# Review of Basic Computer Architecture

החומר בפרק הזה חוזר בקצרה על תוכן הקורס ארכיטקטורה לתואר ראשון במכללת הדסה.
במידה שהחומר הזה לא מוכר מומלץ לפנות לאתר https://cs.hac.ac.il/staff/martin/Architecture

## From Wikipedia, the free encyclopedia

In computer science and engineering, computer architecture refers to specification of the relationship between different hardware components of a computer system. It may also refer to the practical art of defining the structure and relationship of the subcomponents of a computer.



This article needs attention from an expert in computer science.

## Definition 2.0

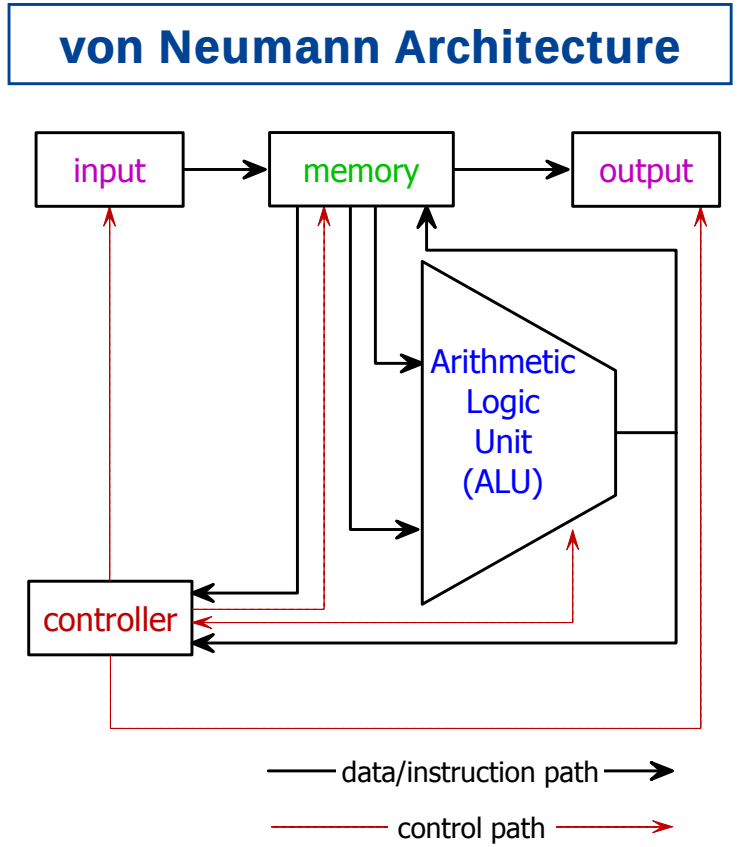
In computer science and engineering, computer architecture refers to the study of performance in computer systems. It also refers to the practical science of applying performance theory to specifying the structure and relationship of the subcomponents of a computer.

From an expert in computer science.

# Theory — Goals — Specification

## Theory

- Computation ⇒ components
  - CPU: ALU + memory + control
  - Instructions
- Performance = run-time speed
  - Run-time of what?
  - Compared to what?




## Requirements

- Word processing
- Number crunching
- Gaming
- Web server
- Real time

## Specification

Requirements + performance theory → component implementation

# Requirements — Relative to Application

## Fastest CPU — Xeon E-2278G

High-end version of Intel x86-64 processor family

IA-32 instruction set on enhanced P6 micro-architecture

Netburst → Ivy Bridge → Haswell → Broadwell → Skylake → … → Golden Cove

Pentium II → Pentium III → Pentium 4 → Multicore

## Smartphones — ARM CPU

Low power

Higher performance / Watt than x86

## Fastest supercomputer — FRONTIER - HPE CRAY EX235A

Node = CPU + 4 GPUs + 5 TB flash + 512 GB RAM

CPU = AMD EPYC 64 core 2GHz

64 cores per CPU + 220 cores per GPU

9,472 nodes = 8,941,568 cores

Total RAM = 4.85 PiB = 4736 TB

# Fundamental Architectural Abstractions

## Digital computer

Machine that can be programmed to process symbols

## Data

Symbol with no intrinsic meaning to machine

User imposes meaning — Integer, float, string, ...

## Operation

Symbol describing processing of data symbols

Machine interprets meaning — transfer, ALU, control, OS, ...

## Instruction

Symbol describing operation on data

Machine language = collection of legal instructions

## Addressing Mode

Specifies data location as operand

Source operand — data input to operation

Destination operand — data output from operation

# Stages in Computer Design

## Instruction Set Architecture (ISA)

1. Define universe of problems to be solved
2. Study candidate operations at level of system programmer
   - Atomic — operations complete sequentially
   - General operation = combination of atomic operations
3. Specify instruction set for machine language
   - Choose minimum set of orthogonal 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

# 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**

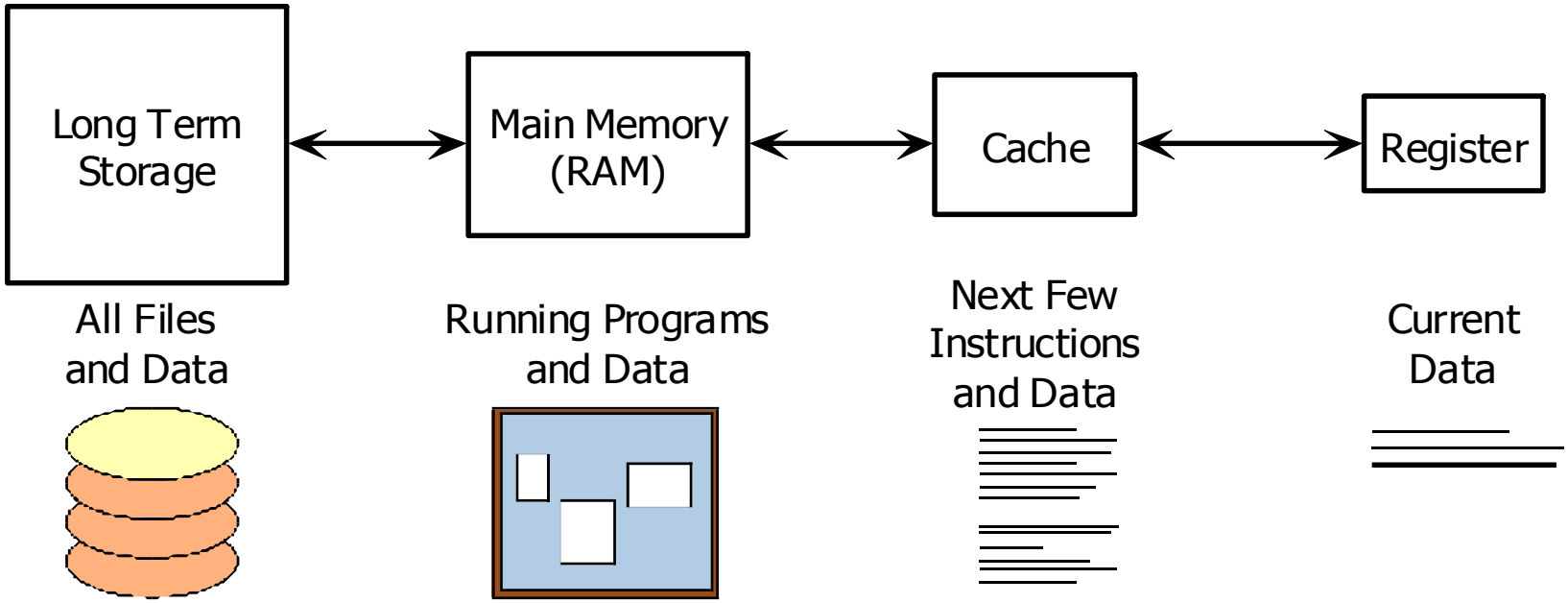
System calls, virtual memory management instructions

