{\text{instructions} / 10^6}{\text{run time}} = \frac{IC}{10^6 \times T} = \frac{IC}{10^6 \times CPI \times IC \times \tau} = \frac{R}{10^6 \times CPI}$$

Example

CPU-1 and CPU-2

Run ALU instructions in 1 cycles and others in 2 cycles

Have clock speed of 1 GHz

CPU-1 compiler produces 50% ALU instructions and 50% other

CPU-2 compiler produces 25% fewer ALU instructions than CPU-1

$$CPI_1 = 1 \times 0.50 + 2 \times 0.50 = 1.50 \Rightarrow MIPS_1 = \frac{10^9\, Hz}{10^6 \times 1.50} \cong 667 \text{ million instructions / sec}$$

For CPU-2

$$IC_1^{ALU} = 0.50 \times IC_1$$

$$IC_2^{ALU} = 0.75 \times IC_1^{ALU} = 0.75 \times \left(0.50 \times IC_1\right) = 0.375 \times IC_1$$

$$IC_2 = IC_2^{ALU} + IC_2^{other} = 0.375 \times IC_1 + 0.50 \times IC_1 = 0.875 \times IC_1$$

$$\frac{IC_2^{ALU}}{IC_2} = \frac{0.375 \times IC_1}{0.875 \times IC_1} = 0.43 \qquad \frac{IC_2^{other}}{IC_2} = \frac{0.50 \times IC_1}{0.875 \times IC_1} = 0.57$$

$$CPI_2 = 1 \times 0.43 + 2 \times 0.57 = 1.57$$

$$MIPS_2 = \frac{10^9\ Hz}{10^6 \times 1.57} \cong 637 \text{ million instructions / sec} < MIPS_1$$

$$\boxed{\frac{MIPS_2}{MIPS_1} = \frac{637}{667} = 0.955}$$

Run time comparison

$$
\begin{aligned}
S &= \frac{T_1}{T_2} \\
&= \frac{CPI_1 \times IC_1 \times \tau_1}{CPI_2 \times IC_2 \times \tau_2} \\
&= \frac{CPI_1 \times IC_1}{CPI_2 \times IC_2} \\
&= \frac{1.50 \times IC_1}{1.57 \times \left(0.875 \times IC_1\right)} \\
&\cong 1.09
\end{aligned}
$$

MIPS is about 5% lower for CPU-2 than CPU-1

CPU-2 is about 9% faster than CPU-1

Replacing Instruction Types

Instruction count

$IC = IC_1 + IC_2 + ... + IC_n$

Examples

Type 1 = ALU

Type 2 = Conditional Branch

New Instruction count

Replace 2 ALU instructions + 1 Branch

```
DEC CX
CMP CX, 0
JNZ target
```

New instruction

```
LOOP target
```

$IC' = IC_1' + IC_2' + ... + IC_n$

Example — Replacing Instructions

A certain CPU has no floating point unit (FPU)

Performs FP calculations by **EMULATION**

Converts FP operations to integer operations

Example

$(2.165 \times 10^{4}) \times (3.247 \times 10^{-3}) \rightarrow 2165 \times 3247$

$\exp = (4 - 3) + (-3 - 3)$

Instruction distribution

Type i	IC_i / IC	CPI_i
ALU	75%	1
Load	10%	2
Store	5%	2
Branch	10%	2

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC}$$

$$= 1 \times 0.75 + 2 \times (0.10 + 0.05 + 0.10)$$

$$= 1.25$$

Replacing Instructions — 2

Enhance CPU with ALU

Replace ALU instructions that emulate FP with new FP instructions

- 2/3 of old ALU instructions emulate FP instructions
- 1 new FPU instruction replaces 10 old ALU emulation instructions
- New FPU instructions run in 4 clock cycles

Type i	IC_i / IC	CPI_i
ALU	75%	1
Load	10%	2
Store	5%	2
Branch	10%	2

ALU^{int}	$1/3 \times 75\% = 25\%$
$ALU^{emulation}$	$2/3 \times 75\% = 50\%$

$$IC'_{ALU} = 0.25\ IC$$

$$IC'_{FPU} = 1/10 \times 0.50\ IC = 0.05\ IC$$

$$IC'_{load} = IC_{load} = 0.10\ IC$$

$$IC'_{store} = IC_{store} = 0.05\ IC$$

$$IC'_{branch} = IC_{branch} = 0.10\ IC$$

$$IC' = 0.25\ IC + 0.05\ IC + 0.10\ IC + 0.05\ IC + 0.10\ IC = 0.55\ IC$$

Replacing Instructions — 3

New instruction distribution

$Type\ i$	IC_i / IC	CPI_i
ALU	0.25/0.55	1
FPU	0.05 /0.55	4
Load	0.10 /0.55	2
Store	0.05 /0.55	2
Branch	0.10 /0.55	2

$$CPI' = \sum_i CPI'_i \times \frac{IC'_i}{IC}$$

$$= 1 \times \frac{0.25}{0.55} + 4 \times \frac{0.05}{0.55} + 2 \times \left(\frac{0.10}{0.55} + \frac{0.05}{0.55} + \frac{0.10}{0.55} \right)$$

$$= \frac{0.95}{0.55}$$

$$\boxed{CPI' = 1.73 \quad > \quad CPI = 1.25}$$

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{1.25 \times IC}{\frac{0.95}{0.55} \times (0.55 IC)} = \frac{1.25}{0.95} \cong 1.32$$

Load-Store versus Register-Memory

CPU-1 is a load-store machine

ALU operands must come from register

Memory operand

Loaded to register before ALU operation

Stored to memory after ALU operation

Instruction distribution

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC}$$
$$= 5 \times 0.25 + 4 \times 0.75$$
$$= 4.25$$

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

Possible enhancement

25% of ALU memory operands used in only 1 ALU operation

Can register-memory ALU operations improve performance?

Load-Store versus Register-Memory

CPU-2 is an "ideal" register-memory machine

- ALU operands may come from register or memory
- 75% of memory operands
  - Used in multiple ALU operations
  - Perfect compiler loads "multiple" memory operands to registers
- 25% of ALU memory operands
  - Used in only a single ALU operation
  - Perfect compiler never loads "single" memory operands to registers

Convert CPU-1 to CPU-2

- Split ALU operations into ALU^{multi} and ALU^{single}
- Replace ALU^{single} with $ALU^{register\text{-}memory}$
- Cancel 1 register load for every ALU^{single}

Detailed Instruction Distribution

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

ALU^{single}	10%
ALU^{multi}	30%

Load^{single}	10%
Load^{multi}	25% - 10% =15%

Type i	IC_i / IC	CPI_i
ALU^{single}	10%	4
ALU^{multi}	30%	4
Load^{single}	10%	5
Load^{multi}	15%	5
Store	15%	4
Branch	20%	4

ALU$^{register\text{-}memory}$	10%	9

$$CPI_{ALU^{reg-mem}} = CPI_{ALU} + CPI_{Load}$$

New Instruction Distribution and Speedup

$Type\ i$	IC_i / IC	CPI_i
ALU^{single}	10%	4
ALU^{multi}	30%	4
Load^{single}	10%	5
Load^{multi}	15%	5
Store	15%	4
Branch	20%	4

ALU$^{register\text{-}memory}$	10%	9

$$IC' = IC'_{ALU^{reg-mem}} + IC_{ALU^{multi}} + IC_{Load^{multi}} + IC_{Store} + IC_{Branch}$$
$$= 0.10IC + 0.30IC + 0.15IC + 0.15IC + 0.20IC$$
$$= 0.90IC$$

$$CPI' = \sum_i CPI'_i \times \frac{IC'_i}{IC}$$
$$= 4 \times \frac{65}{90} + 5 \times \frac{15}{90} + 9 \times \frac{10}{90}$$
$$= \frac{425}{90} = 4.72$$

$Type\ i$	IC_i / IC	CPI_i
ALU$^{register\text{-}memory}$	10/90	9
ALU^{multi}	30/90	4
Load^{multi}	15/90	5
Store	15/90	4
Branch	20/90	4

$$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{4.25 \times IC}{\frac{425}{90} \times (0.90IC)} = 1 \Rightarrow$$

No Change in Performance

Analysis of 8086 Example

8086 program compiled from C source

	Instruction	Clock Cycles	Runs	Type
	MOV WORD PTR [BP-02],0000	13	1	Store
start:	CMP WORD PTR [BP-02],N	10	N	$ALU^{imm-mem}$
	JGE stop	4/13	N-1 / 1	Conditional Branch
	MOV AX,[BP-02]	9	N-1	Load
	SHL AX,1	2	N-1	ALU^{reg}
	MOV [BP-04],AX	12	N-1	Store
	INC WORD PTR [BP-02]	15	N-1	$ALU^{reg-mem}$
	JMP start	JMP 14	N-1	Unconditional Branch
stop:	RET	RET 16	1	Return

CPI for Store and ALU

JGE

Runs N-1 times in 4 clock cycles and 1 time in 13 clock cycles

$$CPI_{JGE} = \frac{JGE \text{ cycles}}{JGE \text{ instructions}} = \frac{4 \times (N-1) + 13 \times 1}{(N-1) + 1} = \frac{4 \times (N-1) + 13}{N}$$

STORE

Runs N-1 times in 12 clock cycles and 1 time in 13 clock cycles

$$CPI_{STORE} = \frac{STORE \text{ cycles}}{STORE \text{ instructions}} = \frac{12 \times (N-1) + 13 \times 1}{(N-1) + 1} = \frac{12 \times (N-1) + 13}{N}$$

Instruction Distribution

Type i	IC_i / IC	CPI_i
Load	$\frac{N-1}{7N-3}$	9
Store	$\frac{1+(N-1)}{7N-3}$	$[12(N–1) +13] / N$
ALU$^{\text{reg-mem}}$	$\frac{N-1}{7N-3}$	15
ALU$^{\text{reg}}$	$\frac{N-1}{7N-3}$	2
ALU$^{\text{imm-mem}}$	$\frac{N}{7N-3}$	10
Conditional Branch	$\frac{N}{7N-3}$	$[4(N–1) + 13] / N$
Unconditional Branch	$\frac{N-1}{7N-3}$	14
Return	$\frac{1}{7N-3}$	16

Instruction Distribution for Loop Length = 100

Type i	IC_i / IC	CPI_i
Load	14.20%	9
Store	14.34%	12.01
$ALU^{reg\text{-}mem}$	14.20%	15
ALU^{reg}	14.20%	2
$ALU^{imm\text{-}mem}$	14.34%	10
Conditional Branch	14.34%	4.09
Unconditional Branch	14.20%	14
Return	0.14%	16

$$N = 100$$

$$IC = 7N - 3 = 697$$

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC}$$

$$= (9+15+2+14)\times 0.1420 + (12.01+10+4.09)\times 0.1434 + 16\times 0.0014 = 9.45$$

$$T = \frac{CPI \times IC}{R} = \frac{(9.45 \text{ cycles / instruction}) \times (697 \text{ instructions})}{4 \text{ MHz}} = 1.646\times 10^{-3} \text{ sec}$$

Estimated run time for
296 MHz UltraSPARC II = 4.71×10^{-7} sec

$$S = \frac{1.646\times 10^{-3}}{4.71\times 10^{-7}} = 3494$$

Register Variables

Improved program

Memory variables replaced with register variables

	Instruction	Clock Cycles	Runs	Type
	MOV SI,0000	4	1	$MOV^{imm-reg}$
start:	CMP SI,+0A	3	N	$ALU^{imm-reg}$
	JGE stop	4/13	N-1 / 1	Conditional Branch
	MOV AX,SI	2	N-1	$MOV^{reg-reg}$
	SHL AX,1	2	N-1	ALU^{reg}
	MOV DI,AX	2	N-1	$MOV^{reg-reg}$
	INC SI	3	N-1	$ALU^{reg-reg}$
	JMP start	JMP 14	N-1	Unconditional Branch
stop:	RET	RET 16	1	Return

New Instruction Distribution

$Type\ i$	IC_i / IC	CPI_i
MOV$^{\text{imm-reg}}$	$\frac{1}{7N-3}$	4
MOV$^{\text{reg-reg}}$	$\frac{2(N-1)}{7N-3}$	2
ALU$^{\text{reg-reg}}$	$\frac{N-1}{7N-3}$	3
ALU$^{\text{reg}}$	$\frac{N-1}{7N-3}$	2
ALU$^{\text{imm-reg}}$	$\frac{N}{7N-3}$	3
Conditional Branch	$\frac{N}{7N-3}$	$[4(N–1) + 13] / N$
Unconditional Branch	$\frac{N-1}{7N-3}$	14
Return	$\frac{1}{7N-3}$	16

New Distribution for Loop Length = 100

$Type\ i$	IC_i' / IC'	CPI_i'
$MOV^{imm\text{-}reg}$	0.14%	4
$MOV^{reg\text{-}reg}$	28.41%	2
$ALU^{reg\text{-}reg}$	14.20%	3
ALU^{reg}	14.20%	2
$ALU^{imm\text{-}reg}$	14.35%	3
Conditional Branch	14.35%	4.09
Unconditional Branch	14.20%	14
Return	0.14%	16

$$N = 100$$

$$IC = 7N - 3 = 697$$

$$CPI' = \sum_i CPI_i' \times \frac{IC_i'}{IC'}$$

$$= (4+16)\times 0.0014 + 2\times 0.2840 + (3+2+14)\times 0.1420 + (3+4.09)\times 0.1435 = 4.31$$

$$T = \frac{CPI' \times IC'}{R} = \frac{(4.31\ \text{cycles / instruction}) \times (697\ \text{instructions})}{4\ \text{MHz}} = 7.515\times 10^{-4}\ \text{sec}$$

Run time with memory variables = 1.646×10^{-3} sec

$$S = \frac{1.646\times 10^{-3}}{7.515\times 10^{-4}} = 2.19$$

Register Variables — Two Threads

Data decomposition

Take N = 100 data points

Each thread runs on 50 data points (loop length)

	Thread 0	Thread 1
	`MOV SI,0000`	`MOV SI,0032`
`start:`	`CMP SI,+32  ; 0x32 = 50`$_{10}$	`CMP SI,+64  ; 0x64 = 100`$_{10}$
	`JGE stop`	`JGE stop`
	`MOV AX,SI`	`MOV AX,SI`
	`SHL AX,1`	`SHL AX,1`
	`MOV DI,AX`	`MOV DI,AX`
	`INC SI`	`INC SI`
	`JMP start`	`JMP start`
`stop:`	`RET`	`RET`

New Distribution for Loop Length = 50 For Each Thread

$Type\ i$	IC_i' / IC'	CPI_i'
$MOV^{imm\text{-}reg}$	0.28%	4
$MOV^{reg\text{-}reg}$	28.41%	2
$ALU^{reg\text{-}reg}$	14.12%	3
ALU^{reg}	14.12%	2
$ALU^{imm\text{-}reg}$	14.41%	3
Conditional Branch	14.41%	4.09
Unconditional Branch	14.12%	14
Return	0.28%	16

$$N = 50$$

$$IC = 7N - 3 = 347$$

$$CPI' = \sum_i CPI_i' \times \frac{IC_i'}{IC'}$$

$$= (4+16)\times 0.0028 + 2\times 0.2841 + (3+2+14)\times 0.1412 + (3+4.09)\times 0.1441 = 4.34$$

$$T = \frac{CPI' \times IC'}{R} = \frac{(4.34 \text{ cycles / instruction}) \times (347 \text{ instructions})}{4 \text{ MHz}} = 3.765\times 10^{-4} \text{ sec}$$

Run time on one thread = 7.525×10^{-4} sec

$$S = \frac{3.765\times 10^{-4}}{7.515\times 10^{-4}} = 1.998$$

Software Content — Instruction Distribution

Type i	CPI_i	IC_i
ALU	2	400
Branch	8	200
Load	4	250
Store	4	150

CPI	Cycles Per Instruction
IC	Instruction Count

Software Content — Instruction Distribution

Type i	CPI_i	IC_i
ALU	2	400
Branch	8	200
Load	4	250
Store	4	150
IC		1000

$$IC = \sum_i IC_i$$

Software Content — Instruction Distribution

Type i	CPI_i	IC_i	IC_i/IC
ALU	2	400	40%
Branch	8	200	20%
Load	4	250	25%
Store	4	150	15%
IC		1000	

Software Content — Instruction Distribution

Type i	CPI_i	IC_i	IC_i/IC	$CPI_i * IC_i/IC$
ALU	2	400	40%	0.8
Branch	8	200	20%	1.6
Load	4	250	25%	1.0
Store	4	150	15%	0.6
IC		1000		

Software Content — Instruction Distribution

Type i	CPI_i	IC_i	IC_i/IC	CPI_i * IC_i/IC
ALU	2	400	40%	0.8
Branch	8	200	20%	1.6
Load	4	250	25%	1.0
Store	4	150	15%	0.6
IC		1000		
CPI				4.0

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC}$$

Software Content — Instruction Distribution

Type i	CPI_i	IC_i	IC_i/IC	$CPI_i * IC_i/IC$
ALU	2	400	40%	0.8
Branch	8	200	20%	1.6
Load	4	250	25%	1.0
Store	4	150	15%	0.6
IC		1000		
CPI				4.0

$$N = CPI \times IC = 4.0 \times 1000 = 4000$$

$$T = CPI \times IC \times \tau = 4000\tau$$

Software Content — Instruction Distribution

Type i	CPI_i	IC_i	IC_i/IC	$CPI_i * IC_i/IC$	N_i
ALU	2	400	40%	0.8	800
Branch	8	200	20%	1.6	1600
Load	4	250	25%	1.0	1000
Store	4	150	15%	0.6	600
IC		1000			
CPI				4.0	

$$N_i = CPI_i \times IC_i$$

Software Content — Instruction Distribution

Type i	CPI_i	IC_i	IC_i/IC	$CPI_i * IC_i/IC$	N_i
ALU	2	400	40%	0.8	800
Branch	8	200	20%	1.6	1600
Load	4	250	25%	1.0	1000
Store	4	150	15%	0.6	600
IC		1000			
CPI				4.0	

$$N = CPI \times IC = 4.0 \times 1000 = 4000$$

$$N = \sum_i N_i = 800 + 1600 + 100 + 600 = 4000$$

Software Content — Instruction Distribution

Type i	CPI_i	IC_i	IC_i/IC	$CPI_i * IC_i/IC$	N_i	F_i
ALU	2	400	40%	0.8	800	20%
Branch	8	200	20%	1.6	1600	40%
Load	4	250	25%	1.0	1000	25%
Store	4	150	15%	0.6	600	15%
IC		1000				
CPI				4.0		

$$F_i = \frac{t_i}{T} = \frac{CPI_i \times IC_i \times \tau}{CPI \times IC \times \tau} = \frac{CPI_i \times IC_i}{CPI \times IC} = \frac{N_i}{N}$$

Enhancement

Type i	CPI_i'	IC_i	IC_i/IC	$CPI_i' * IC_i/IC$	N_i'	F_i'
ALU	2	400	40%	0.8	800	25%
Branch	8 → 4	200	20%	1.6 → 0.8	800	25%
Load	4	250	25%	1.0	1000	31%
Store	4	150	15%	0.6	600	19%
IC		1000				
CPI'				3.2		

CPU Equation $$S = \frac{T}{T'} = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{CPI}{CPI'} = \frac{4.0}{3.2} = 1.25$$

Amdahl Equation $$S = \frac{T}{T'} = \frac{1}{1 - F_e + \frac{F_e}{S_e}} = \frac{1}{1 - 0.4 + \frac{0.4}{2}} = \frac{1}{0.6 + 0.2} = 1.25$$

$$S_e = \frac{t_e}{t_e'} = \frac{8 \times IC \times \tau}{4 \times IC \times \tau} = 2$$

Instruction Distributions

CPU analysis

Permits performance analysis of machine design "on drawing board"

Evaluate proposed design without building CPU implementation

Summary of procedure

Specify Instruction Set Architecture (ISA)

- Describes machine language for proposed CPU
- Provides human-readable assembly language
- Determines CPI_i for each instruction group i
  - Count clock cycles to implement a single instruction in ISA

Write C, C++, Fortran compilers for proposed machine language

Compile representative programs to machine language

- Can use programs from SPEC CINT and CFP

Sort instructions into groups to find relative instruction count IC_i/IC

Calculate average CPI and run time T

Compare run time with reference machine

From CISC to RISC

CISC Creates the Anti-CISC Revolution

Digital Equipment Company (DEC) introduces VAX (1977)

Commercially successful 32-bit CISC minicomputer

In 1970s and 1980s

CISC minicomputers became cheaper

Serious computers became available to small organizations

UNIX developed as minicomputer operating system

TCP/IP developed to support networks of minicomputers

Computer Science emerged as separate academic discipline

Students needed topics for final projects, theses, dissertations

Research results on CISC performance

Most machine instructions are never used

CISC implementations give up speed in favor of generality

CISC machines run slowly to support unnecessary features

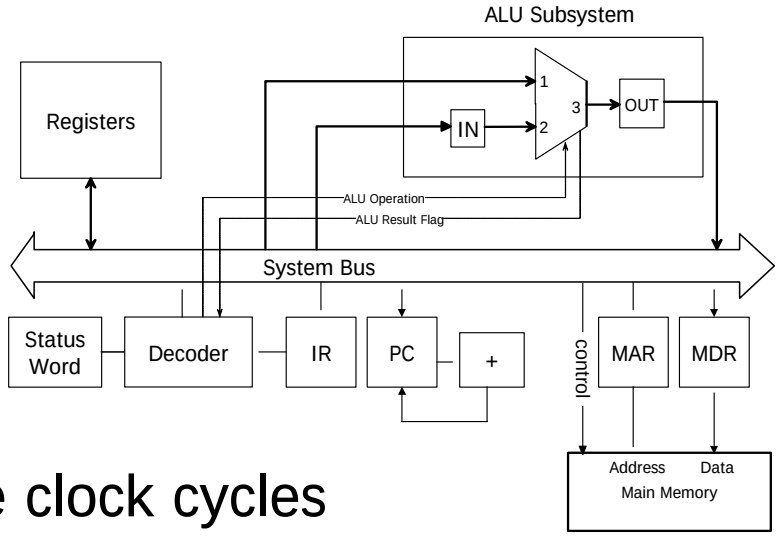
CISC Limitations

CISC instruction set requires microcode

- Many different instruction types
  - Each instruction requires different implementation
- Complex operations
  - Many instructions require complex decoding and sequencing

Central bus organization

- Atomic microcode operations
- System bus = bottleneck
  - Microcode operations — sequential
  - Machine instructions — sequential
- Machine instruction executes in multiple clock cycles



Memory access

- Operation complexity — non-uniform instruction length
- Instruction fetch — multiple clock cycles to load instruction

RISC "Philosophy"

Technological developments from 1975 to 1990

Price of RAM — from $5000 / MByte (1975) to $5 / MByte (1990)

Compilers — powerful and efficient with extensive optimization

Unix, C, and TCP/IP — practical portable code

Principal research result on CISC performance

~ 90% of run time = ~ 10% of VAX ISA

~ 90% of VAX instruction set < 10% of run time

Reduced Instruction Set Computer (RISC) — 1984

Apply Amdahl's "Law" to Instruction Set Architecture (ISA)

- Speed up operations accounting for most of run time
- Ignore performance degradation to other instructions

RISC ISA — keep most important instructions from CISC ISA

- Other CISC instructions implemented as multiple RISC instructions

Simple hardware implementation — faster execution

RISC Microprocessors

Simpler ISA

- Fewer machine instructions
- All instructions are same length

Simpler hardware design

- Allows lower CPI_i and higher clock speed
- No microcode — all instructions implemented in similar way
- No dedicated system bus
- CPU can process several instructions at once
- An instruction completes execution on almost every clock cycle

High level program compiled to RISC

- Larger IC_i — more machine instructions than compilation for CISC
- Run more quickly than same high level programs on CISC

All processors today use RISC technology

- Pure RISC (IBM Power, SPARC, MIPS, ARM, ...)
- RISC technology for CISC language Intel x86 (Pentium, Core, Xeon)
- Explicitly parallel RISC (Intel Itanium, IBM mainframes)

CISC vs. Pure RISC

	CISC	RISC
Instruction Types	300	50
Addressing Modes	15	5
Data Types	10	2
Procedure Handling	Automated	Coded
Implementations	Complex	Simple
Memory Organization	Complex	Simple

$$S = \frac{T}{T'} = \frac{CPI^{CISC} \times IC^{CISC} \times \tau^{CISC}}{CPI^{RISC} \times IC^{RISC} \times \tau^{RISC}} = \frac{CPI^{CISC}}{CPI^{RISC}} \times \frac{IC^{CISC}}{IC^{RISC}} \times \frac{\tau^{CISC}}{\tau^{RISC}}$$

$$= (\approx 6) \times \left(\approx \frac{1}{2} \right) \times \frac{\tau^{CISC}}{\tau^{RISC}}$$

$$\approx 3 \times \frac{\tau^{CISC}}{\tau^{RISC}}$$

Designing a RISC ISA

Considerations for a RISC ISA

Goals

- Simple — no instruction should require more steps than others
- Complete — able to perform any desired computation
- Orthogonal — only one way to encode any given computation

Choices

- Computation model
  - Register-register
  - Register-memory
- Range and type of operations
- Operands
  - Data types
  - Data sizes
- Addressing modes
  - Displacement sizes
- Branch types
  - Conditional
  - Unconditional
  - Procedural (call/return)
  - Branch offset (length of jump)

Instruction Types

Representative instruction distribution

Five programs from SPECint92 benchmark suite

Compile for x86 instruction set (ISA for Intel 386/486/Pentium)

Instruction	Relative Proportion of Total Run Time
Load	22%
Conditional branch	20%
Compare	16%
Store	12%
Add	8%
And	6%
Sub	5%
Move reg-reg	4%
Call	1%
Return	1%
Other	5%
Total	**100%**

First 10 instructions account for 95% of run time

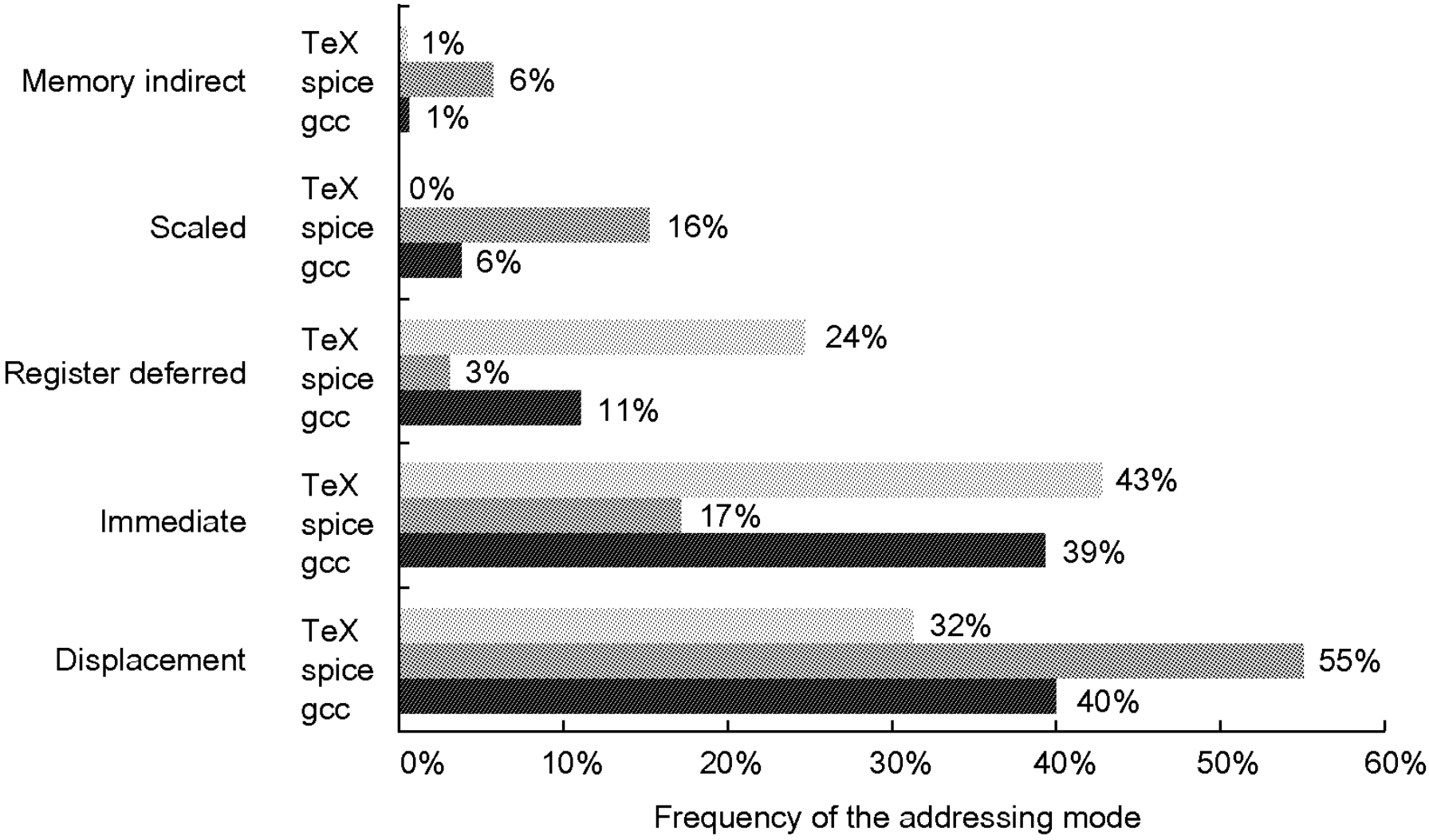
Amdahl's "Law"

Fast implementation of 95%
Other 5% will not seriously degrade performance

Must include unconditional branch for completeness

Ref: Hennessy / Patterson, figure 2.11

Addressing Modes Graph



Ref: Hennessy / Patterson, figure 2.6

Addressing Modes

Representative instruction distribution

Three programs from SPEC CINT92 and SPEC CFP95 benchmarks

Compile for VAX instruction set

Mode	tex	spice	gcc	Example of Mode
register deferred	24	3	11	`mem[R1]`
immediate	43	17	39	`#11223344`
displacement	32	55	40	`mem[R1 + disp]`
memory indirect	1	6	1	`mem[mem[R1]]`
scaled	0	16	6	`mem[R1 + R2 * d + disp]`
other	0	3	3	
total	**100**	**100**	**100**	
total (top 3)	**99**	**75**	**90**	

First three addressing modes

Account for more than 75% of all operand accesses

Ref: Hennessy / Patterson, figure 2.6

Instruction Length

Instructions should be of uniform length

Simplifies instruction **DECODING**

- No need to calculate instruction length
- Instruction fields are always in same place

Enables **INSTRUCTION FETCH** in 1 clock cycle

Practical instruction lengths

Most RISC machines for servers/workstations use 32-bit instructions

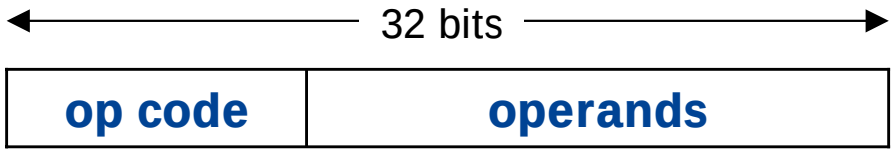
Special purpose RISC machines use longer instructions

Itanium and mainframes use 128-bit instructions

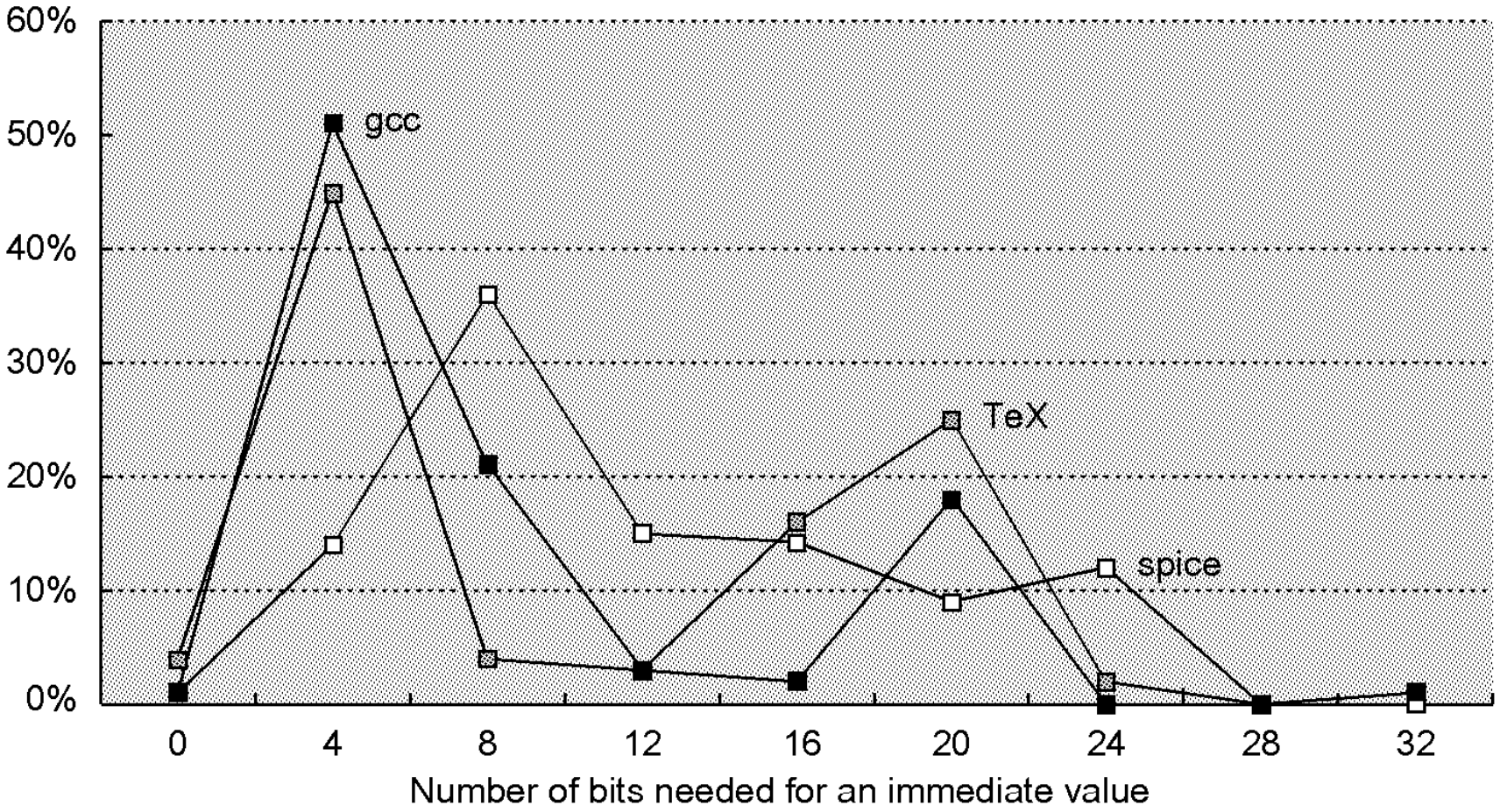
ISA defines 32-bit instructions

No single field can be 32 bits long

Includes address displacements, immediates, branch length



Length of Immediate Operand Graph



Ref: Hennessy / Patterson, figure 2.9

Length of Immediate Operand

Representative instruction distribution

Three programs from SPEC CINT92 and SPEC CFP95 benchmarks

Compile for VAX instruction set

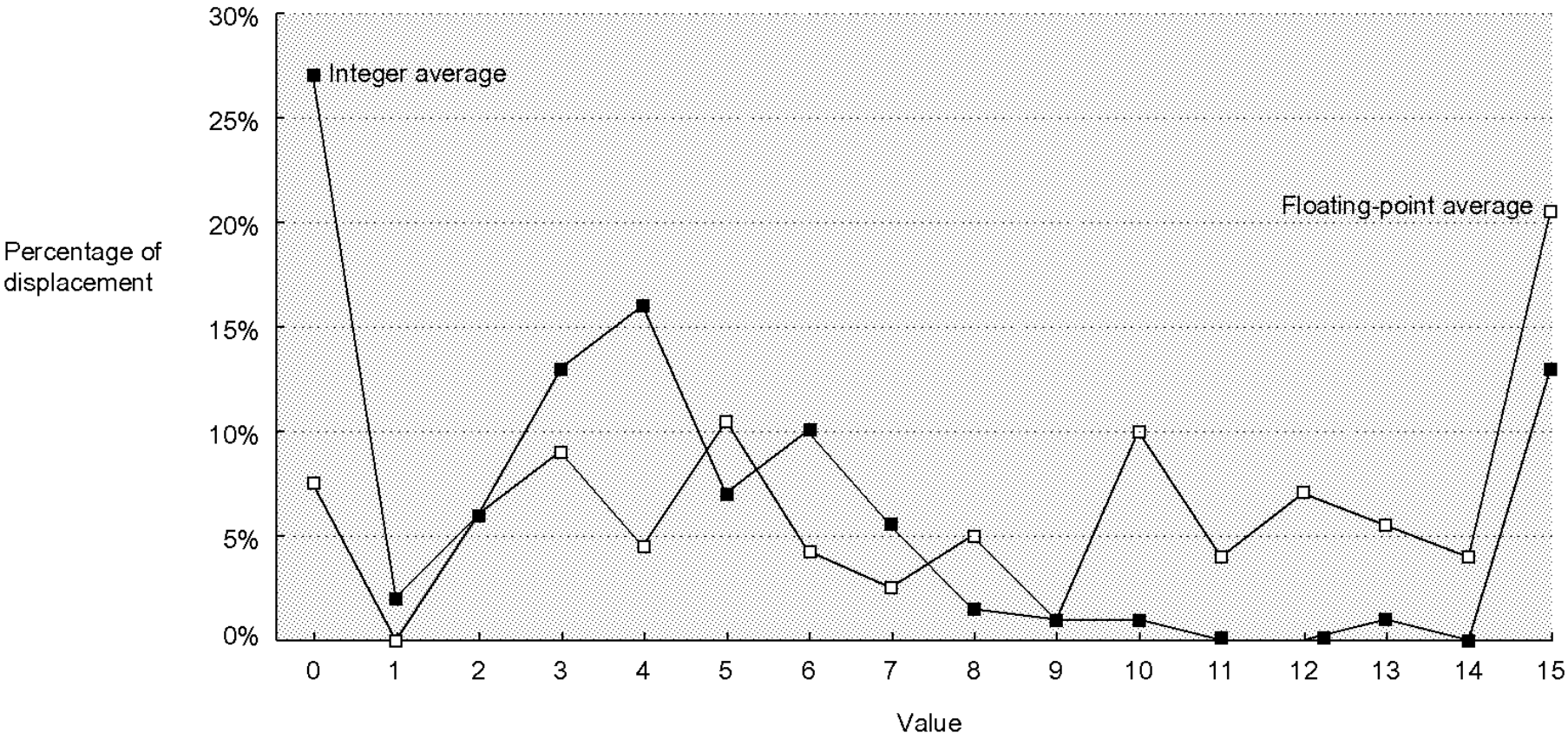
Immediate size	tex	spice	gcc
0	3	1	1
4	45	13	50
8	4	35	22
12	3	15	4
16	15	14	3
20	25	10	18
24	2	12	0
28	1	0	0
32	2	0	2
Total	**100**	**100**	**100**
Total to 16 bits	**70**	**78**	**80**

#1122

Allocating **16 bits** in 32-bit instruction for **immediate operands** covers more than 70% of cases

Ref: Hennessy / Patterson, figure 2.9

Displacement Length Graph



Ref: Hennessy / Patterson, figure 2.7

Displacement Length

Representative instruction distribution

Programs from SPEC CINT92 and SPEC CFP95 benchmarks

Compile for VAX instruction set

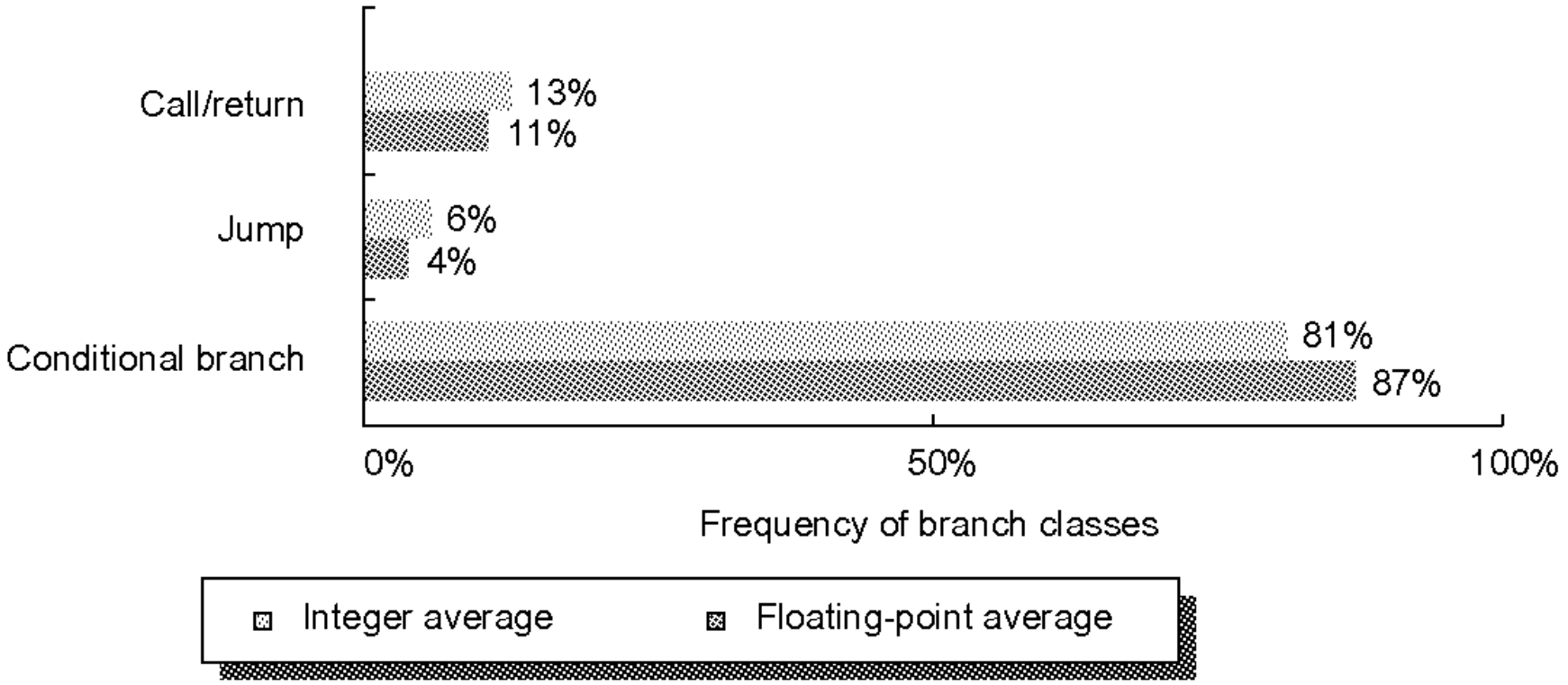
Bits in address displacement	int	FP
0	26	7
1	1	0
2	6	6
3	12	8
4	16	5
5	6	10
6	10	4
7	6	3
8	2	5
9	1	1
10	1	10
11	0	4
12	0	7
13	1	6
14	0	4
15	12	20
Total	**100**	**100**

Allocating **16 bits** for **address displacements** covers almost all cases

`mem[R1 + 1122]`

Ref: Hennessy / Patterson, figure 2.7

Branch Instructions Graph



Ref: Hennessy / Patterson, figure 2.12

Branch Instructions

Representative instruction distribution

Programs from SPEC CINT92 and SPEC CFP95 benchmarks

Compile for VAX instruction set

	Integer	FP
Call / Return	13	10
Unconditional Branch	6	4
Conditional branch	81	86
Total	100	100
Total of Conditional and Unconditional Branch	87	90

Conditional branch accounts for more than 80% of all branch instructions

Unconditional branch must be included for completeness

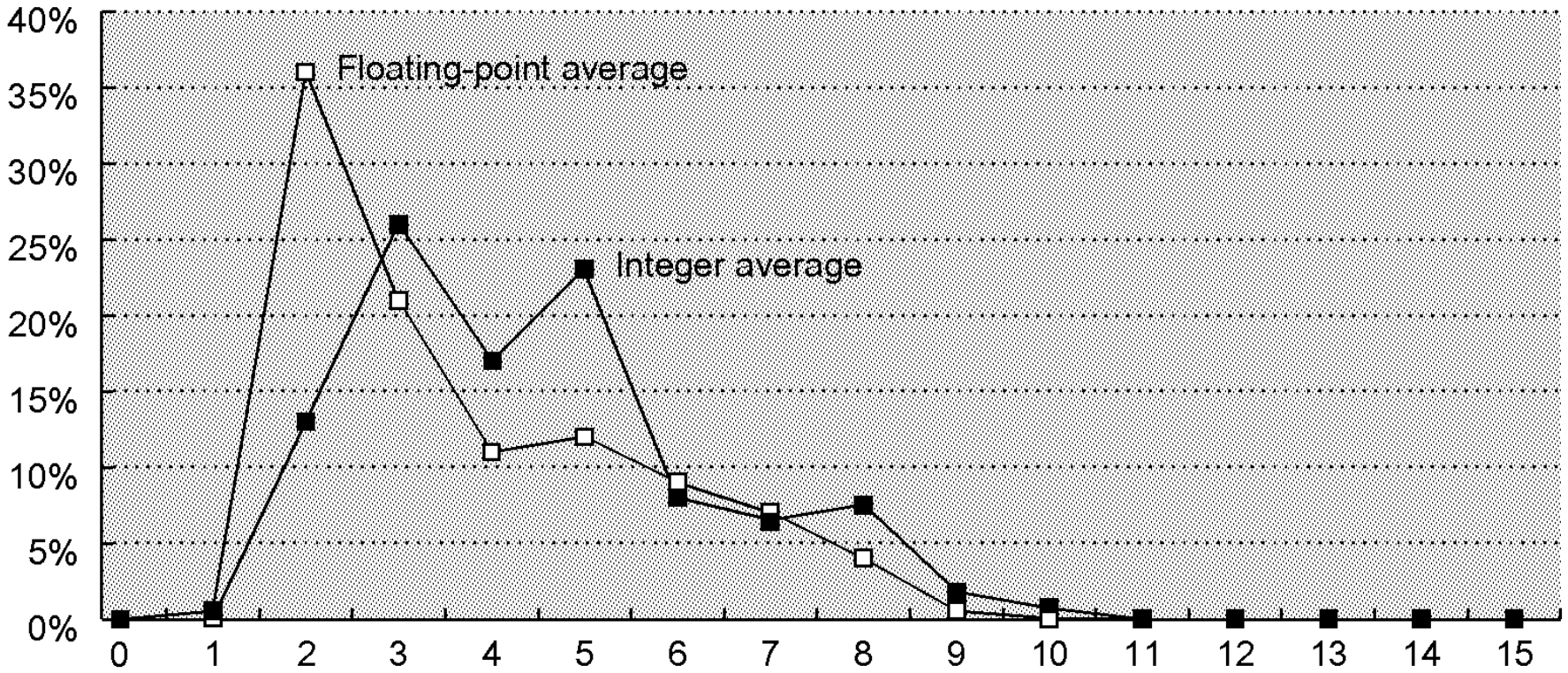
Call and return

Include many steps — saving registers and branching

Are difficult to implement

Ref: Hennessy / Patterson, figure 2.12

Branch Offset Graph



Ref: Hennessy / Patterson, figure 2.13

Branch Offset

Representative instruction distribution

Programs from SPEC CINT92 and SPEC CFP95 benchmarks

Compile for VAX instruction set

Offset bits for branch address	int	FP
0	0	0
1	1	0
2	13	36
3	26	21
4	16	11
5	24	12
6	6	9
7	5	6
8	6	4
9	2	1
10	1	0
11	0	0
12	0	0
13	0	0
14	0	0
15	0	0
Total	**100**	**100**

Allocating **16 bits** for **branch offsets** covers almost all cases

PC ← PC + 1122

Ref: Hennessy / Patterson, figure 2.13

Summary — RISC ISA By the Numbers

Instruction Types

10 instructions cover 95% of run time

Choose 30 – 50 most necessary / convenient instructions

Addressing Modes

Register

Immediate

Displacement

75% – 90% of run time addressing modes

Instruction Length

32-bit instructions

Branch Instructions

Conditional branch

Unconditional branch

75% – 90% of run time addressing modes

Length of immediate values

16-bit length for

Immediate operand	70% – 80% of run time immediates
Displacement	100% of run time address displacements
Branch offset	100% of run time branch offsets

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

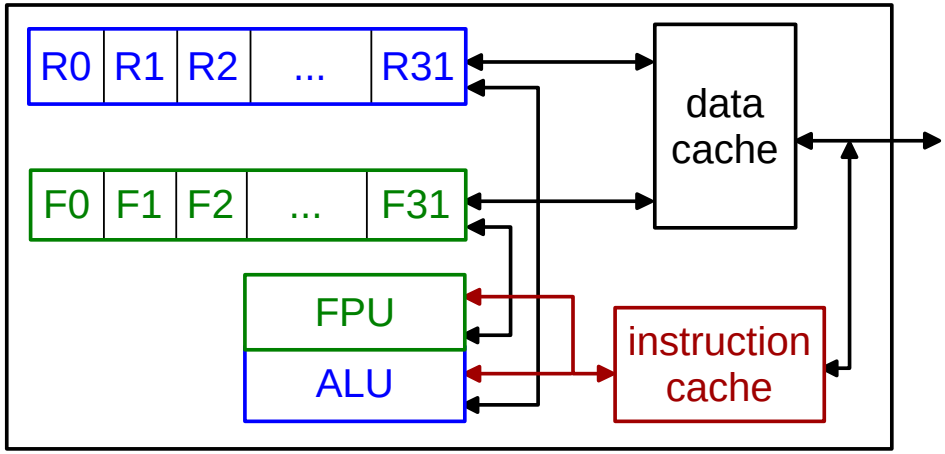
- Fewer instructions
- Simpler hardware implementation
- Appropriate for teaching

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	$\begin{pmatrix}Reg[F0]\\Reg[F1]\end{pmatrix} \xleftarrow{64} \begin{pmatrix}Reg[F2]\\Reg[F3]\end{pmatrix} + \begin{pmatrix}Reg[F4]\\Reg[F5]\end{pmatrix}$
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** **($-2^{25} \le$ 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 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)

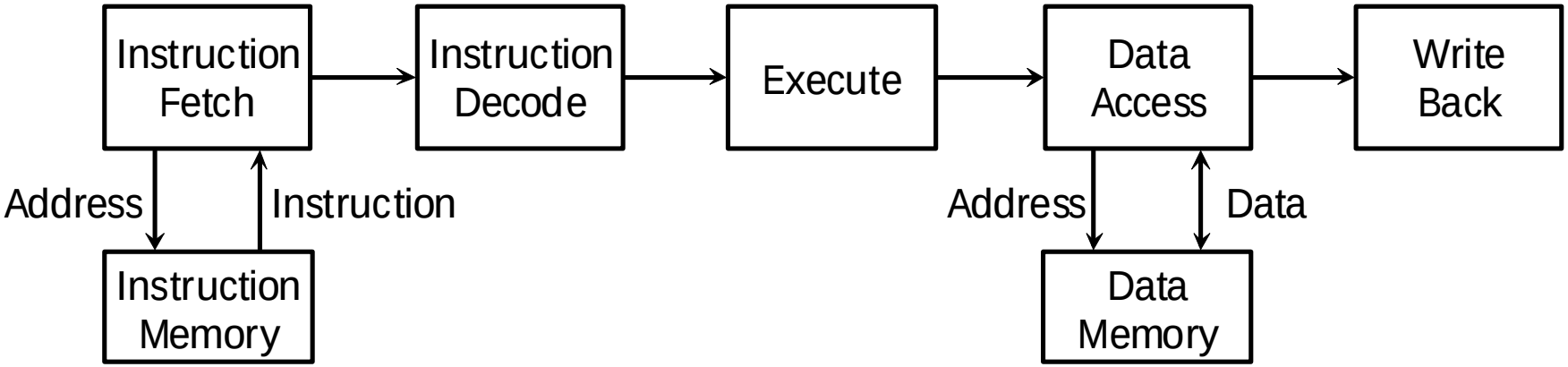
First two stages identical for all instructions — **FETCH** and **DECODE**

Last three stages operate according to instruction

EXECUTE (ALU instructions and address calculations)

MEMORY ACCESS (Load/Store instructions)

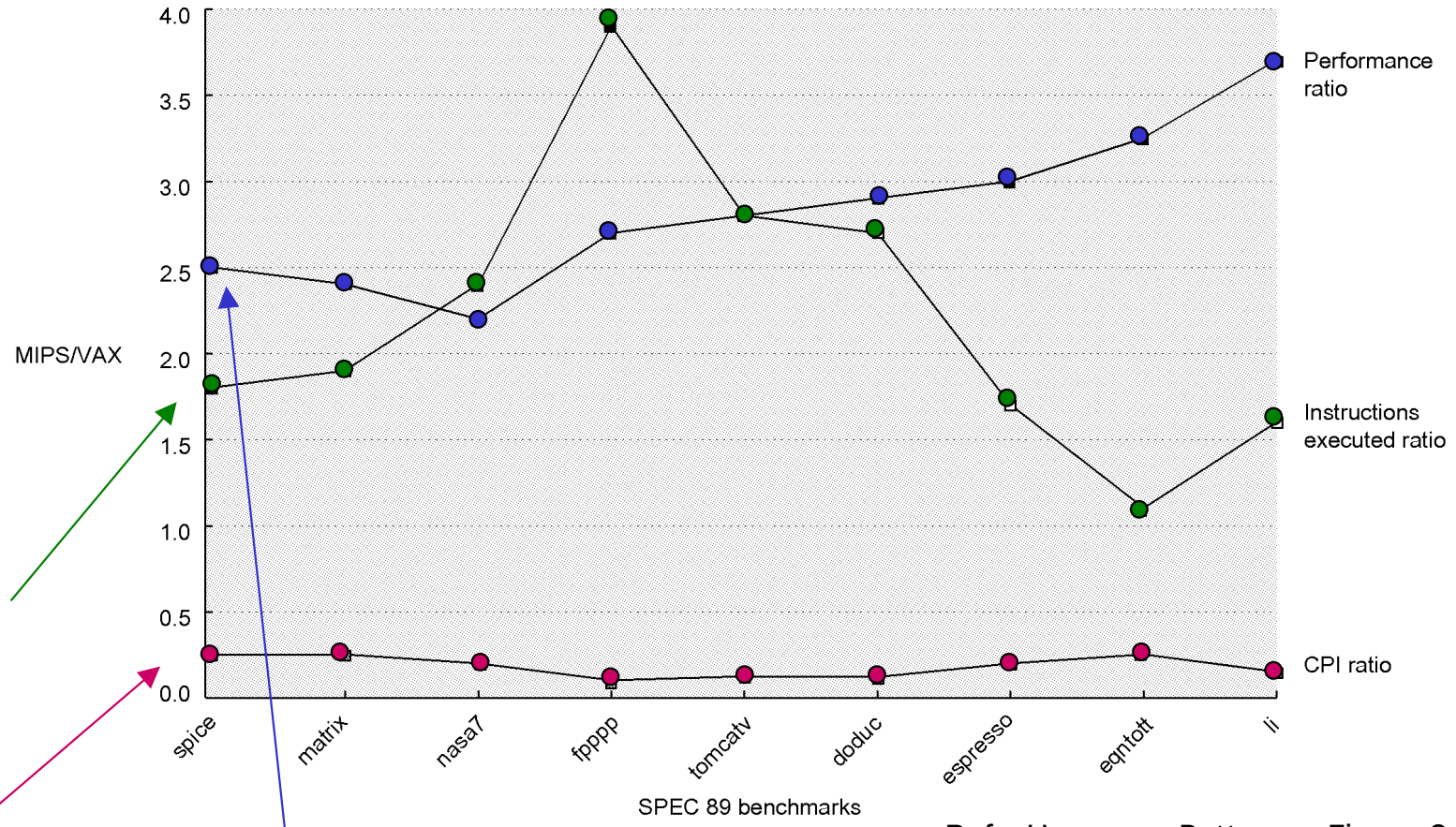
WRITE BACK (register update for Load and ALU instructions)



RISC Performance

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

Same clock rate



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

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

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

Ref: Hennessy-Patterson Figure 2-30

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



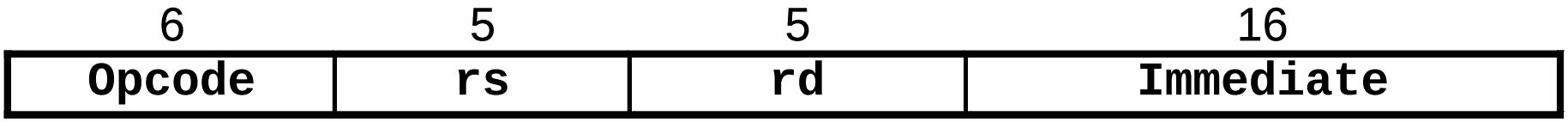
Encodes:

- 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

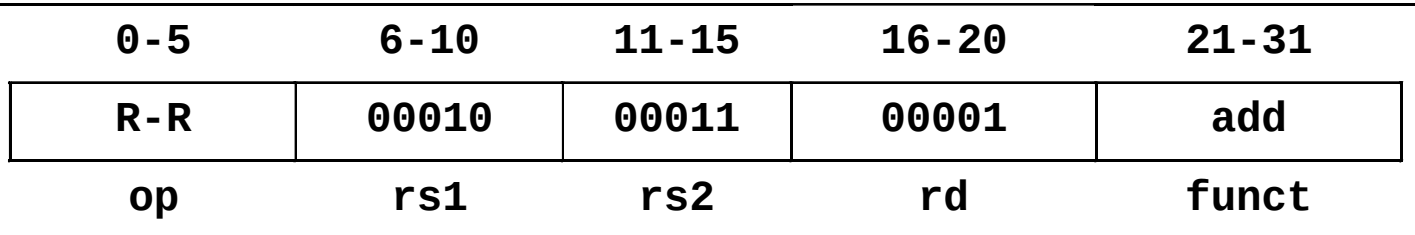
Condition flag

Result of test for conditional branch

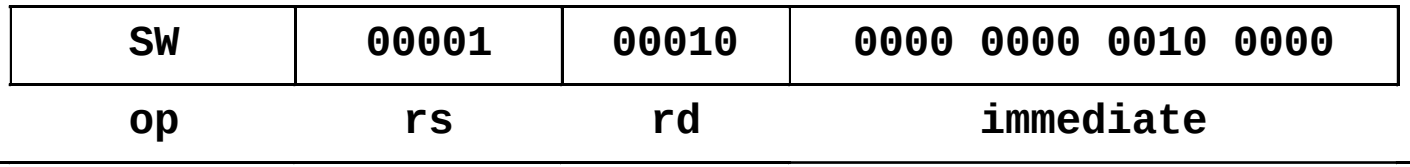
Example Type-I ALU Instruction

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

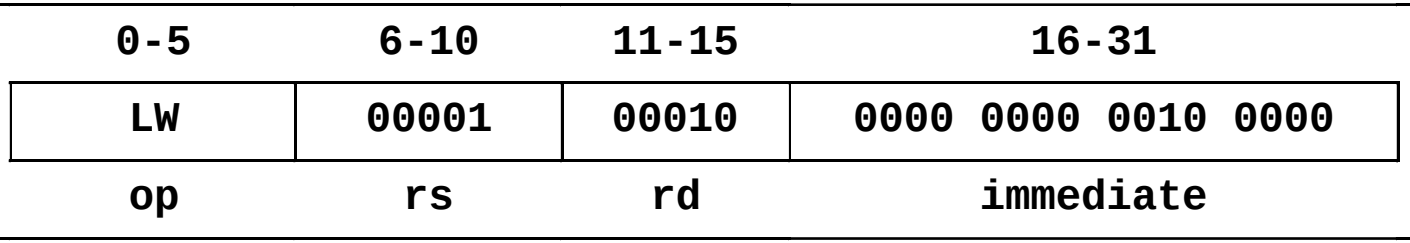
Example Type-R ALU Instruction

Instruction	`add R1, R2, R3`
Operation	`Reg[R1] ← Reg[R2] + Reg[R3]`
Encoding	
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← 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] */ $I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$
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 ← 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 ← 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	

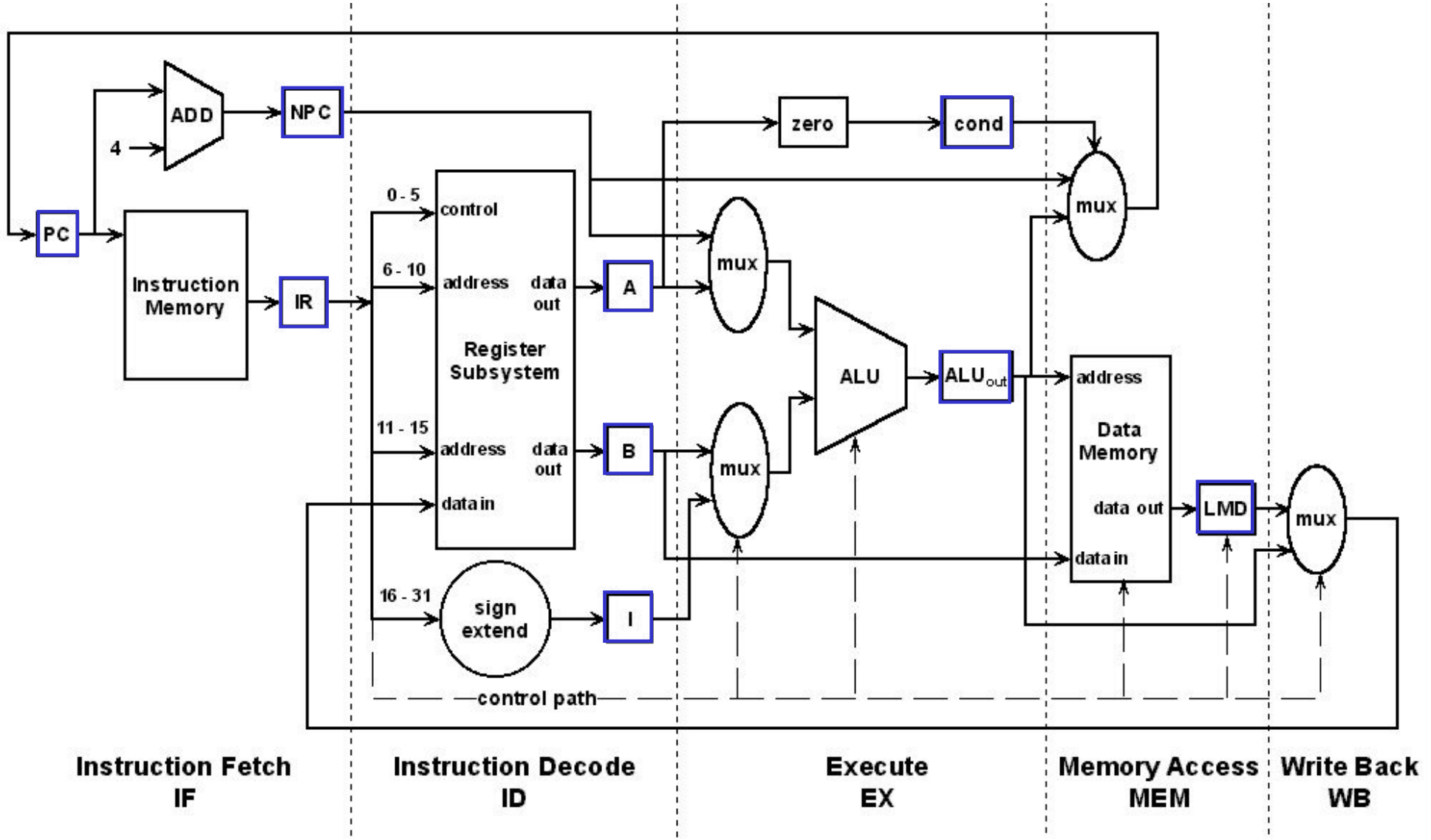
Example Type-I Load Instruction

Instruction	`LW R2, 32(R1)`
Operation	`Reg[R2] ← Mem[32+Reg[R1]]`
Encoding	
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

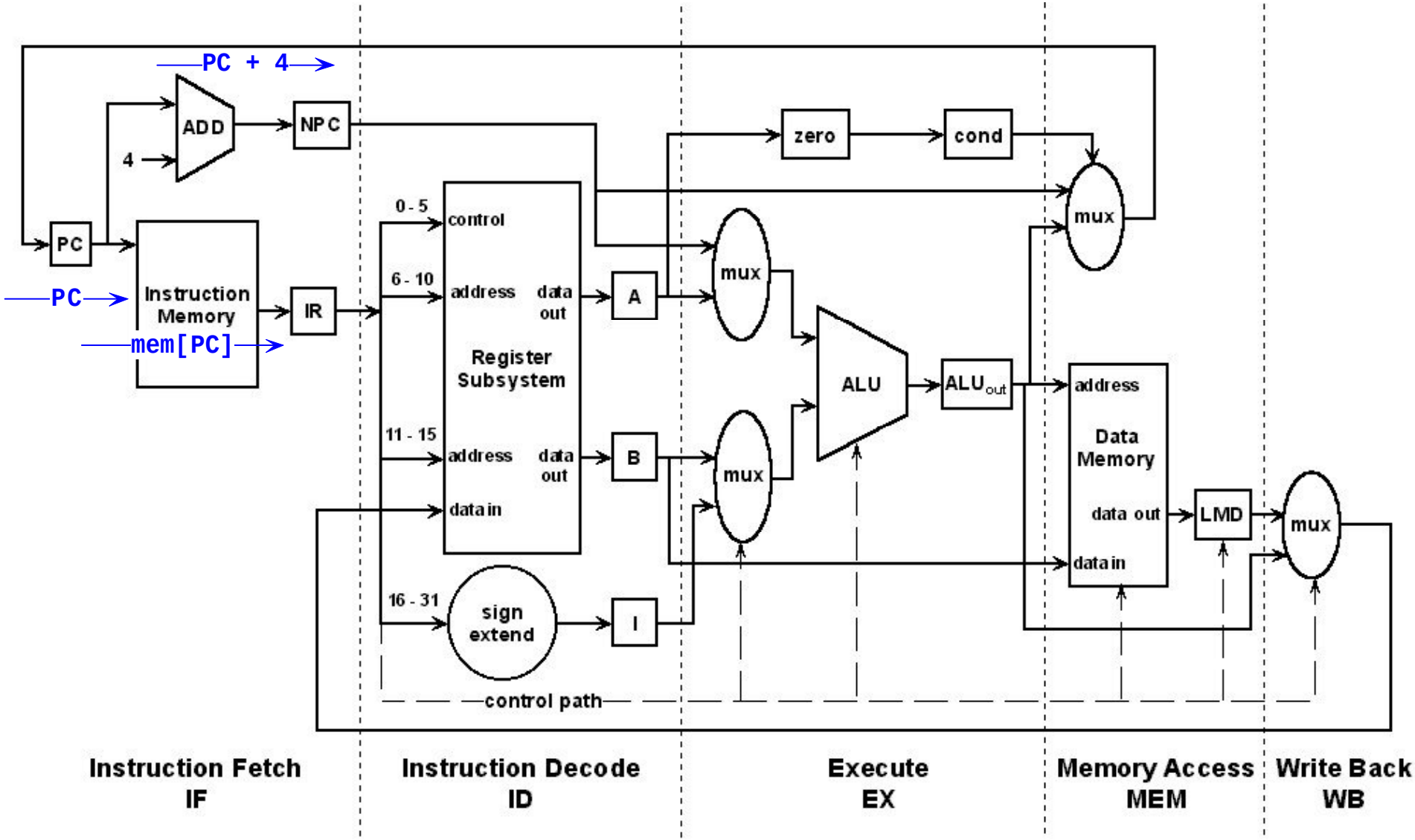
Instruction	`beqz R1, 1024`
Operation	`if (Reg[R1] == 0) PC ← NPC + 1024` `else PC ← NPC`
Encoding	0-5 | 6-10 | 11-15 | 16-31 beqz | 00001 | 00000 | 0000 0100 0000 0000 op | rs | rd | immediate
Hardware Stage 1	`IR ← Mem[PC]` `NPC ← PC + 4`
Hardware Stage 2	`A ← Reg[IR6-10]` /* A ← Reg[R1] */ `B ← Reg[IR11-15]`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



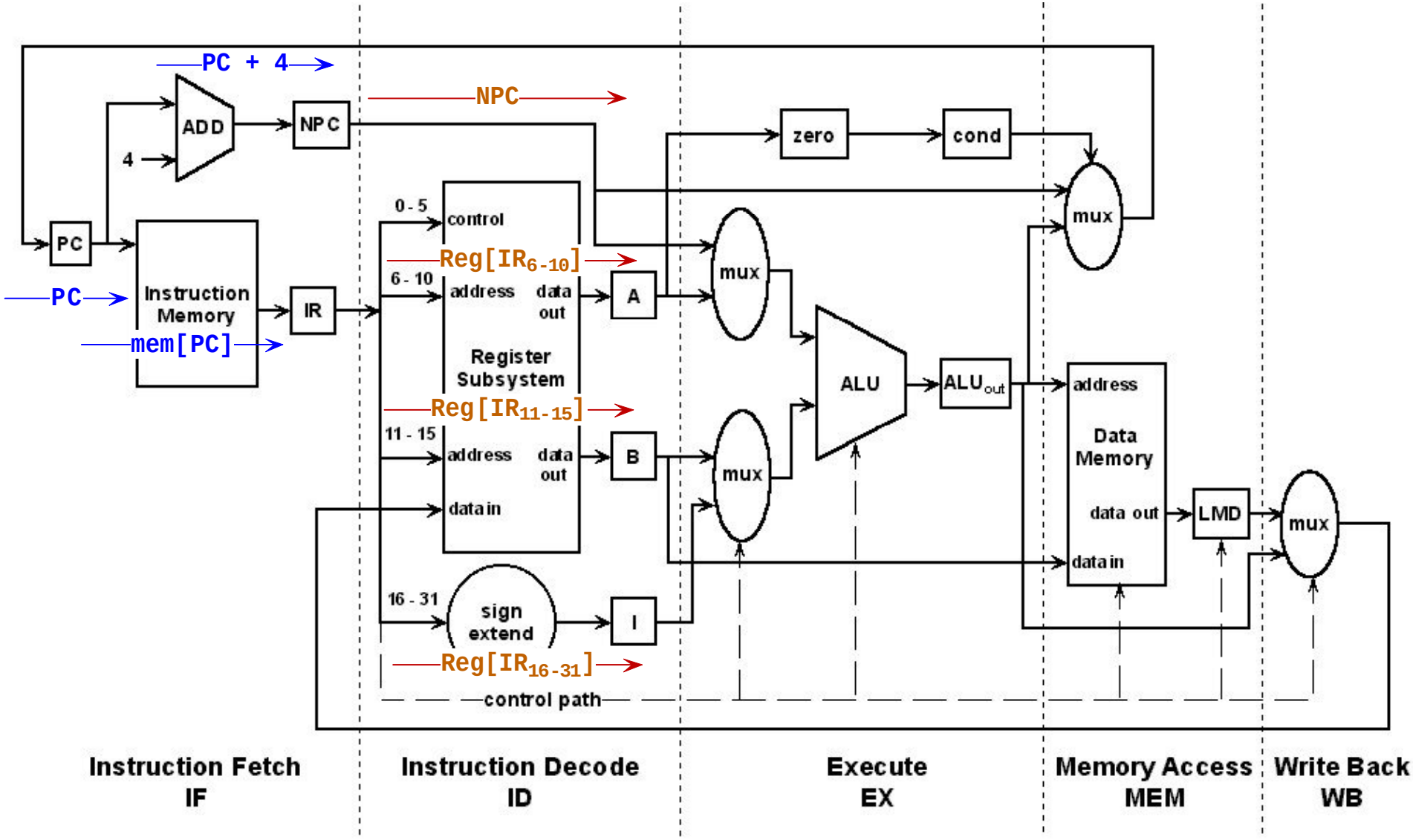
mux (multiplexer) — chooses 1 output from N inputs

Type-I ALU Instruction — 1



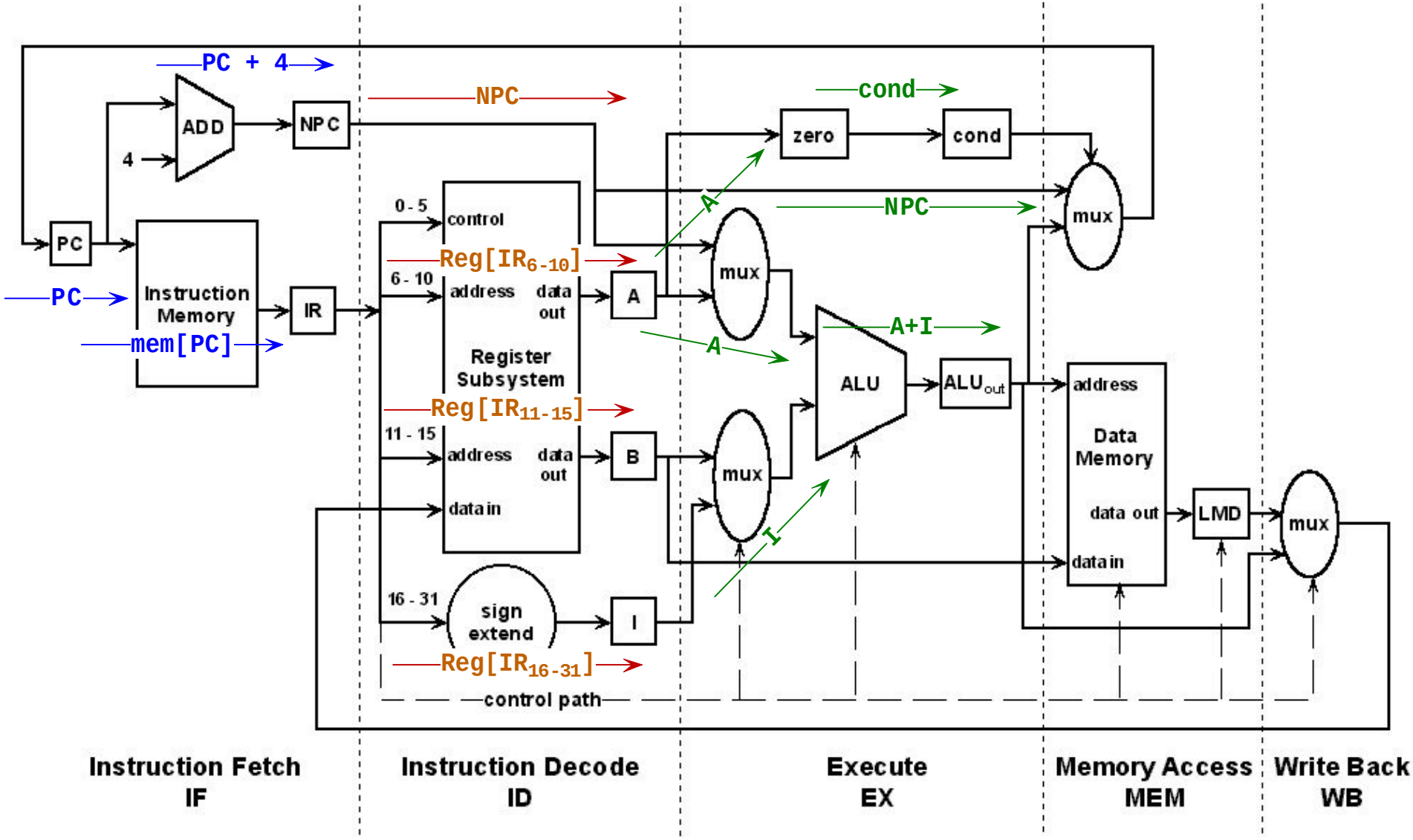

```
addi r1, r2, #5        regs[r1] ← regs[r2] + 5
```

Type-I ALU Instruction — 2



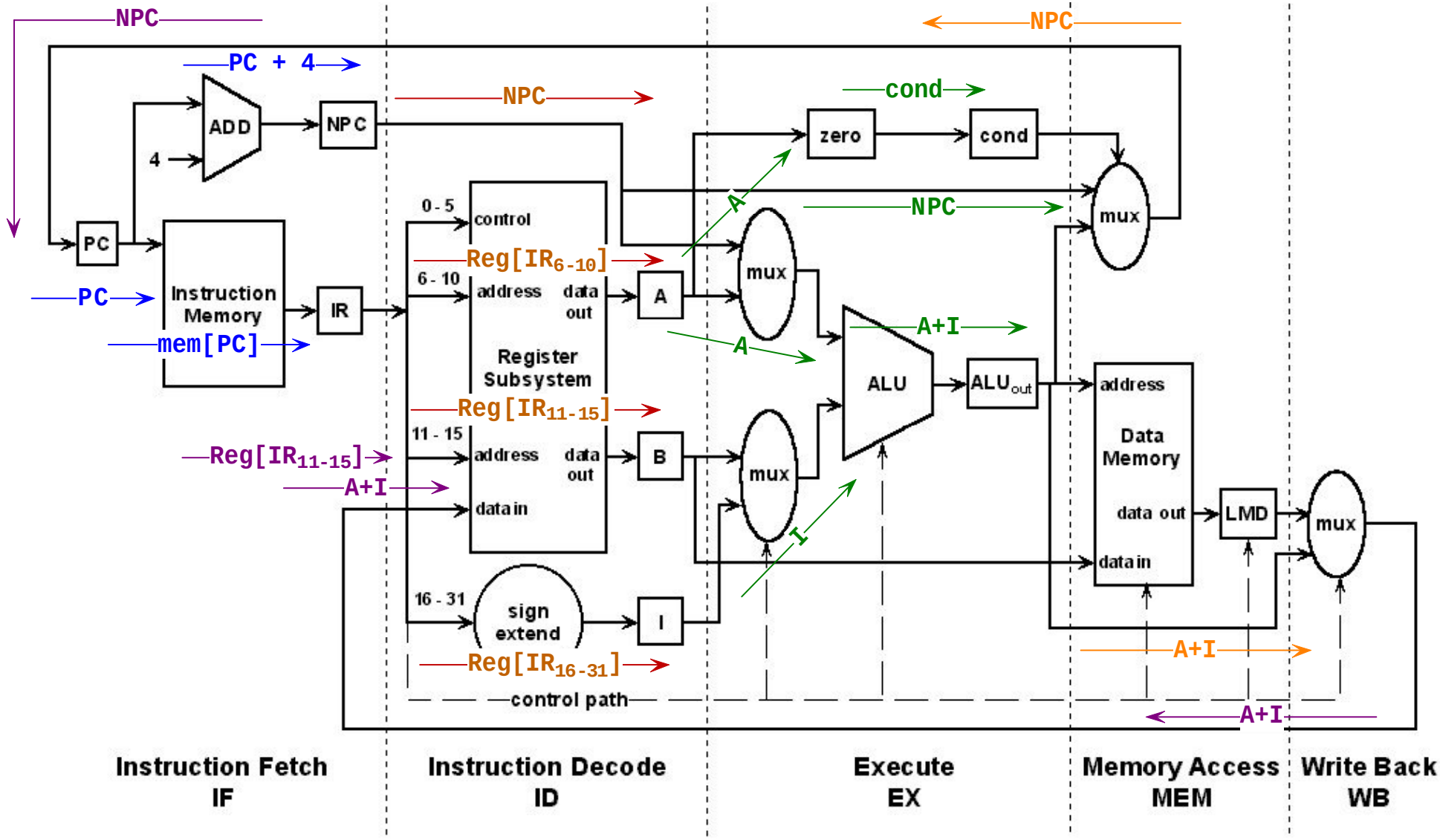

```
addi r1, r2, #5        regs[r1] ← regs[r2] + 5
```

Type-I ALU Instruction — 3



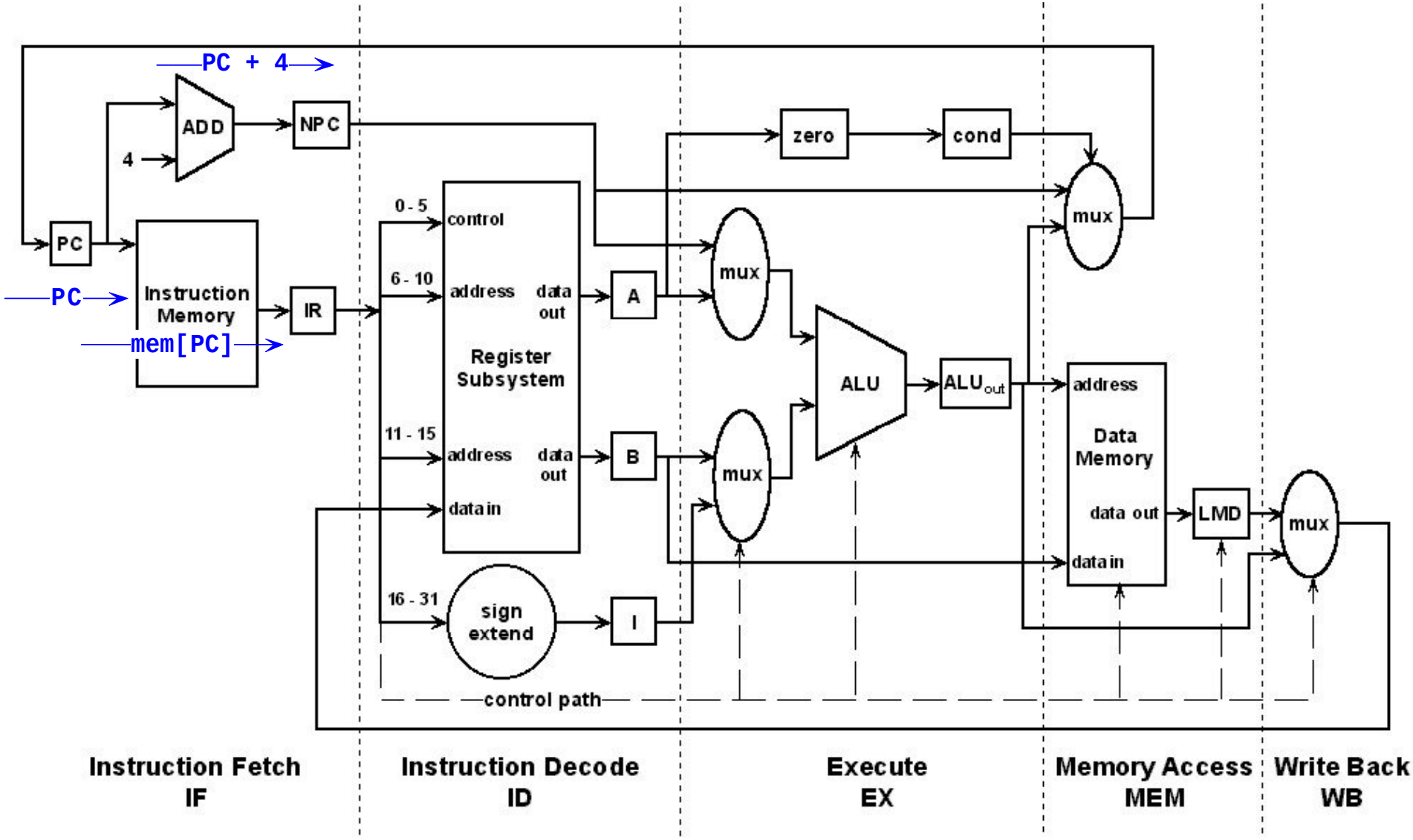

```
addi r1, r2, #5        regs[r1] ← regs[r2] + 5
```

Type-I ALU Instruction — 4



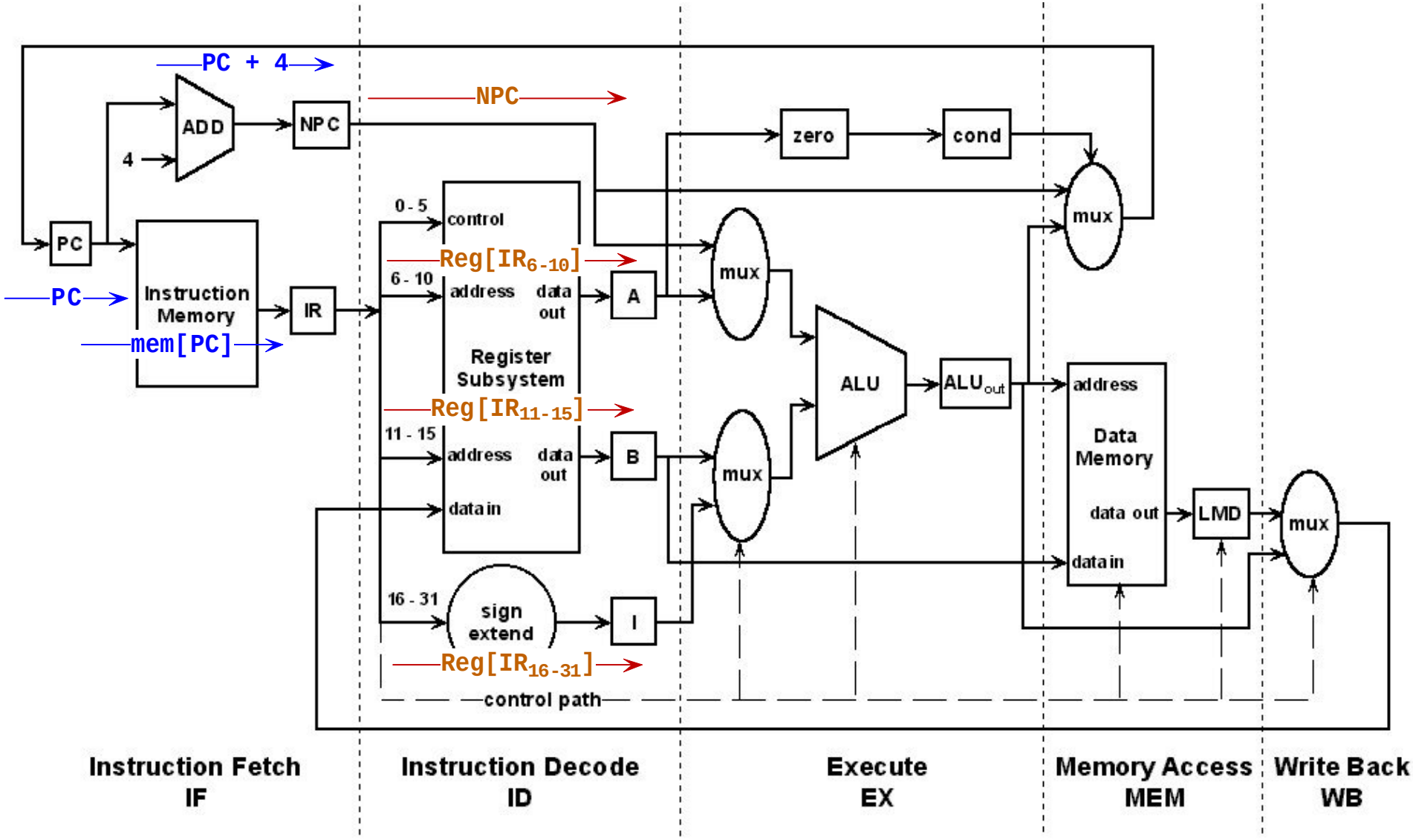

```
addi r1, r2, #5        regs[r1] ← regs[r2] + 5
```