**Graphics**

Pixel operations, compression/decompression operations

# 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 |



# CPU and Memory Hierarchy

## CPU controller accesses L1 cache

if (L1 cache hit) {access performed in 1 clock cycle}

else {

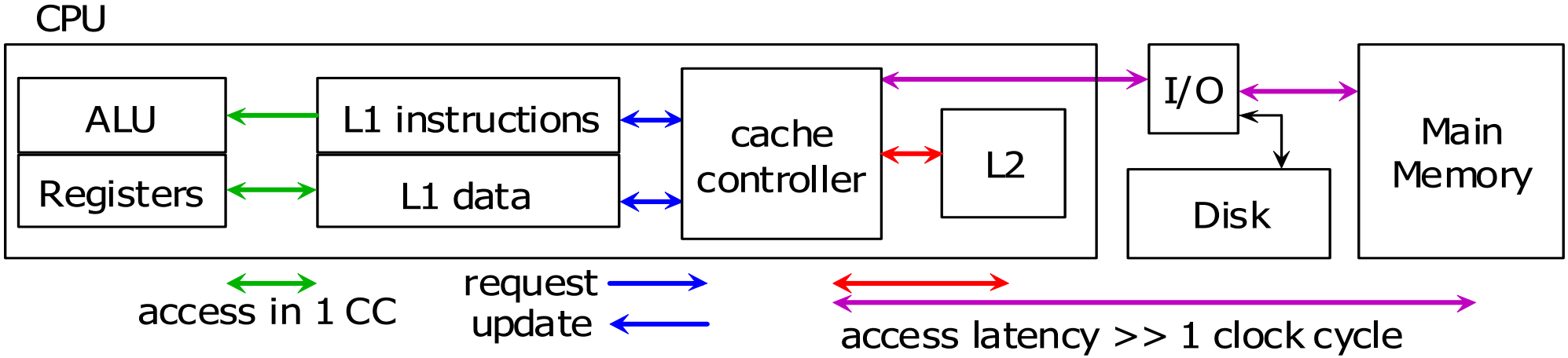
L1 cache miss — L1 cache accesses cache controller

cache controller initiates access to L2 and main memory

if (address in L2 cache) {controller copies contents to L1 from L2}

else {controller copies location to L1 from main memory}

}




## Cache miss penalty

Address not in L1 ⇒ delay in memory access

# 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**

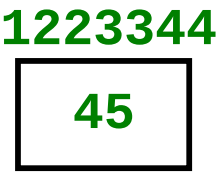



## Memory operands

**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**

# 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]]<br>Regs[R2] ← Regs[R2]+d | Stack access |
| Auto Decrement | -(R2) | Regs[R2] ← Regs[R2]-d<br>Mem[Regs[R2]] | Stack access |
| Scaled | 100(R2)[R3] | Mem[100+Regs[R2]+Regs[R3]*d] | Indexing arrays |
| PC-relative | (PC) | Mem[PC+value] | Load instruction to data register |
| PC-relative deferred | 1001(PC) | Mem[PC+Mem[1001]] | Load instruction to data register |

# Commitment to State

## Internal registers

Temporary registers used in executing machine instructions

Not visible to programs

## Architectural state

CPU registers visible to programs

## System state

All data resources visible to programs

Architectural state + system memory

## Commitment to state

Update of system state

Write to architectural state / system memory

# Complex Instruction Set Computer (CISC)

## Classic Machine Design

300 instruction types
15 addressing modes
10 data types
Complex machine implementations

## Mainframes (1955 — 2000)

Large, expensive, centralized computers for big business and government
Manufacturers: IBM, Control Data, Burrows, Honeywell

## Minicomputers (1965 — 1990)

Smaller computers for smaller organizations
Manufacturers: Digital (PDP/**VAX**), Data General (Eclipse)

## CISC microprocessors (1979 — 1996)

6800 (1974) and 8086 (1978) designed as tiny CISC on chip
- Apple II (1977) — 6502 (1975)
- IBM PC (1981) — 8088 (1979)

Intel x86 for PC/Mac = last CISC ISA still manufactured.

# Why CISC?

## Semantic Gap Argument

Computer language should imitate natural language

Large vocabulary + high redundancy → flexibility + power

## Terrible compilers

Limited optimization

Limited error messaging

Efficient code written or optimized in assembly language

## Expensive memory

RAM ~ $5000/MB wholesale in 1977

RAM < $0.01/MB since 2015

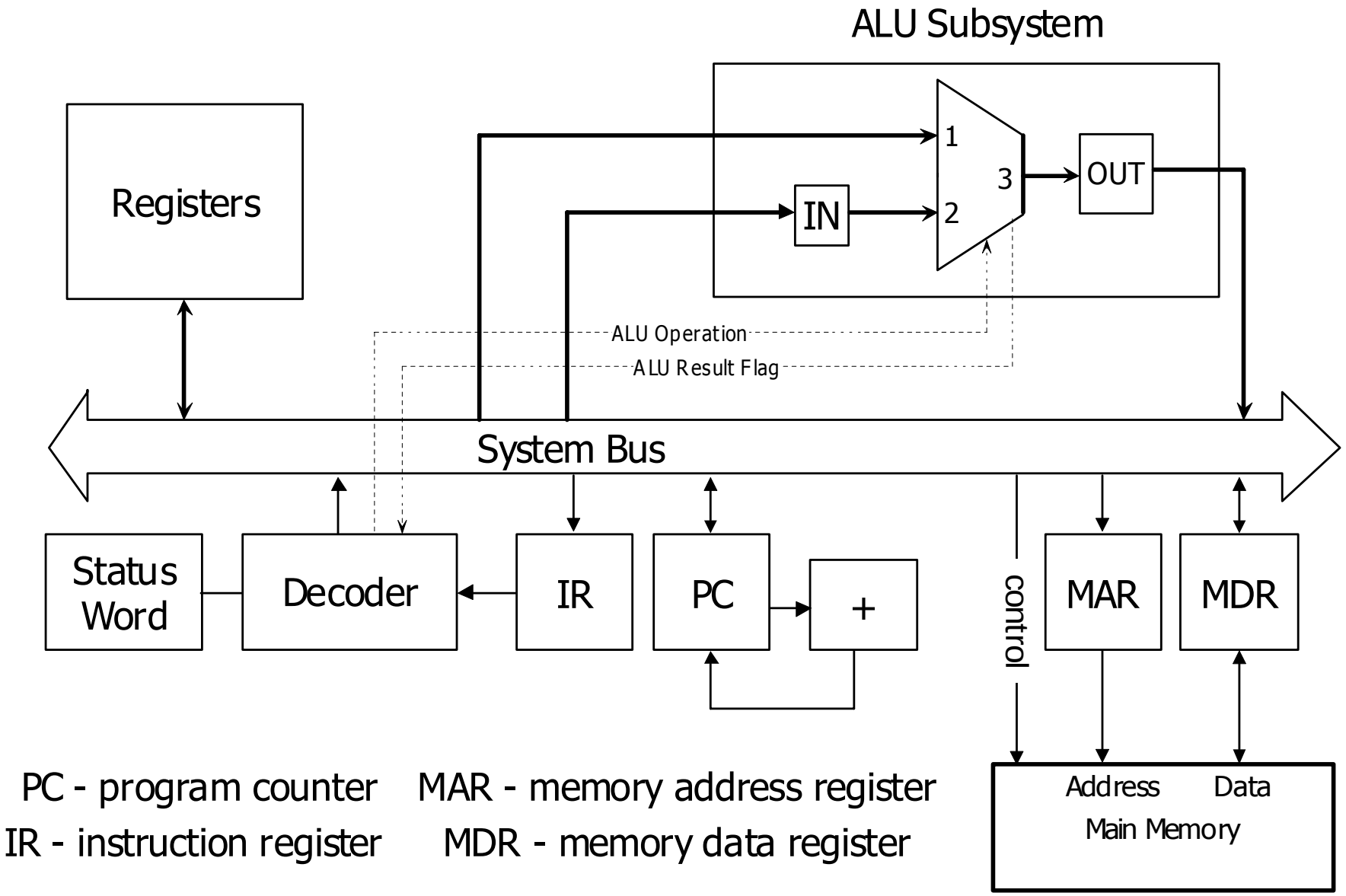
## Implications for machine language

Design for user-friendly programming and small memory use

Many highly specific instructions using many addressing modes

Compact instruction codes that perform a lot of work

# Physical Implementation of CISC — Generic Machine




PC - program counter
IR - instruction register
MAR - memory address register
MDR - memory data register