Type-R ALU Instruction — 1



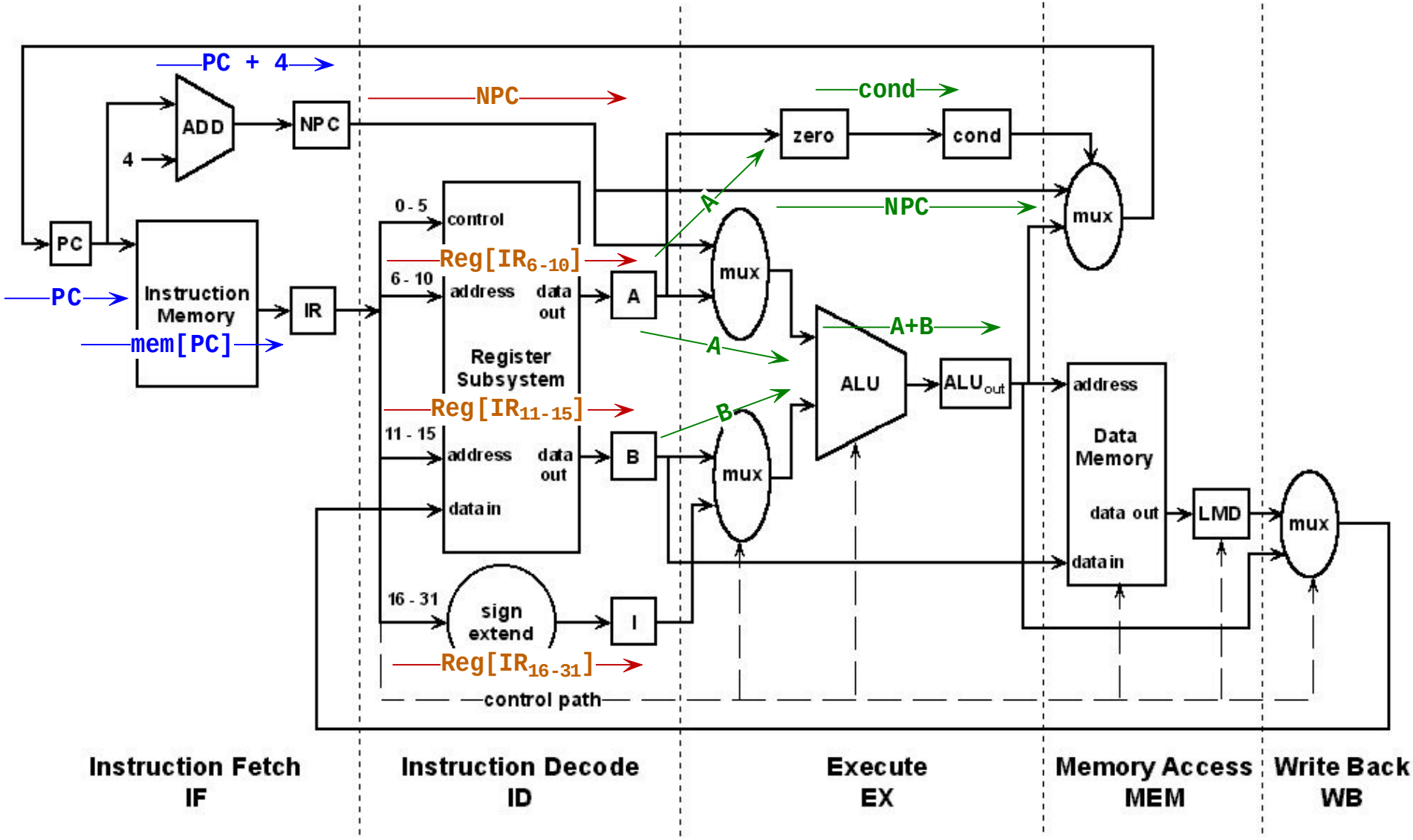

```
add r1, r2, r3          regs[r1] ← regs[r2] + regs[r3]
```

Type-R ALU Instruction — 2



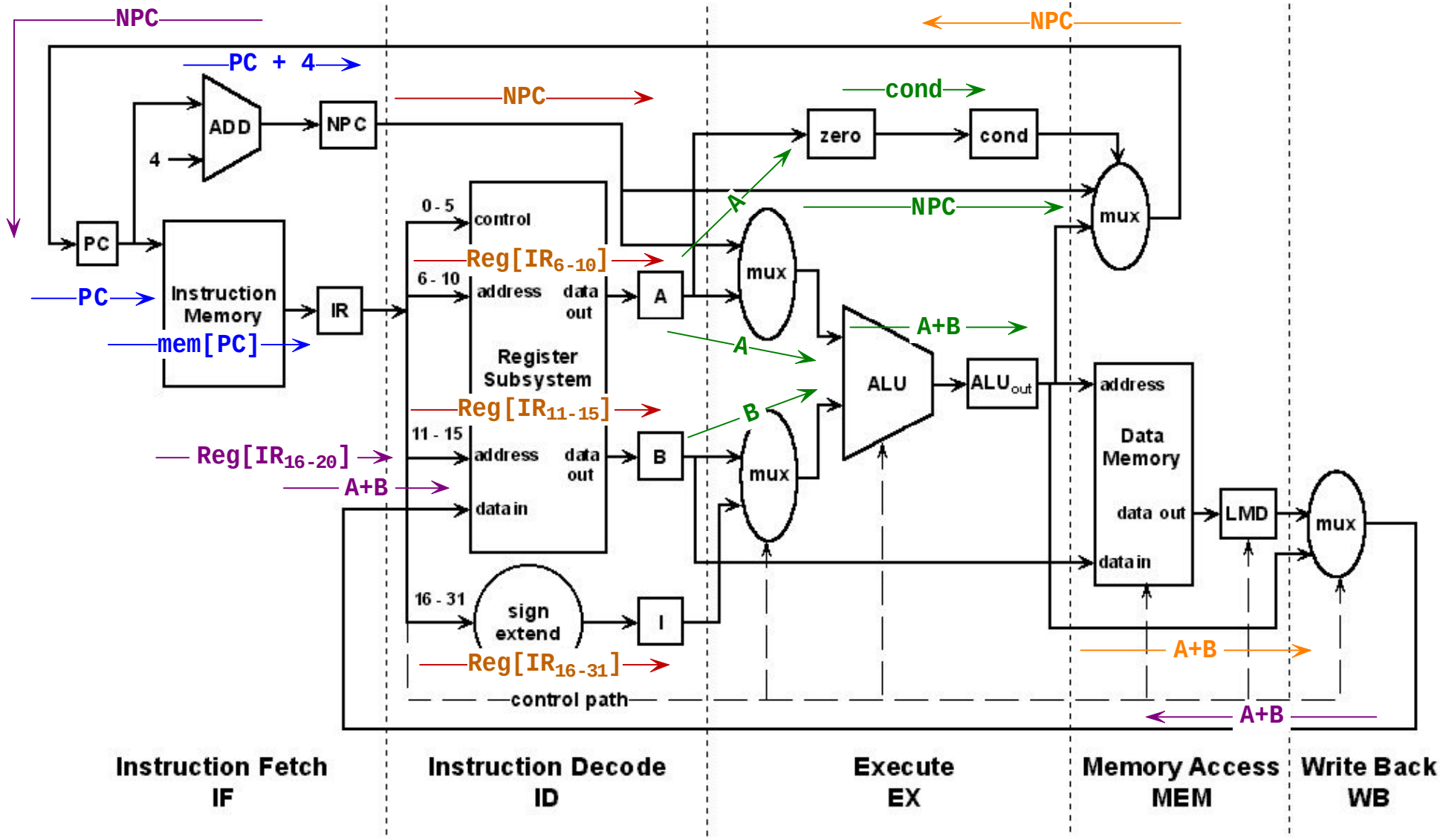
`add r1, r2, r3` `regs[r1] ← regs[r2] + regs[r3]`

Type-R ALU Instruction — 3



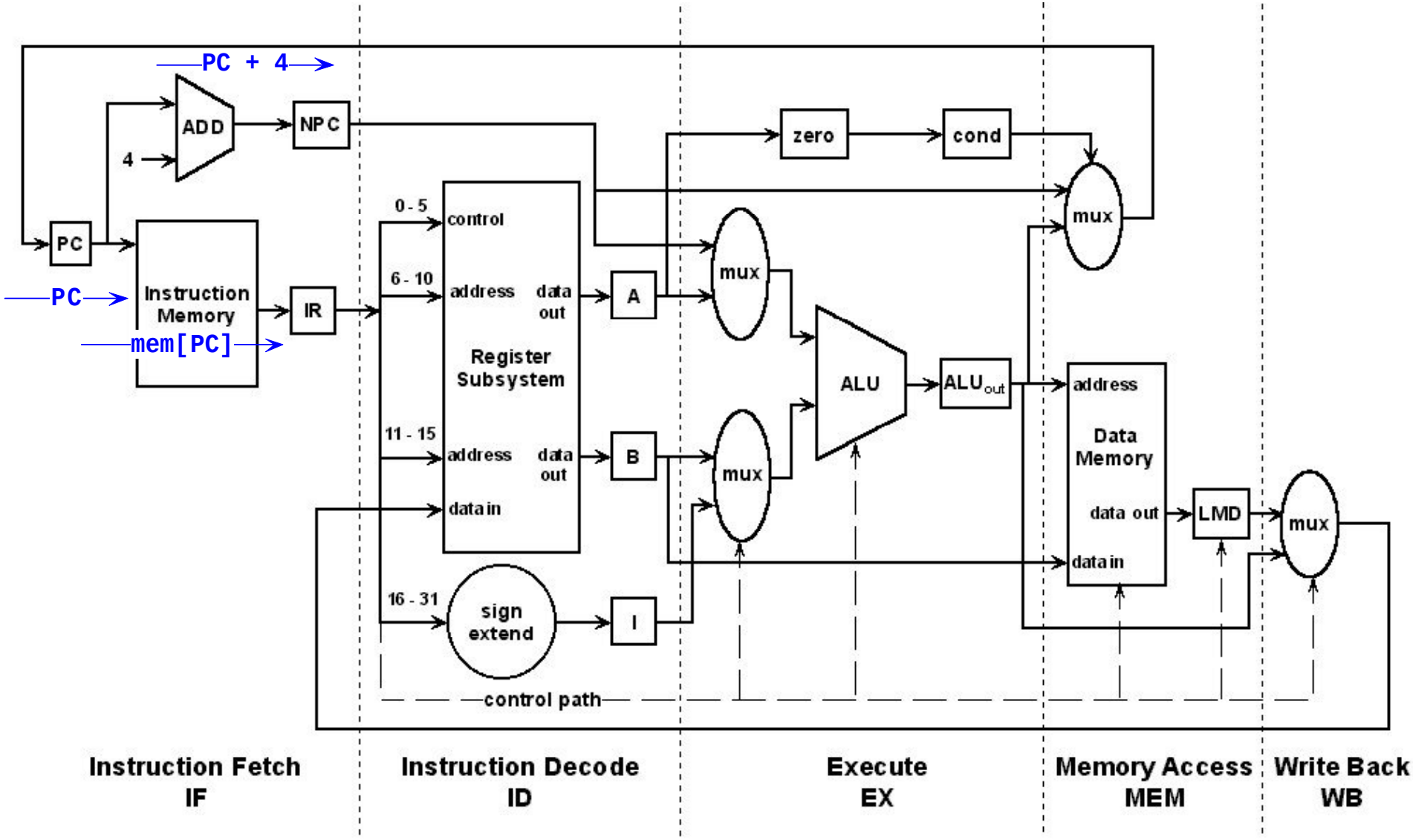

```
add r1, r2, r3          regs[r1] ← regs[r2] + regs[r3]
```

Type-R ALU Instruction — 4



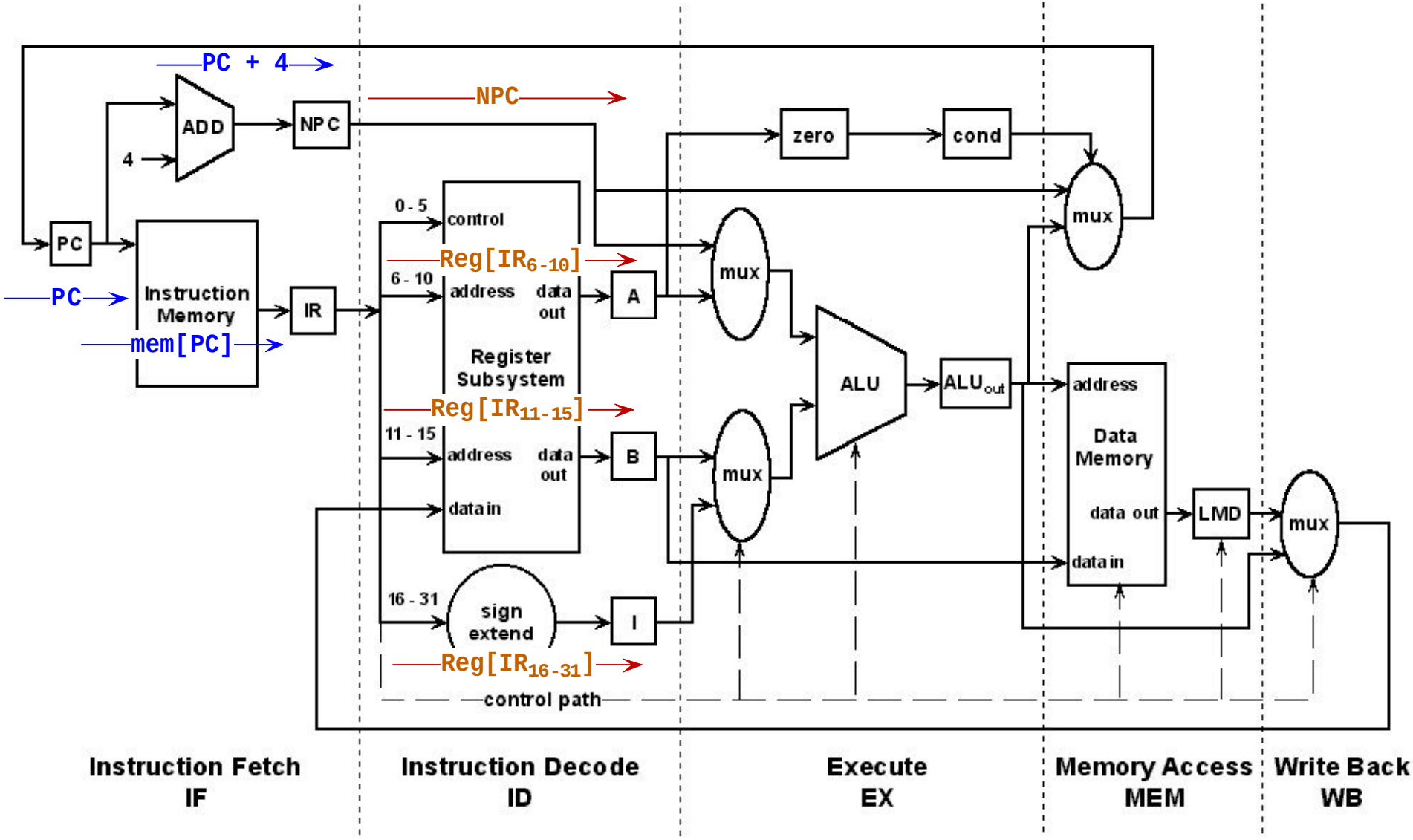

```
add r1, r2, r3          regs[r1] ← regs[r2] + regs[r3]
```

Type-I Store Instruction — 1



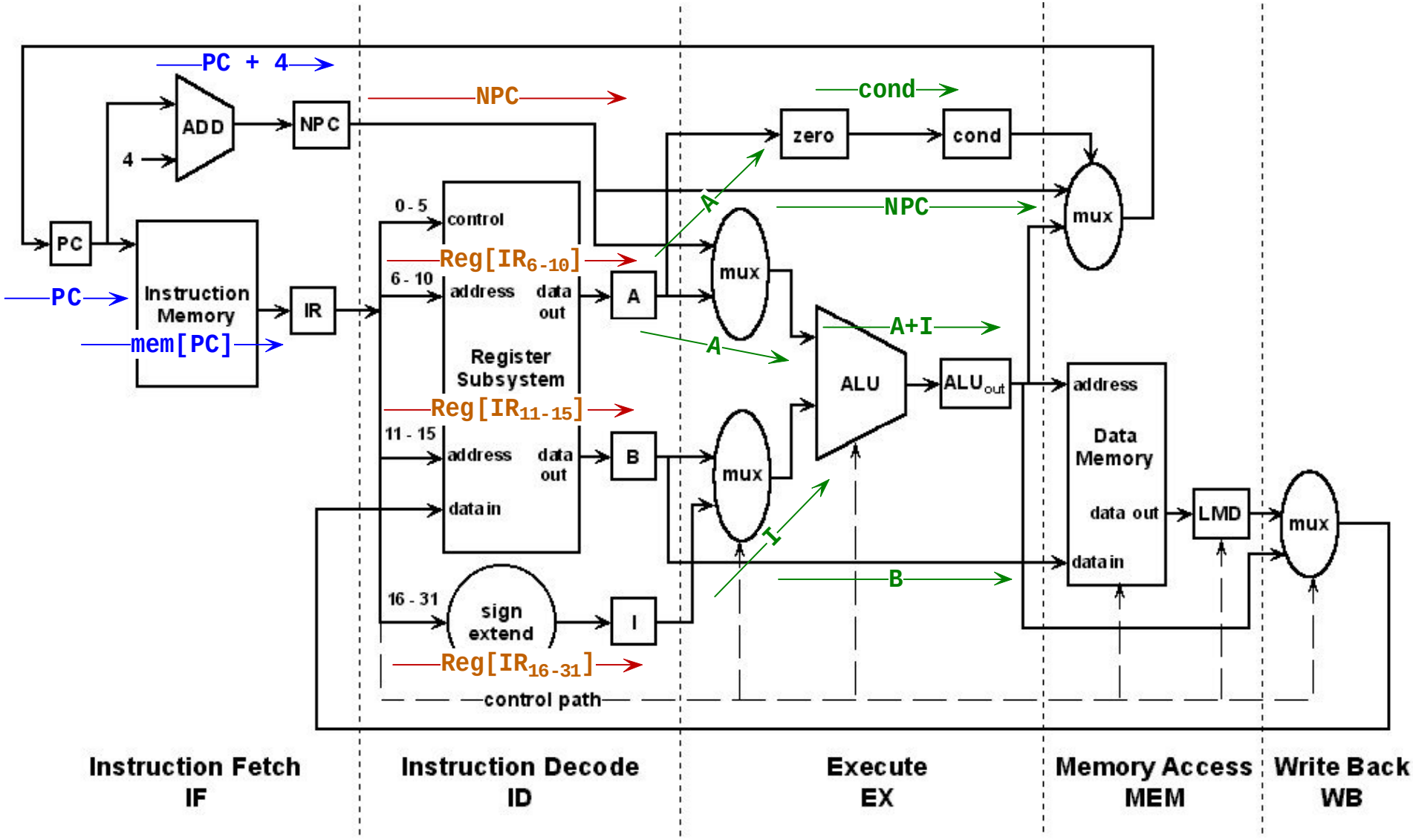
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Store Instruction — 2



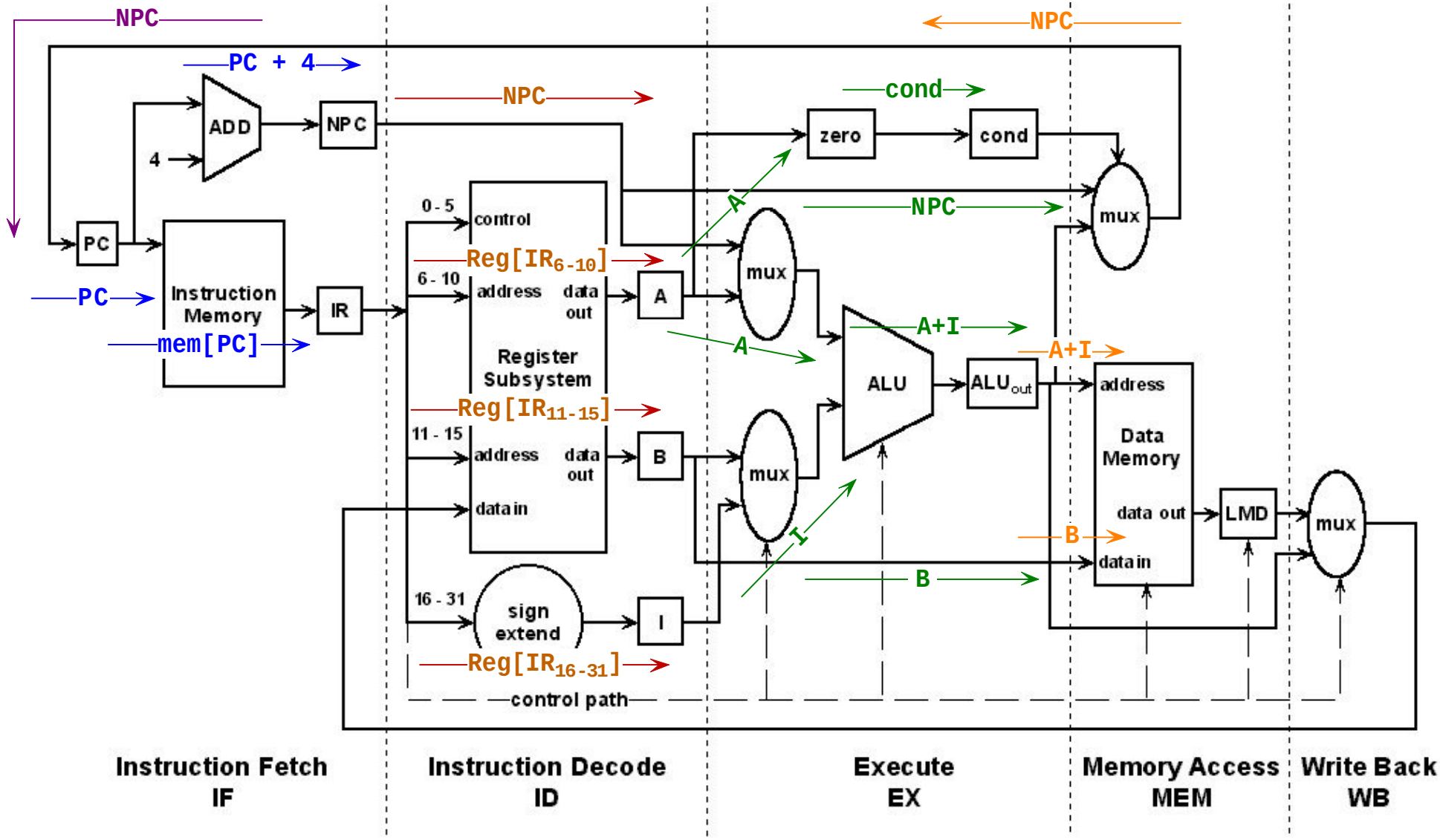
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Store Instruction — 3



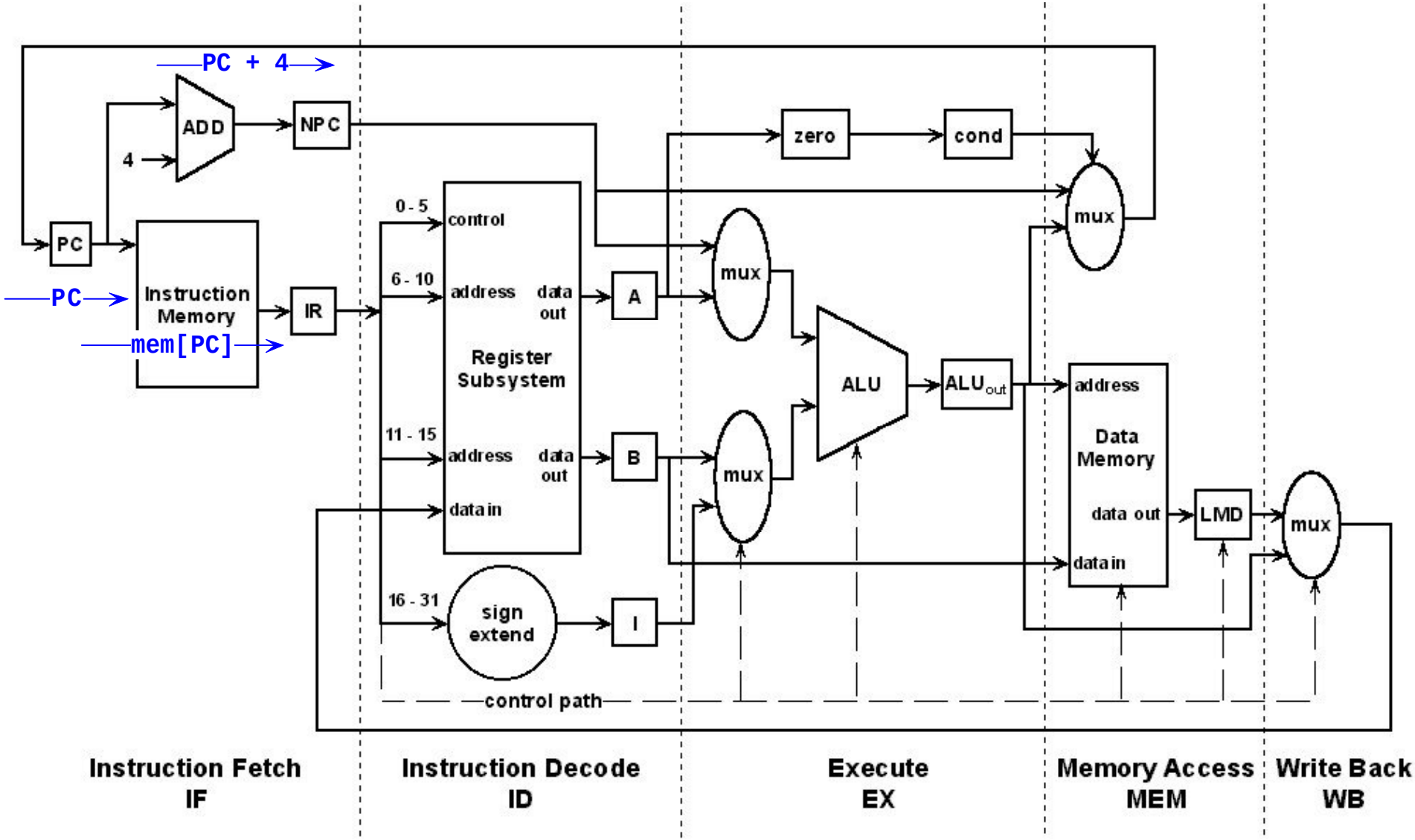
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Store Instruction — 4



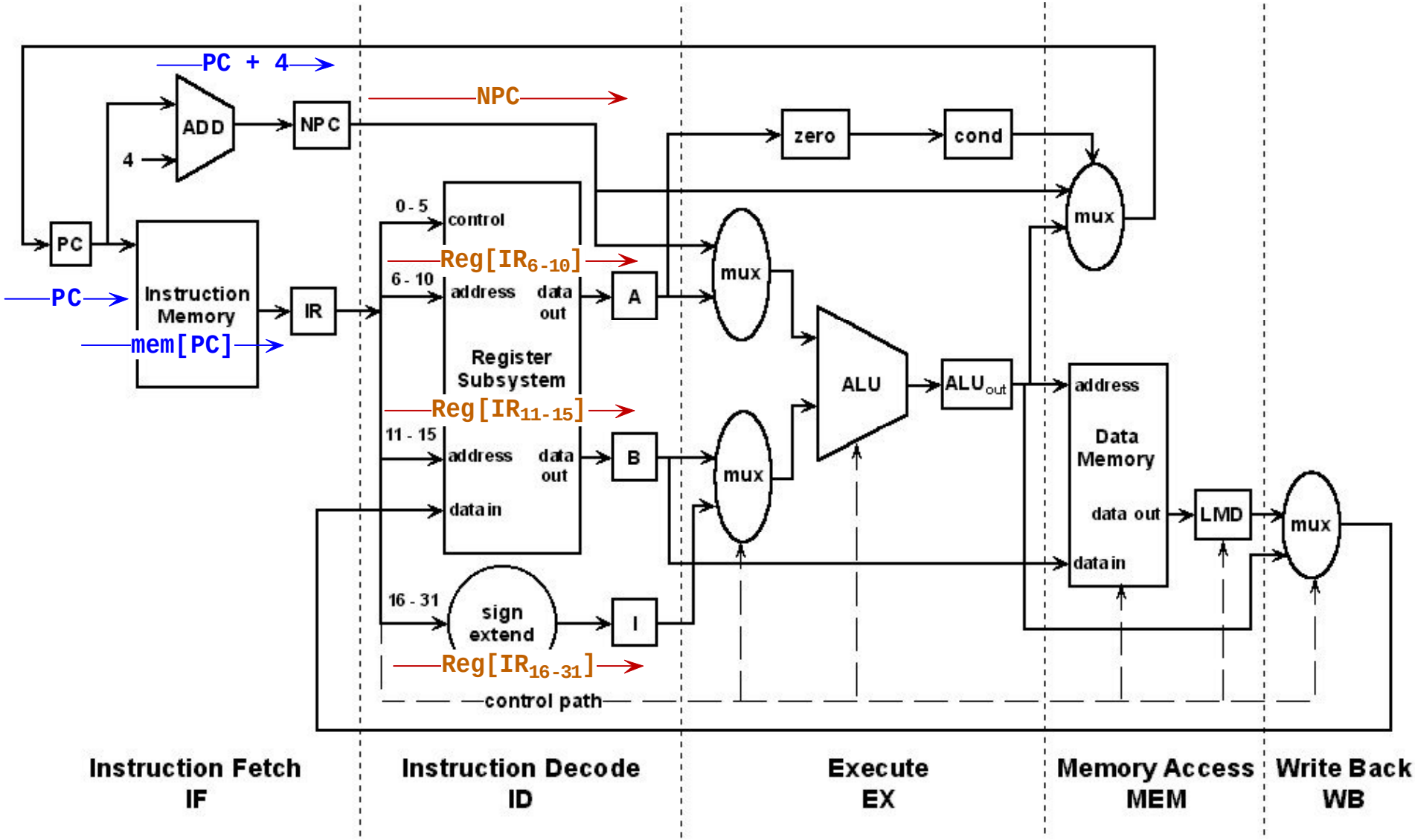
`sw 32(r1), r2` `mem[32+ regs[r1]] ← regs[r2]`

Type-I Load Instruction — 1



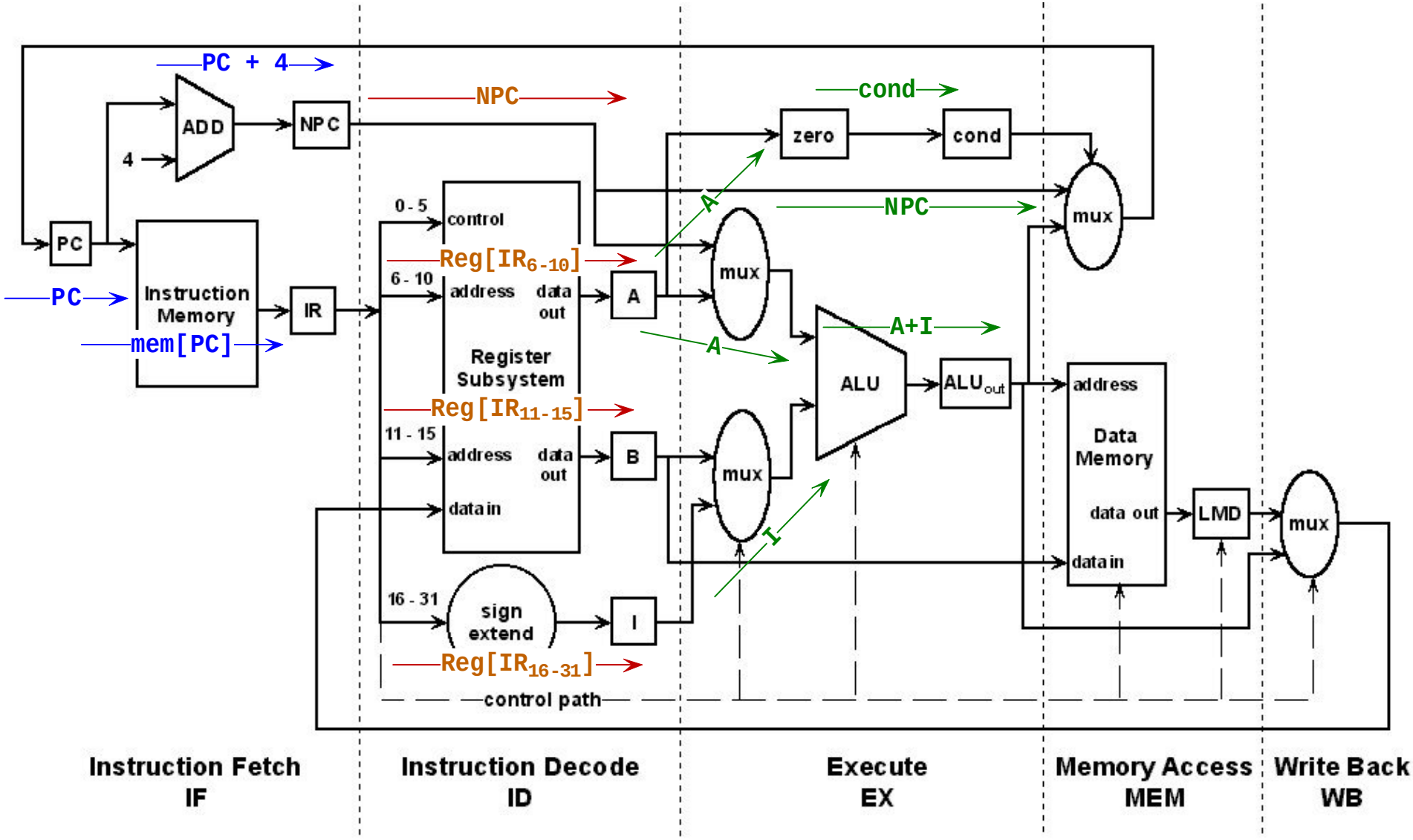
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 2



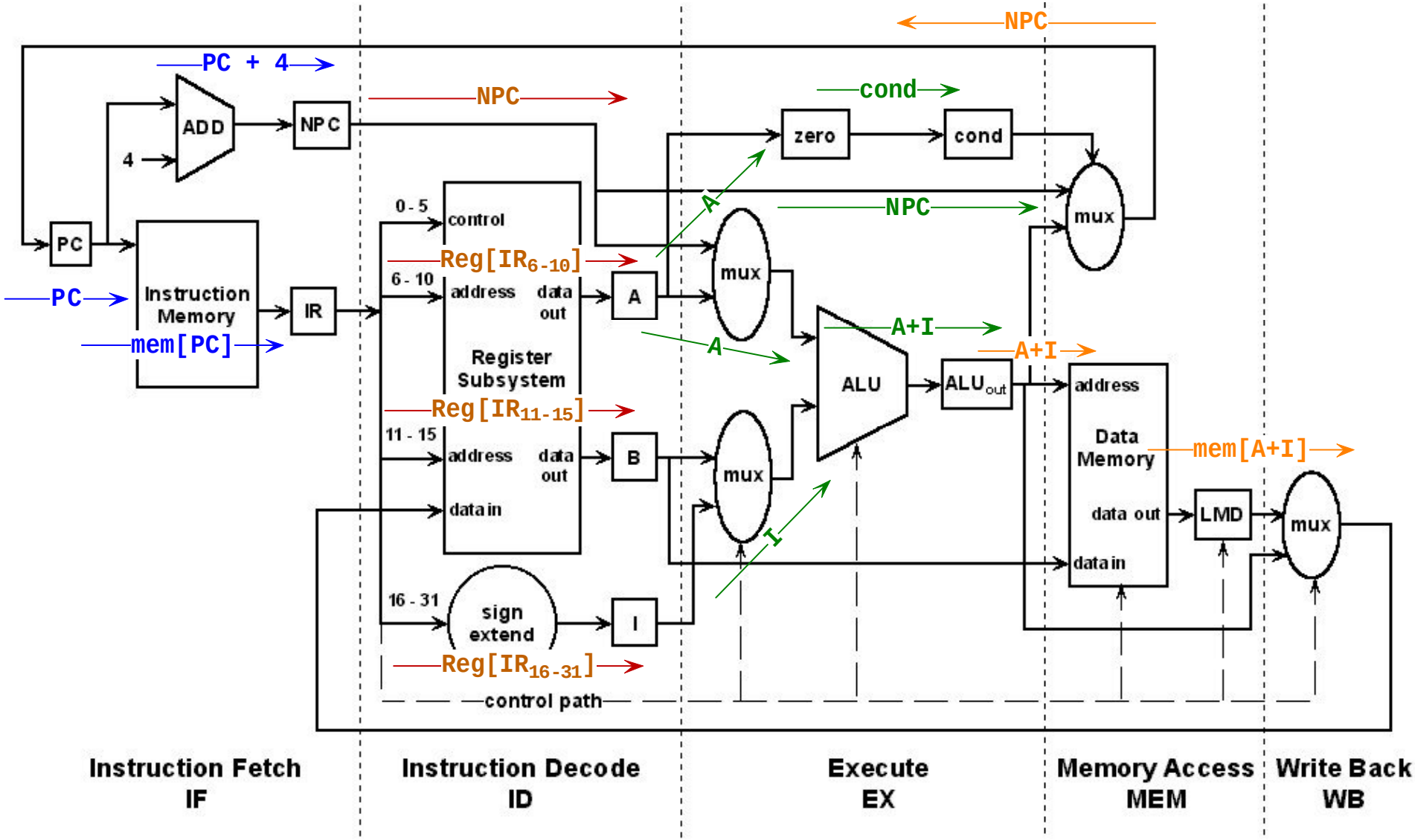
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 3



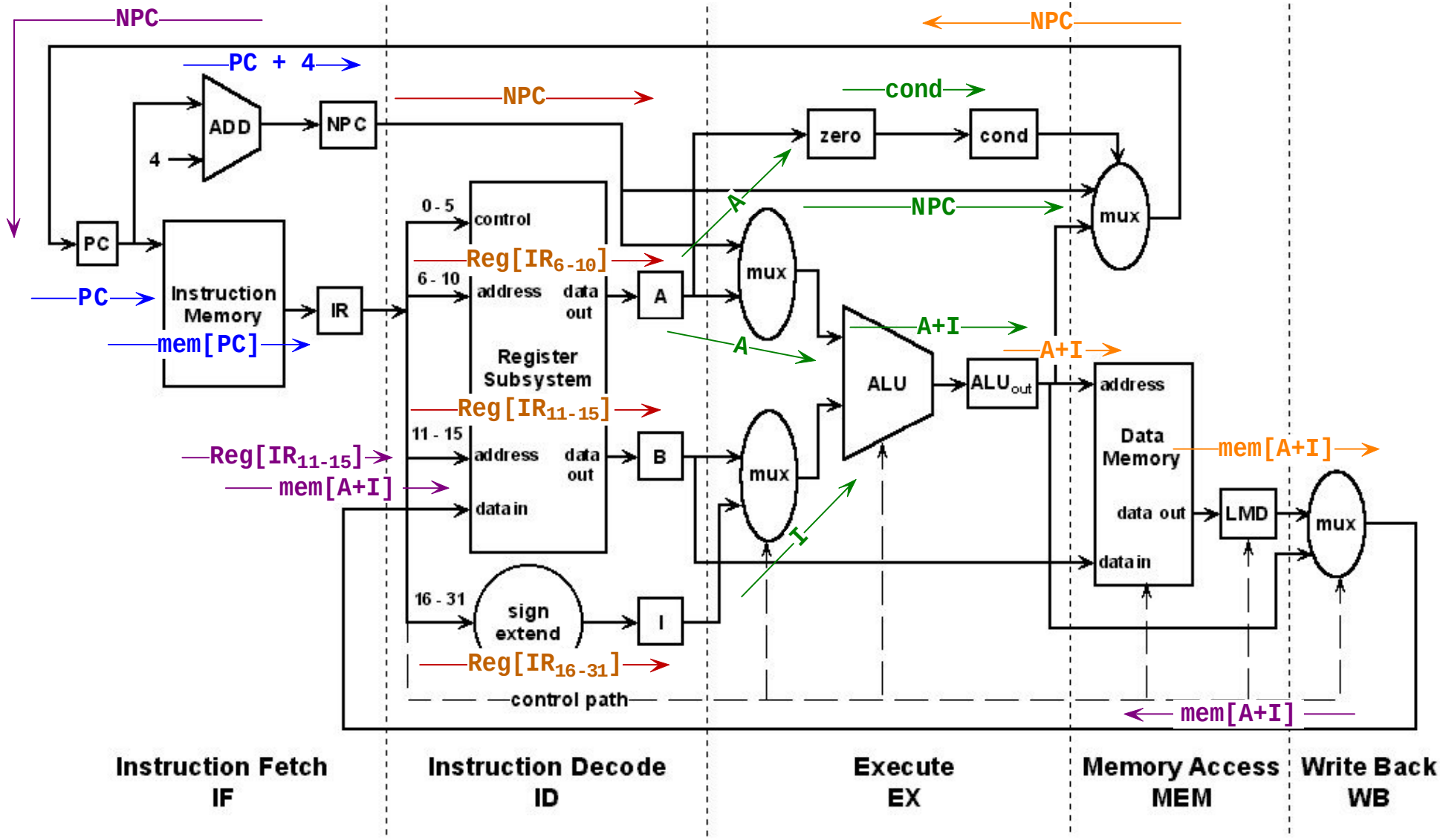
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 4