# Decoding Machine Instructions

## Machine Language Instruction

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

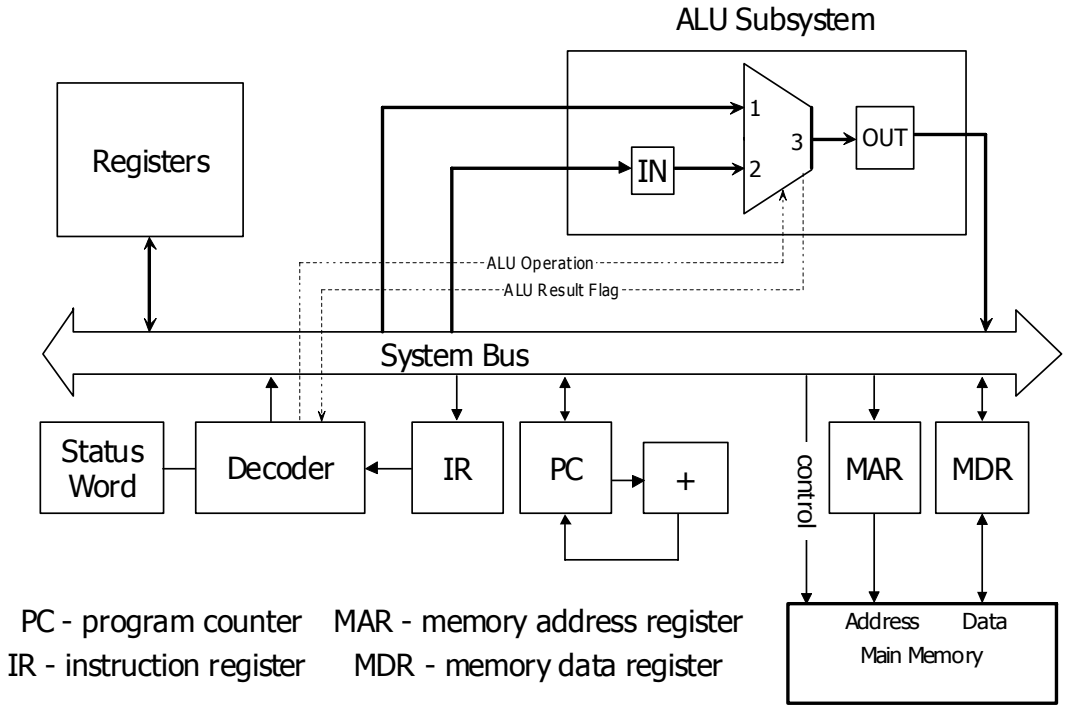
## Microcode Instruction Sequence (Microprogram)

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

### Microcode instruction

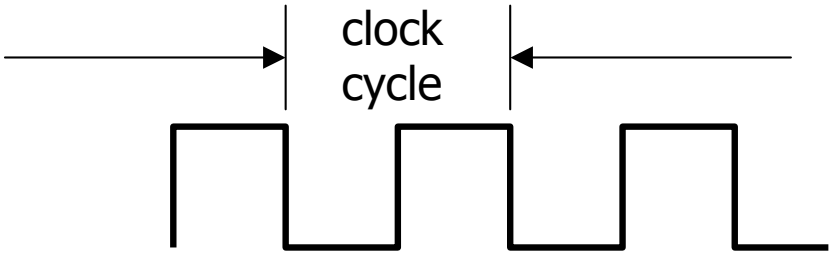
Hardware level atomic operation

9 lines = 9 clock cycles



# Run Time and Clock Cycles

## CPU is timed by periodic signal called clock (CLK)




## Clock Cycle (CC) time = seconds per cycle

Instruction requires 1 or more clock cycles to process

Clock Rate = cycles per second = Hz (Hertz)

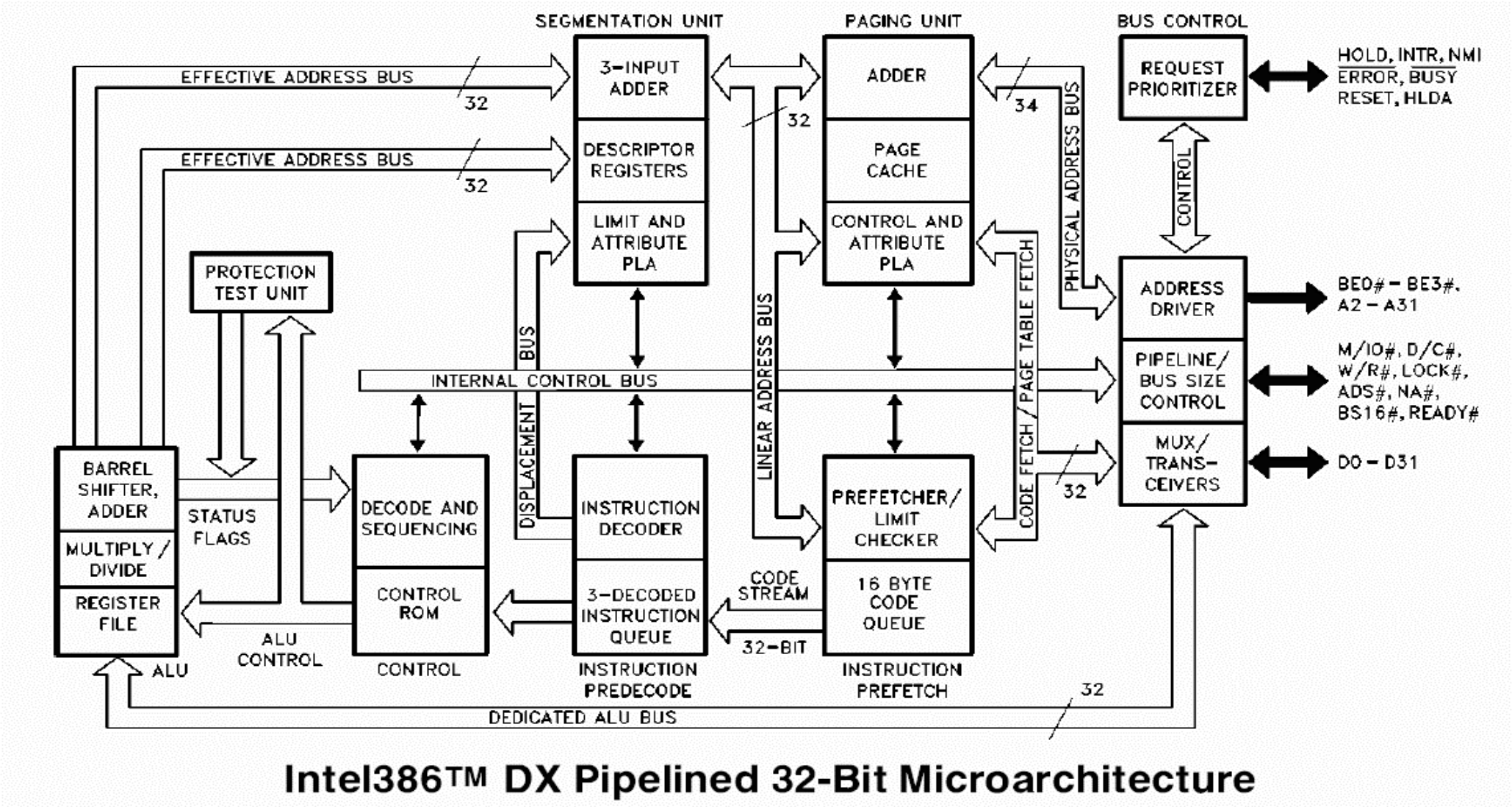
$$\textbf{Run time} = \text{clock cycles to run program} \times \text{seconds per clock cycles}$$

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

Higher clock rate $\Rightarrow$ shorter run time

More clock cycles (at constant clock rate) $\Rightarrow$ longer run time

# Intel 386 Microprocessor — Last Pure CISC Intel CPU




**Intel386™ DX Pipelined 32-Bit Microarchitecture**

# Basic Performance Measures

## Run Time

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

## Latency

Excess response time — depends on context

## Throughput

Number of defined tasks performed per unit time

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

## Enhancement

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

## Speedup

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

# 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 $i$

$\tau$ = 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 $i$

# 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

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

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

# Amdahl Equation

$$F_i = \frac{t_i}{T} = \textbf{relative run time of instructions in group } i$$

$$S_i = \frac{t_i}{t_i'} = \textbf{speedup for instructions in group } i$$

$$S = \frac{T}{T'} = \frac{\sum_i t_i}{\sum_i t_i'} = \frac{\sum_i t_i}{\sum_i \frac{t_i}{S_i}} = \frac{\sum_i F_i T}{\sum_i \frac{F_i}{S_i} T} = \frac{\sum_i F_i}{\sum_i \frac{F_i}{S_i}} = \frac{1}{\sum_i \frac{F_i}{S_i}}$$

**Enhancement to group** $e$

$$S = \frac{1}{\frac{1-F_e}{S_{\bar{e}}} + \frac{F_e}{S_e}} = \frac{1}{1-F_e+\frac{F_e}{S_e}}$$

**Amdahl's "Law"**

Speedup limited by $1 - F_e$

Enhance maximum $F_e$

Accept impairment to small $1 - F_e$

# Amdahl 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

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

# 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 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

# Benchmarking a Processor Design

## Specify Instruction Set Architecture (ISA)

- Specifies machine language for proposed CPU
- Provides human-readable assembly language
- Determines $CPI_i$ for each instruction group $i$
  - Count clock cycles required to implement each instruction in ISA

## Write compilers for proposed machine language

- C, C++, Fortran

## Compile benchmark programs to machine language

- Programs from SPEC CINT and CFP

## Analyze compiler output (executable programs)

- Sort machine instructions into groups
- Calculate relative instruction count $IC_i/IC$ for each group
- Calculate average $CPI$ and overall run time $T$
- Compare run time with reference machine

# 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

# RISC "Philosophy"

## Technological developments from 1975 to 1990

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

Compilers become powerful and efficient with extensive optimization

Portable code made practical by minicomputer, Unix, C, and TCP/IP

## Principal research results on CISC performance

~ 90% of run time devoted to ~ 10% of instruction set

~ 90% of instructions in ISA rarely used

## Reduced Instruction Set Computer (RISC)

Apply Amdahl's "Law" CISC ISA

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

RISC ISA — only most important CISC instructions

- Other CISC instructions = multiple RISC instructions

RISC implementation executes its ISA in fast dedicated hardware

# 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

# RISC Microprocessors

## Simpler ISA

Small set of uniform length machine instructions

## Simpler hardware

No microcode — standard instruction implementation

No central system bus

CPU process several instructions at once

Lower $CPI_i$ + higher clock speed

Instruction completes on (almost) every clock cycle

## All processors today use RISC technology

Pure RISC (PowerPC, Sparc, MIPS, ARM, …)

RISC technology for CISC language (Pentium II – 4, Centrino, Core)

Explicitly parallel RISC (Intel Itanium, IBM mainframes)

# Typical RISC ISA

## Data types

32-bit / 64-bit integer and floating point

## Flat memory model with 32-bit / 64-bit address

Address mode: **`disp(Rn) ~ Mem[Regs[Rn] + disp]`**

## Register-register operation model

32 – 128 integer registers + 32 – 128 FP registers

OS (kernel mode) registers

Result flags

Read-only (value = 0) and write-only (null) registers

## Instruction types

Load, store, move register-register

Integer add, sub, mult, div, shift, compare

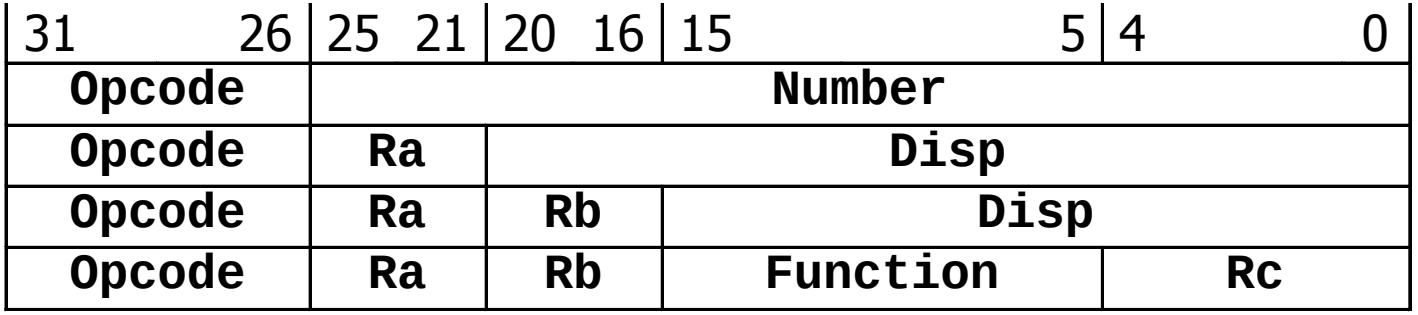
Boolean and, or, xor

Floating point add, sub, mult, div, sqrt, compare

Jump, jump register, jump and link, conditional branch

# Typical Instruction Encoding

## Instruction types for Alpha 64-bit RISC processor




**Opcode** (6 bits) identifies operation to CPU

**Ra, Rb** (5 bits) identify register names (**R0** to **R32**)

**PALcode** (Privileged Architecture Library) — hardware support for OS

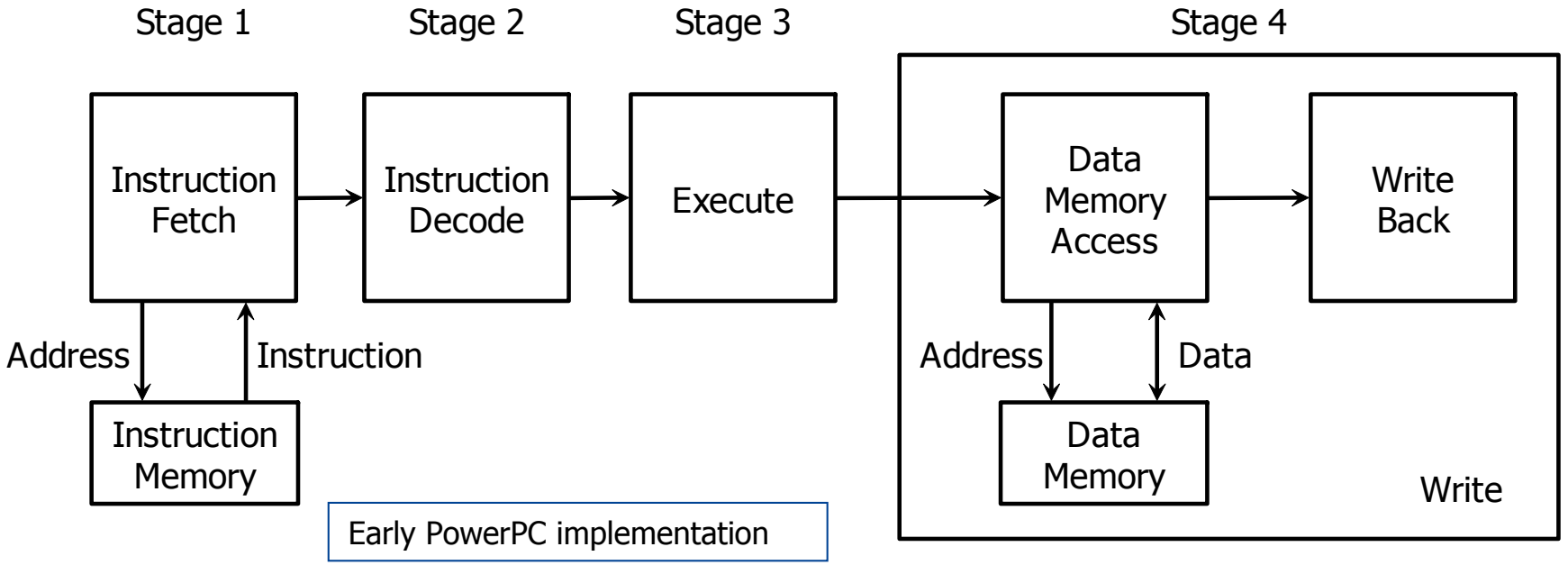
**Branch** — test **Ra**, true ⇒ **Ra** ← **PC**, **PC** ← **PC** + **Disp**

**Memory** — move between **Ra** and **Mem[Regs[Rb] + Disp]**

**Operate** R/R — **Rc** ← **Ra** function **Rb** (register name)

**Operate** R/I— **Rc** ← **Ra** function **Imm** (in **Rb** and 3 bits of function)

# Simple RISC Physical Implementation




**No system bus — instructions proceed from left to right (assembly line)**

**Separate cache memory for instructions and data**

**Simple repetitive operations**

1. Fetch uniform-length instructions
2. Instruction decode — read source operands from registers
3. Execute ALU instructions and calculate addresses
4. Access memory and/or write destination operands (commit to state)