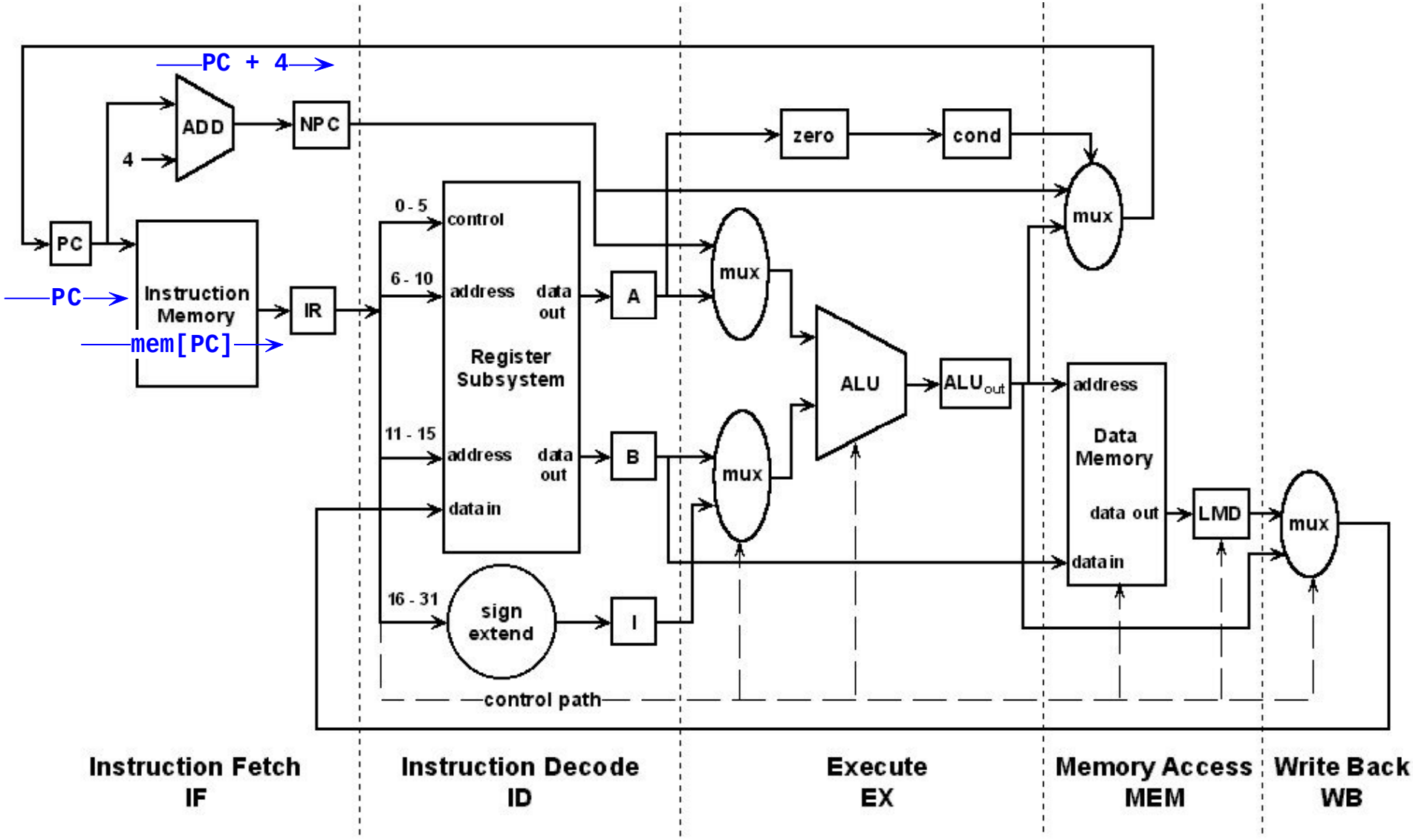
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Load Instruction — 5



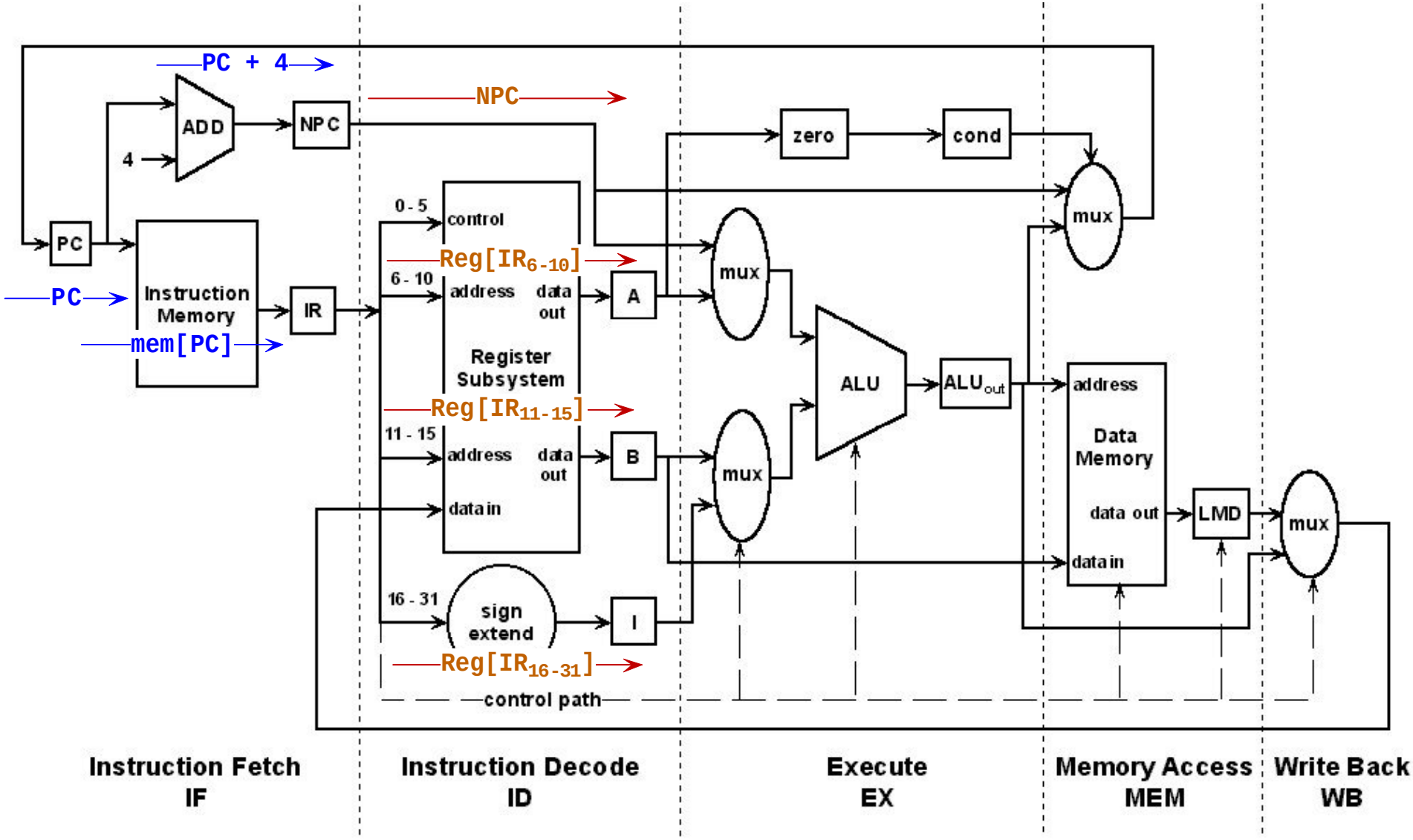
`lw r2, 32(r1)` `regs[r2] ← mem[32+ regs[r1]]`

Type-I Branch Instruction — 1



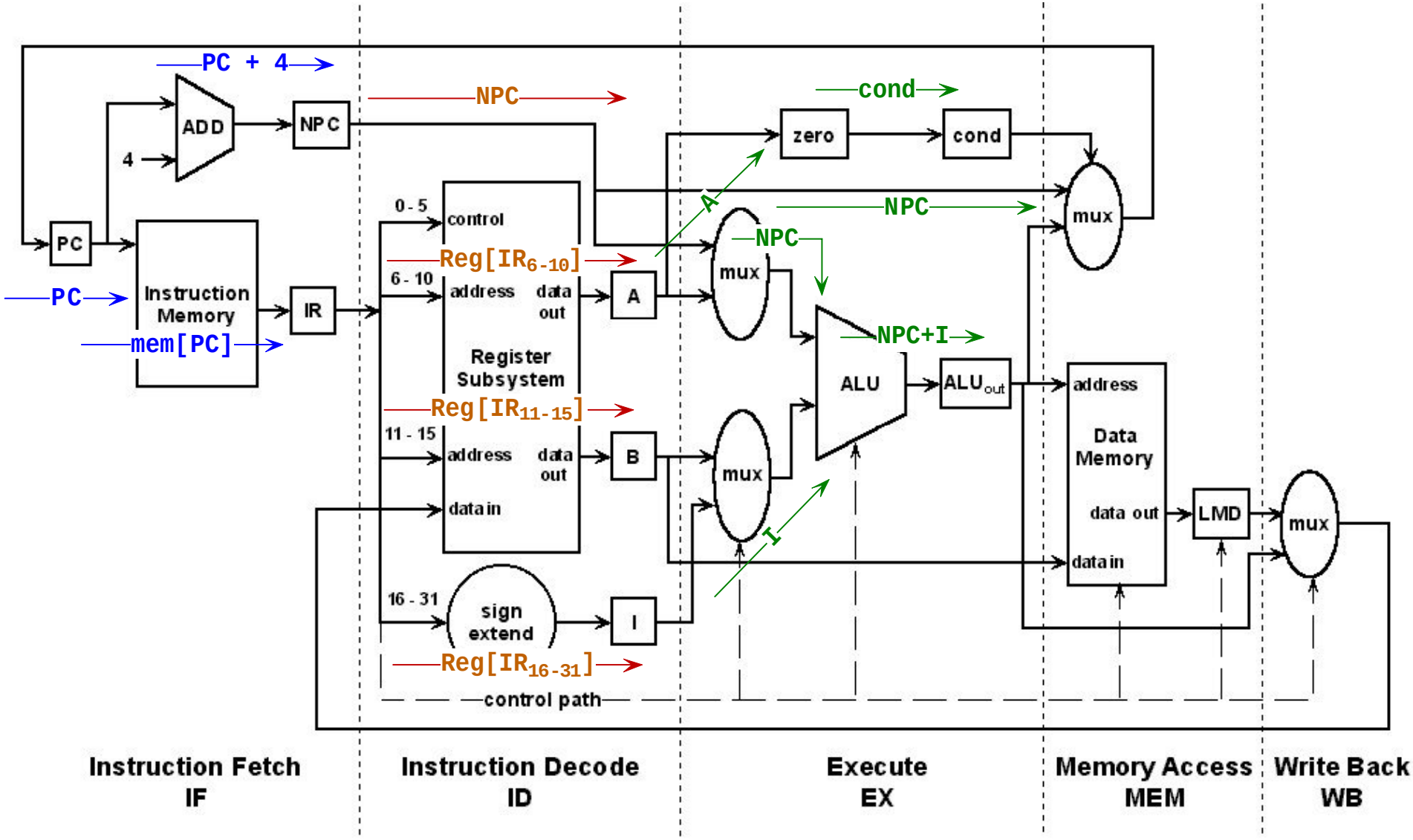

```
beqz r1, 1024      if (regs[r1] == 0) PC ← NPC + I else PC ← NPC
```

Type-I Branch Instruction — 2



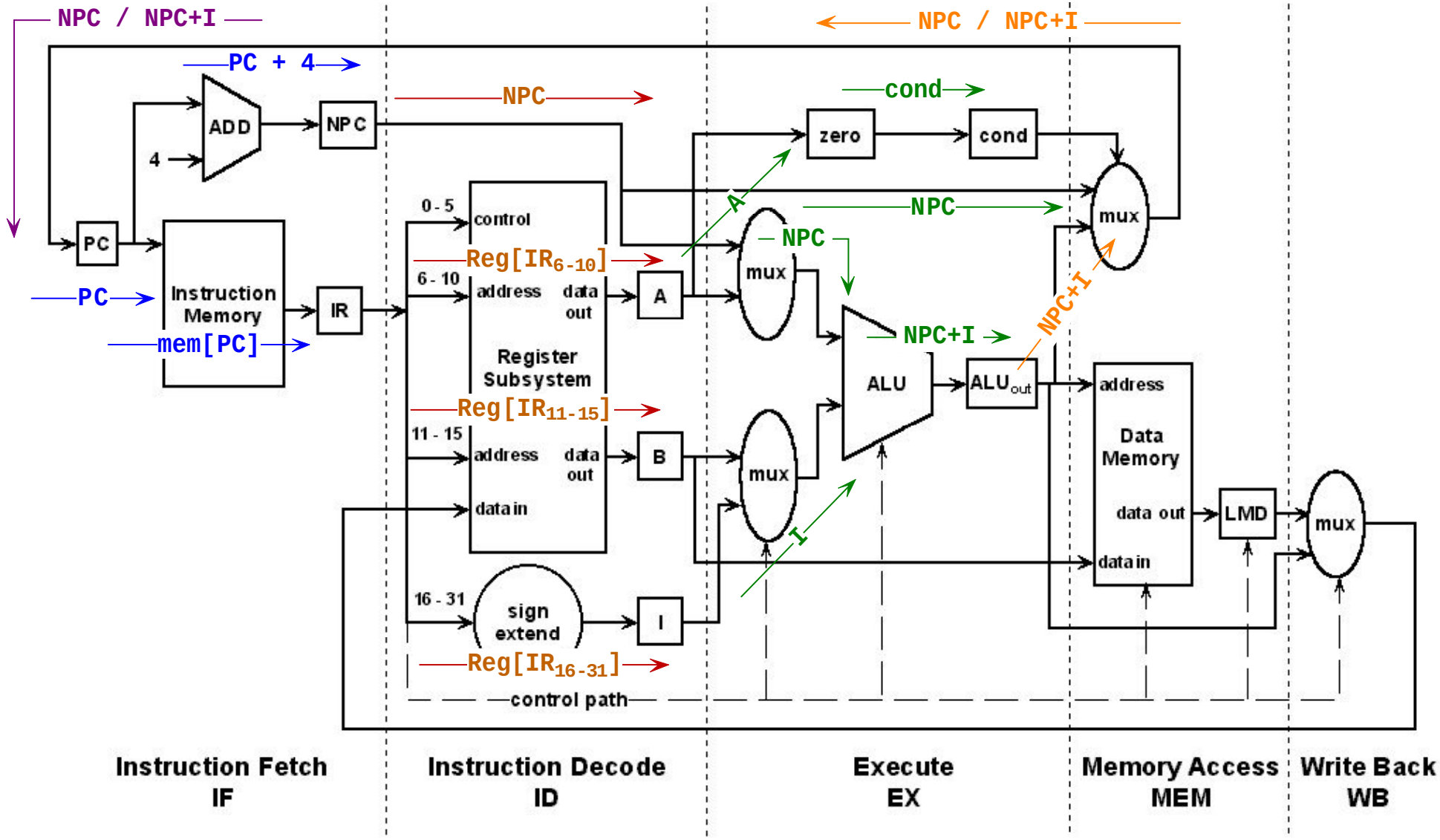

```
beqz r1, 1024      if (regs[r1] == 0) PC ← NPC + I else PC ← NPC
```

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

$$CPI = \sum_i CPI_i \times \frac{IC_i}{IC}$$

$$= 4 \times 0.40 + 5 \times 0.25 + 4 \times 0.15 + 4 \times 0.25$$

$$= 4.25$$

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

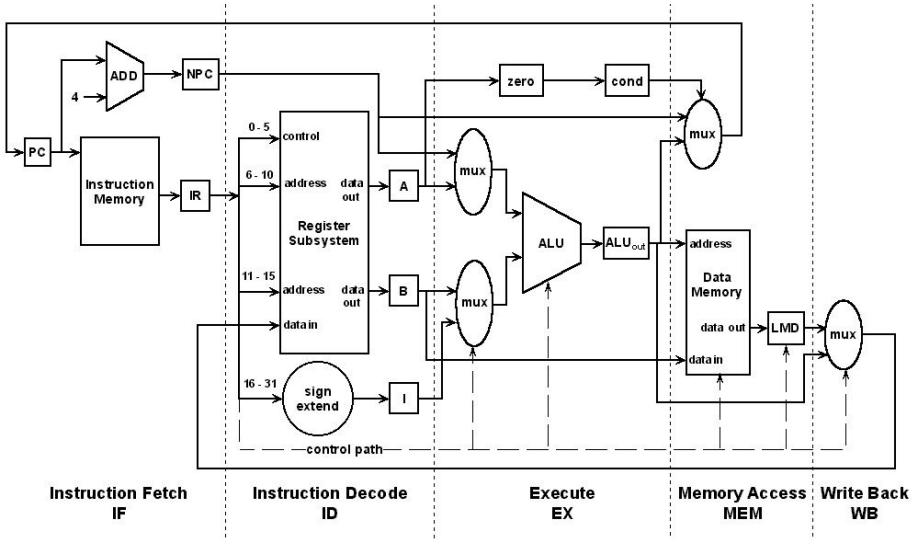
Instruction Decode (ID) holds state fixed

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

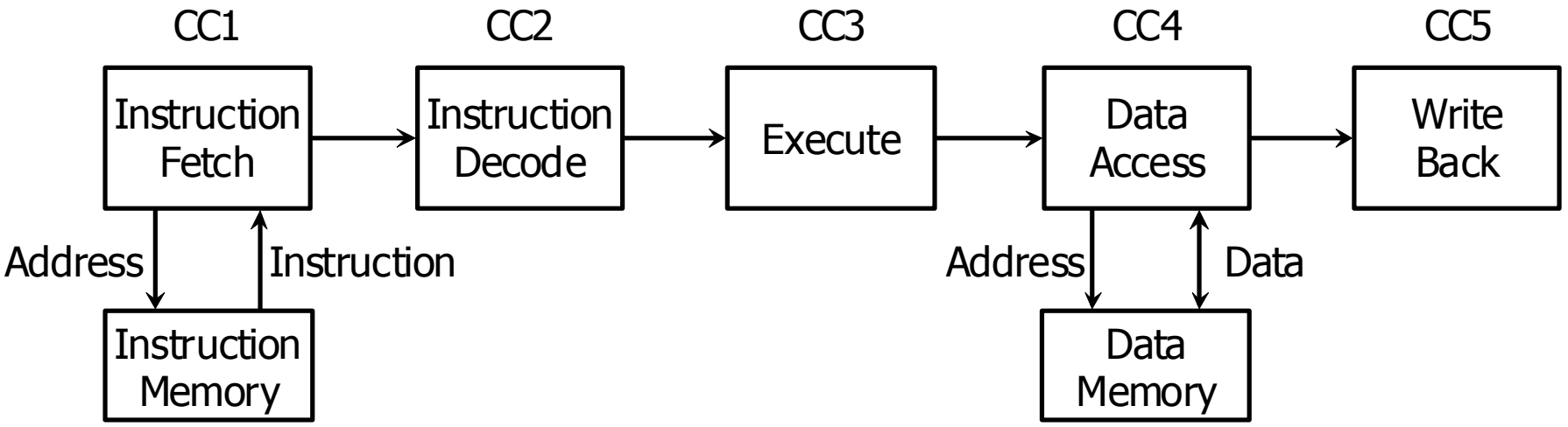
DLX based on assembly line

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

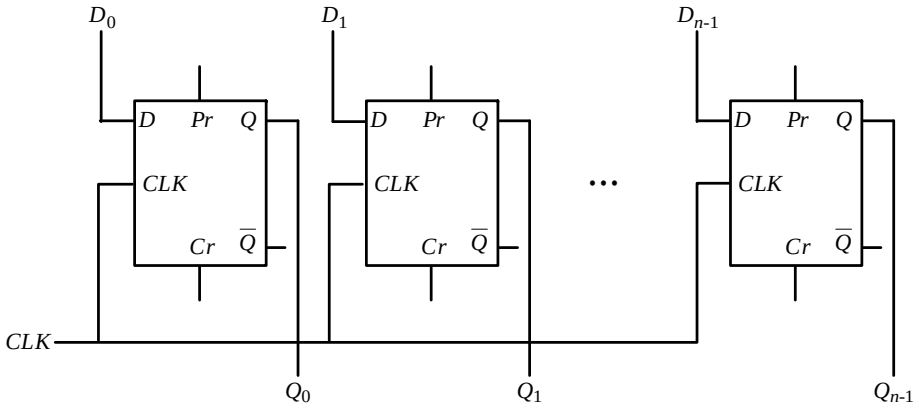
Example

- **`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



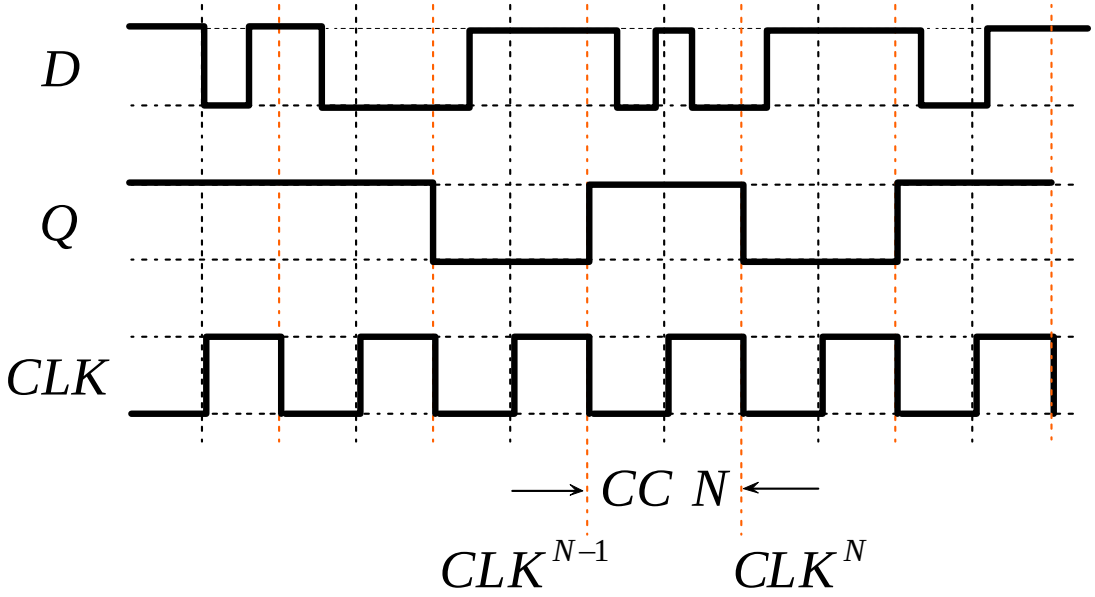
D-flip/flop (edge-triggered latch)

Input D

- Output of some digital system

Output Q

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



Clock Cycle N

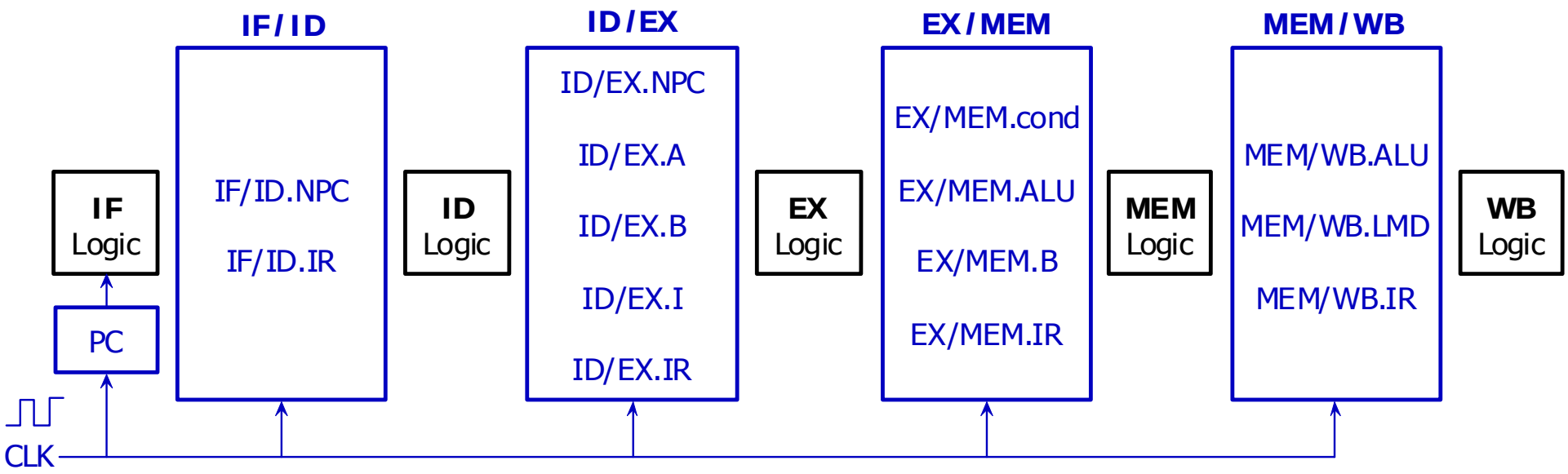
$CC\ N$ begins on CLK^{N-1}

- 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

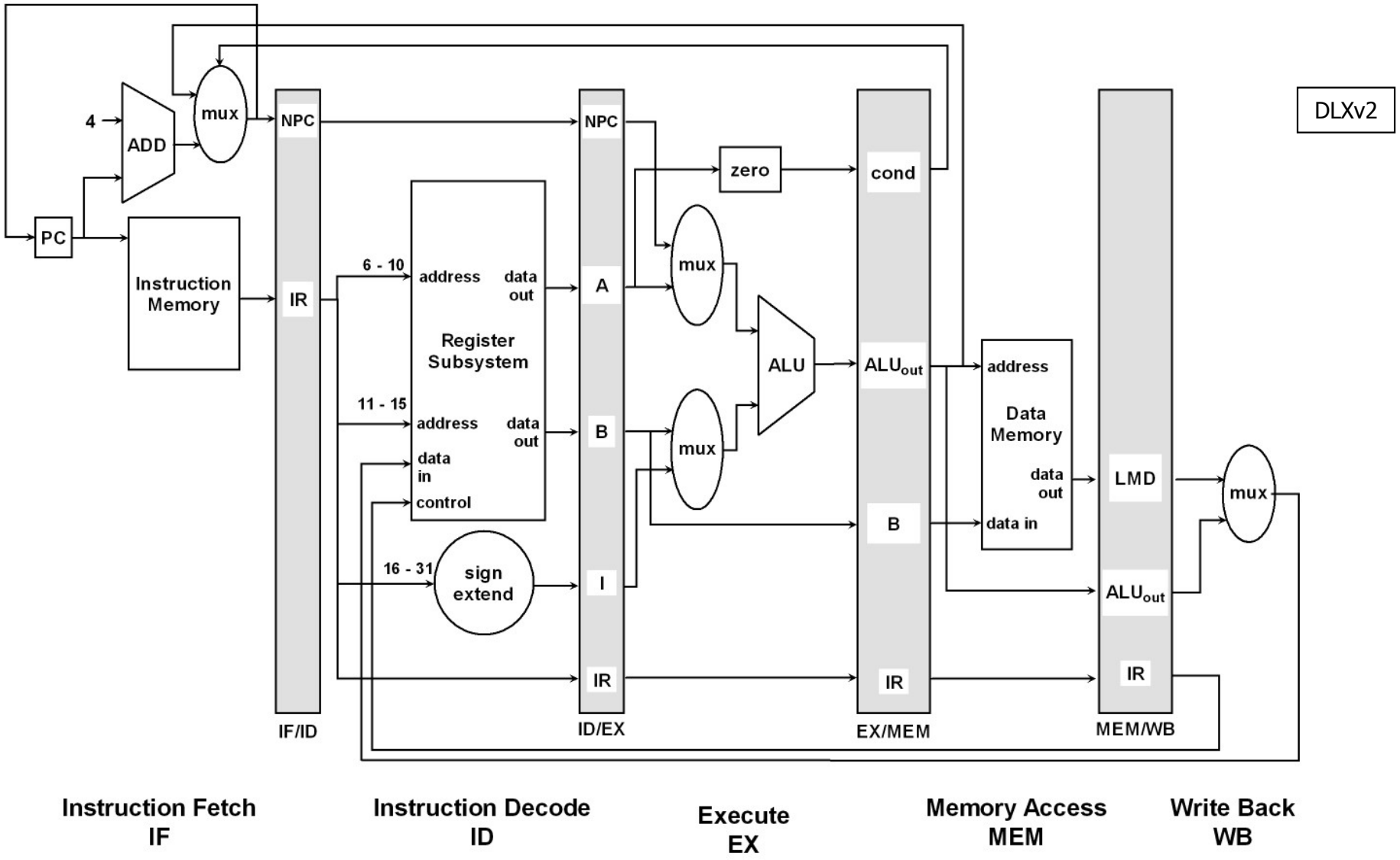
Synchronous sequential logic

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)

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

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

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

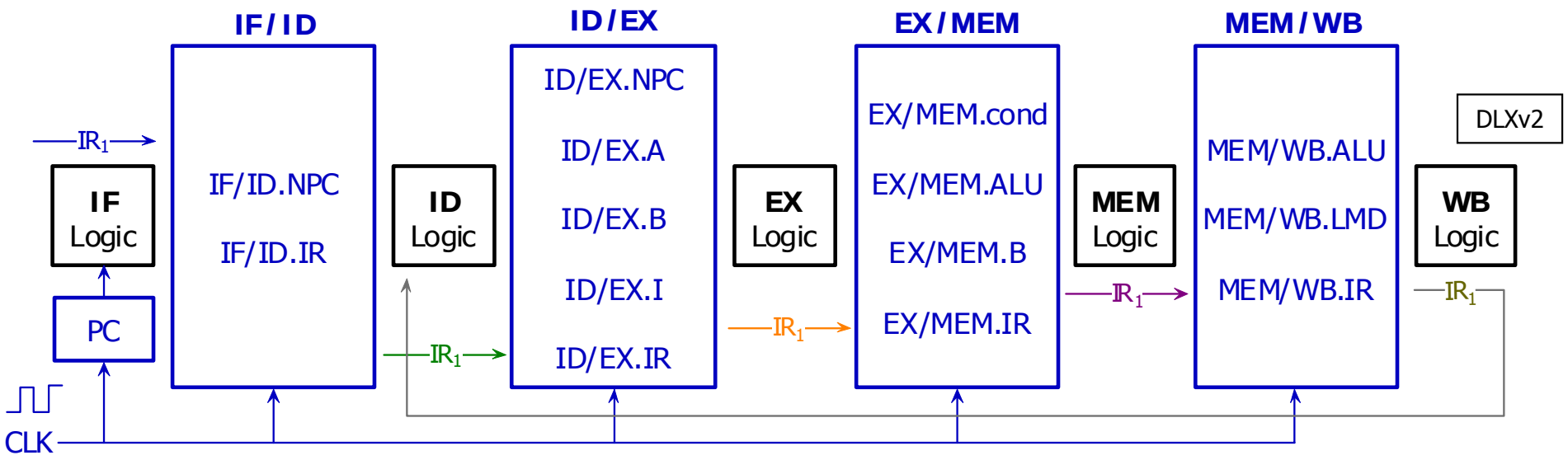
$$MEM/WB.IR \leftarrow EX/MEM.IR$$

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

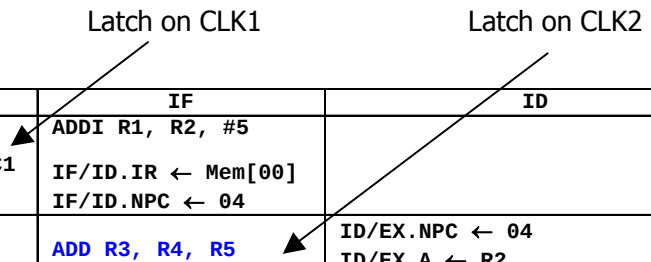


CLK 0	CC 1 begins	Memory ← PC(I_1)	IF/ID.IR "sees" Mem[PC(I_1)]
CLK 1	CC 2 begins	IF/ID.IR ← Mem[PC(I_1)] Memory ← PC(I_2)	ID/EX.IR "sees" Mem[PC(I_1)] IF/ID.IR "sees" Mem[PC(I_2)]
CLK 2	CC 3 begins	ID/EX.IR ← Mem[PC(I_1)] IF/ID.IR ← Mem[PC(I_2)] Memory ← PC(I_3)	EX/MEM.IR "sees" Mem[PC(I_1)] ID/EX.IR "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

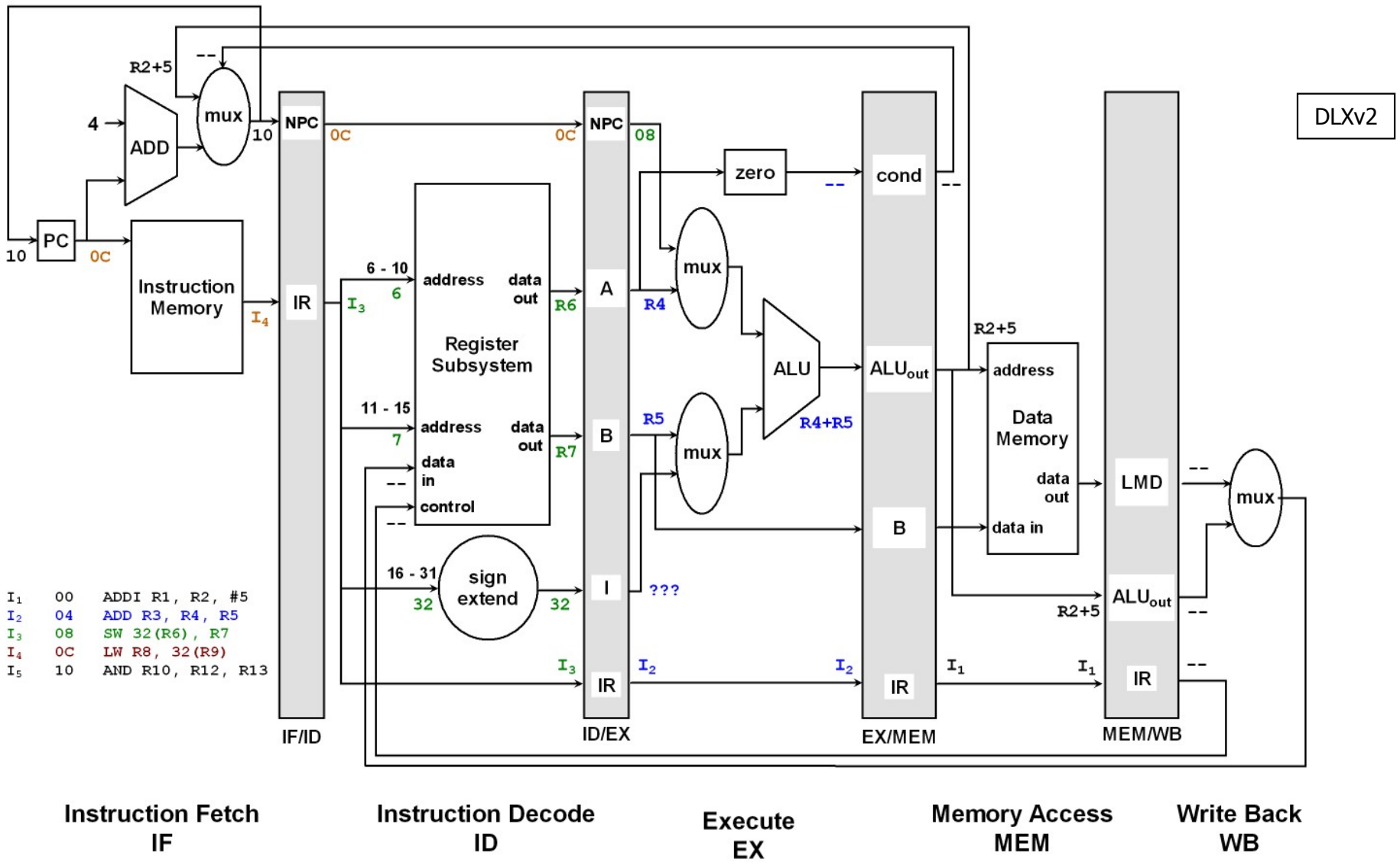
After CLK 1

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

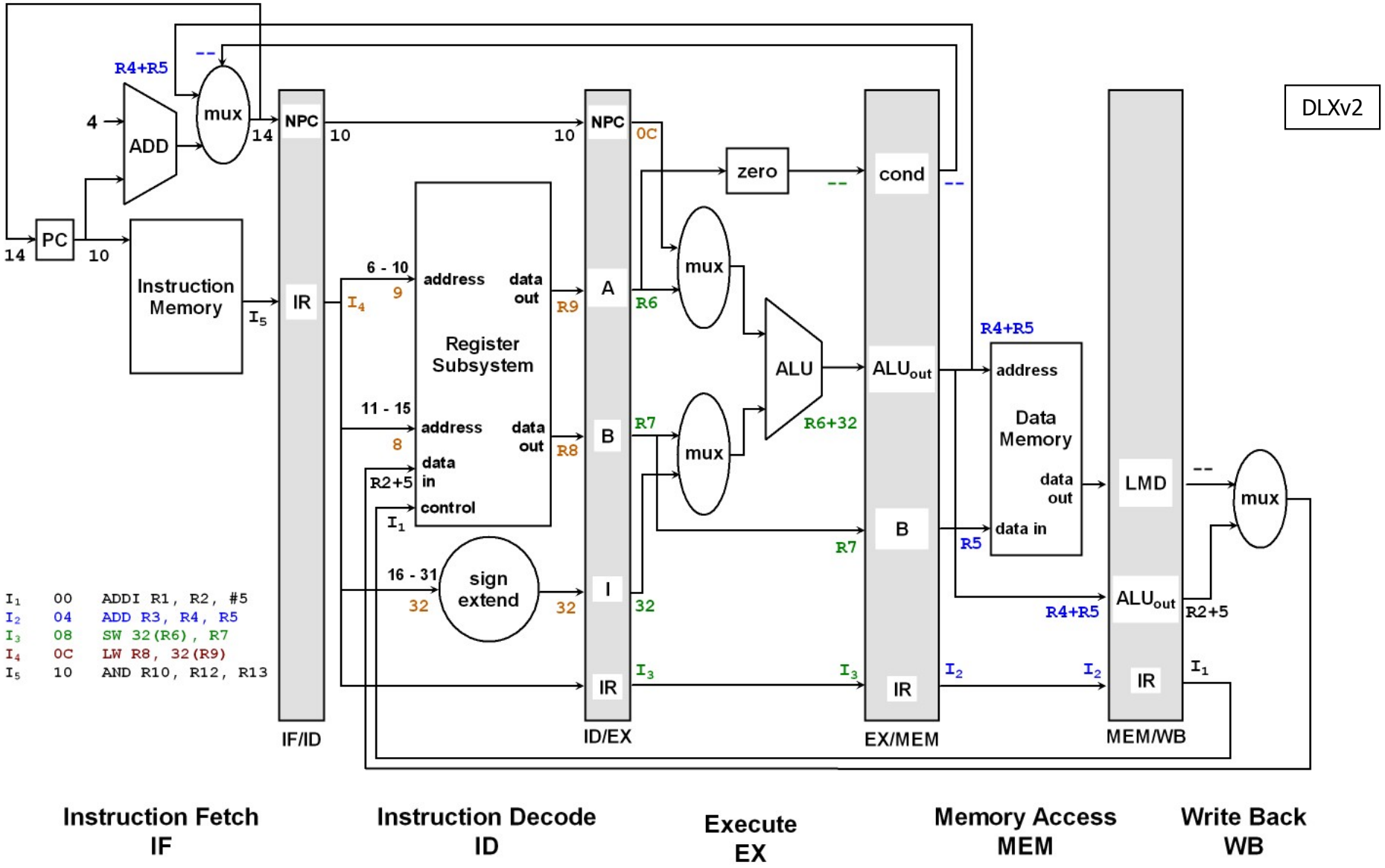
Registers "see" IF/ID.IR and ID/EX.A,B,I "see" R2, R1, 5 as inputs

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

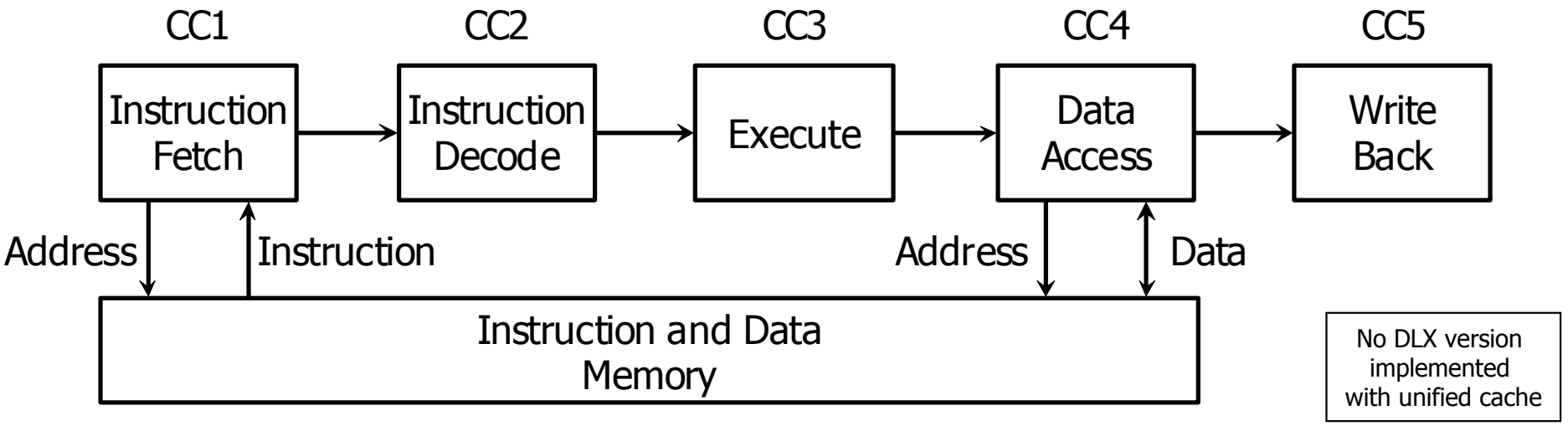
Typical structural hazard — unified cache hazard

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

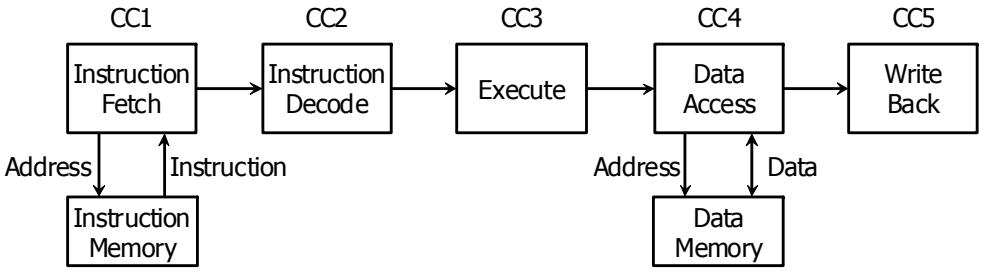
Data Hazards in DLXv2

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**

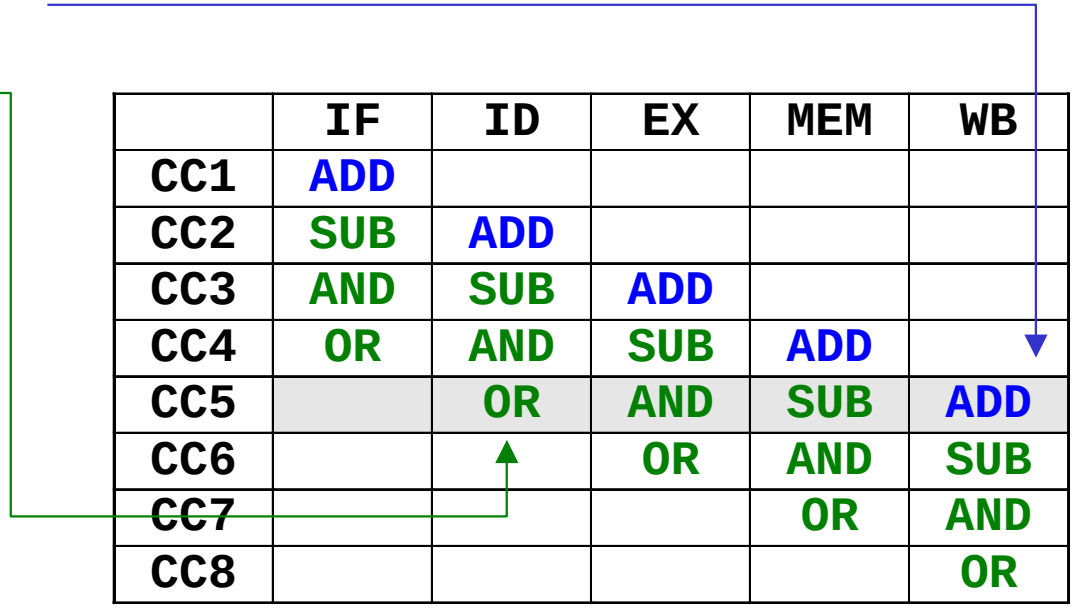
Bad timing (uncorrected execution)