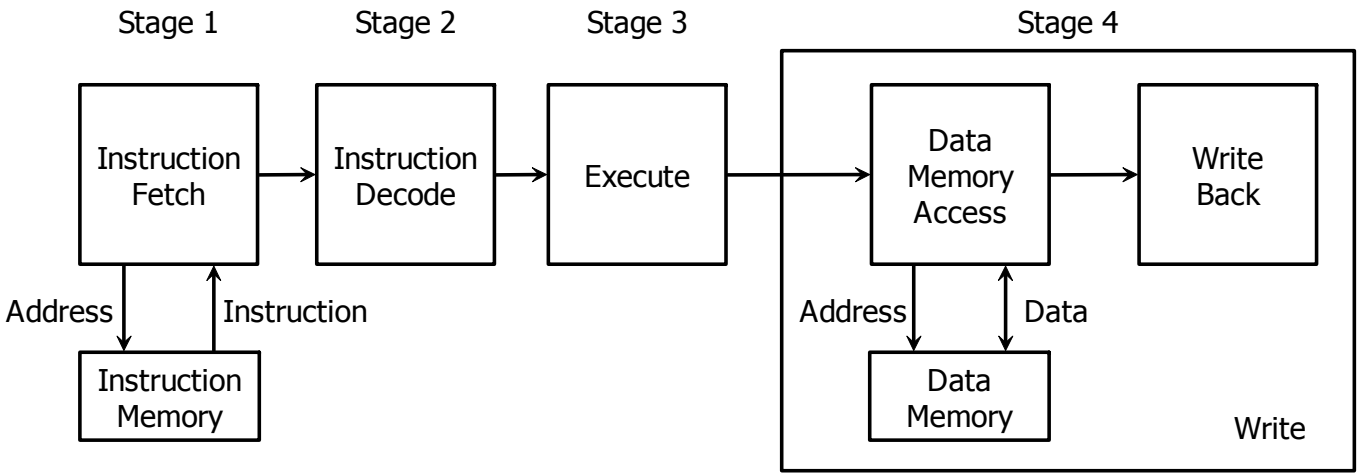
**One CC per stage per instruction ⇒ 4 clock cycles per instruction**

# Pipelining — The RISC Advantage

## Instruction Level Parallelism (ILP)

Hardware starts second instruction before first completes

Typically 4 instructions in various stages of execution at one time




| CC | Stage 1 | Stage 2 | Stage 3 | Stage 4 |
|---|---|---|---|---|
| 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 | $I_5$ | $I_4$ | $I_3$ | $I_2$ |
| 6 | $I_6$ | $I_5$ | $I_4$ | $I_3$ |

# Instruction-Oriented View

Clock Cycles →

Instructions ↓

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $I_1$ | IF | ID | EX | W | | |
| $I_2$ | | IF | ID | EX | W | |
| $I_3$ | | | IF | ID | EX | W |
| $I_4$ | | | | IF | ID | EX |
| $I_5$ | | | | | IF | ID |
| $I_6$ | | | | | | IF |

| | |
|---|---|
| IF | Instruction Fetch |
| ID | Instruction Decode |
| EX | Execute |
| W | Write |

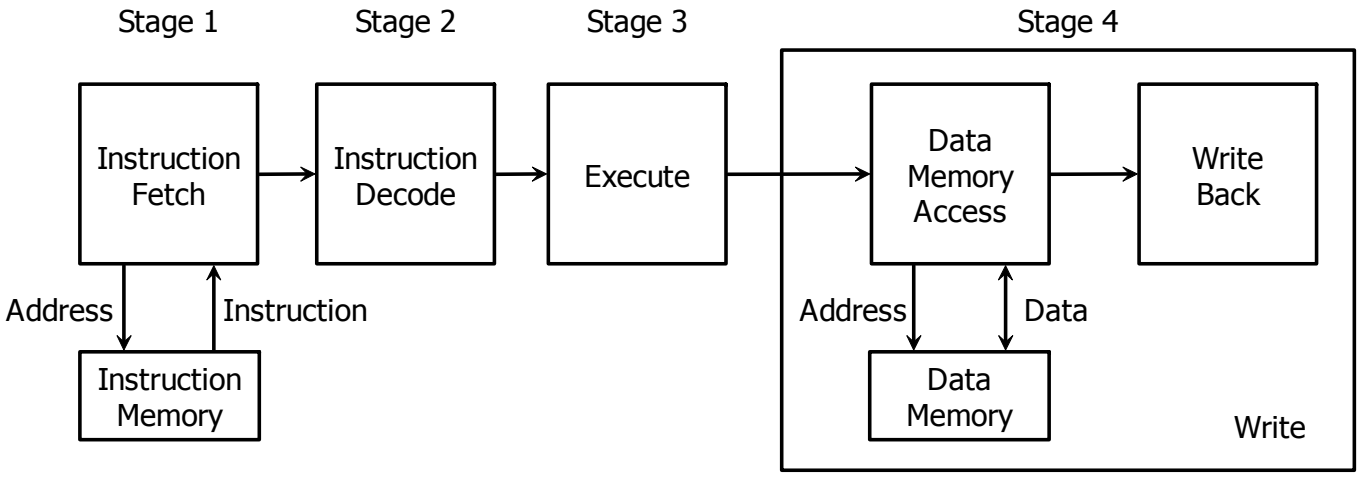
$$N^{ideal} = IC + (\text{pipeline length} - 1)$$

$$CPI^{\text{ideal}} = \frac{N^{ideal}}{IC}$$

$$= \frac{IC + (\text{pipeline length} - 1)}{IC} = 1 + \frac{\text{pipeline length} - 1}{IC} \xrightarrow[IC \text{ large}]{} 1$$

$$T = CPI \times IC \times \tau \xrightarrow[IC \text{ large}]{} IC \times \tau = \frac{\text{instructions}}{\text{clock rate}}$$

# Pipeline Imbalance




**Instruction executes in 4 clock cycles**

**Clock cycle time determined by LOAD instruction**

Longest execution time

$$\tau_{\text{fetch}} \approx \tau_{\text{decode}} \approx \tau_{\text{execute}} \approx \tau_{\text{memory access}} \approx \tau_{\text{register write-back}} = \tau_{\text{minimum}}$$

$$\tau_{\text{clock cycle}} > \tau_{\text{memory access}} + \tau_{\text{register write-back}} \approx 2 \times \tau_{\text{minimum}}$$

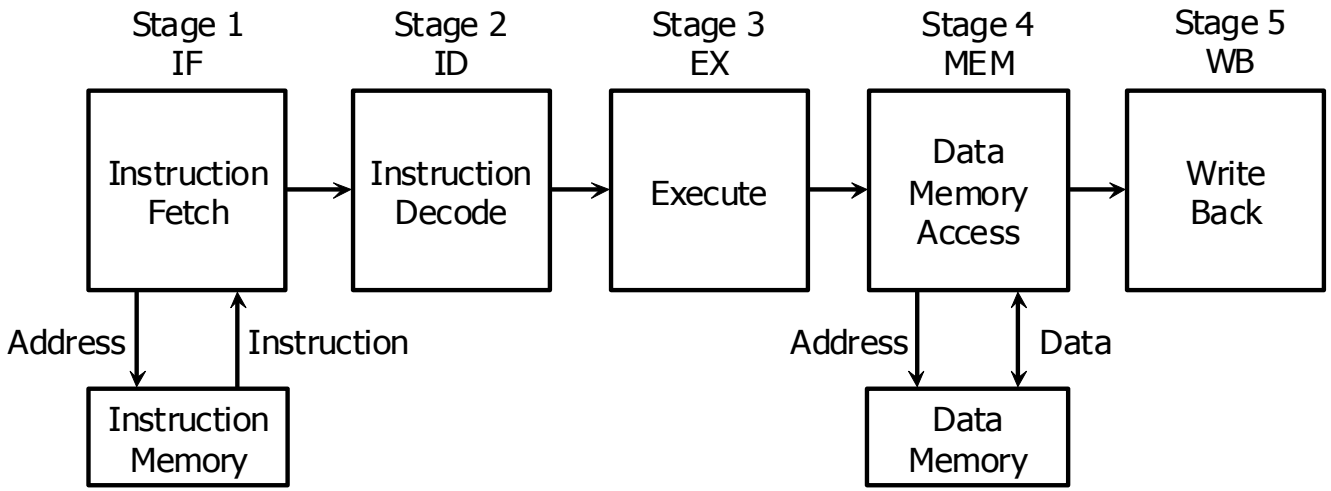
**Most instructions do not access data memory in stage 4**