I_1 updates R1 in WB during CC5

I_2 reads R1 in ID during CC3

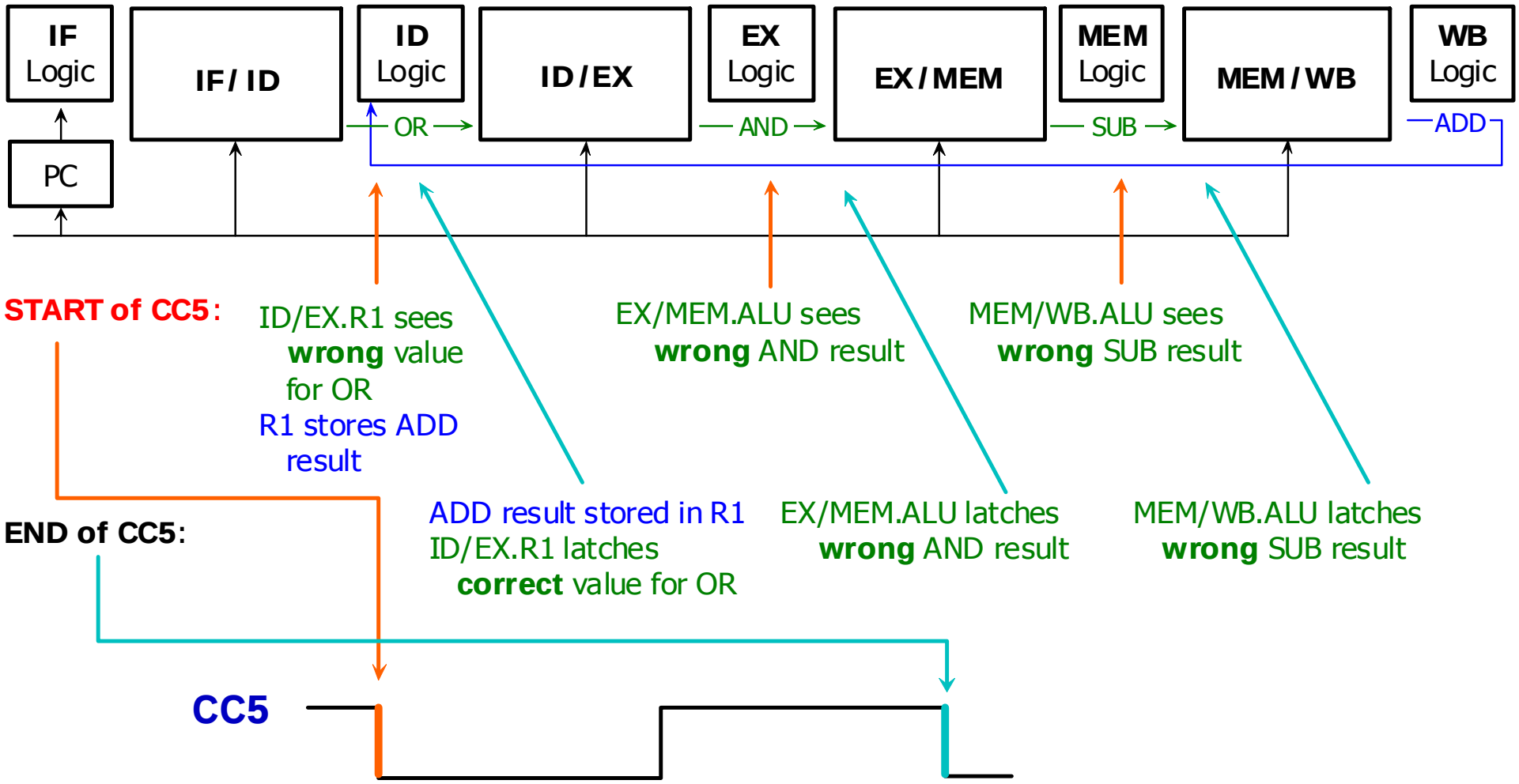
I_3 reads R1 in ID during CC4

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**

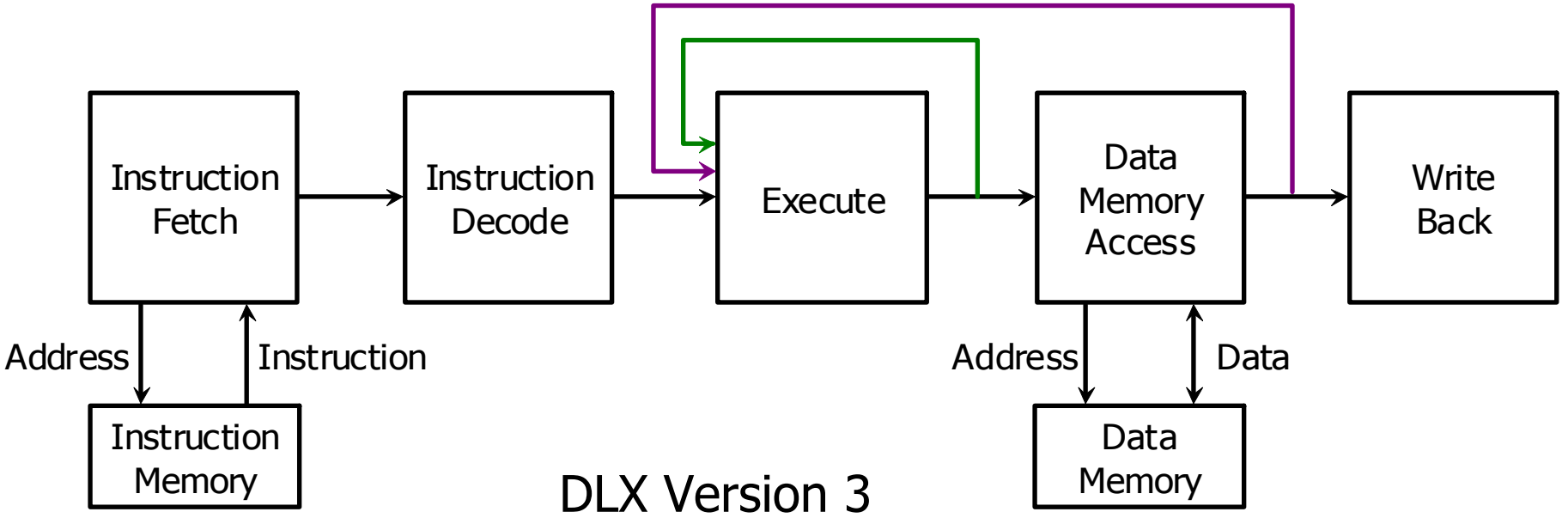
	IF	ID	EX	MEM	WB
CC1	ADD				
CC2	SUB	ADD			
CC3	AND	SUB	ADD		
CC4	OR	AND	SUB	ADD	
CC5		OR	AND	SUB	ADD
CC6			OR	AND	SUB
CC7				OR	AND
CC8					OR

Trick to prevent stall

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

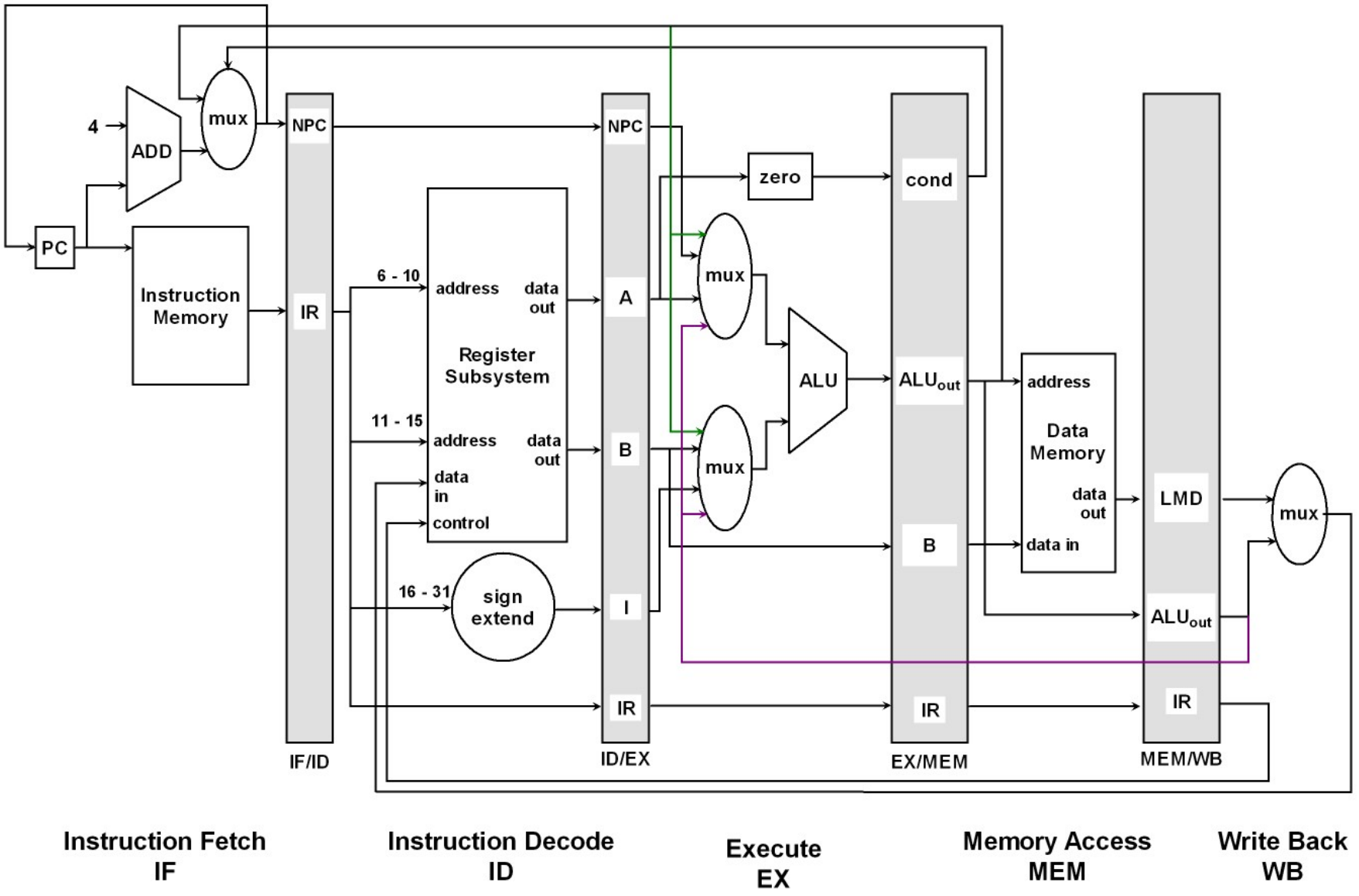
Allow **SUB** and **AND** to read incorrect value in **ID**

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

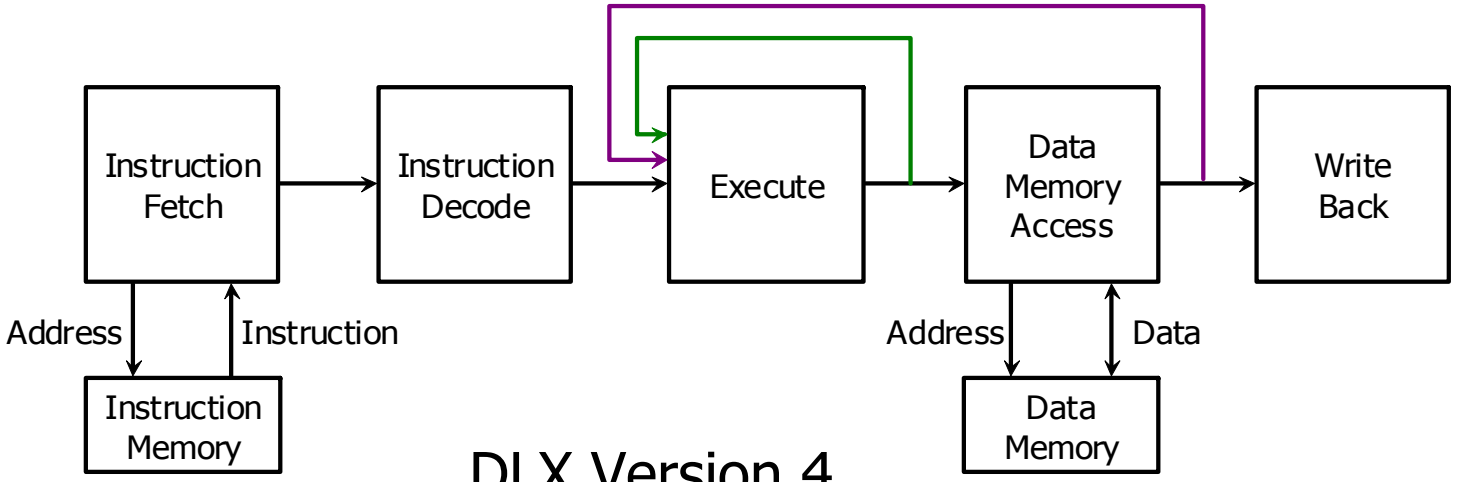
Trick to minimize stall

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

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

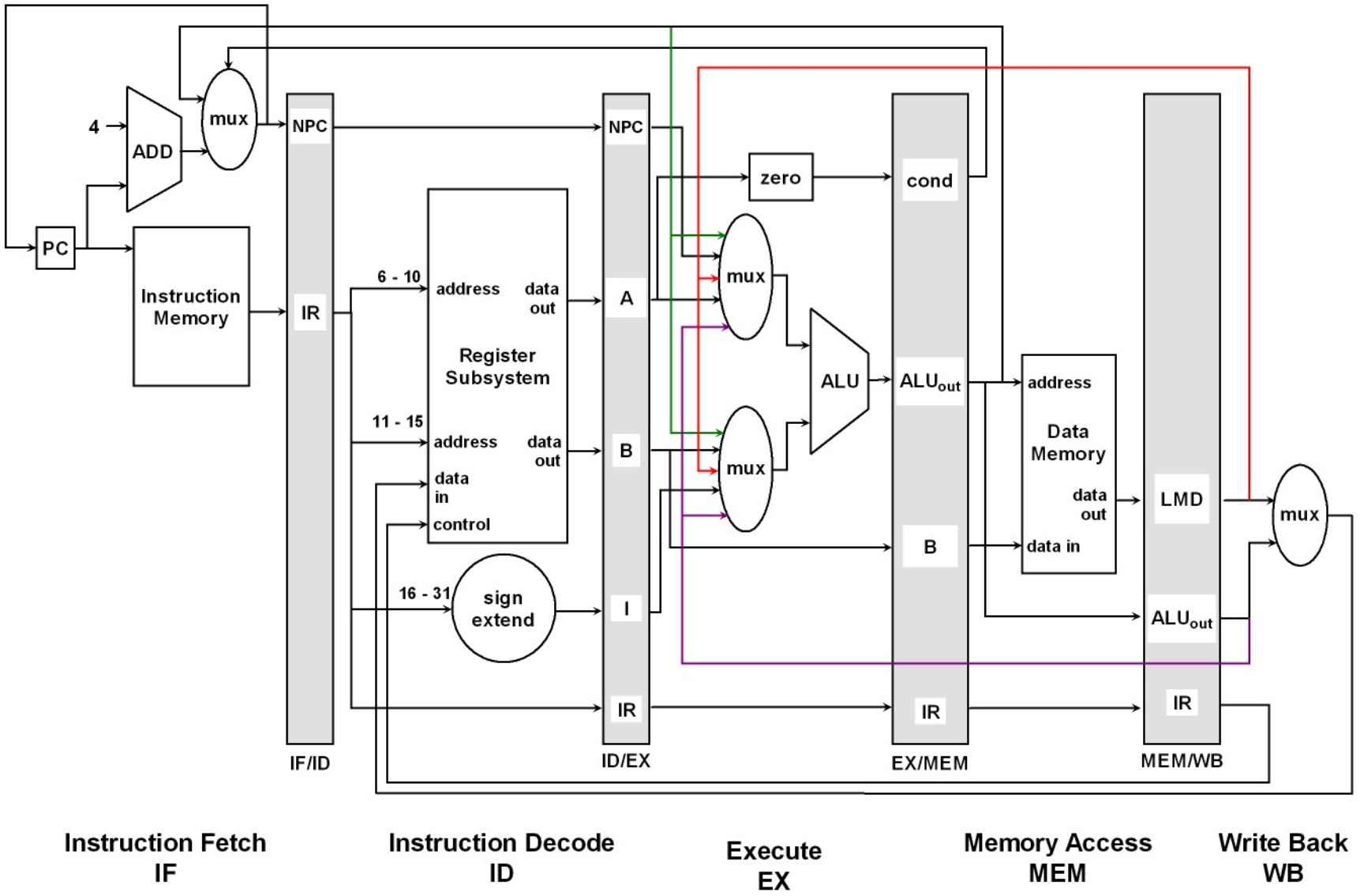
Stall **SUB** for 1 clock cycle in ID (load performed later than ALU operation)

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

I_1 `SUB R1,R5,R4` — I_1 has R1 as **destination**

I_2 `SW 32(R2),R1` — 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

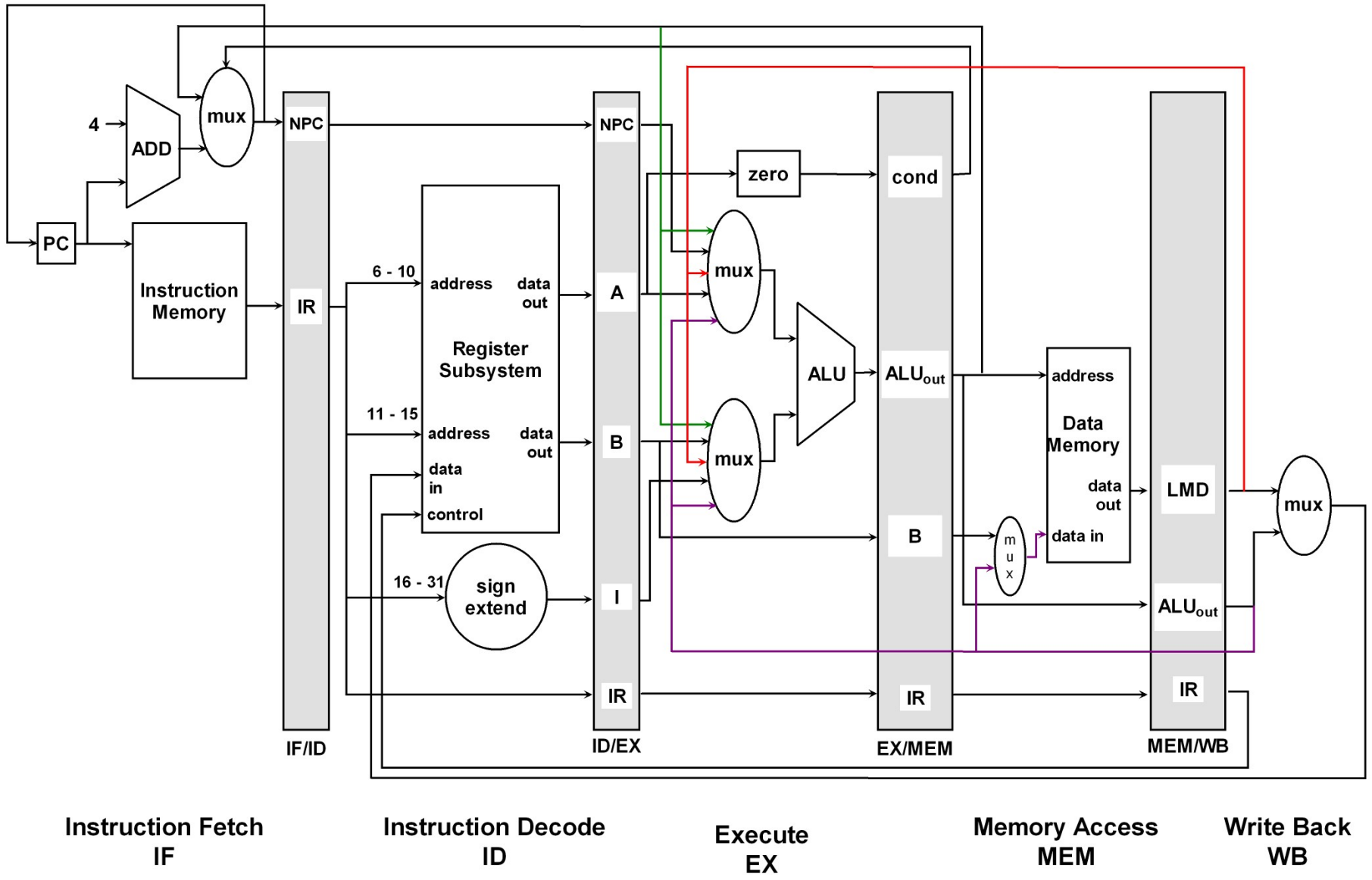
	IF	ID	EX	MEM	WB
CC1	SUB				
CC2	SW	SUB			
CC3		SW	SUB		
CC4			SW	SUB	
CC5				SW	SUB
CC6					SW

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

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

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

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

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

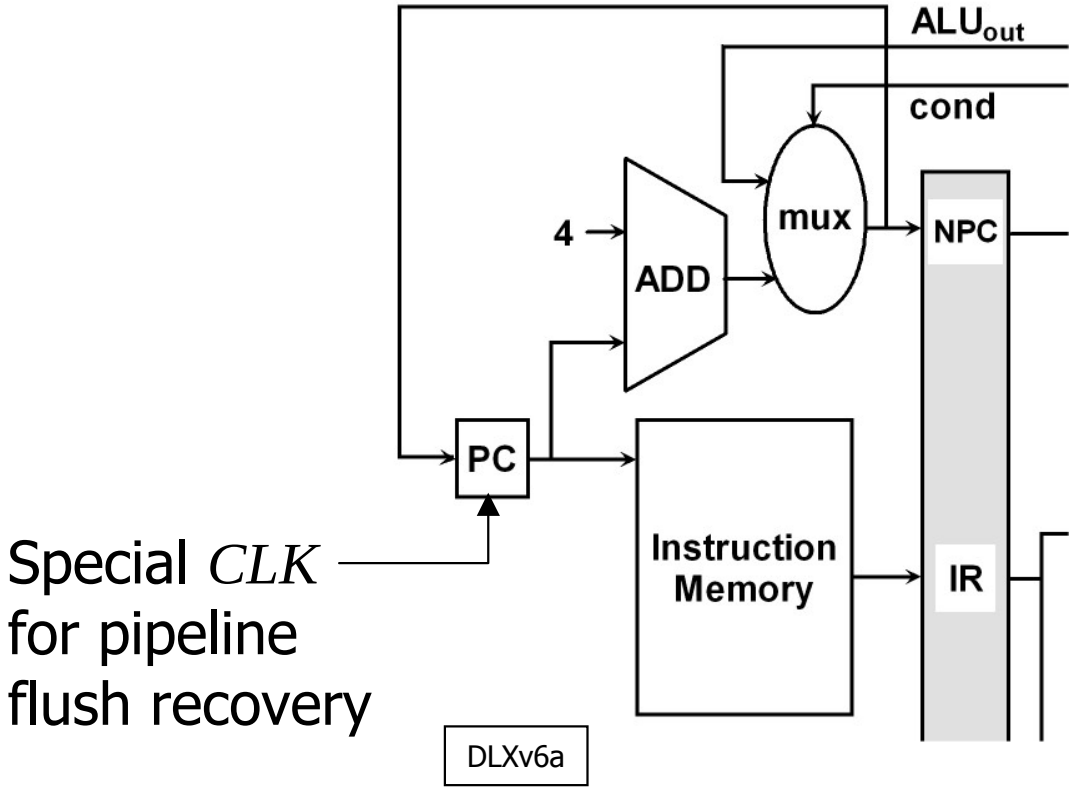
Improving Branch Performance — 1

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

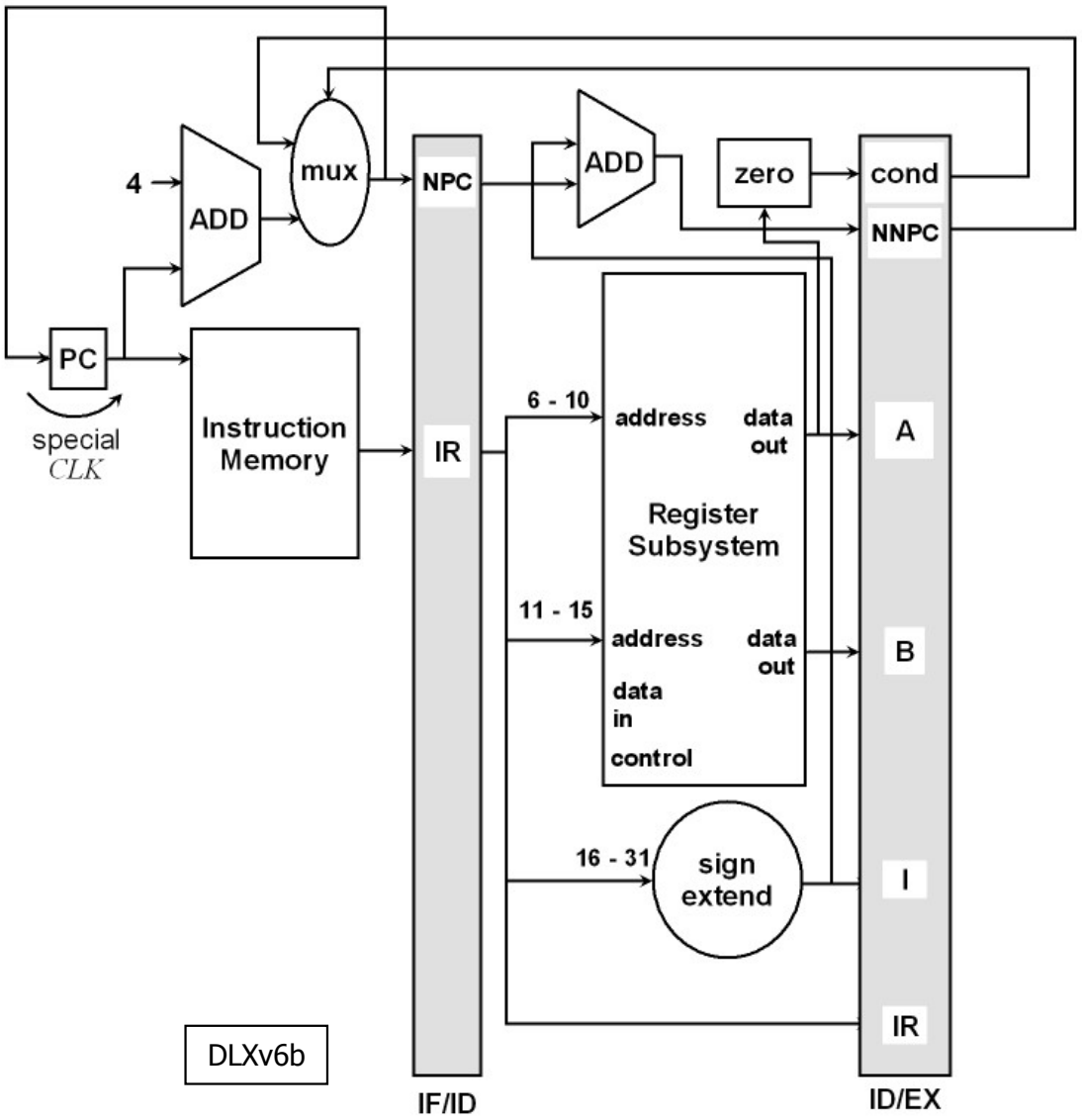
$$CPI^{stall} = 2 \times 0.20 \frac{\text{cycles}}{\text{instruction}}$$

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

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

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

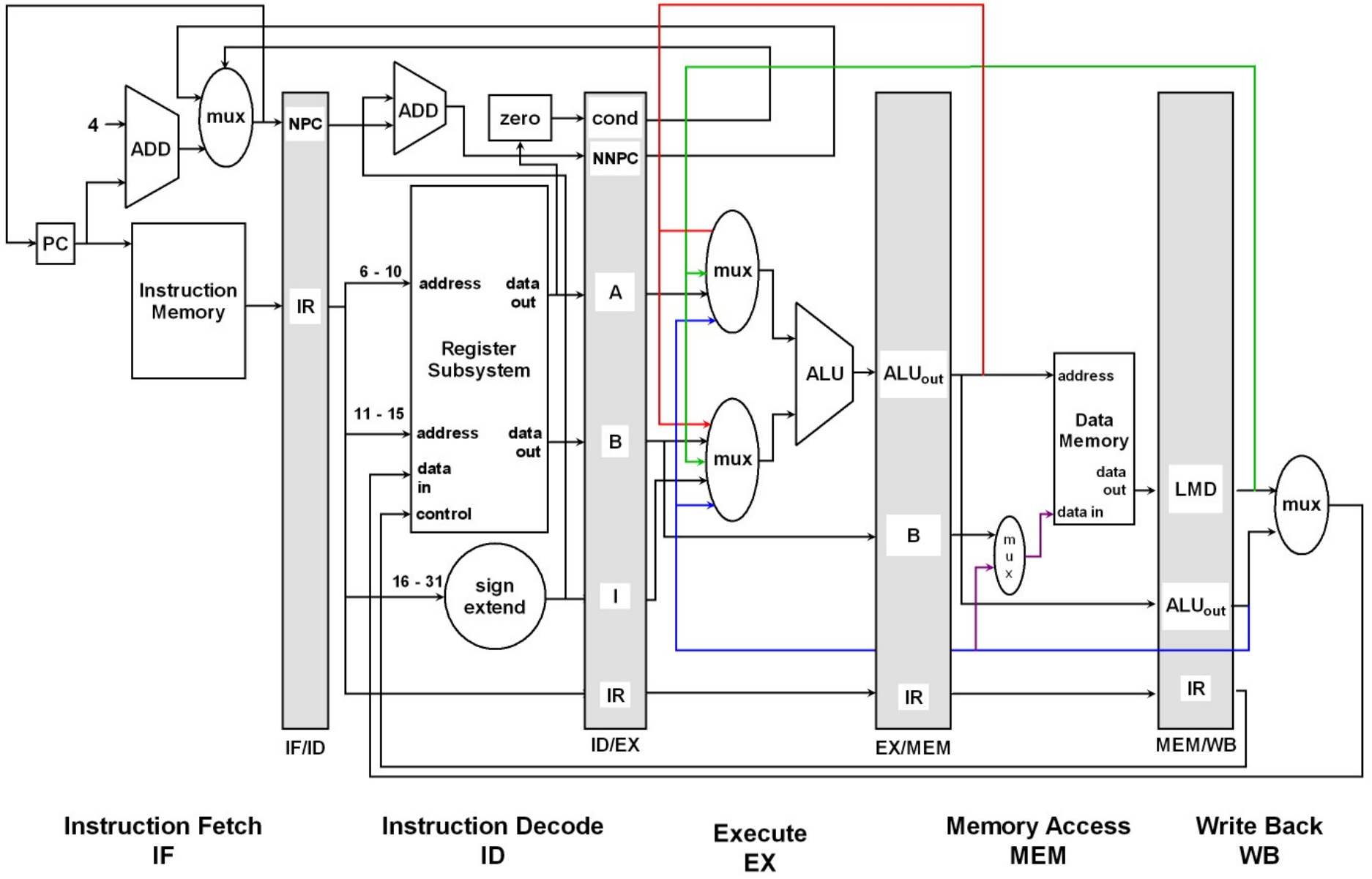
		1	2	3	4	5	6	7	8	9
I_1	BEQZ R1,I_T	IF	ID	EX	MEM	WB				
I_2	Fall-Through		IF	ID	EX	MEM	WB			
I_3				IF						
. . .	…	…	…	…	…	…	…	…	…	…
I_T	Target			IF	ID	EX	MEM	WB		

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

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

DLXv6c

DLX Version 6c



Version 6c Branch Processing — 1

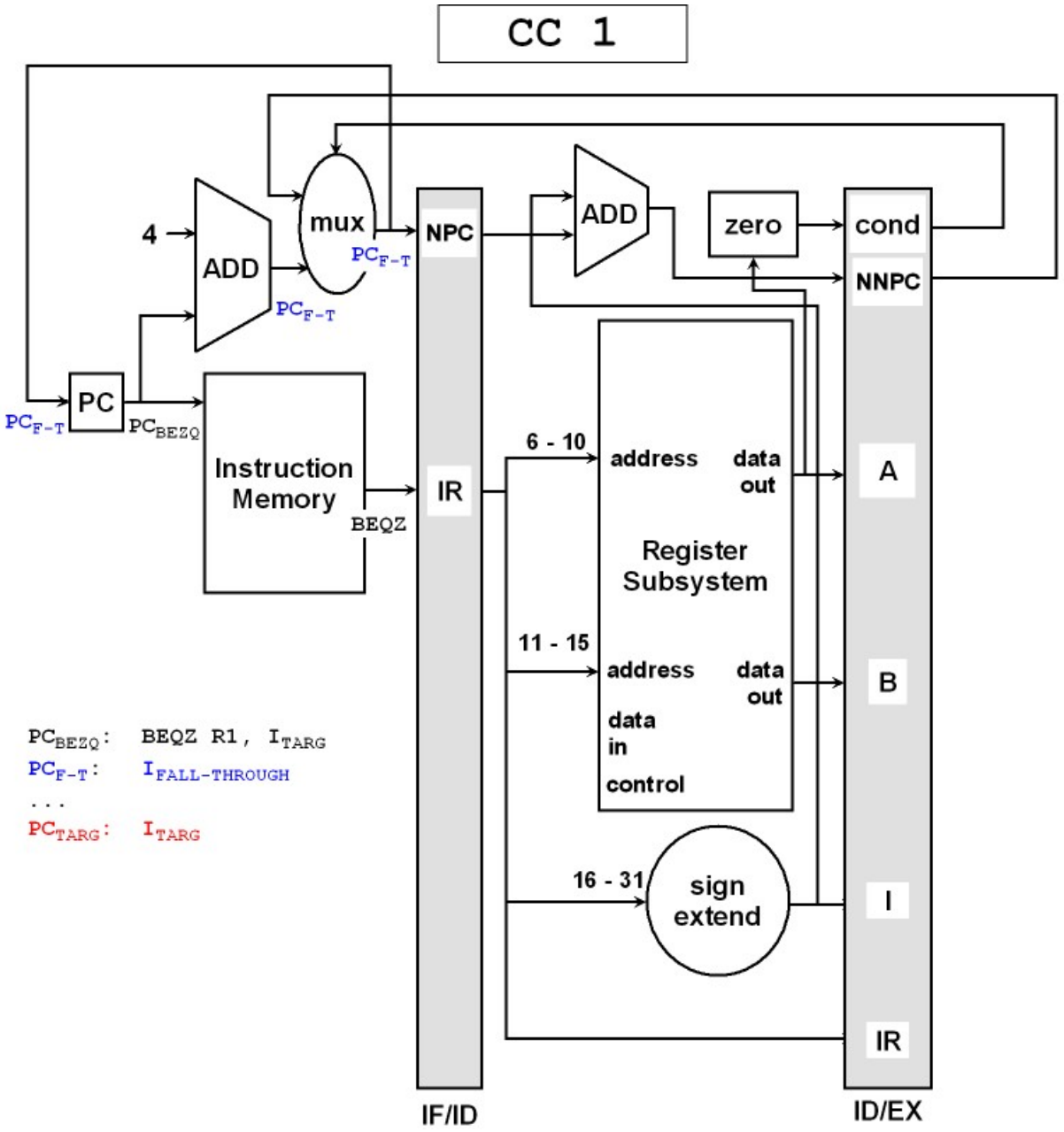
CC1

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

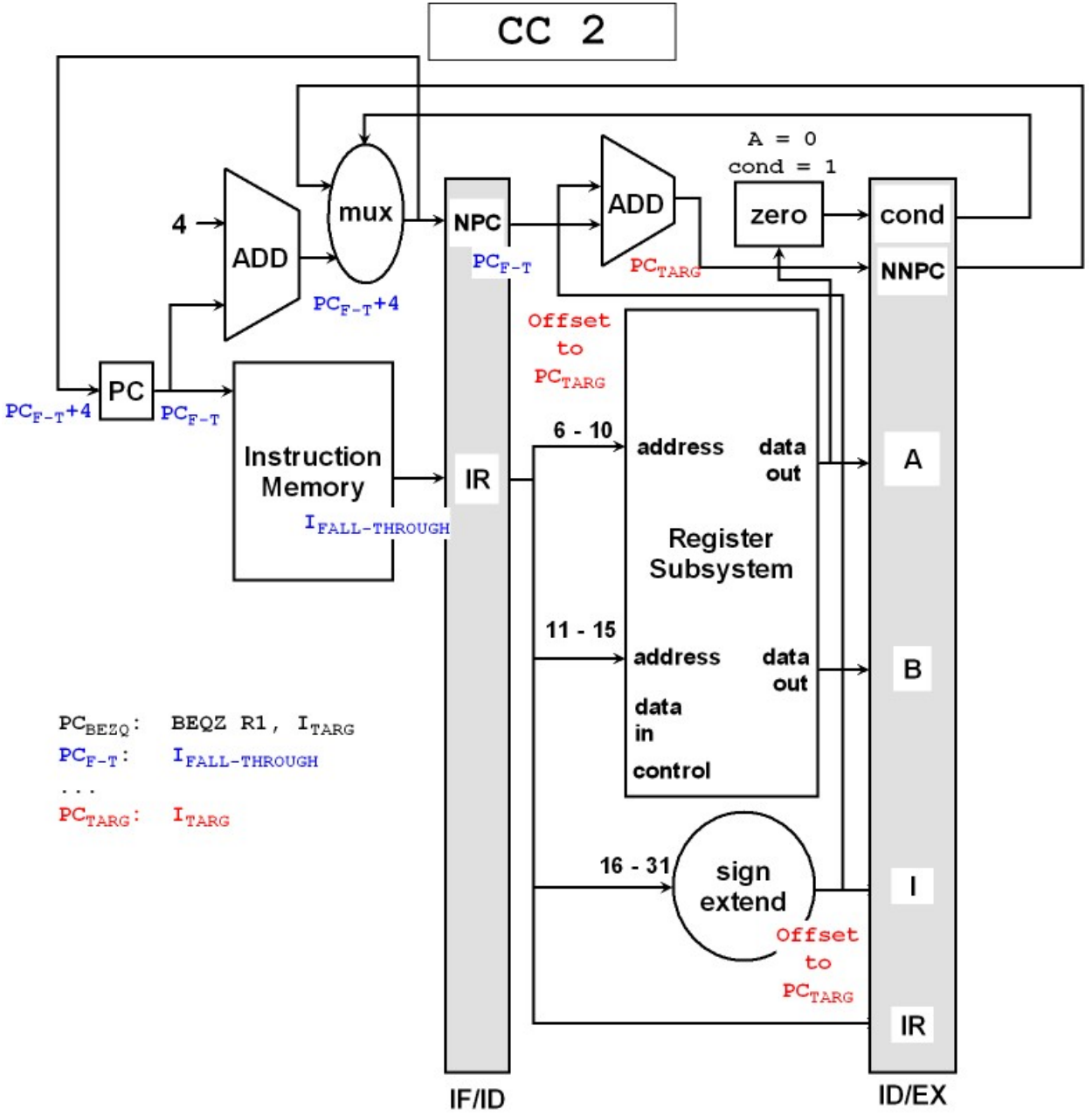
I_{TARG} = NPC+I

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**

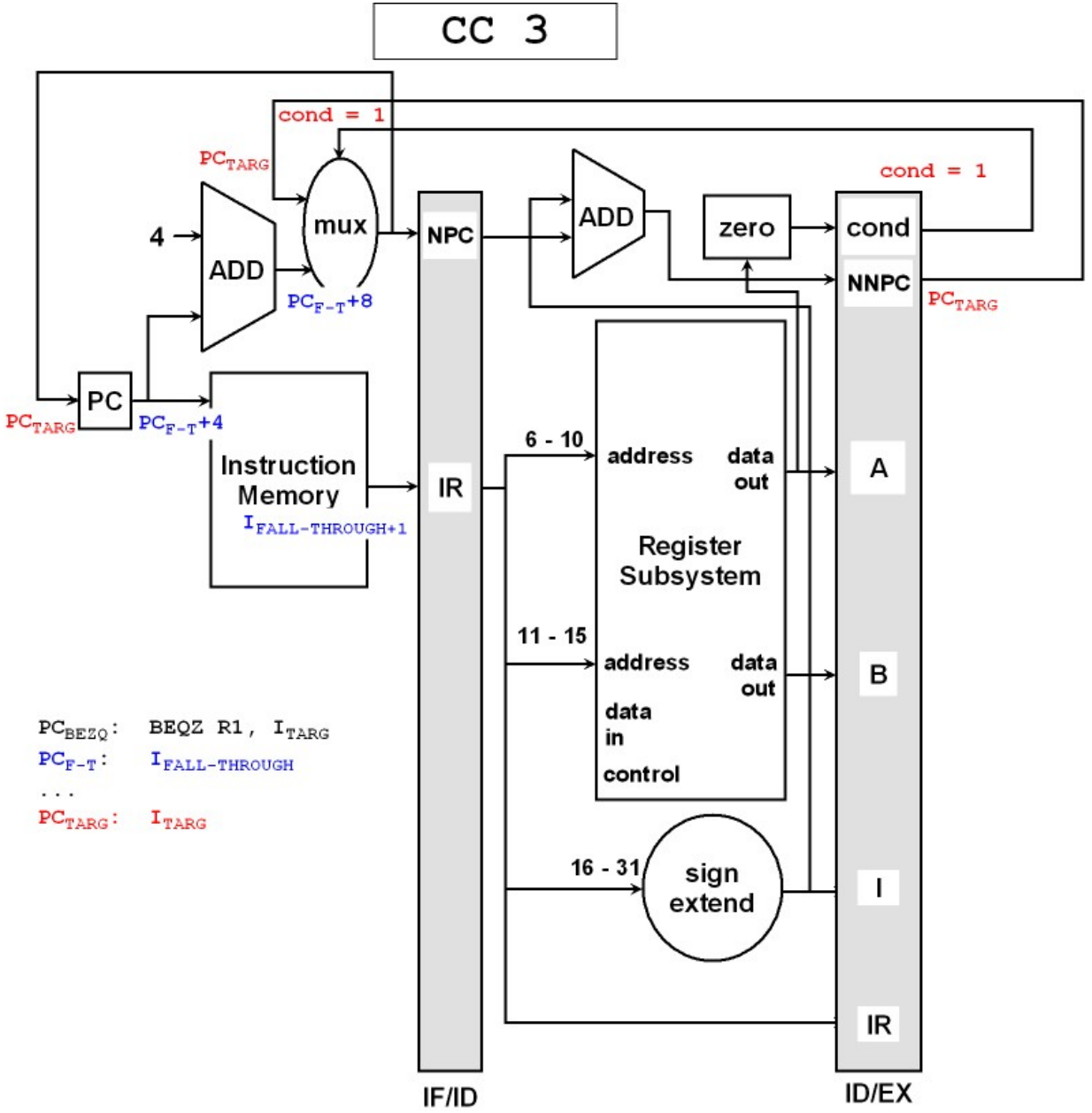
ID/EX latches

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"

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

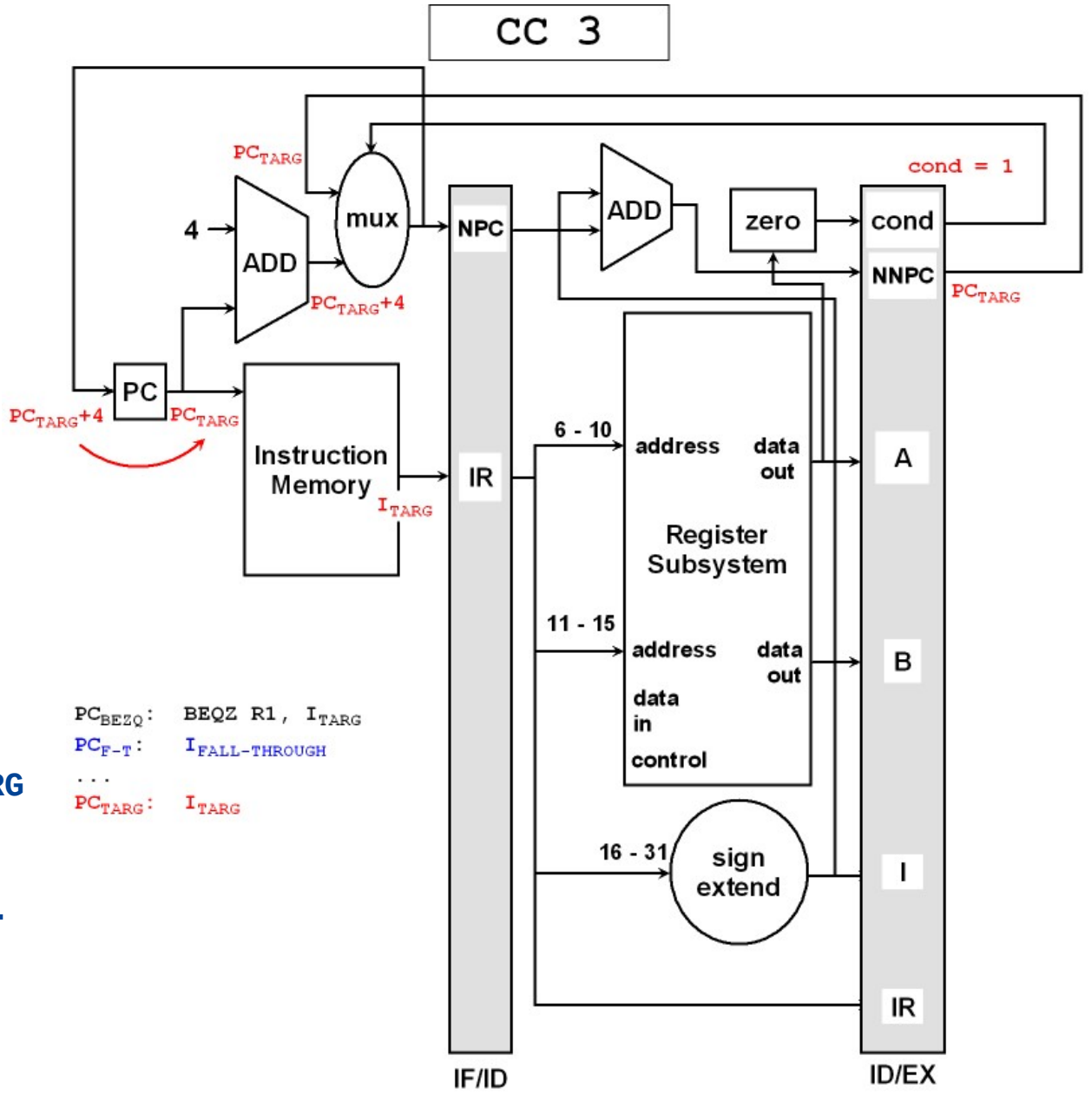
Points to I_{TARG+1}

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%

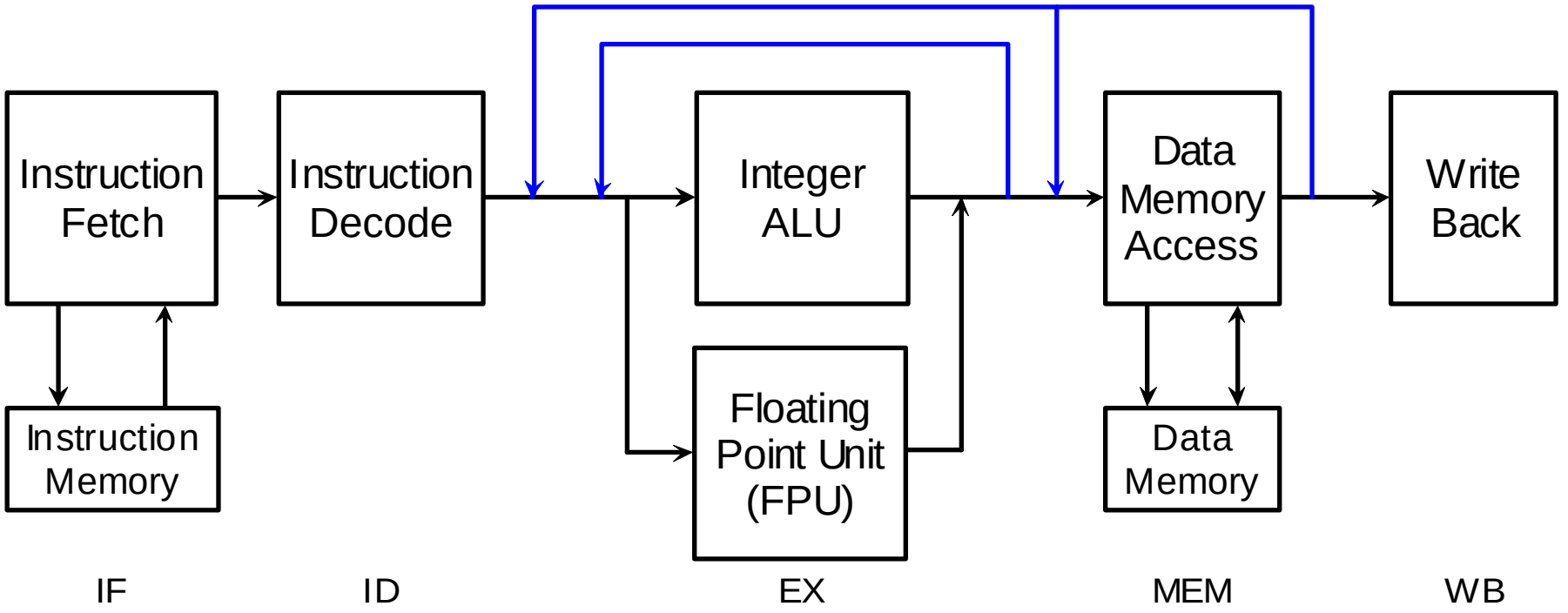
$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stalls}} \times \frac{\text{stalls}}{\text{instruction type}} \times \frac{\text{instruction types}}{\text{instruction}}$$

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

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

$CPI = 1.13 \Rightarrow$ (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

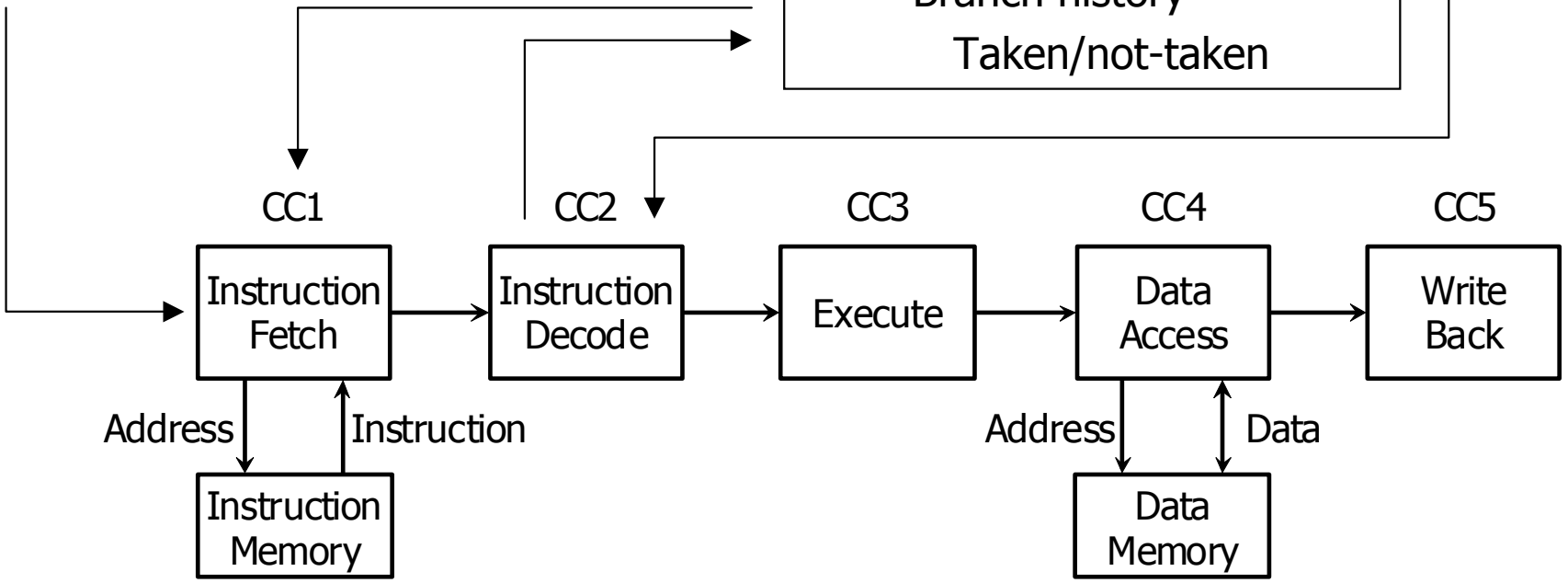
3. After validation

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

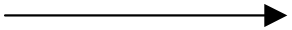
Inlining

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

Computer Arithmetic

Unsigned Integers

Binary representation

$$k_2 = a_{n-1} \quad a_{n-2} \quad a_{n-3} \quad \ldots \quad a_1 \quad a_0 \qquad a_i \in \{0,1\}$$

$$\underbrace{00 \ \ldots \ 0}_{n} \le k_2 \le \underbrace{11 \ \ldots \ 1}_{n}$$

Decimal value

$K(k_2) =$ decimal value of k_2

$$k_{10} = K(k_2) = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \ \ldots \ + a_1 \times 2^1 + a_0 \times 2^0$$

$$(k_2)_{\min} = 00 \ \ldots \ 0 \quad \Rightarrow \quad K\left((k_2)_{\min}\right) = 0$$

$$(k_2)_{\max} = \underbrace{11 \ \ldots \ 1}_{n} \quad \Rightarrow \quad (k_2)_{\max} + 1 = 1\underbrace{00 \ \ldots \ 0}_{n}$$

$$K\left((k_2)_{\max} + 1\right) = \left(1 \times 2^n + 0 \times 2^{n-2} + \ \ldots \ + 0 \times 2^0\right) = 2^n$$

$$K\left((k_2)_{\max}\right) = 2^n - 1,$$

$$0 \le k_{10} = K(k_2) \le 2^n - 1$$

$k > 2^n - 1$ cannot be represented in n bits

Signed Integers

Binary representation (two's complement)

$$k_2 = a_{n-1} \quad a_{n-2} \quad a_{n-3} \quad \ldots \quad a_1 \quad a_0 \qquad a_i \in \{0,1\}$$

Non-negative values

$$0 \le k_{10} \le 2^{n-1} - 1$$

$$k_{10} = K(k_2) = K(0 \quad a_{n-2} \quad a_{n-3} \quad \ldots \quad a_1 \quad a_0)$$

$$= 0 \times 2^{n-1} + a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \ldots + a_1 \times 2^1 + a_0 \times 2^0$$

Negative values

$$0 > k_{10} \ge -2^{n-1} \Rightarrow k_{10} = -K((k_2)'+1)$$

$$k_2 = 11 \quad \ldots 1 \Rightarrow k_{10} = -K((11 \quad \ldots 1)'+1) = -K((00 \quad \ldots 0)+1) = -1$$

$$k_2 = 11 \quad \ldots 0 \Rightarrow k_{10} = -K((11 \quad \ldots 0)'+1) = -K((00 \quad \ldots 1)+1) = -2$$

$$\vdots$$

$$k_2 = 10 \quad \ldots 0 \Rightarrow k_{10} = -K((10 \quad \ldots 0)'+1) = -K((01 \quad \ldots 1)+1) = -2^{n-1}$$

Negative Signed Numbers

$$0 > k_{10} \geq -2^{n-1} \Rightarrow k_{10} = -K\left(\left(k_2\right)'+1\right)$$

$$k_2 = a_{n-1} \quad a_{n-2} \quad a_{n-3} \quad \ldots \quad a_1 \quad a_0$$

$$\left(k_2\right)' = (1-a_{n-1}) \quad (1-a_{n-2}) \quad (1-a_{n-3}) \quad \ldots \quad (1-a_1) \quad (1-a_0)$$

$$K\left(k_2 + \left(k_2\right)'+1\right) = K\left(a_{n-1} + (1-a_{n-1}) \quad a_{n-2} + (1-a_{n-2}) \quad \ldots \quad a_0 + (1-a_0)\right)+1$$

$$= K\left(\underbrace{11 \ldots 1}_{n}\right)+1$$

$$= \left(2^n - 1\right)+1$$

$$= 2^n \text{ which has no representation in } n \text{ bits}$$

$$K\left(\left(k_2\right)'+1\right) = 2^n - K\left(k_2\right) = -K\left(k_2\right) + \text{overflow bit (ignored for signed)}$$

General Formula for Signed Numbers

$$k_{10} = \begin{cases} K\left(0\ a_{n-2}\ \ldots\ a_0\right) & , \quad a_{n-1} = 0 \\ -K\left(a_{n-1}\ a_{n-2}\ \ldots\ a_0\right) & , \quad a_{n-1} = 1 \end{cases}$$

$$\boxed{K\left(\left(k_2\right)'+1\right) = 2^n - K\left(k_2\right)}$$

$$= \begin{cases} 0\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots\ + a_0 & , \quad a_{n-1} = 0 \\ -\left[2^n - \left(1\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots\ + a_0\right)\right] & , \quad a_{n-1} = 1 \end{cases}$$

$$= \begin{cases} 0\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots\ + a_0 & , \quad a_{n-1} = 0 \\ -\left[2\times 2^{n-1} - 1\times 2^{n-1}\right] + a_{n-2}\times 2^{n-2} + \ \ldots\ + a_0 & , \quad a_{n-1} = 1 \end{cases}$$

$$= \begin{cases} 0\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots\ + a_0 & , \quad a_{n-1} = 0 \\ -1\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots\ + a_0 & , \quad a_{n-1} = 1 \end{cases}$$

$$= -a_{n-1}\times 2^{n-1} + a_{n-2}\times 2^{n-2} + \ \ldots\ + a_0$$

Multiplying Unsigned Integers

Long Multiplication

```
          0  1  1
      ×   1  0  1
    ---------------
          0  1  1
       0  0  0
    0  1  1
    ---------------
    0  1  1  1  1
```

Algorithm

Operands $a = a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0$

$b = b_{n-1}\ b_{n-2}\ b_{n-3}\ \cdots\ b_0$

Zero temporary register $P = P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \leftarrow 0$ $\quad c_{out} \leftarrow 0$

n times $\{$

$$c_{out}\ P \leftarrow \begin{cases} P, \text{ if } a_0 = 0 \\ P + b, \text{ if } a_0 = 1 \end{cases}$$

shift right bits $c_{out}\ [P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \quad a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0]$

to form new $[P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \quad a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0]$

$\}$

Result is found in bits $P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \quad a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0$

Multiplying Signed Integers

Algorithm

Operands $a = a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0$ (non-negative)

$b = b_{n-1}\ b_{n-2}\ b_{n-3}\ \cdots\ b_0$

Zero temporary register $P = P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \leftarrow 0$

n times {

$$P \leftarrow \begin{cases} P, & \text{if } a_0 = 0 \\ P + b, & \text{if } a_0 = 1 \end{cases}$$

Uses P_{n-1} instead of c_{out}

shift right bits $P_{n-1}\ [P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0\ \ a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0]$

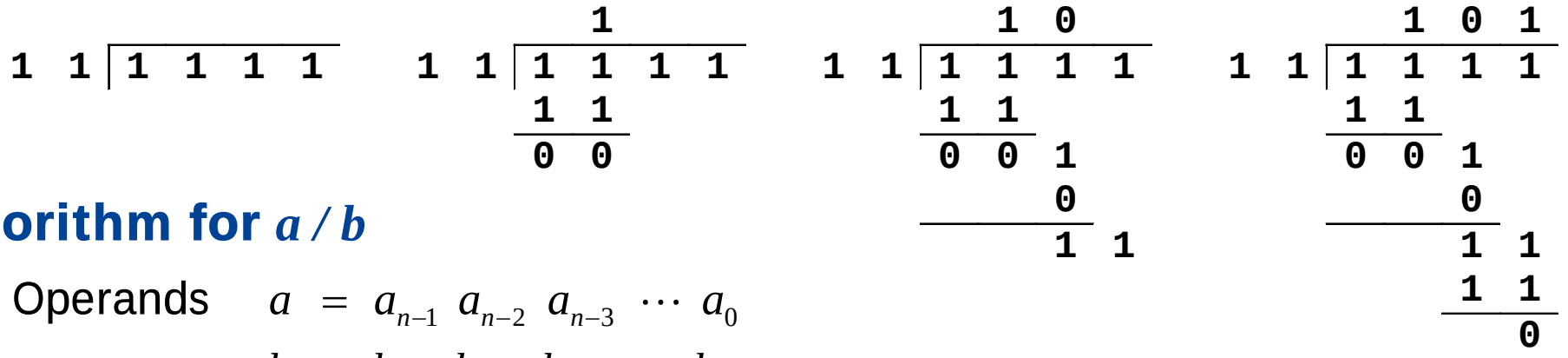
to form new $P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0\ \ a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0$

}

Result is found in bits $P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0\ \ a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0$

Dividing Unsigned Integers

Long Division



Algorithm for a / b

Operands $a = a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0$

$b = b_{n-1}\ b_{n-2}\ b_{n-3}\ \cdots\ b_0$

Zero temporary register $P = P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \leftarrow 0$

n times $\{$

shift bits $[P\ a]$ 0 left to form $\underbrace{P_{n-1}}_{discard}\left[\underbrace{P_{n-2}\ P_{n-3}\ \cdots\ P_0\ a_{n-1}}_{new\ P}\ \ \underbrace{a_{n-2}\ a_{n-3}\ \cdots\ a_0\ 0}_{new\ a}\right]$

if $P \geq b$, $\begin{cases} P \leftarrow P - b \\ a_{n-2}\ \cdots\ a_0\ 0 \leftarrow a_{n-2}\ \cdots\ a_0\ 1 \end{cases}$

$\}$

Remainder is in P and Quotient is in a

Floating Point Numbers (IEEE-754 Standard)



Single precision

s — 1 bit

e — 8 bits

f — 23 bits

$$N = (-1)^s \times (1.f)_2 \times 2^{e-127},$$

$$1 \le e \le 254$$

$$-126 \le e - 127 \le 127$$

Double precision

s — 1 bit

e — 11 bits

f — 52 bits

$$N = (-1)^s \times (1.f)_2 \times 2^{e-1023},$$

$$1 \le e \le 2046$$

$$-1022 \le e - 1023 \le 1023$$

Special values

e	f	N
0	0	0
0	Not zero	$N = (-1)^s \times 0.f \times 2^{-127}$ | $N = (-1)^s \times 0.f \times 2^{-1023}$
255	0	$N = (-1)^s \times \infty$
255	Not zero	NaN (not a number — overflow / underflow)

Floating Point Operations

Addition

$$1.010\times2^{-3}+1.010\times2^{-4}=10100\times2^{-7}+1010\times2^{-7}$$

$$\begin{array}{r} 10100 \\ +\ 01010 \\ \hline 11110 \end{array}$$

$$11110\times2^{-7}=1.1110\times2^{-3}$$

Multiplication

$$1.010\times2^{-3}\times1.010\times2^{-4}=10100\times2^{-7}\times1010\times2^{-7}$$

$$\begin{array}{r} 10100 \\ \times\ 01010 \\ \hline 11001000 \end{array}$$

$$11001000\times2^{-7}\times2^{-7}=1.1001000\times2^{-7}\rightarrow1.100\times2^{-7}$$

Floating Point Multiplication

$$x = (-1)^{s_1} \times 1.f_1 \times 2^{e_1-127}$$

$$y = (-1)^{s_2} \times 1.f_2 \times 2^{e_2-127}$$

$$x \times y = (-1)^{s_1+s_2} \times [1.f_1 \times 1.f_2] \times 2^{(e_1+e_2-127)-127} = (-1)^{s} \times [1.f_1 \times 1.f_2] \times 2^{e-127}$$

Multiply unsigned numbers

$$P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \quad a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0 \quad \leftarrow \quad 1.f_1 \times 1.f_2$$

Rounding algorithm

If $P_{n-1} = 0$ assemble

X_{n-1}	X_{n-2}	...	X_1	X_0	r
P_{n-2}	P_{n-3}		P_0	a_{n-1}	a_{n-2}

If $P_{n-1} = 1$ assemble

X_{n-1}	X_{n-2}	...	X_1	X_0	r	$e \leftarrow e+1$
P_{n-1}	P_{n-2}		P_1	P_0	a_{n-1}	

$X_{n-1}\ .\ X_{n-2}\ \ldots\ X_0\ +\ r \times (2^{-(n-1)})$ r is rounding bit

Floating Point Addition — 1

$$\left.\begin{aligned} x_1 &= (-1)^{s_1} \times 1.f_1 \times 2^{e_1-127} \\ x_2 &= (-1)^{s_2} \times 1.f_2 \times 2^{e_2-127} \end{aligned}\right\} \quad X = x_1 + x_2 = (-1)^{s} \times 1.f \times 2^{e-127}$$

1a. If $e_2 > e_1$ then $x_1 \leftrightarrow x_2$

1b. $d = e_1 - e_2$

1c. $e = e_1$

1d. $w = 0$

2. If $s_1 \neq s_2$ then $x_2 \leftarrow (x_2)$" (two's complement) and $w \leftarrow 1$

3a. Construct from x_2

P_{n-1}	.	P_{n-2}	P_{n-3}	...	P_0
x_{n-1}	.	x_{n-2}	x_{n-3}	...	x_0

3b. Shift P right by d places, shifting in w from left

P_{n-1}	.	P_{n-2}	...	P_{n-d}	P_{n-1-d}	...	P_0	g	r
w	.	w	...	w	P_{n-1}	...	P_d	P_{d-1}	P_{d-2}

4. Add $P \leftarrow P + 1.f_1$

Floating Point Addition — 2

$$\left.\begin{aligned} x_1 &= (-1)^{s_1} \times 1.f_1 \times 2^{e_1-127} \\ x_2 &= (-1)^{s_2} \times 1.f_2 \times 2^{e_2-127} \end{aligned}\right\} \quad X = x_1 + x_2 = (-1)^{s} \times 1.f \times 2^{e-127}$$

5. If $s_1 \neq s_2$ and $P_{n-1} = 1$ and $c_{out} = 0$ then $P \leftarrow (P)$''

6. If $s_1 = s_2$ and $c_{out} = 1$ then

   shift right $P_{n-1}\ P_{n-2}\ P_{n-3}\ \ldots\ P_1\ P_0 \leftarrow 1\ P_{n-1}\ P_{n-2}\ P_{n-3}\ \ldots\ P_1$

   $r \leftarrow P_0$

   $e \leftarrow e + 1$

   Otherwise shift P left L times until $P_{n-1} = 1$

   $P = 0\ 0\ 0\ \ldots\ 1\ P_s\ \ldots\ S_1\ P_0 \Rightarrow P \leftarrow 1\ P_s\ P_{s-1}\ P_{s-2}\ \ldots\ P_0\ g\ r\ 0\ \ldots\ 0$

   $1.f \leftarrow P$

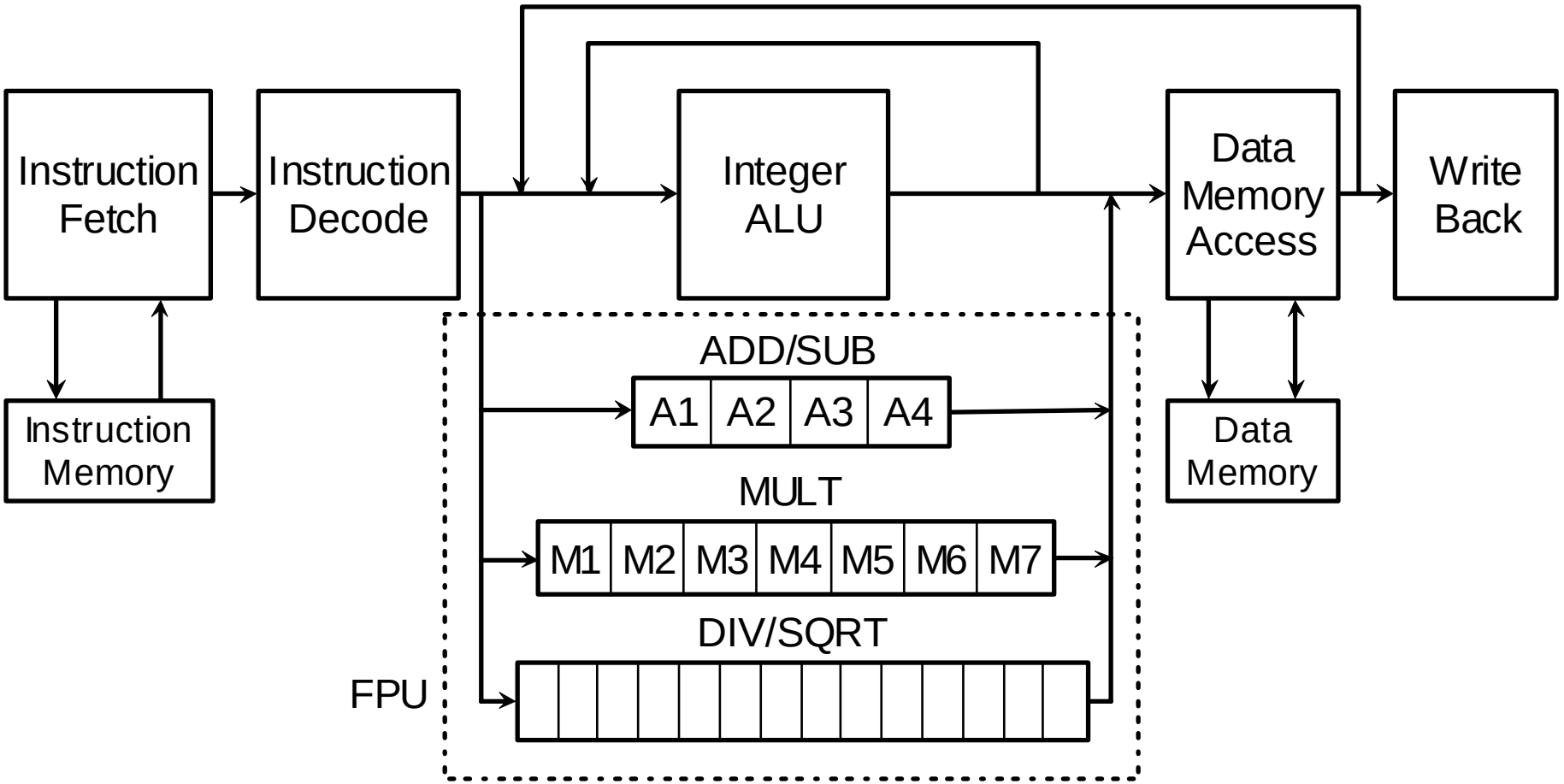
   $e \leftarrow e - L$

   $L = 0 \Rightarrow r \leftarrow g$

   $L = 1 \Rightarrow r$ is rounding bit

   $L > 1 \Rightarrow r \leftarrow 0$

DLX FP Pipeline



Floating Point Unit (FPU)

ADD/SUB: 4 pipelined stages

MULT: 7 pipelined stages

DIV/SQRT: 24 stages — 15 non-pipelined

FP Execution

F = IF D = ID X = EX M = MEM W = WB

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F4,0(R2)	F	D	X	M	W												
ADDD F6,F8,F10		F	D	A1	A2	A3	A4	M	W		Pipelined FP ADD						
ADDD F12,F14,F16			F	D	A1	A2	A3	A4	M	W							

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F4,0(R2)	F	D	X	M	W												
MULTD F2,F4,F8		F	D	M1	M2	M3	M4	M5	M6	M7	M	W		Pipelined FP MULT			
MULTD F0,F2,F6			F	D	M1	M2	M3	M4	M5	M6	M7	M	W				

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F4,0(R2)	F	D	X	M	W						Dependency stalls Forwarding						
ADDD F2,F4,F8		F	D	D	A1	A2	A3	A4	M	W							
MULTD F0,F2,F6			F	F	D	D	D	D	M1	M2	M3	M4	M5	M6	M7	M	W
SD 0(R2),F0					F	F	F	F	D	X	X	X	X	X	X	M	W

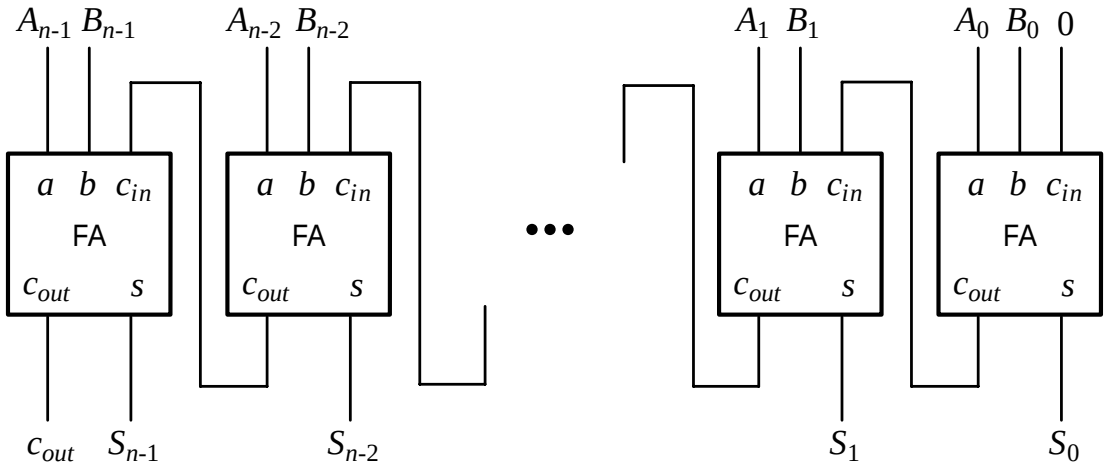
More Hardware → Fewer Steps → Speedup

Ripple Adder

Adds order by order

$c_{out} \rightarrow$ next order as c_{in}

$n-1$ propagation delays from order to order



Look Ahead Adder

Each stage produces

$$\begin{cases} s_i = a_i \oplus b_i \oplus c_i \\ c_i = a_{i-1}b_{i-1} + (a_{i-1} + b_{i-1})c_{i-1} = g_{i-1} + p_{i-1}c_{i-1} \end{cases}$$

Calculate $n-1$ values for c_{in} in (large) dedicated hardware

$$c_1 = g_0 + p_0 c_0$$

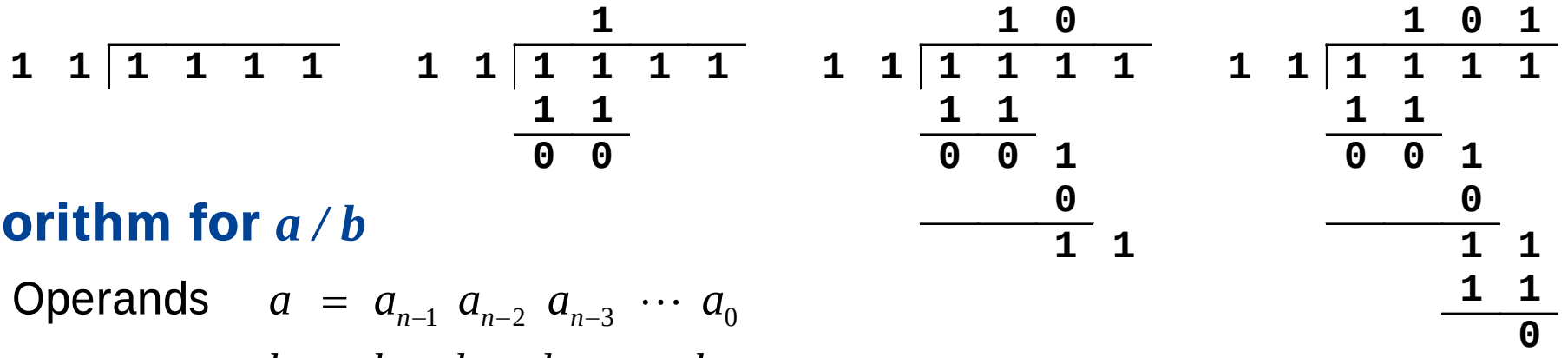
$$c_2 = g_1 + p_1 c_1 = g_1 + p_1[g_0 + p_0 c_0] = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2[g_1 + p_1 g_0 + p_0 p_1 c_0] = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

$$c_k = g_{k-1} + p_{k-1}g_{k-2} + p_{k-1}p_{k-2}g_{k-3} + p_{k-1}p_{k-2}p_{k-3}g_{k-4} + p_{k-1}p_{k-2}p_{k-3}p_{k-4}g_{k-5}$$
$$+ \ldots + p_{k-1}p_{k-2}\cdots p_1 g_0 + p_{k-1}p_{k-2}\cdots p_0 c_0$$

Dividing Unsigned Integers

Long Division



Algorithm for a / b

Operands $a = a_{n-1}\ a_{n-2}\ a_{n-3}\ \cdots\ a_0$

$b = b_{n-1}\ b_{n-2}\ b_{n-3}\ \cdots\ b_0$

Zero temporary register $P = P_{n-1}\ P_{n-2}\ P_{n-3}\ \cdots\ P_0 \leftarrow 0$

n times $\{$

shift bits $[P\ a]$ 0 left to form $\underbrace{P_{n-1}}_{discard}\left[\underbrace{P_{n-2}\ P_{n-3}\ \cdots\ P_0\ a_{n-1}}_{new\ P}\ \ \underbrace{a_{n-2}\ a_{n-3}\ \cdots\ a_0\ 0}_{new\ a}\right]$

if $P \geq b$, $\begin{cases} P \leftarrow P - b \\ a_{n-2}\ \cdots\ a_0\ 0 \leftarrow a_{n-2}\ \cdots\ a_0\ 1 \end{cases}$

$\}$

Remainder is in P and Quotient is in a

Advanced Architectures

General Outlook

Fundamental performance parameters

$$T = CPI \times IC \times \tau$$

$$CPI = CPI^{ideal} + CPI^{stall}$$

Technological limitations

IC grows with software complexity

τ reaching physical limit $\approx 10^{-10}$ seconds $= \left(10 \text{ GHz}\right)^{-1}$

$CPI^{ideal} = 1$ for integer pipeline

$$CPI^{stall} = CPI^{stall}_{data\,dependency} + CPI^{stall}_{cache\,miss} + CPI^{stall}_{branch\,penalty}$$

Areas for possible improvement

Instruction and thread level parallelism to achieve $CPI^{ideal} < 1$

Reducing instruction dependency stalls to lower $CPI^{stall}_{data\,dependency}$

Reducing cache latency to lower $CPI^{stall}_{cache\,miss}$

Reducing branch stalls to lower $CPI^{stall}_{branch\,penalty}$

Practical Successes

Instruction Level Parallelism (ILP)

Provide multiple copies of hardware units in each processor

Begin executing multiple instructions on same clock cycle

Multiple instructions finish on every clock cycle $\Rightarrow$ CPI < 1

Reducing instruction dependency stalls

Compiler rescheduling or dynamic rescheduling (out-of-order execution)

Improving floating point performance

Parallel process FP instructions

Reducing branch stalls

Advanced branch prediction

Reducing cache latency

Processor pre-fetches cache blocks based on address prediction

Optimization of data structures

Thread Level Parallelism

Provide multiple complete processor cores

Divide code into independently executing threads

Types of Parallelism

Pipelining

Instruction I_{n+1} begins before In completes

1 instruction completes on every clock cycle τ

R = 1 / τ instructions complete every second

Superscalar

M > 1 copies of pipeline in parallel

Execute M instructions start on same clock cycle

M instructions complete on every clock cycle

Superpipelining

Divide pipeline into smaller stages — less work per stage

Less work $\Rightarrow$ shorter clock cycle $\tau' < \tau \Rightarrow$ higher clock rate R' > R

R' > R instructions complete every second

Multiprocessor

N > 1 program sections running on N processors

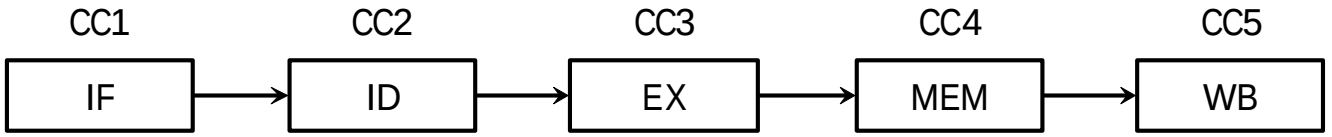
Overall program runs in less time

DLX Pipeline

Five pipeline stages

New instruction begins on each clock cycle

One instruction completes on each 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	

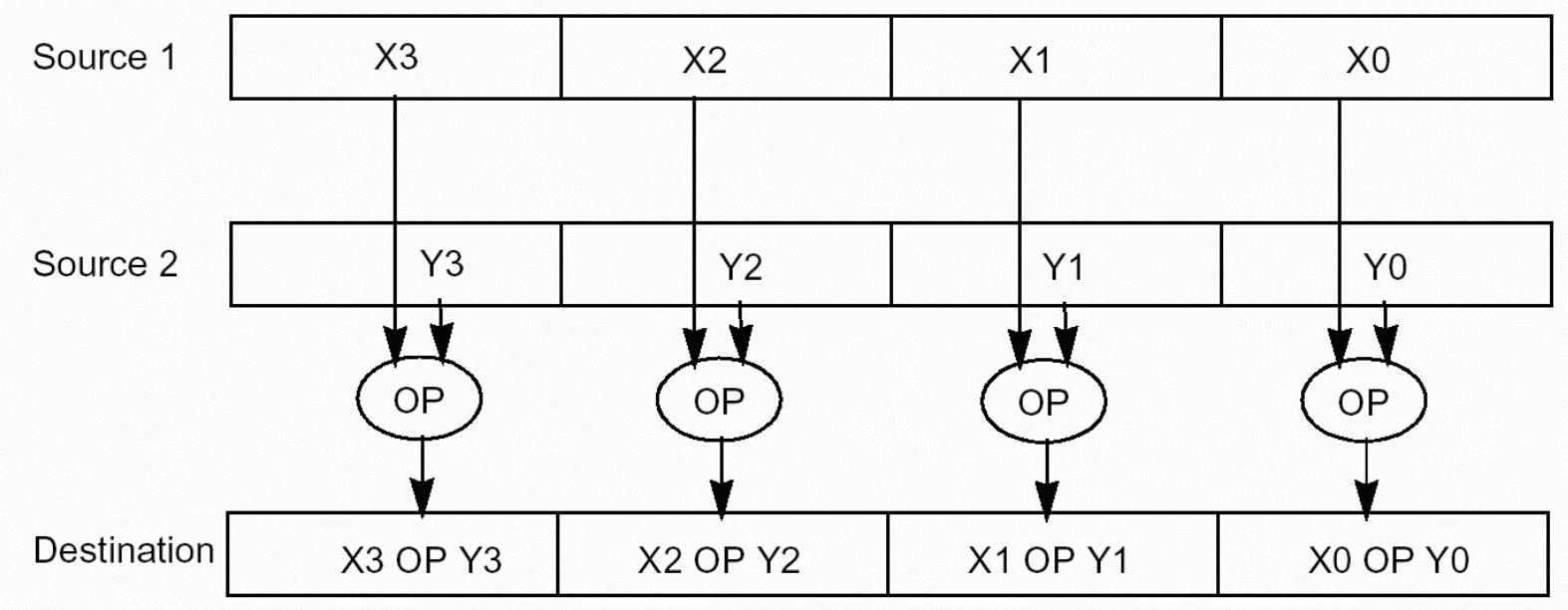
$$CPI^{ideal} = \frac{\text{cycles}}{\text{instruction}} = \frac{IC+4}{IC} \xrightarrow[IC \to \text{large}]{} 1$$

$$\text{Run-Time}^{\text{ideal}}_{\text{pipeline}} = CPI^{ideal} \times IC \times \tau \xrightarrow[IC \to \text{large}]{} IC \times \tau$$

Vector Processors

SIMD execution model

Single Instruction performed in parallel on Multiple Data



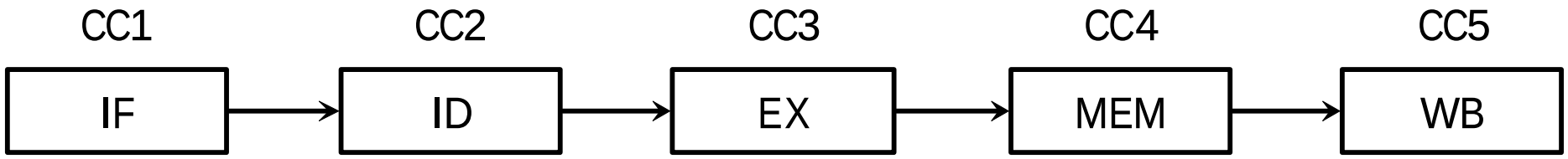
Typical applications for vector operations

Data compression/decompression

Audio processing

Graphics processing

DLX Vector Pipeline Example



p_LW P1, 400(R1)	Load 4 memory words (16 bytes) to register P1
p_LW P2, 800(R1)	Load 4 memory words (16 bytes) to register P2
p_ADD P3, P1, P2	Perform 4 word additions in parallel

	1	2	3	4	5	6	7	8
p_LW P1, 400(R1)	**IF**	**ID**	**EX**	**MEM**	**WB**			
p_LW P2, 800(R1)		**IF**	**ID**	**EX**	**MEM**	**WB**		
p_ADD P3, P1, P2			**IF**	**ID**	**ID**	**EX**	**MEM**	**WB**

Examples of Vector Processing

Intel MMX Technology

Integer operations on 64-bit registers

8 bytes, 4 words, or 2 dwords

Intel SSE (SSE2/SSE3/AVX) Technology

Similar to MMX for Floating Point operations

	SSE	SSE2/SSE3/AVX
Register width	128 bits = 16 bytes	256 bits = 32 bytes
single-precision FP operations	4	8
double-precision FP operations	2	4

PowerPC AltiVec vector processor

Similar to SSE

128-bit registers

Compiler Support

Vector instructions part of processor instruction set

Reasonable compiler supports vectorization

Intel Vectorization Example

```
int i;
float a[100], red;
…
red = 0;
for (i = 0; i < 100; i++) {
     red += a[i];
}
```