Only LOAD and STORE access data memory

Only LOAD performs both memory access and register write-back

**Most operations can complete in time** $\tau_{\text{minimum}}$

# Superpipelining




## Divide stage 4 into two stages

Only load/store do useful work in MEM Stage

## Divide clock cycle time (double clock rate)

$$\tau' = \tau_{\text{minimum}} \approx \tau / 2$$

$$S = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{1 \times IC \times \tau}{1 \times IC \times \tau / 2} = 2$$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| $I_1$ | F | D | E | M | W | | |
| $I_2$ | | F | D | E | M | W | |
| $I_3$ | | | F | D | E | M | W |
| $I_4$ | | | | F | D | E | M |
| $I_5$ | | | | | F | D | E |

$$CPI^{\text{ideal}'} = CPI^{\text{ideal}} = 1$$

## Programs can run twice as fast

# Pipeline Hazards

## Instruction dependencies

Result of one instruction is source for later instruction

## Hazard condition

Processor runs uninterrupted but provides incorrect answers

## Pipeline hazard

Several instructions in various stages of execution

Pipeline uses a resource value before update by earlier instruction

Example

```
ADD  R1,R2,R3
SUB  R4,R5,R1     ;  hazard if SUB reads R1 before ADD writes R1
```

## Hazard Types

| | |
|---|---|
| Structural Hazard | conflict over access to resource |
| Data Hazard | instruction result not ready when needed |
| Control Hazard | branch address and condition 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 (wasted time) to instruction execution

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

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

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

## Eliminate cause of stall

Improve implementation based on analysis of stalls

Main activity of hardware architects

# Structural Hazards

**Conflict over access to resource**

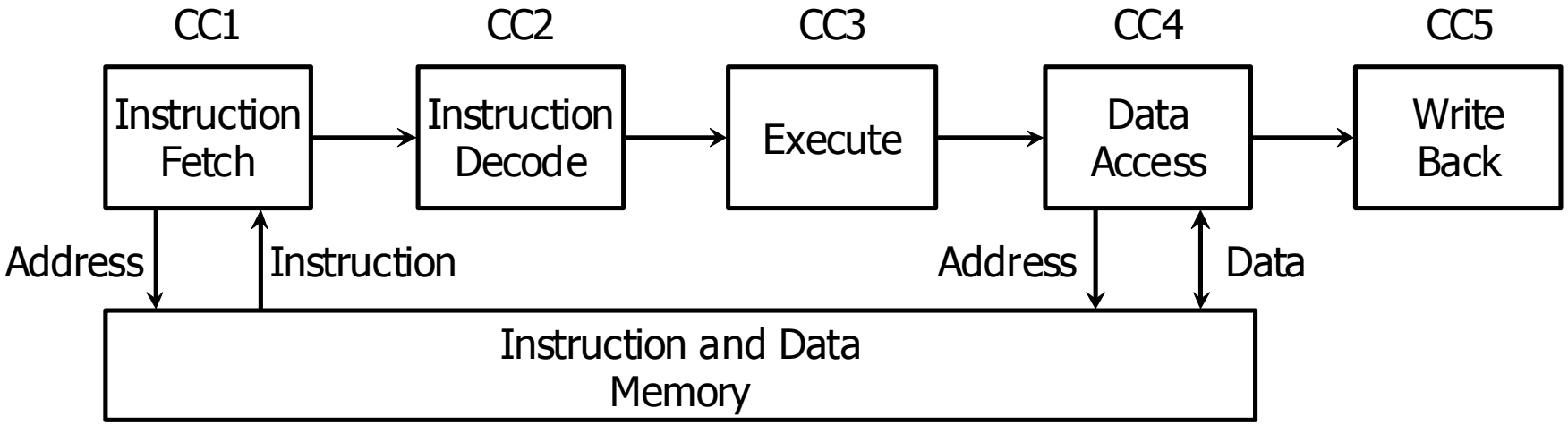
**Typical structural hazard — unified cache hazard**

Instructions and data in same memory device

Cannot access data and fetch instruction on same clock cycle

**To prevent hazard**

Stall **INSTRUCTION FETCH** during data **MEMORY ACCESS**




**unified cache**

# Stall Implementation for 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 | $I_3$ |
| CC9 | | | | $I_4$ | ϕ |
| CC10 | | | | | $I_4$ |

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

| | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 | CC10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $I_1$ | IF | ID | EX | MEM | WB | | | | | |
| LW | | 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 |

# Calculating Effect of Cache Hazard on CPI

Assume:
- Loads ~ 25%
- Stores ~ 15%
- Other ~ 60%

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

$$= \sum_{i=\text{types}} \left( \frac{\text{stall cycles}}{\text{stall}} \right)_i \times \left( \frac{\text{stalls}}{\text{instruction}} \right)_i \times \frac{\text{instructions}_i}{\text{instruction}}$$

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

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

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

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

$$CPI = CPI^{ideal} + CPI^{stall} = 1.40 \Rightarrow (\text{degradation} = \frac{0.4}{1.4} = 29\%)$$

# Data Hazards

**Instruction result not ready when needed**

**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

**To prevent hazard — stall pipeline until result is ready**

# Control Hazards

## Branch outcome affects program counter (PC)

- Taken
  - Branch condition is true and **`PC`** ← **`PC`** + **`Disp`**
- Not taken
  - Branch condition is false and **`PC`** not changed
- Target
  - Result of calculation **`PC`** ← **`PC`** + **`Disp`**

## Branch hazard

- Outcome not known until branch execution finishes
- Pipeline automatically fetches (default) instruction following branch
- Default instruction not correct if branch taken

## To prevent hazard

- Flush default instructions
- Stall pipeline until branch condition and branch target are ready
- Delay in processing branch instructions is called branch penalty

# Exception Hazards

**Exception**

- Hardware or software condition requiring special service routine

**Interrupt**

- Service response to external hardware event
- Usually asynchronous
  - Not triggered by program instructions
  - Does not affect validity of running instructions

**Trap**

- Service response to software condition in running program
- Usually synchronous
  - Triggered by program instructions
  - May stall or affect validity of running instructions

**Hazard**

- Multiple instructions in various stages of execution in pipeline
- How/where/when to interrupt pipeline
- Where is return-point?

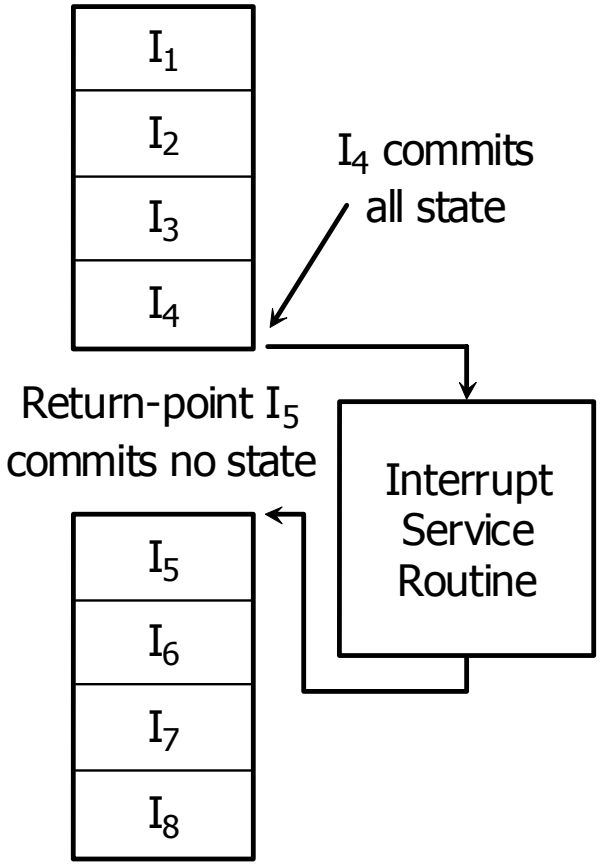
# Precise Exception

## Return-point

- Follows atomic operation
- Previous operations commit all results to state
- No following operations commit any results to state