Loop runs **100** times

Operates on one array element in each loop iteration

```
int i;
float a[100], red;
…
red = 0;
ASM p_XOR xmm0,xmm0              /* zero 128-bit accumulator */
for (i = 0; i < 25; i++) {
     ASM p_ADD xmm0,a[4*i]       /* four 32-bit additions */
}
ASM h_ADD xmm0                   /* add four 32-bit FP registers to one */
ASM p_MOV red, xmm0              /* move FP sum to memory location */
```

Loop runs **25** times

Operates on 4 array elements in each loop iteration

Cache Refresh Latency

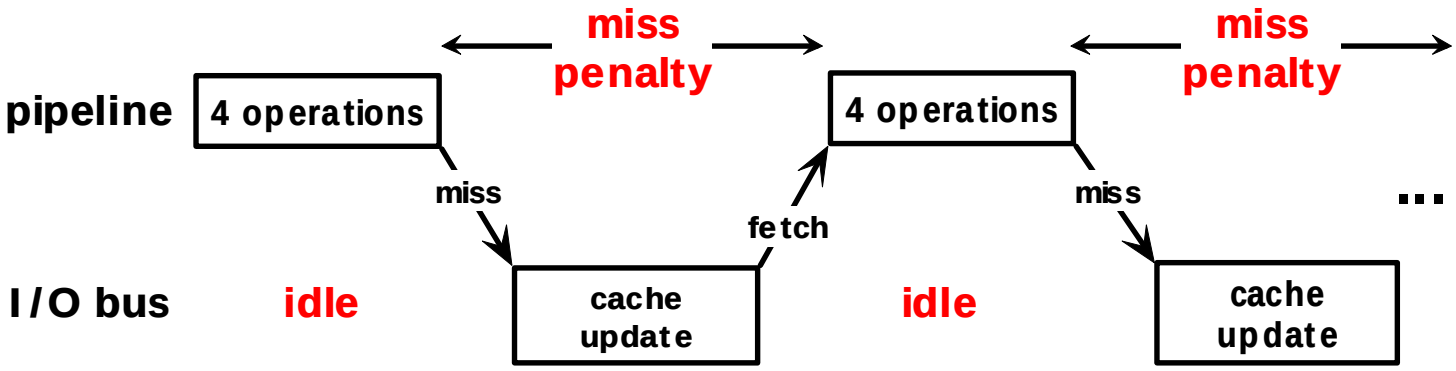
```
for (i = 0; i < 1024; i++) {
     for (j = 0; j < 1024; j++) {
          SSE_operation a[i][j];
     }
}
```

Reads 1024 × 1024 SSE operands (16-byte operands) = 16 MB

Reads sequentially, without repeated access to same data

Pentium 4 has 64 byte block size = 4 x 16-byte operand

Will miss in L1 on every 4 accesses



Cache Prefetch

```
for (i = 0; i < 1024; i++) {
     for (j = 0; j < 1024; j++) {
          prefetch a[i][j+8];
          SSE_operation a[i][j];
     }
}
```

Reads 1024 × 1024 SSE operands (16-byte values) = 16 MB

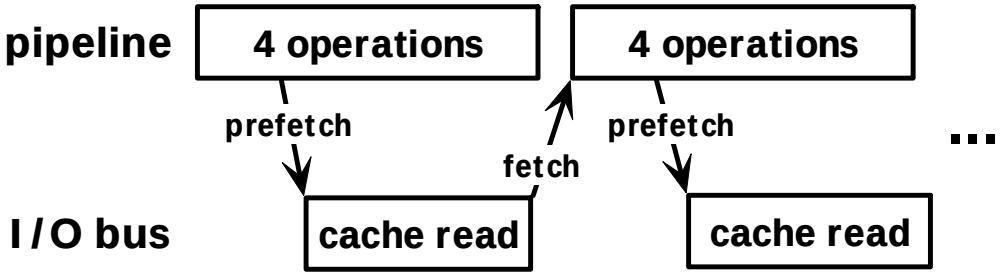
On Pentium 4 prefetching,

Software prefetch loads 128 bytes (2 cache blocks)

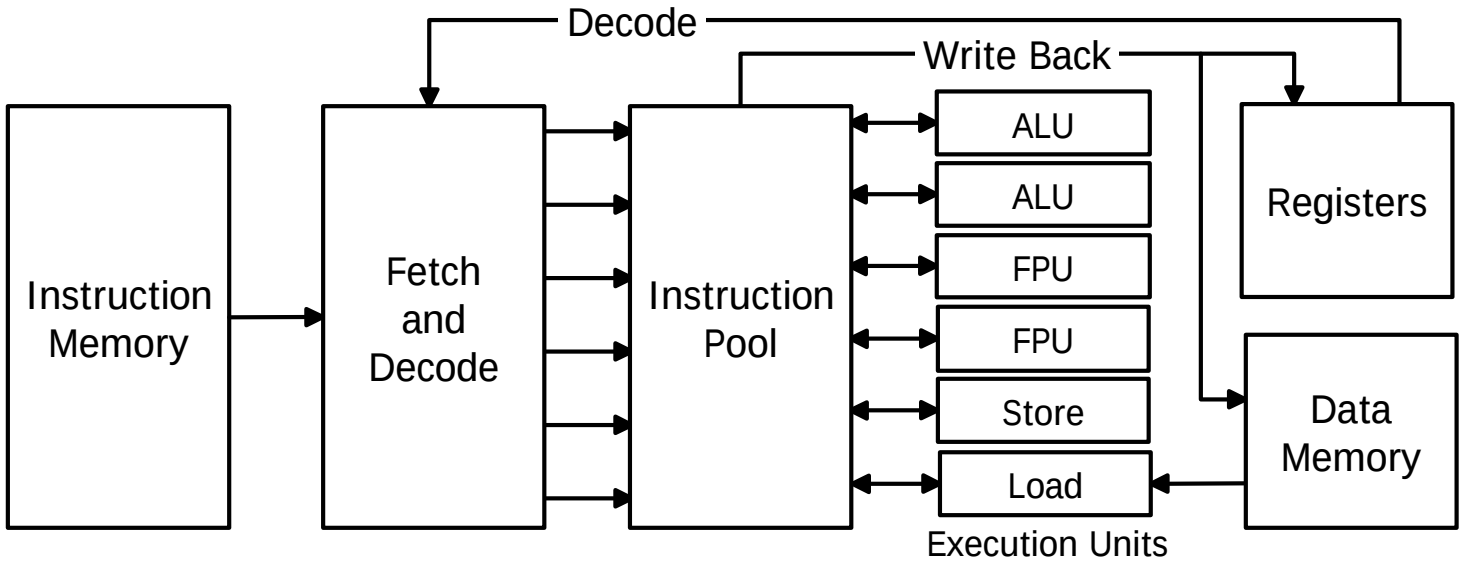
128 bytes = (128/16) = 8 SSE operands

Prefetch 8 operands forward

NOP on prefetch of cache hit



P6 Superscalar with Dynamic Rescheduling



Intel P6 architecture for Intel x86 since Pentium II

Fetch / Decode

Convert IA-32 instructions to 1 – 6 RISC-type **micro-ops** per CC

Instruction pool — out-of-order dynamic rescheduling

Holds micro-ops until ready for execution

Scheduler issues micro-ops to parallel execution in ALU, FPU, Load, Store

Finished micro-ops return to instruction pool with execution results

Retirement (in-order write back)

Finished micro-ops write in original program order

Instruction Scoreboard

Status field assigned to instructions in program listing

NR	Not Ready	At least one source operand not available
R	Ready	All source operands available
X	Executed	Instruction executed — destination operand not available
F	Finished	Instruction executed — all destination operand(s) available

Instructions executed according to

- Status fields
  - Only instructions marked Ready can be executed
- Scheduling policy
  - Depends on hardware organization

Update scoreboard after each clock cycle

- Completed instructions marked Finished
- Instructions marked Ready as operands become available

Scoreboard Example for DLX

Instruction status after EX clock cycle (ignoring IF, ID, MEM, WB)

Instruction	1	2	3	4	5	6	7	8	9	10	11	12
LW R2,[X]	X	F	F	F	F	F	F	F	F	F	F	F
ADD R2,R2,#123	NR	R	F	F	F	F	F	F	F	F	F	F
SW [X],R2	NR	NR	R	F	F	F	F	F	F	F	F	F
LW R3,[Y]	R	R	R	R	X	F	F	F	F	F	F	F
SUB R3,R3,#456	NR	NR	NR	NR	NR	R	F	F	F	F	F	F
SW [Y],R3	NR	NR	NR	NR	NR	NR	R	F	F	F	F	F
LW R4,[Z]	R	R	R	R	R	R	R	R	X	F	F	F
SUB R4,R4,#789	NR	NR	NR	NR	NR	NR	NR	NR	NR	R	F	F
SW [Z],R4	NR	NR	NR	NR	NR	NR	NR	NR	NR	NR	R	F

Scheduling rules

Issue only ready instructions

Choose instructions in **ORIGINAL PROGRAM ORDER**

Scoreboard generates inefficient code

NR	Not Ready
R	Ready
X	Executed
F	Executed and destination operand available

Scoreboard Example for Dynamic DLX

Instruction status after EX clock cycle (ignoring IF, ID, MEM, WB)

Instruction	1	2	3	4	5	6	7	8	9
LW R2,[X]	X	F	F	F	F	F	F	F	F
ADD R2,R2,#123	NR	R	R	F	F	F	F	F	F
SW [X],R2	NR	NR	NR	R	R	R	F	F	F
LW R3,[Y]	R	X	F	F	F	F	F	F	F
SUB R3,R3,#456	NR	NR	R	R	F	F	F	F	F
SW [Y],R3	NR	NR	NR	NR	R	R	R	F	F
LW R4,[Z]	R	R	X	F	F	F	F	F	F
SUB R4,R4,#789	NR	NR	R	R	R	F	F	F	F
SW [Z],R4	NR	NR	NR	NR	NR	R	R	R	F

Scheduling rules

Issue only ready instructions

Choose instructions in **ORDER THEY BECOME READY**

Scoreboard generates

compiler-rescheduled code

NR	Not Ready
R	Ready
X	Executed
F	Executed and destination operand available

Scoreboard Example for P6

Execution condition after each clock cycle (ignoring fetch, decode, write-back)

Instruction	Execution Unit	CC1	CC2	CC3	CC4	CC5	CC6
`LW R2,[X]`	Load	F	F	F	F	F	
`ADD R2,R2,#123`	ALU	R	F	F	F	F	
`SW [X],R2`	Store	NR	R	F	F	F	
`LW R3,[Y]`	Load	R	F	F	F	F	
`SUB R3,R3,#456`	ALU	NR	R	F	F	F	
`SW [Y],R3`	Store	NR	NR	R	F	F	
`LW R4,[Z]`	Load	R	R	F	F	F	
`SUB R4,R4,#789`	ALU	NR	NR	R	F	F	
`SW [Z],R4`	Store	NR	NR	NR	R	F	

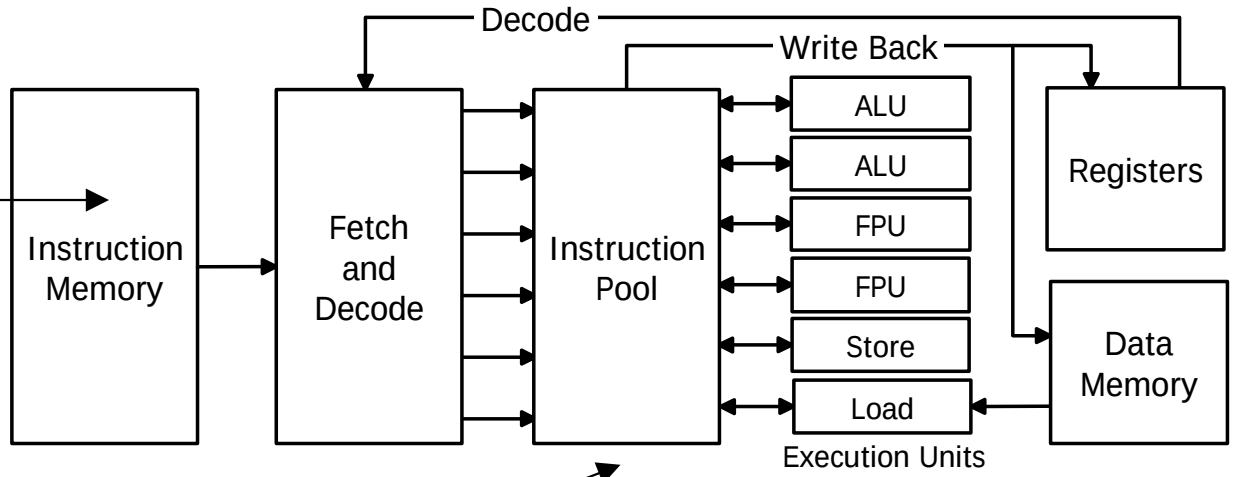
Scheduling rules

Issued only ready instructions
Among ready instructions, maintain program list order
Only 1 load and 1 store per CC
Up to 2 ALU and 2 FPU instructions per CC

NR	Not Ready
R	Ready
F	Finished

Program Execution in P6

```
ADD [X],123
SUB [Y],567
SUB [Z],789
```



IA-32 instructions decoded in 2 CC to RISC micro-ops with register renaming

Dynamic scheduling

```
LW   R2,[X]
ADD  R2,R2,#123
SW   [X],R2
LW   R3,[Y]
SUB  R3,R3,#567
SW   [Y],R3
LW   R4,[Z]
SUB  R4,R4,#789
SW   [Z],R4
```

`LW R2,[X]`	Load	CC1
`ADD R2,R2,#123` `LW R3,[Y]`	ALU Load	CC2

`SW [X],R2`
`SUB R3,R3,#567`
`LW R4,[Z]`
	Store
	ALU
	Load
		CC3

`SW [Y],R3` `SUB R4,R4,#789`	Store ALU	CC4
`SW [Z],R4`	Store	CC5

Hardware Utilization

Unit	CC1	CC2	CC3	CC4	CC5
ALU	IDLE	ADD R2,R2,#123	SUB R3,R3,#567	SUB R4,R4,#789	IDLE
ALU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
FPU	IDLE	IDLE	IDLE	IDLE	IDLE
Load	LW R2,[X]	LW R3,[Y]	LW R4,[Z]	IDLE	IDLE
Store	IDLE	IDLE	SW [X],R2	SW [Y],R3	SW [Y],R4

Good program efficiency

Program executes in minimum number of sequential cycles

Low hardware utilization

Most execution units idle in most clock cycles

Higher ILP ⇒ higher utilization of execution units

Higher utilization ⇒ larger pool of independent instructions

Speculation — deep branch prediction

Many instructions executed before program flow determined

Hardware multithreading

Instructions from different threads are independent

Deep Superpipeline for DLX

Divide each pipeline stage into 2 smaller stages

Each new stage does half the work in half the time

New stage finishes in half the time $\Rightarrow$ Double clock speed

	1	2	3	4	5	6	7	8	9	10	11	12
I_1	IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2		
I_2		IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2	
I_3			IF1	IF2	ID1	ID2	EX1	EX2	MEM1	MEM2	WB1	WB2

Double clock speed $\Rightarrow$ $\tau_{\text{superpipeline}} = \dfrac{\tau_{\text{pipeline}}}{2}$

$$CPI_{\substack{\text{ideal}\\\text{superpipeline}}} = \frac{IC+9}{IC} \xrightarrow[IC \to \text{large}]{} 1$$

$$T_{\substack{\text{ideal}\\\text{superpipeline}}} = CPI_{\substack{\text{ideal}\\\text{superpipeline}}} \times IC \times \frac{\tau_{\text{pipeline}}}{2} \xrightarrow[IC \to \text{large}]{} IC \times \frac{\tau_{\text{pipeline}}}{2} = \frac{1}{2} T_{\substack{\text{ideal}\\\text{pipeline}}}$$

Problems with deep superpipeline

Some instructions cannot be effectively split

Some operations do not scale in time — faster clock $\Rightarrow$ more stall cycles

Cache update, branch penalty, page fault, etc

Pentium 4 Superpipeline

Pentium III

10 stage pipeline at clock speed up to about 1.5 GHz

Pentium 4

20 stage pipeline at clock speed up to about 4.0 GHz

Expect

1.5 GHz processor ~ **50%** faster than same processor at 1.0 GHz

Measurement on SPEC CINT2000

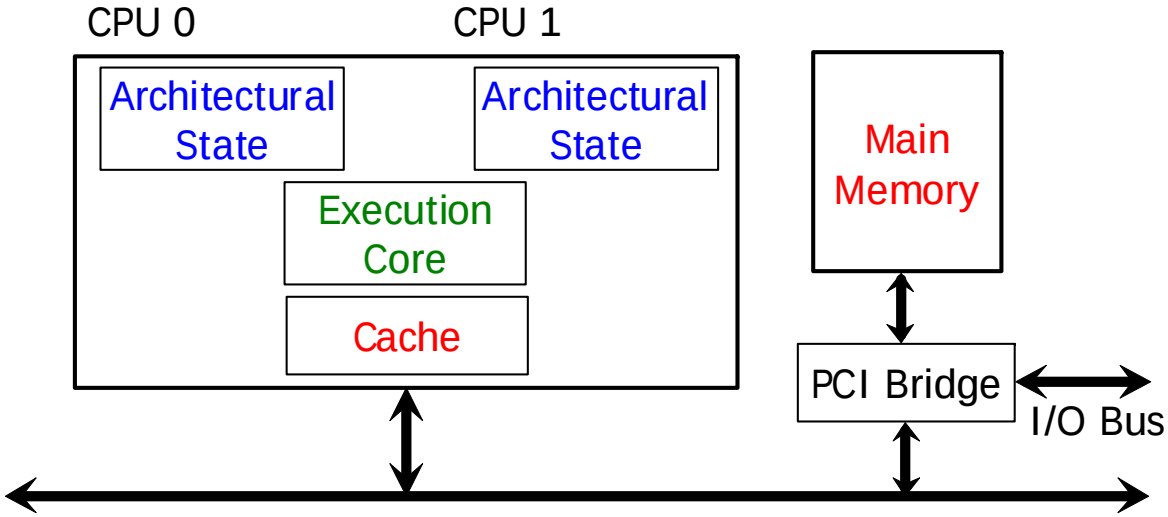
1.5 GHz Pentium-4 ~ **20%** faster than 1.0 GHz Pentium-III

$$S = \frac{CPI^{PIII} \times IC^{PIII} \times \frac{1}{1.0\ \text{GHz}}}{CPI^{P4} \times IC^{P4} \times \frac{1}{1.5\ \text{GHz}}} = \frac{CPI^{PIII}}{CPI^{P4}} \times \frac{1.5}{1.0} = 1.2 \Rightarrow \boxed{CPI^{P4} = 1.25 \times CPI^{PIII}}$$

$$CPI^{P4}_{ideal} + CPI^{P4}_{stall} = 1.25 \times \left(CPI^{PIII}_{ideal} + CPI^{PIII}_{stall} \right)$$

$$CPI^{P4}_{ideal} = CPI^{PIII}_{ideal} = 1 \Rightarrow \boxed{CPI^{P4}_{stall} = 1.25 \times CPI^{PIII}_{stall} + 0.25 \approx 0.5}$$

Hyper-Threading



Architectural State

Registers, stack pointers and program counter

Execution Core

ALU, FPU, vector processors, memory unit

Two copies of architectural state — one execution core

OS sees two sets of registers — looks like two CPUs

OS assigns threads to CPU 0 and CPU 1

CPU 0 and CPU 1 issue instructions to shared execution core

No stall in either thread

CPU 0 and CPU 1 issue instructions on alternate clock cycles

Stall in one thread

Other CPU issues instructions on each clock cycle until stall ends

Both CPUs keep working on most clock cycles

Expected Improvement from Hyper-Threading

Without hyper-threading

$$CPI^{stall} = \frac{\text{stall cycles}}{\text{stall}} \times P(\text{stall}) = CPS \times P_S$$

$CPS =$ cycles per stall

With hyper-threading

$$CPI^{stall}{}' = \frac{\text{stall cycles}}{\text{stall}} \times P(\text{2 simultaneous stalls}) \approx CPS \times \left[P_S\right]^2$$

Speedup

$$S = \frac{CPI \times IC \times \tau}{CPI^{HT} \times IC^{HT} \times \tau^{HT}} = \frac{CPI}{CPI^{HT}} = \frac{1 + CPS \times P_S}{1 + CPS \times \left[P_S\right]^2}$$

Take for Pentium-4

$$CPI^{stall} \approx 0.5$$

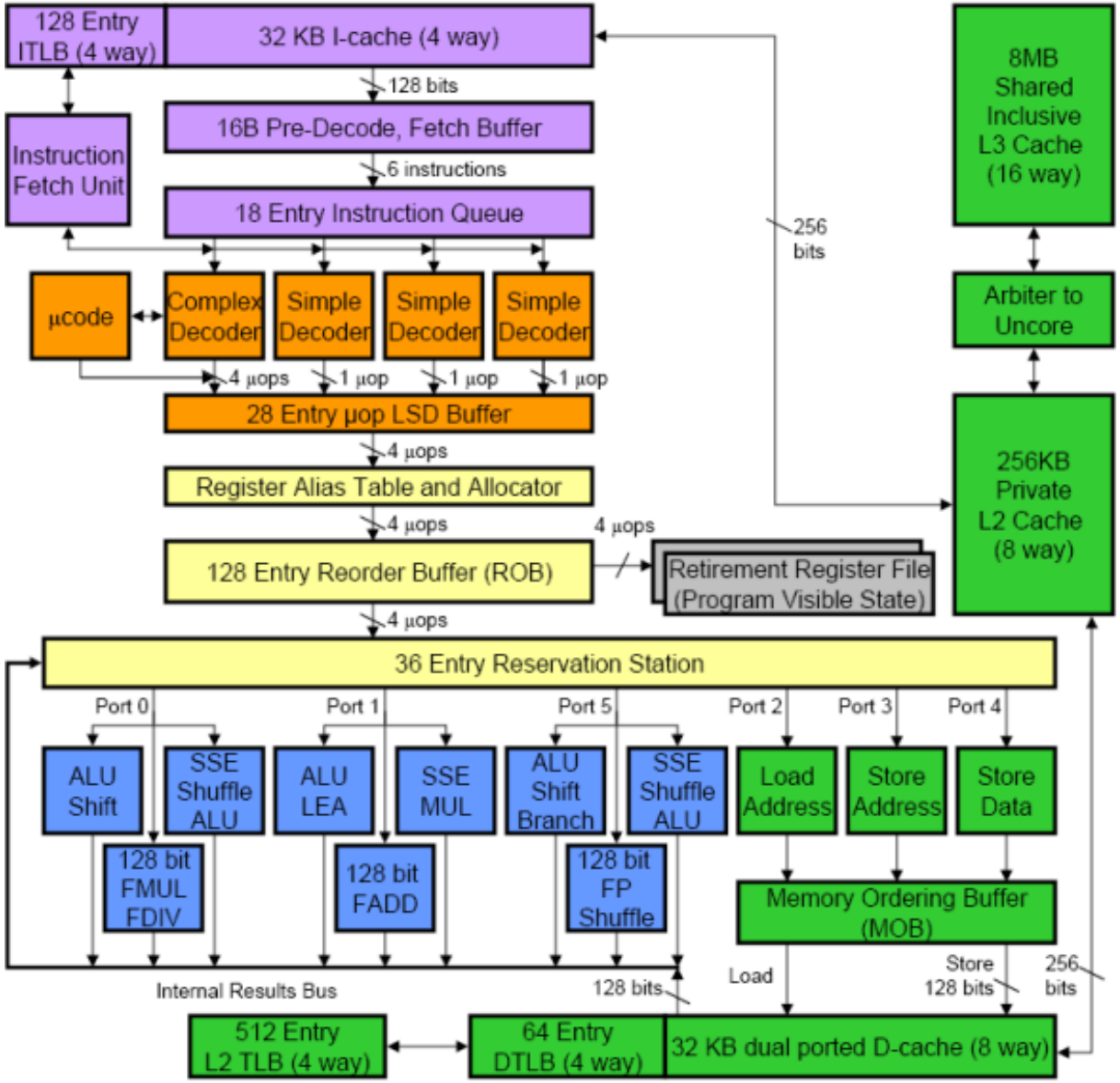
$$CPS = 2 \text{ cycles per stall} \Rightarrow P_S = 0.5 / 2 = 0.25$$

$$S = \frac{1 + 2 \times 0.25}{1 + 2 \times \left[0.25\right]^2} = 1.2$$

Measured improvement $= 1.2$

Intel, "Hyper-Threading Technology Architecture and Microarchitecture"

Intel Nehalem Micro-Architecture



David Kanter, "Inside Nehalem: Intel's Future Processor and System", http://realworldtech.com/includes/templates/articles.cfm?ArticleID=RWT040208182719&mode=print

Amdahl's Equation in Parallel Processing

$$CPI^{N-processors} = \frac{CPI^{N=1}}{N} \times P(\text{work can be parallelized})$$

$$+CPI^{N=1} \times P(\text{work cannot be parallelized})$$

$$= \frac{CPI^{N=1}}{N} \times F_P + CPI^{N=1} \times (1 - F_P)$$

$F_P =$ Fraction of processing that can be performed independently

$N =$ Number of processing units

$$S = \frac{CPI^{N=1} \times IC \times \tau}{CPI^{N-processors} \times IC \times \tau} = \frac{CPI^{N=1}}{CPI^{N-processors}} = \frac{1}{(1 - F_P) + \frac{F_P}{N}}$$

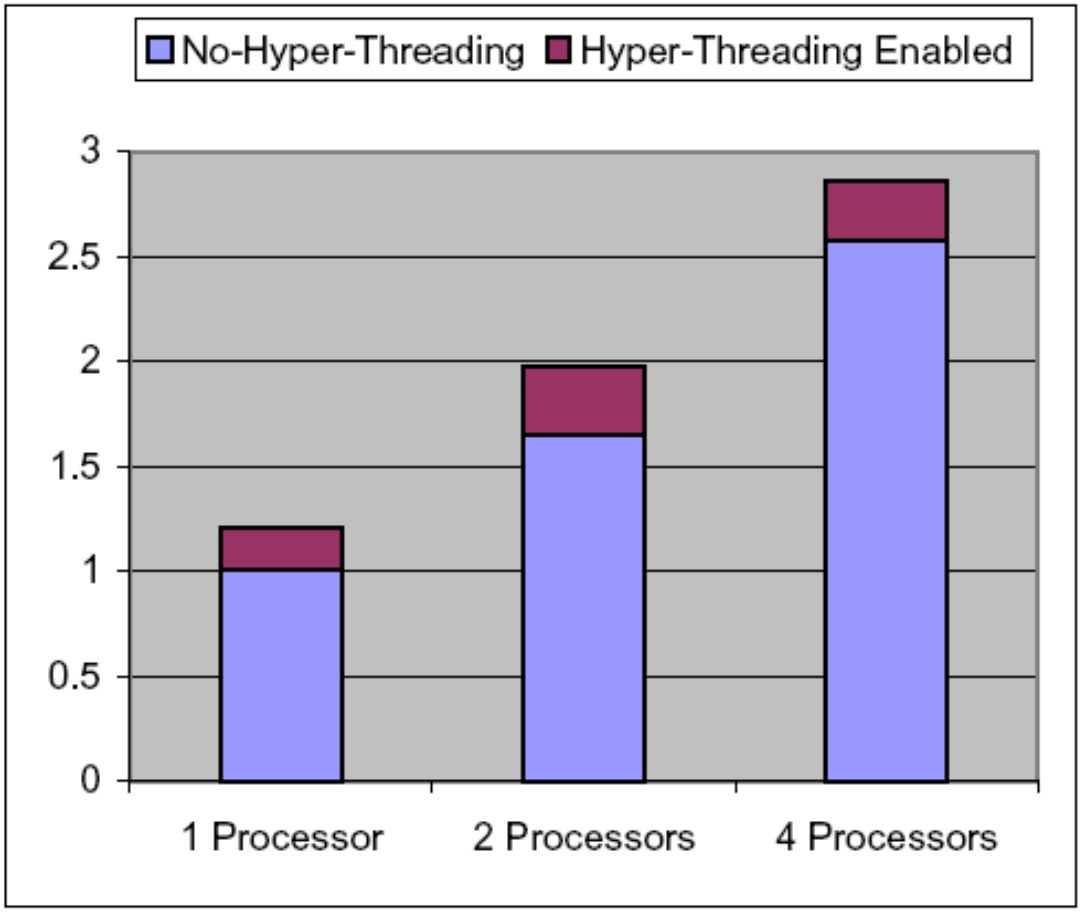
MP and HT Performance Enhancements

MP Without Hyper Threading

CPUs	S	S/CPU
2	1.7	0.85
4	2.6	0.65

Hyper Threading Without MP

CPUs	S	S/CPU
2	1.2	0.60

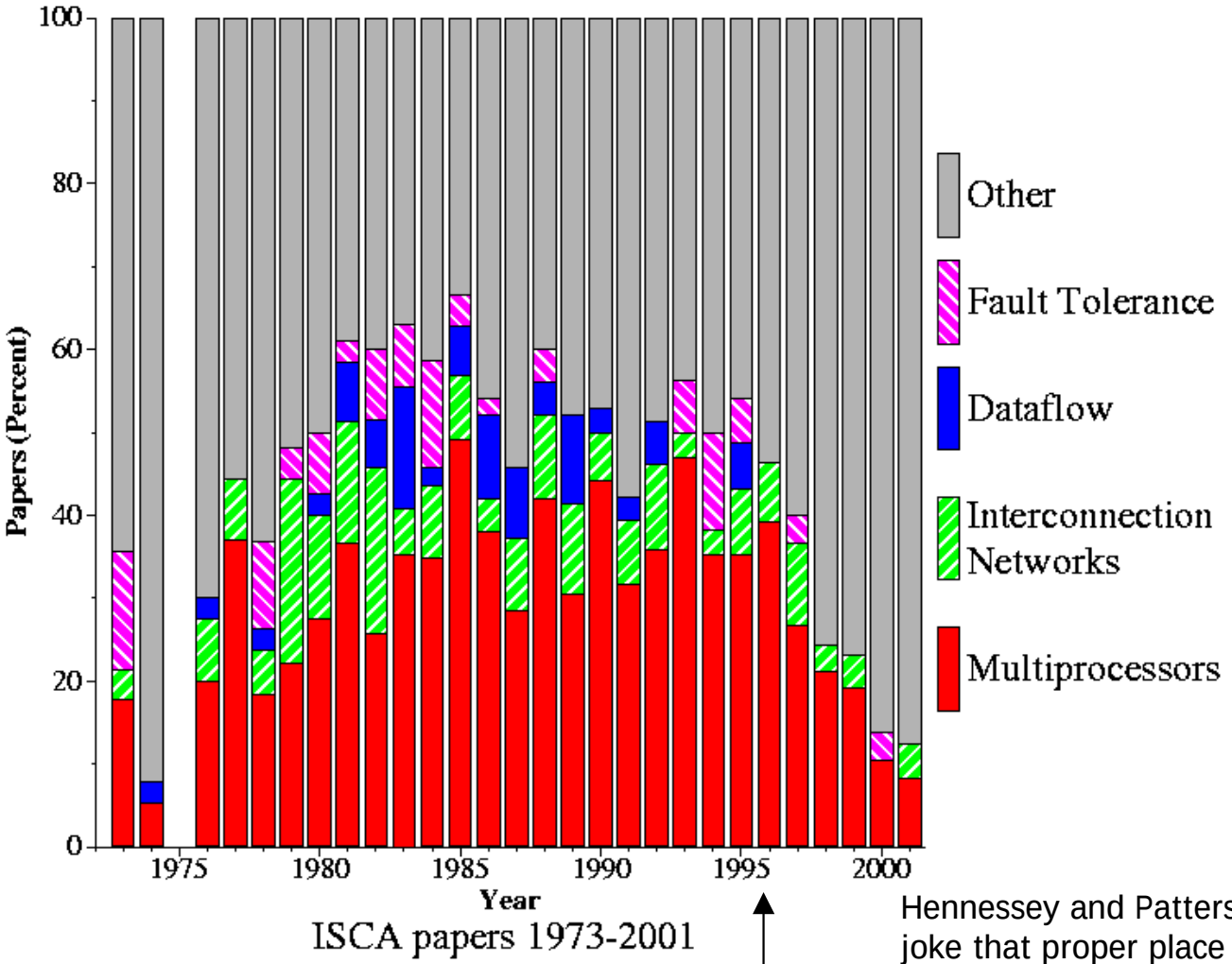


Speed-up (S) for On Line Transaction Processing (OLTP) Workload

Rise and Fall of Multiprocessor R&D

Topics of papers submitted to ISCA 1973 to 2001

Sorted as percent of total



ISCA papers 1973-2001

ISCA — International Symposium on Computer Architecture

Hennessey and Patterson joke that proper place for multiprocessing in their book is Chapter 11 (a section of US business law on bankruptcy)

Ref: Mark D. Hill and Ravi Rajwar, "The Rise and Fall of Multiprocessor Papers in the International Symposium on Computer Architecture (ISCA)", http://pages.cs.wisc.edu/~markhill/mp2001.html

Basic Interprocess Communication Models

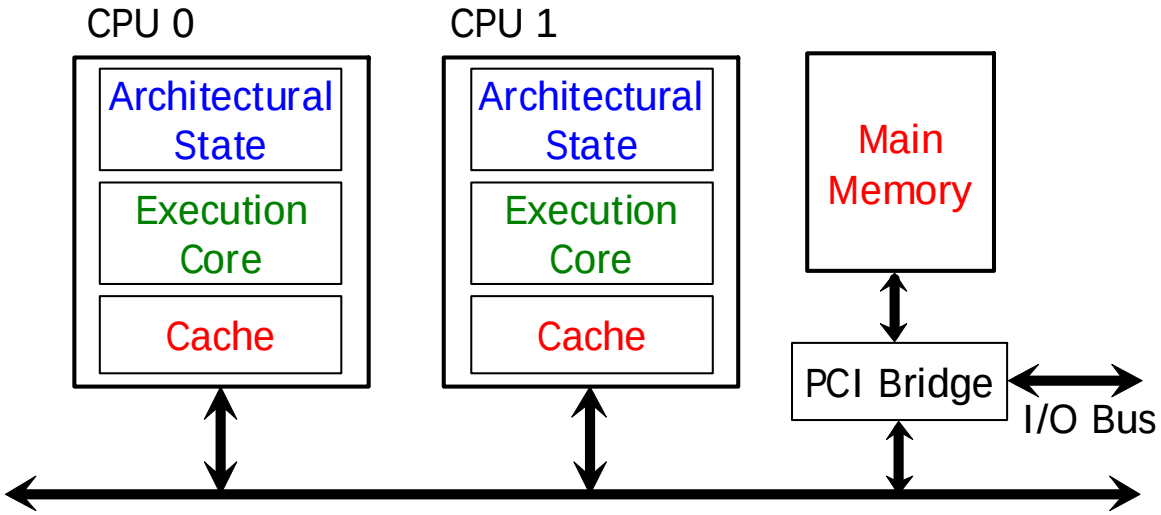
Shared memory system

- Interprocess communication — write/read shared memory location
  - Single shared address space
- Sequential coherence enforced by cache snooping
  - Bus imposes write / read order
  - Cache coherency overhead

Message passing system

- Interprocess communication — send/receive structured messages
  - Send / request data
  - Provide requested data or status
- Sequential coherence enforced by message content + synchronization
  - No snooping or snooping overhead
  - Message management contributes overhead

Multiprocessor Shared Memory Multi-Threading



One or more physical microprocessors

Architectural state

Registers, including stack pointers and program counter

Execution core

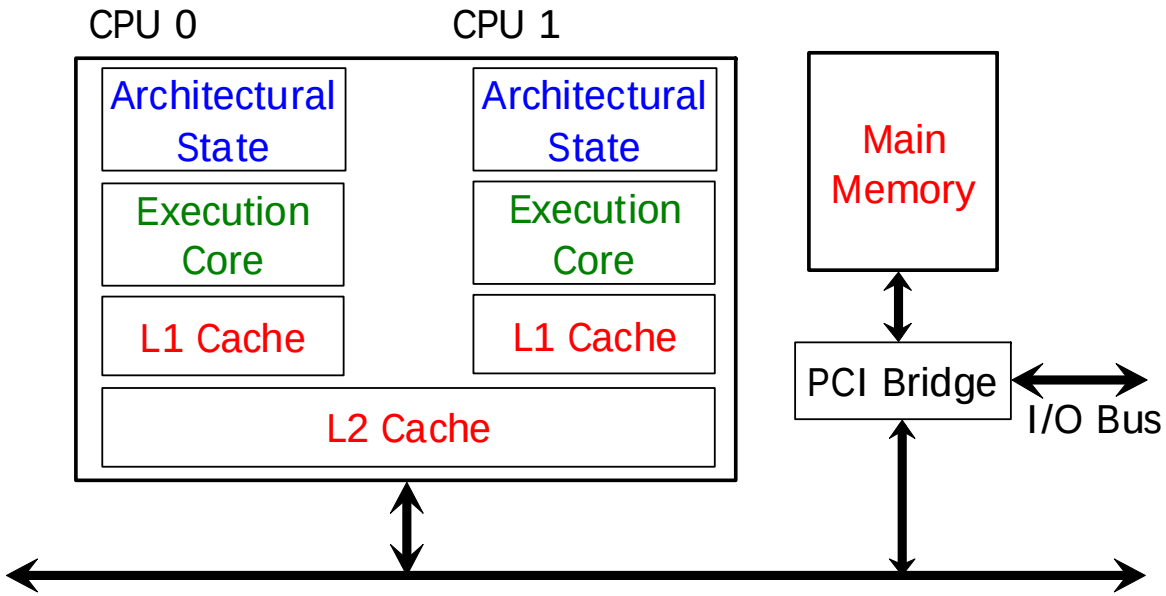
Integer ALUs, FPUs, vector processors, memory access

OS assigns a thread to each processor

Each thread runs independently

On long stall (page fault) a CPU can switch threads

Multi-Core Shared Memory Multi-Threading



Multiprocessor system on one physical chip

Cheaper than multi-microprocessor system

Can be bottleneck at memory bus

- Both processors need to update cache simultaneously
- One processor must wait

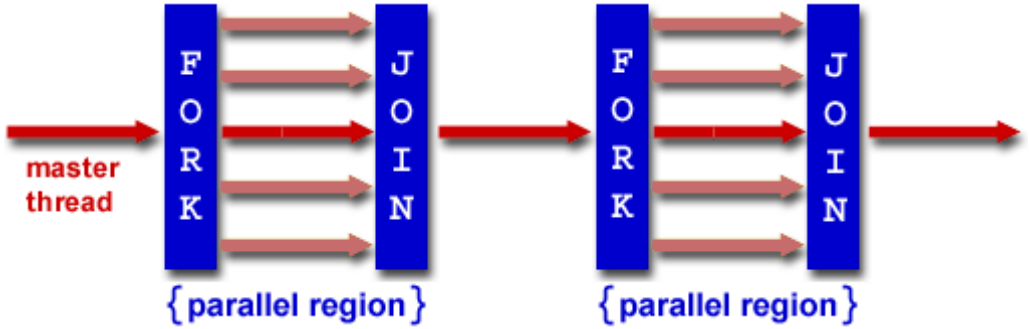
Open MP for Shared Memory Systems

Application Program Interface (API) for multiprocessing

- Supports shared memory applications in C/C++ and Fortran
- Provides directives for explicit thread-based parallelization
- Simple programming models on shared memory machines

Fork — Join Model

- Master thread (consumer thread)
  - Programs initiate as single thread
  - Executes sequentially until parallel construct is encountered
- Fork (producer thread)
  - Master thread creates team of parallel threads
  - Program statements in parallel construct execute in parallel
- Join
  - Team threads complete
  - Synchronize and terminate
  - Master thread continues
- Nesting
  - Forks can be defined within parallel sections



Ref: https://computing.llnl.gov/tutorials/openMP/

General Code Structure

```
#include <omp.h>
main ()  {
int var1, var2, var3;
/* Serial code */
     ...
#pragma omp parallel private(var1, var2) shared(var3)
   {
  /* Parallel section executed by all threads */
     ...
  /* All threads join master thread and disband */
   }
/* Resume serial code */
     ...
}
```

Variables private to each thread
Each thread has private copy

Variables shared among all threads
One copy accessed by all threads

"Hello Worlds" Program

```
#include <omp.h>
main ()  {
int nthreads, tid;
/* Fork team of threads with private variables */
#pragma omp parallel private(tid)
  {
  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);
  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

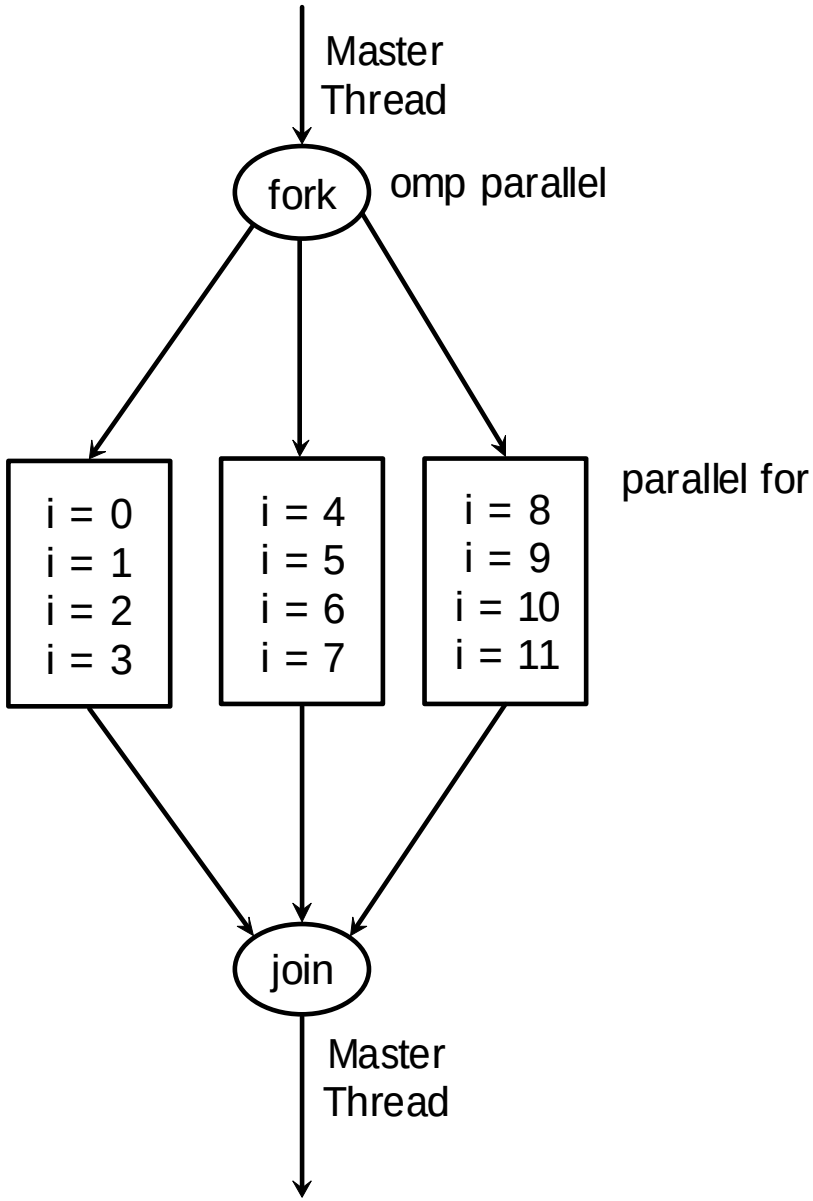
Parallel For