## Precise exception

- Exception with well-defined return-point
- Service exception following atomic operation
- Restart execution at return point without error



# Exception Hazards in 5-Stage Pipeline

## Exceptions specific to each stage

- Memory access exception in IF or MEM
- Instruction exception in ID
- Arithmetic exception in EX

## 5 instructions in various stages of execution

- Where is return-point?
- How to handle subsequent partially executed instructions?

| | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 |
|---|---|---|---|---|---|---|---|---|---|
| $I_1$ | IF | ID | EX | MEM | WB | | | | |
| $I_2$ | | IF | ID | EX | MEM error | WB | | | |
| $I_3$ | | | IF | ID | EX | MEM | WB | | |
| $I_4$ | | | | IF | ID | EX | MEM | WB | |
| $I_5$ | | | | | IF | ID | EX | MEM | WB |

# Berkeley Solution

**Attach exception status field and source PC to instruction in IF**

**Instruction raises exception**

Mark status field with exception

Continue pipeline until prior instruction completes (reaches WB)

**RETURN-POINT** ← PC of instruction that raises exception

Flush pipeline (mark instructions in IF — MEM as NOP to cancel WB)

PC ← **EXCEPTION SERVICE ROUTINE** (ESR)

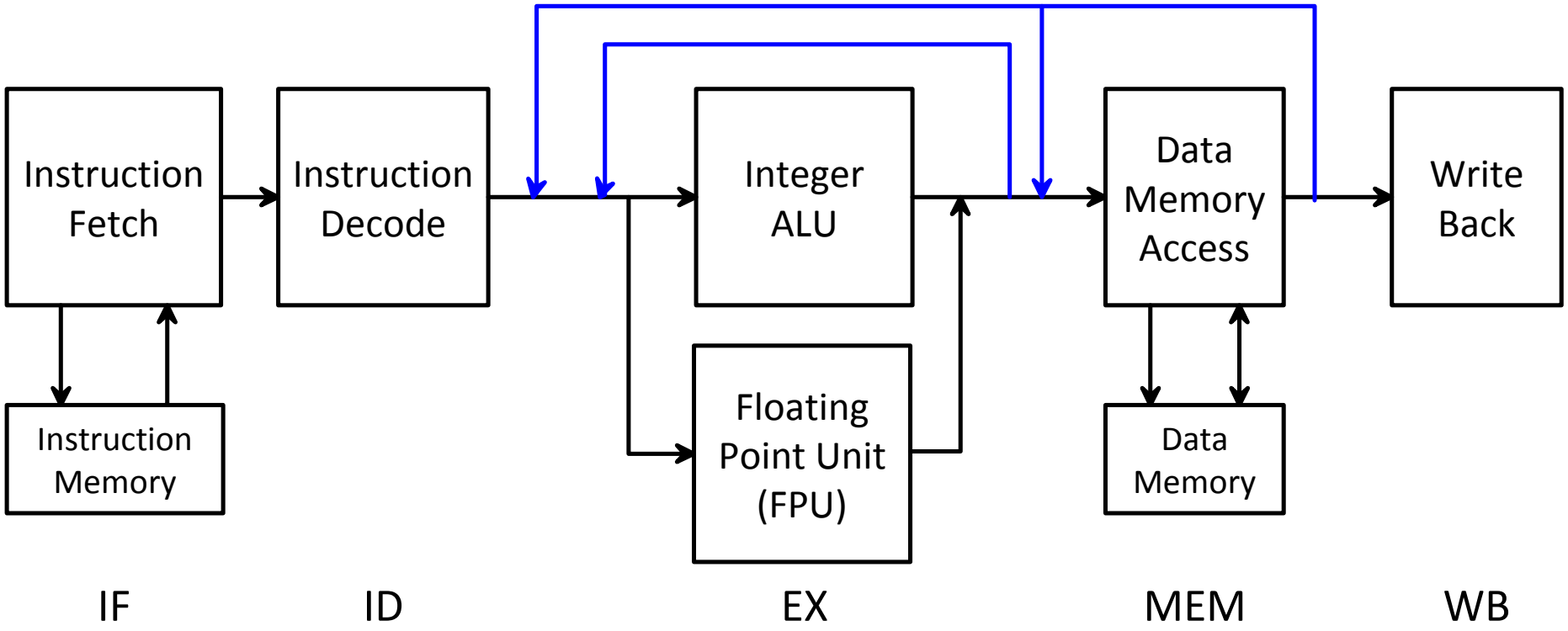
Return from ESR depends on exception type

| | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 | CC10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $I_1$ | IF | ID | EX | MEM | WB | $I_1$ completes atomically | | | | |
| $I_2$ | | IF | ID | EX | error | ϕ | return point = $I_2$ | | | |
| $I_3$ | | | IF | ID | EX | ϕ | ϕ | | | |
| $I_4$ | | | | IF | ID | ϕ | ϕ | ϕ | | |
| $I_5$ | | | | | IF | ϕ | ϕ | ϕ | ϕ | |
| ESR | | | | | | IF | ID | EX | MEM | WB |

Ref: http://www-inst.eecs.berkeley.edu/~cs252/

# DLX: A Simplified RISC Model

# DLX Pipeline




**Hardware definition based on MIPS 2000 commercial microprocessor**

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

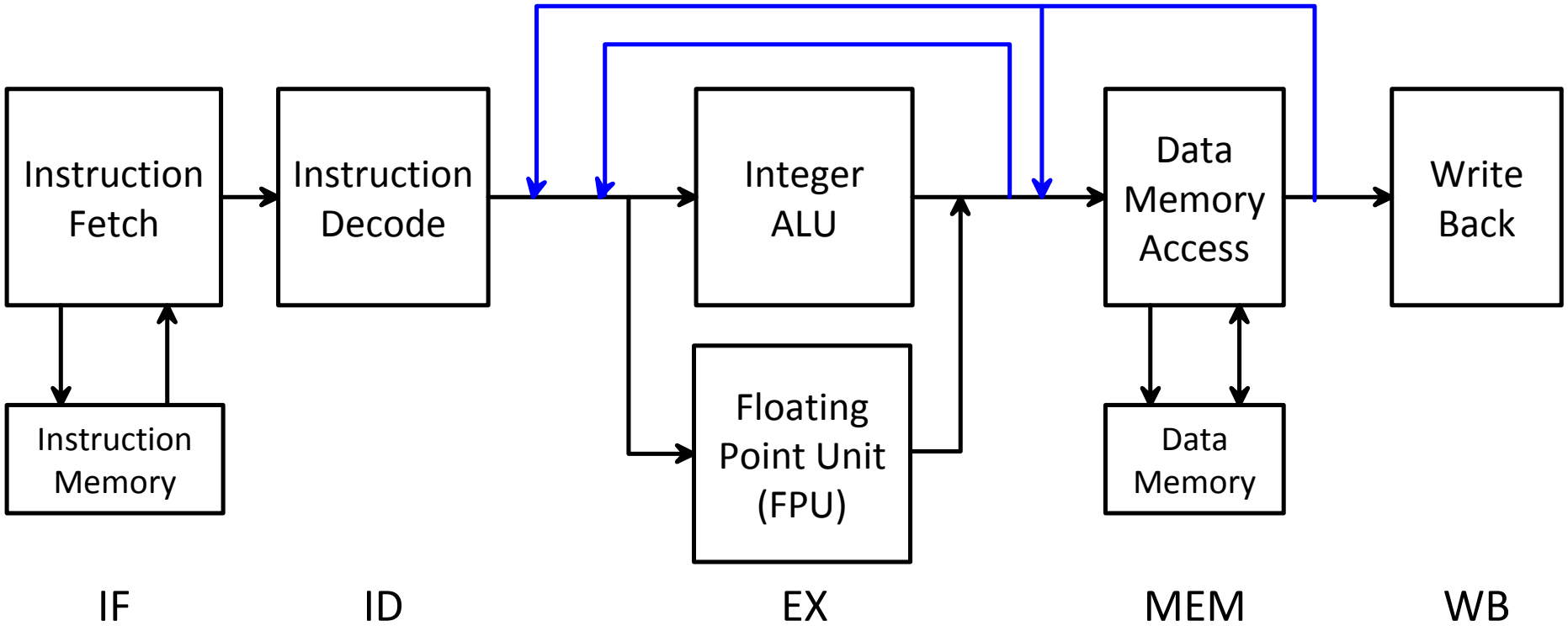
32 integer registers R0, R1, ... , R31

Regs[R0] = 0 (read-only)