```
#pragma omp parallel
{
#pragma omp for
        for (i = 0; i < 12; i++)
                c[i] = a[i] + b[i];
}
```

Data decomposition

12 loop iterations divided among 3 CPU cores

Each core executes 4 loop iterations in parallel



Sections

```
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections
    {

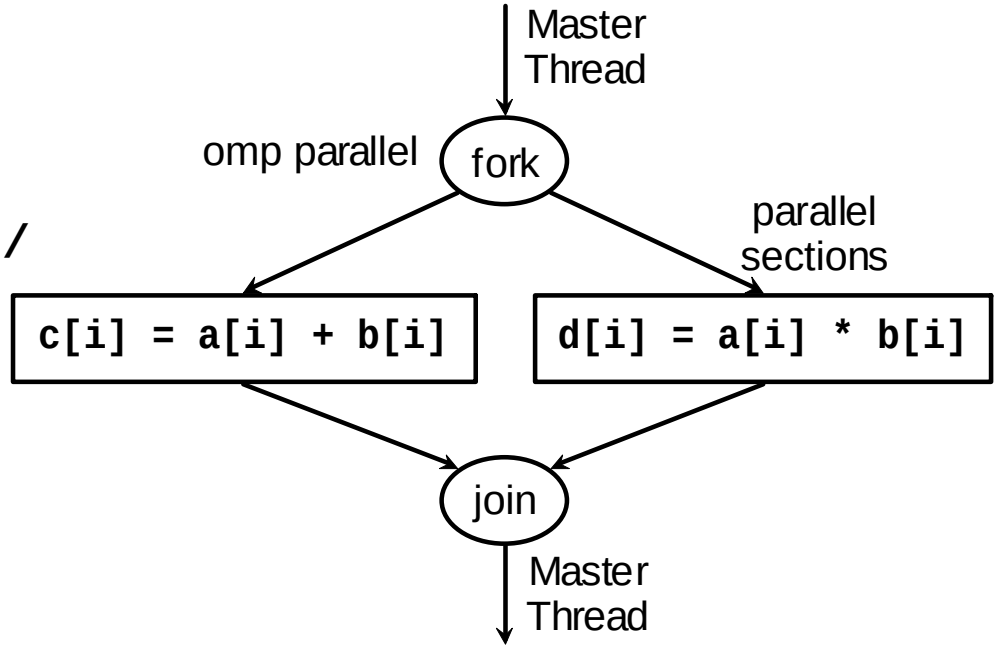
    #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

    #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];

    }  /* end of sections */
  }  /* end of parallel section */
}
```

Functional decomposition

Enclosed sections of code divided among threads in team



Message Passing Example — Vector Product

Compute $\sum_{i=0}^{3} a[i]*b[i]$ from data pre-distributed to nodes

P0	P1	P2	P3
load R_a, a	load R_a, a	load R_a, a	load R_a, a
load R_b, b	load R_b, b	load R_b, b	load R_b, b
$R_a \leftarrow R_a * R_b$	$R_a \leftarrow R_a * R_b$	$R_a \leftarrow R_a * R_b$	$R_a \leftarrow R_a * R_b$
send P1, R_a →	recv P0, R_b	send P3, R_a →	recv P2, R_b
	$R_a \leftarrow R_a + R_b$		$R_a \leftarrow R_a + R_b$
	send P3, R_a →		recv P1, R_b
			$R_a \leftarrow R_a + R_b$
			store p, R_a

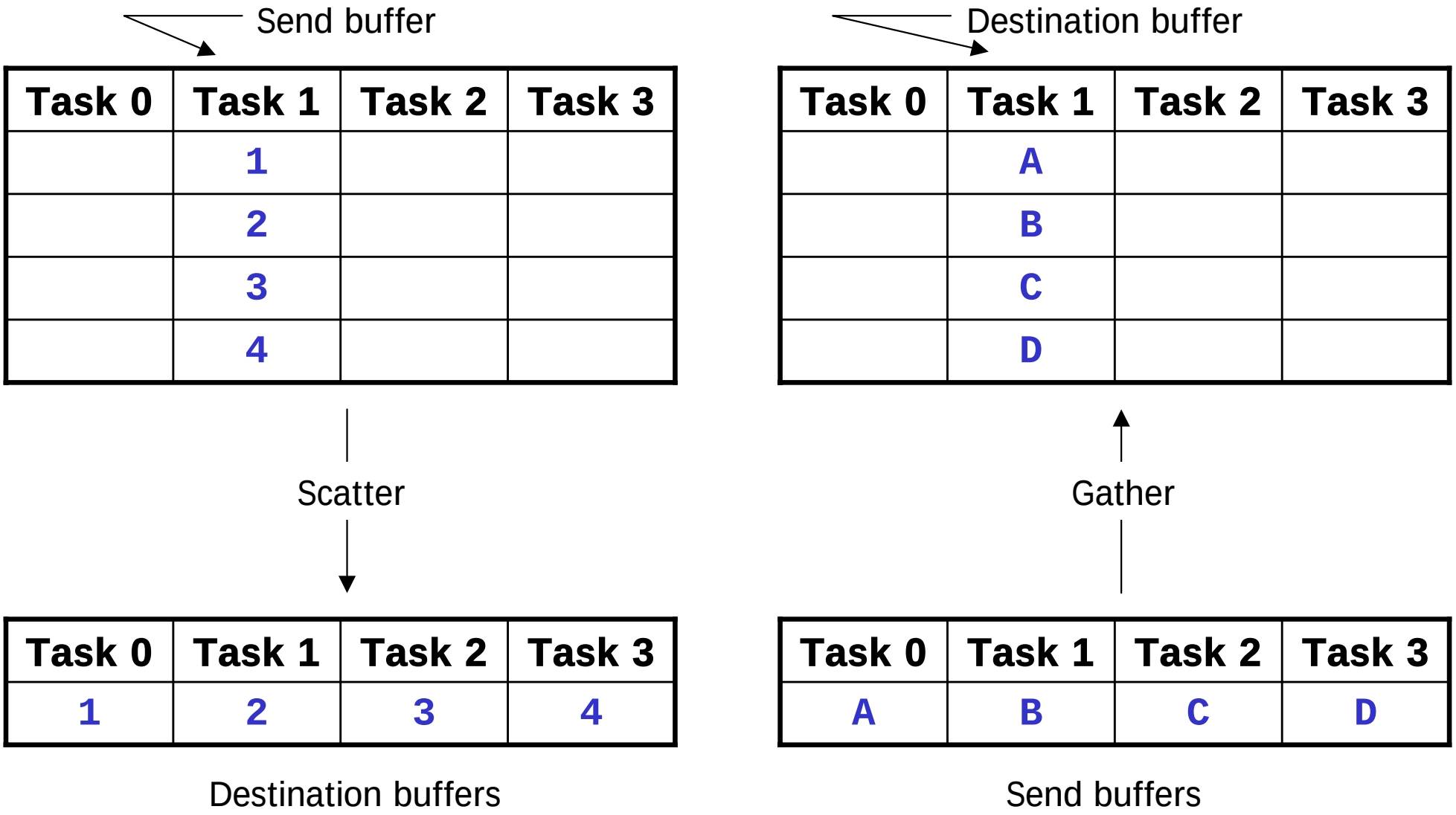
Message overhead

- Source or destination
- Time of creation

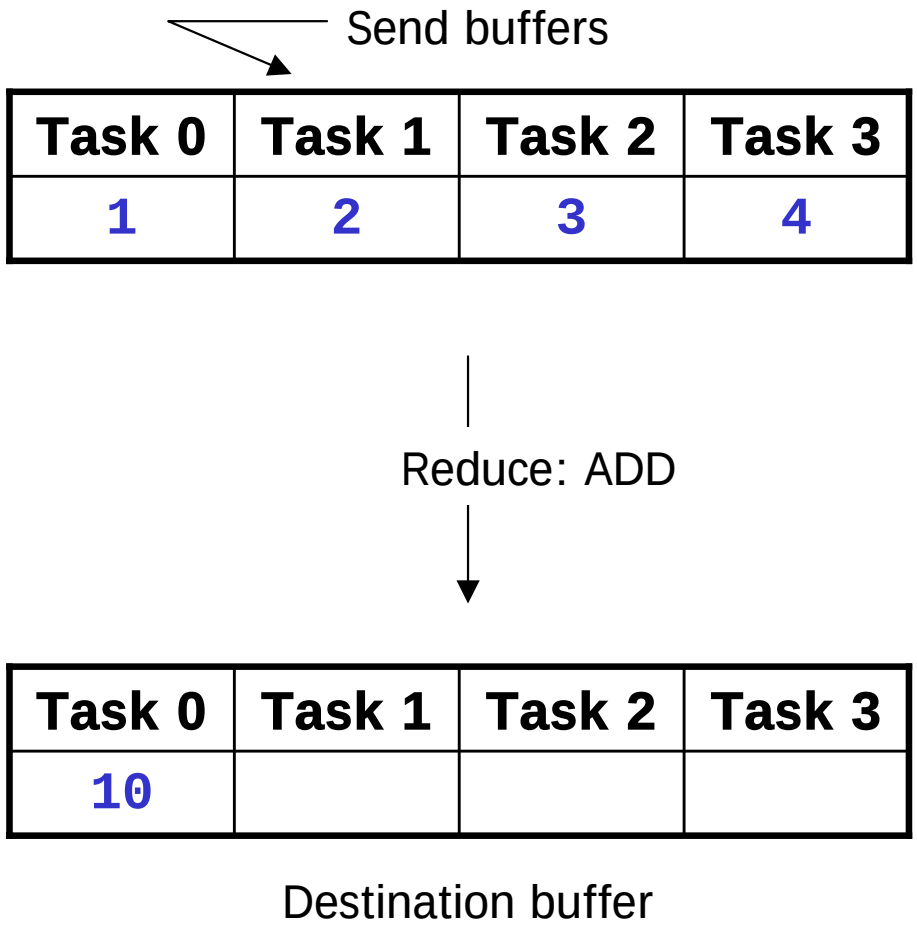
Sequential consistency guaranteed by message overhead

- P3 distinguishes two reads (receives) from P1 and P2 by source address
- No data hazard

Scatter and Gather



Reduce


Send buffers
Task 0
Task 1
Task 2
Task 3
1
2
3
4
Reduce: ADD
Task 0
Task 1
Task 2
Task 3
10
Destination buffer

MPI "Hello World"

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
  char message[20];
  int myrank;              /* myrank = this process number */
  MPI_Status status;       /* MPI_Status = error flags */
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
                           /* MPI_COMM_WORLD = list of active MPI processes */
  if (myrank == 0)         /* code for process zero */
  {
    strcpy(message,"Hello, there");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
  }
  else                     /* code for process one */
  {
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    printf("received :%s:\n", message);
  }
  MPI_Finalize();
}
```

Ref: "MPI: A Message-Passing Interface Standard Version 1.3"

Real-Life RISC

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

WinMIPS

Cross-compiler

Develop applications for MIPS in C / C++ on Windows platforms

Extensive documentation

Tools

GNU GCC compiler and assembler

GNU-Utils for compiler/linker

Download

https://www.ocf.berkeley.edu/~davelai/al/labs/lab10/

Requires cygwin1.dll

Convert C code to assembly code

```
cc1.exe filename.c -o filename.asm
```

MIPS Compilation 1-1

C source

```
main()
{
      int x = 0;
      while (x < 10){
         x++;
      }
}
```

Assembly output of compiler

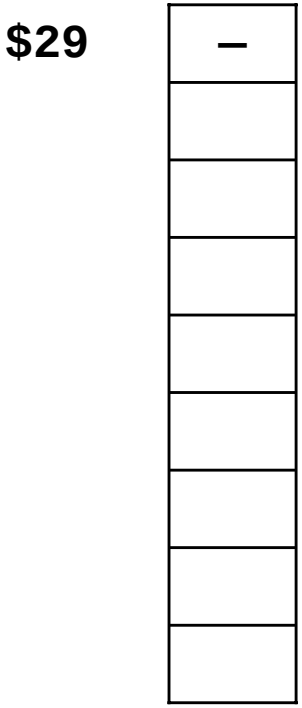
```
main:
      sub $29,$29,32
      sw $31,28($29)
      sw $30,24($29)
      addi $30,$29,$0
      jal __main
      sw $0,16($30)
$L2:
      lw $2,16($30)
      slt $2,$2,10
      bne $2,$0,$L4
      j $L3
$L4:
      lw $2,16($30)
      addu $2,$2,1
      sw $2,16($30)
      j $L2
$L3:
      addi $29,$30,$0
      lw $31,28($29)
      lw $30,24($29)
      addu $29,$29,32
      j $31
```

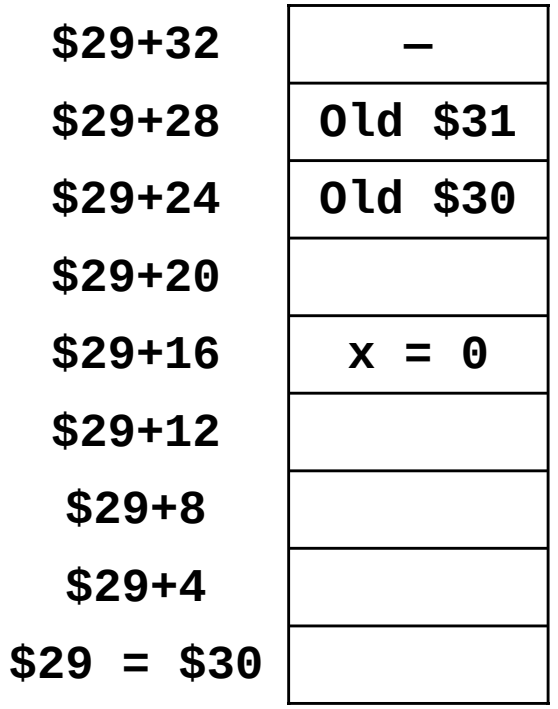
MIPS Compilation 1-2

Building data frame

```
sub $29,$29,32          ; $29 used as stack pointer
sw $31,28($29)          ; save $31 = old return pointer
sw $30,24($29)          ; save $30 = old frame pointer
addi $30,$29,$0         ; set $30 = $29
jal __main              ; OS function call to set up C
                        ; saves return pointer in $31

sw $0,16($30)           ; store x = 0 in data frame
```





MIPS Compilation 1-3

Executing loop

```
$L2:
     lw $2,16($30)          ; $2 ← x
     slt $2,$2,10           ; if ($2 < 10) $2 ← 1, else $2 ← 0
     bne $2,$0,$L4          ; if ($2 != 1) jump to $L4
     j $L3                  ; jump to $L3 (exit loop)
$L4:
     lw $2,16($30)          ; $2 ← x
     addu $2,$2,1           ; $2 ← $2 + 1
     sw $2,16($30)          ; x ← $2
     j $L2                  ; jump to $L2 (continue loop)
$L3:
     addi $29,$30,$0        ; restore $29
     lw $31,28($29)         ; restore $31 (old return pointer)
     lw $30,24($29)         ; restore $30 (old frame pointer)
     addu $29,$29,32        ; restore original $29
     j $31                  ; return to calling point
```

MIPS Compilation 2-1

C source

```
main()
{
  int x , y;
  for (x = 0 ; x < 10 ; x++){
    y = x + 4;
  }
}
```

Assembly output of compiler

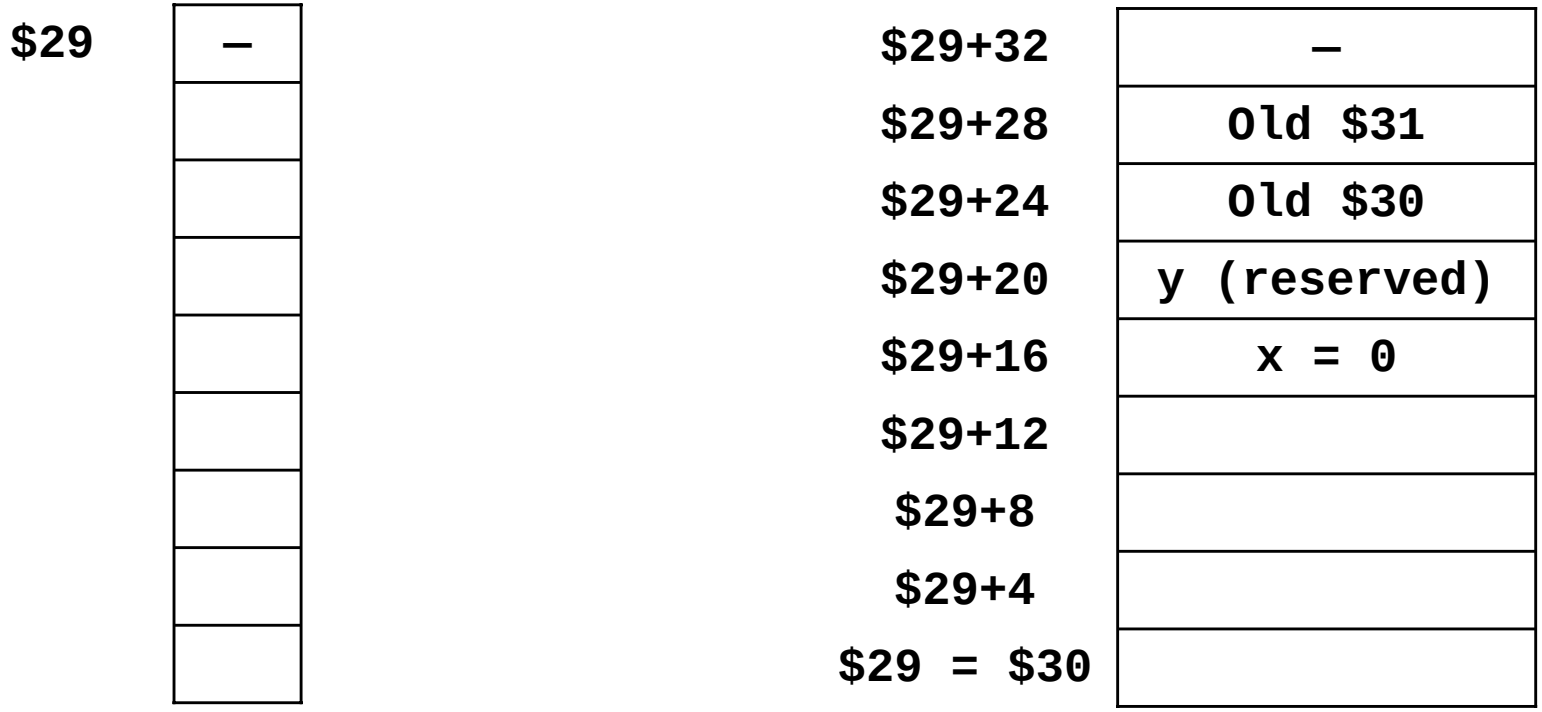
```
main:
        sub $29,$29,32
        sw $31,28($29)
        sw $30,24($29)
        addi $30,$29,$0
        jal__main
        sw $0,16($30)
$L2:
        lw $2,16($30)
        slt $2,$2,10
        bne $2,$0,$L5
        j $L3
$L4:
        lw $2,16($30)
        addu $2,$2,4
        sw $2,20($30)
        lw $2,16($30)
        add $2,$2,1
        sw $2,16($30)
        j $L2
$L3:
        addi $29,$30,$0
        lw $31,28($29)
        lw $30,24($29)
        add $29,$29,32
        j $31
```

MIPS Compilation 2-2

Building data frame

```
sub $29,$29,32          ; $29 used as stack pointer
sw $31,28($29)          ; save $31 = old return pointer
sw $30,24($29)          ; save $30 = old frame pointer
addi $30,$29,$0         ; set $30 = $29
jal __main              ; OS function call to set up C
                        ; saves return pointer in $31

sw $0,16($30)           ; store x = 0 in data frame
```



MIPS Compilation 3-3

Executing loop

```
$L2:
      lw $2,16($30)              ; $2 ← x
      slt $2,$2,10               ; if ($2 < 10) $2 ← 1, else $2 ← 0
      bne $2,$0,$L4              ; if ($2 != 1) jump to $L4
      j $L3                      ; jump to $L3 (exit loop)
$L4:
      lw $2,16($30)              ; $2 ← x
      addu $2,$2,4               ; $2 ← $2 + 4
      sw $2,20($30)              ; y ← $2
      lw $2,16($30)              ; $2 ← x
      add $2,$2,1                ; $2 ← $2 + 1
      sw $2,16($30)              ; x ← $2
      j $L2                      ; jump to $L2 (continue loop)
$L3:
      addi $29,$30,$0            ; restore $29
      lw $31,28($29)             ; restore $31 (old return pointer)
      lw $30,24($29)             ; restore $30 (old frame pointer)
      addu $29,$29,32            ; restore original $29
      j $31                      ; return to calling point
```

ARM Overview

Microprocessor and microcontroller for embedded systems

Advanced **R**ISC **M**achine developed by ARM Limited

ARM Ltd primarily licenses ISA implementations to developers

Most widely used 32-bit RISC ISA

- Over 50 billion ARM processors used in phones, games, peripherals
- 98 percent of mobile phones use ARM



RISC Architectural Features

Data Types

- Byte 8 bits
- Halfword 16 bits (in ARMv4 and higher)
- Word 32 bits

Standard RISC

- Load/store architecture
- Large uniform register file
- Simple addressing modes
- Uniform and fixed-length instruction fields
- Scalar in-order pipeline

Additional ARM architectural features

- Shift + ALU operations
- Auto-increment / auto-decrement addressing modes for loops
- Load and Store Multiple instructions
- Conditional execution of most instructions
  - Cancel instructions on certain condition flags
  - Replaces control hazards in forward jumps

ARM Versions

Architecture Version	Processor Family
ARMv1	ARM1
ARMv2	ARM2
	ARM3
ARMv3	ARM6
	ARM7
ARMv4	ARM7TDMI
	StrongARM
	ARM8
	ARM9TDMI
ARMv5TE ARMv5TEJ	ARM9E
	ARM10E
	XScale
ARMv6	ARM11
ARMv7	Cortex
ARMv8	Cortex

Extension Features	
J	Jazelle (Java)
T	Thumb
D	Debugger
M	Multiplier (64-bit result)
I	ICE
E	DSP Enhancement (implies TDMI)

Seven Operating Modes

User	Normal (non-privileged) program execution mode No access to protected resources
FIQ	Supports high speed data transfer or DMA process Entered on high priority (fast) interrupt
IRQ	General purpose interrupt handling Entered on low priority (normal) interrupt
Supervisor	Protected mode for operating system Entered on reset and on Software Interrupt
Abort	Implements virtual memory and/or memory protection Handles memory access violations
Undef	Supports software emulation of hardware coprocessors
System	Runs privileged operating system tasks (ARMv4 and above) Accesses user mode registers

Registers

32-bit general purpose registers

- 16 architectural registers
  - r0, ... , r15
  - r11 — FP = Frame Pointer
  - r12 — IP = intra-procedure-call scratch register
  - r13 — SP = Stack Pointer used by push/pop instructions
  - r14 — LR = Link Register used to return from function calls
  - r15 — PC = Program Counter
- 31 physical registers
  - Multiple copies of r8, ... , r14
  - Each copy accessible in specific operating mode

32-bit status registers

- Current Program Status Register (CPSR) visible in all modes
  - 32 bits wide
- 5 Saved Program Status Registers (SPSR)
  - Privileged modes copy previous CPSR

Modes and Visible Registers

User	FIQ	IRQ	Supervisor	Abort	Undef
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8	r8	r8	r8	r8
r9	r9	r9	r9	r9	r9
r10	r10	r10	r10	r10	r10
r11 (FP)	r11 (FP)	r11 (FP)	r11 (FP)	r11 (FP)	r11 (FP)
r12 (IP)	r12 (IP)	r12 (IP)	r12 (IP)	r12 (IP)	r12 (IP)
r13 (SP)	r13 (SP)	r13 (SP)	r13 (SP)	r13 (SP)	r13 (SP)
r14 (LR)	r14 (LR)	r14 (LR)	r14 (LR)	r14 (LR)	r14 (LR)
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR	SPSR	SPSR	SPSR	SPSR

Instruction Sets

ARM

- 32-bit instructions
- Aligned on 32-bit boundaries
  - Lowest 2 bits of PC (r15) always 0

Thumb

- 16-bit instructions 1-to-1 mapped to 32-bit instructions
- Shortened versions with restricted options and implicit operands
  - Example — **`add dest, src1, src2`** becomes **`add dest, src`**
- Aligned on 16-bit boundaries
  - Lowest bit of PC (r15) always 0
- Set by T = 1 in CPSR

Jazelle

- Executes Java bytecode directly
- ARM reads 4 8-bit instructions per instruction fetch
- Set by J = 1 in CPSR

Conditional Execution

Flag set suffix S

ALU instructions set CPSR flags

N — Negative Z — Zero

C — Carry V — oVerflow

Conditional execution suffix

Execute instructions if flag combination is true

Example — operate-compare-if-else

Usual Execution	Conditional Execution
`SUB r3, r3, #1`	`SUBS r3, r3, #1`
`CMP r3, #0`	
`BEQ L1 ; branch on 0`	
`ADD r0, r1, r2 ; skip if r3 = 0`	`ADDNE r0, r1, r2`
`B L2 ; jump to L2`	
`L1: SUB r0, r1, r2 ; skip if r3 != 0`	`SUBEQ r0, r1, r2`
`L2: ADD r4, r5, r6`	`ADD r4, r5, r6`

Basic Instructions

Transfer

MOV — Move register-to-register or immediate-to-register

MVN — Move Not

LDR, STR — load / store

Branch

B — Add sign-extended 24-bit signed immediate to PC (r15)

BL — Branch and store PC+4 in link register (r14)

Conditional branch — conditional execution of **B** or **BL**

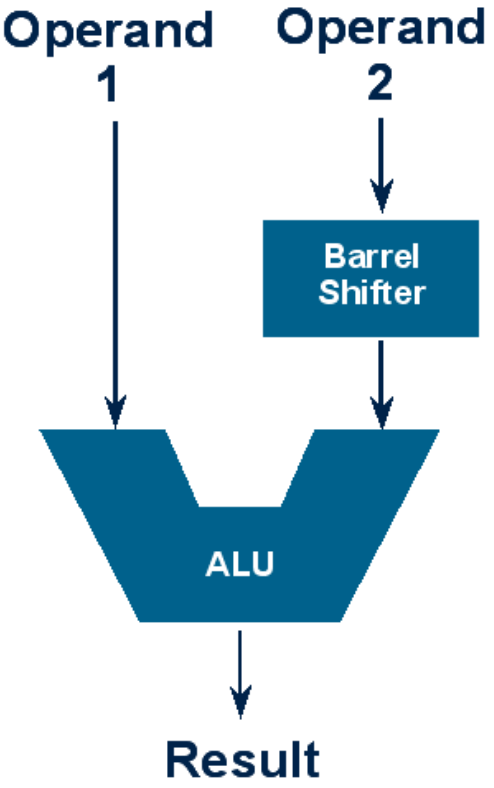
ALU

ADD	Add	**MLA**	Multiply and ADD
ADC	Add with Carry	**ORR**	Logical OR
AND	Logical AND	**SUB**	Subtract
BIC	Logical Bit Clear	**SBC**	Subtract with Carry
CMP	Compare	**RSB**	Reverse Subtract
CMN	Compare Negative	**RSC**	Reverse Subtract with Carry
EOR	Logical XOR	**TST**	Test
MUL	Multiply	**TEQ**	Test Equivalence

Shift and Rotate

Legal ALU instruction operands

- 32-bit source / destination register contents
- Sign-extended 12-bit immediate
- Shifted-operand
  - Shifted / rotated 32-bit source register contents
  - Number of shifts set by 8-bit immediate



Shifts

- LSL — Logical Left Shift (unsigned)
- ASR — Arithmetic Right Shift
- LSR — Logical Shift Right

Rotates

- ROR — Rotate Right
- RRX — Rotate Right Extended (CF into MSB)

VFP Extension

No FP operations in basic instruction set

- Not needed in simple embedded applications
- VFP implements FP ISA extension as optional coprocessor
- Since ARM10

Vector Floating Point (VFP)

- Single precision and double precision floating point computation
  - ANSI/IEEE Std 754 compliant
- Single Instruction Multiple Data (SIMD) FP unit
  - One FP operation performed in parallel on 256-bit vector
    - 8 single-precision (4-byte) FP numbers
    - 4 double-precision (8-byte) FP numbers
  - Accesses 32 single precision FP registers (32-bit width)

VFPv3

- Operates on 8 double-precision (8-byte) FP numbers (512-bit vector)

VFP Instructions

Transfer

- Load / store FP values into registers from memory
- Transfer / copy 32-bit values between VFP and ARM GP registers
- Conversions between float, double, unsigned / signed integers

FPU

- Add
- Subtract
- Multiply
- Divide
- Square root
- Combined multiply-accumulate
- Compare FP values in registers

VFPv3

- Store FP constant in register

DSP Enhancements

Digital Signal Processing

- Operations on sampled-digitized-encoded analog information
- Typical applications
  - Audio / video, speech processing, modems, medical instruments
- Typical algorithms
  - D / A, normalization, correlation, convolution, FFT, encoding / decoding
  - Real time control

Practical applications

- GSM-AMR (Adaptive Multi-Rate) speech codec in 3G GSM phones
- Servo motor control (HDD/DVD)
- Audio encode/decode (MP3, AAC, WMA)
- MPEG4 decode
- Voice and handwriting recognition

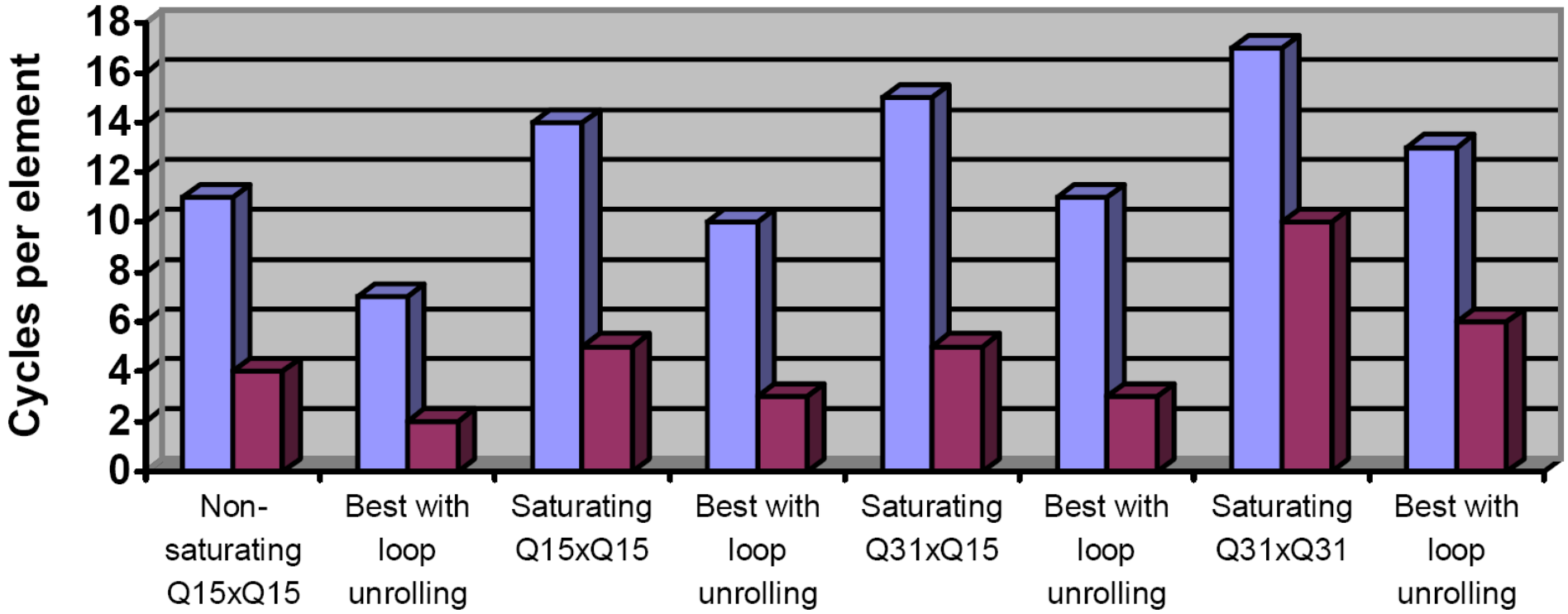
DSP Instructions

Instruction	Operation	Purpose
SMLAxy{cond}	**16 x 16 + 32 → 32**	Signed MAC
SMLAWy{cond}	**32 x 16 + 32 → 32**	Signed MAC wide
SMLALxy{cond}	**16 x 16 + 64 → 64**	Signed MAC long
SMULxy{cond}	**16 x 16 → 32**	Signed multiply
SMULWy{cond}	**16 x 32 → 32**	Signed multiply long
QADD Rd, Rm, Rs	**SAT(Rm + Rd)**	Saturating add
QDADD Rd, Rm, Rs	**SAT(Rm + SAT(Rs x 2))**	Saturating add double
QSUB Rd, Rm, Rs	**SAT(Rm – Rd)**	Saturating subtract
QDSUB Rd, Rm, Rs	**SAT(Rm – SAT(Rs x 2))**	Saturating subtract double
CLZ{cond} Rd, Rm	**COUNTZ(Rm)**	Count leading zeros

16 — halfword
32 — word
64 — doubleword

MAC	Multiply – Accumulate (**Rd ← R1 x R2 + R3**)
Saturating	Pin overflow result at max or min No modulo arithmetic or report of overflow

Performance Comparisons



ARM9 before DSP enhancements

ARM10 with DSP enhancements

Q15 / Q31 — integer arithmetic techniques used in DSP

Apple iPhone 5 Hardware

Apple A6 Application Processor

- Dual ARMv7 cores + 3 GPU
- Hand-optimized layout

Memory

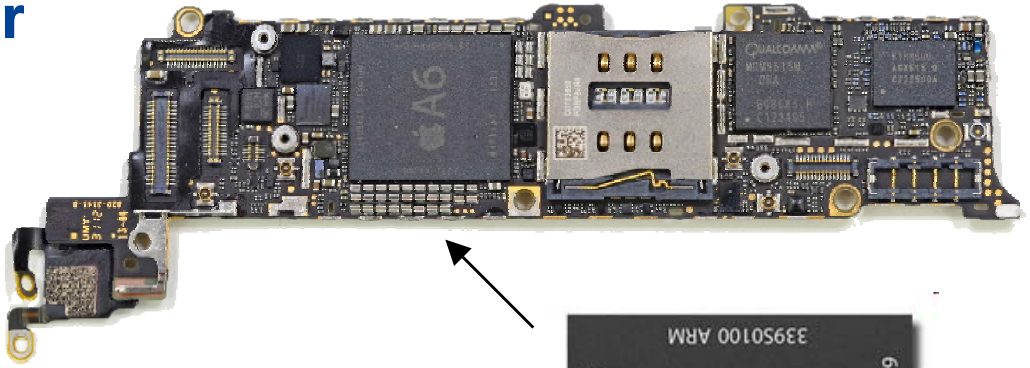
- Hynix 16 GB Flash

Network processors

- Skyworks GSM / GPRS / EDGE module
- Skyworks CDMA module
- Triquint WCDMA / HSUPA / UMTS
- Qualcomm LTE processor
- Murata WiFi module

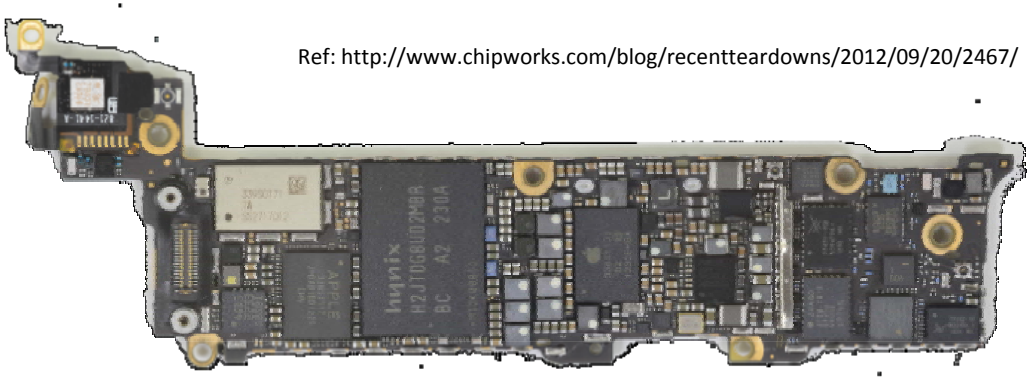
Interface controllers

- Apple Power Management IC
- Apple Audio CODEC
- Texas Instruments touch screen controller
- STMicroelectronics 3-axis gyro
- STMicroelectronics 3-axis linear accelerometer





Ref: http://www.chipworks.com/blog/recentteardowns/2012/09/20/2467/



WinARM

Cross-compiler

Develop applications for ARM in C / C++ on Windows platforms

Extensive documentation

Tools

GNU GCC compiler

GNU-Utils for compiler/linker

ARM header-files

Sample applications with source-code

Download

http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects

Convert C code to assembly code

```
arm-elf-gcc -S filename.c -o filename.asm
```

ARM Compilation 1-1

C source

```
main()
{
     int x = 0;
     while (x < 10){
        x++;
     }
}
```

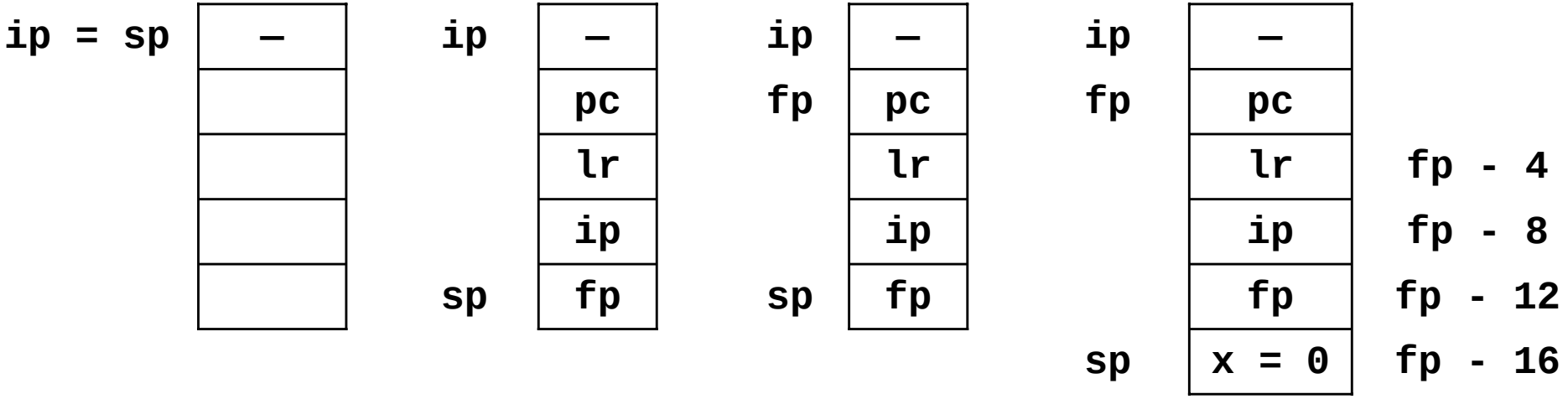
Assembly output of compiler

```
main:
     mov        ip, sp
     stmfd      sp!, {fp, ip, lr, pc}
     sub        fp, ip, #4
     sub        sp, sp, #4
     mov        r3, #0
     str        r3, [fp, #-16]
     b  .L2
.L3:
     ldr        r3, [fp, #-16]
     add        r3, r3, #1
     str        r3, [fp, #-16]
.L2:
     ldr        r3, [fp, #-16]
     cmp        r3, #9
     ble        .L3
     ldmfd      sp, {r3, fp, sp, pc}
```

ARM Compilation 1-2

Building data frame

```
mov       ip, sp
stmfd     sp!, {fp, ip, lr, pc}
   ; push fp, ip, lr, pc to stack
sub       fp, ip, #4
   ; fp = ip – 4
   ; sp = ip – 16 = fp - 12
sub       sp, sp, #4
   ; sp = fp – 16
mov       r3, #0
str       r3, [fp, #-16]
```



ARM Compilation 1-3

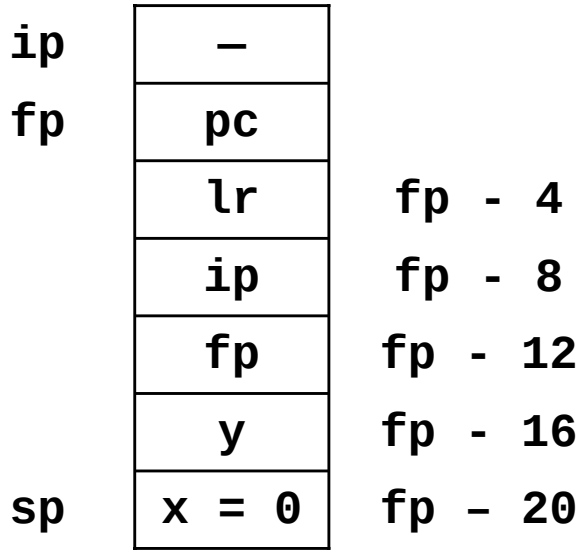
Executing loop

```
        b       .L2                     ; branch to .L2
.L3:
        ldr     r3, [fp, #-16]          ; r3 ← x
        add     r3, r3, #1              ; r3++
        str     r3, [fp, #-16]          ; x ← r3
.L2:
        ldr     r3, [fp, #-16]          ; r3 ← x
        cmp     r3, #9                  ; compare r3 , 9
        ble     .L3                     ; jump .L3 if r3 ≤ 9
        ldmfd   sp, {r3, fp, sp, pc}    ; restore registers
```

ARM Compilation 2-1

C source

```
main()
{
  int x , y;
  for (x = 0 ; x < 10 ; x++){
    y = x + 4;
  }
}
```



Assembly output of compiler

```
main:
        mov     ip, sp
        stmfd   sp!, {fp, ip, lr, pc}
        sub     fp, ip, #4
        sub     sp, sp, #8 ; 2 integers
        mov     r3, #0
        str     r3, [fp, #-20]
        b  .L2
.L3:
        ldr     r3, [fp, #-20]
        add     r3, r3, #4
        str     r3, [fp, #-16]
        ldr     r3, [fp, #-20]
        add     r3, r3, #1
        str     r3, [fp, #-20]
.L2:
        ldr     r3, [fp, #-20]
        cmp     r3, #9
        ble     .L3
        sub     sp, fp, #12
        ldmfd   sp, {fp, sp, pc}
```

ARM Compilation 2-2

Executing loop

```
     b  .L2
.L3:
     ldr    r3, [fp, #-20]          ; r3 ← x
     add    r3, r3, #4              ; r3 ← r3 + 4
     str    r3, [fp, #-16]          ; y ← r3
     ldr    r3, [fp, #-20]          ; r3 ← x
     add    r3, r3, #1              ; r3++
     str    r3, [fp, #-20]          ; x ← r3
.L2:
     ldr    r3, [fp, #-20]          ; r3 ← x
     cmp    r3, #9                  ; compare r3 , 9
     ble    .L3                     ; jump .L3 if r3 ≤ 9
     sub    sp, fp, #12             ; sp ← fp – 12
     ldmfd  sp, {fp, sp, pc}        ; restore registers
```