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

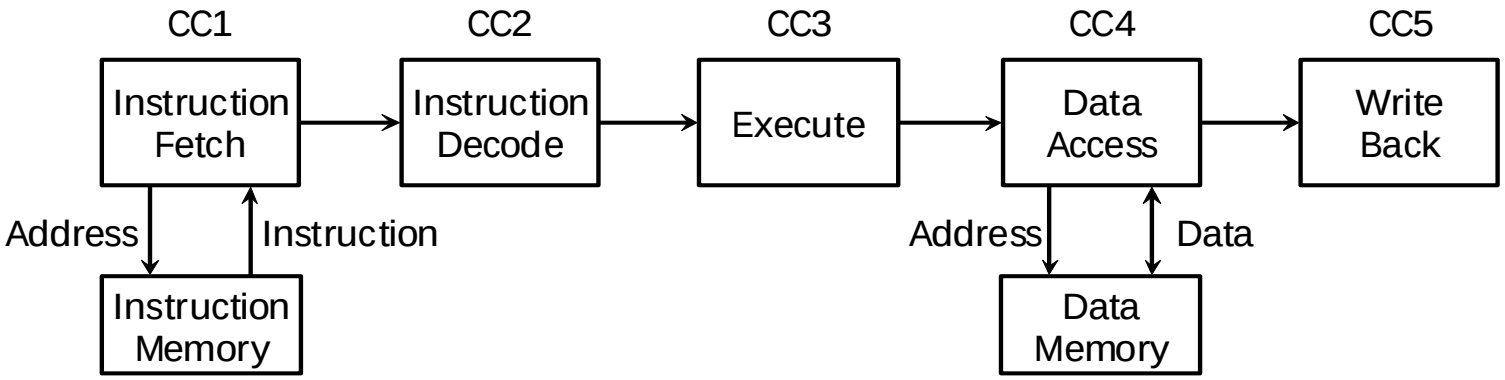
**Reference: Hennessey and Patterson, 2nd ed, chapter 2**

# DLX Pipeline Stages




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

# Ideal Instruction Pipelining — Instruction View




clock cycle →

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

# DLX Instruction Formats

## Three 32-bit instruction formats

**J-type** — absolute branch (jump) instructions

**`PC ← PC + OFFSET`**

**R-type** — register-register ALU instructions

**`rd ← ALU_function (rs1, rs2)`**

**I-type** — all other instructions

Load: **`rd ← imm(rs)`**

Store: **`imm(rs) ← rd`**

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

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

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

# Data Transfer Instructions

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

# Arithmetic/Logic Instructions

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

# Floating Point Instructions

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

# Control Instructions

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

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

# Programming in DLX Assembly

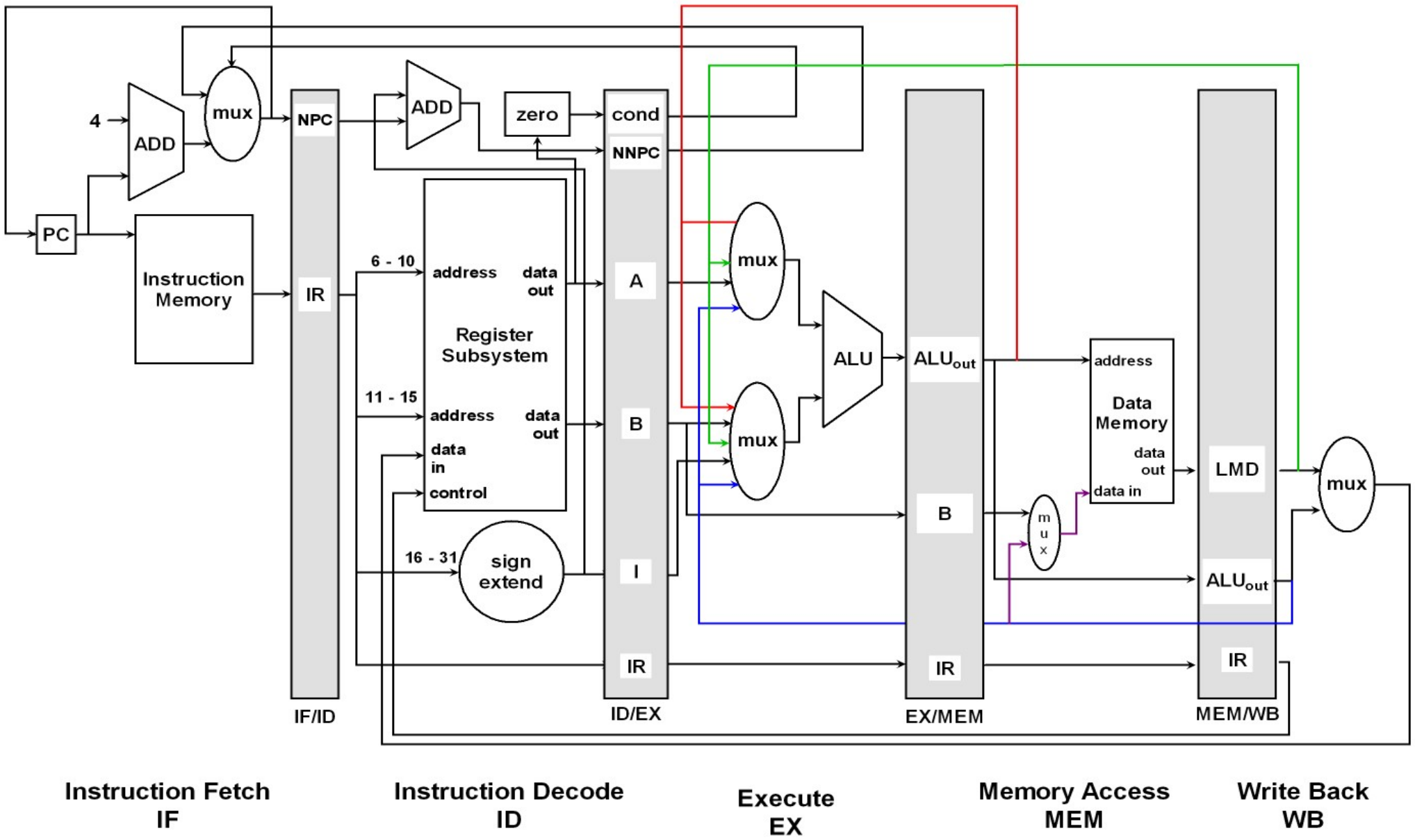
## C program

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

## DLX version

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

# DLX Implementation (Integer Pipeline)



# Temporary Registers in DLX Implementation

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

Store and forward instruction states between 5 stages

Update on falling edge of system clock

**PC**

Program Counter

Memory address of next instruction

**IR**

Instruction Register

Holds fetched instruction during execution

**NPC**

Next Program Counter

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

**A, B, I**

Operand buffers

Values read from data registers

**$ALU_{out}$**

ALU output

Result of ALU operation

**LMD**

Load Memory Data

Data loaded from memory

**Cond**

Condition flag

Result of test for conditional branch

# DLX Formal Specification (Integer Pipeline) — 1

## Instruction Fetch (IF)

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

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

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

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

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

## Instruction Decode (ID)

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

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

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

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

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

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

# DLX Formal Specification (Integer Pipeline) — 2

## Execute (EX)

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

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

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

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

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

## Memory (MEM)

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

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

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

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

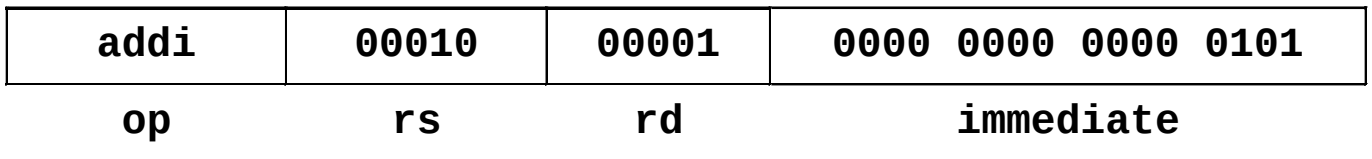
$$MEM/WB.IR \leftarrow EX/MEM.IR$$

## Write Back (WB)

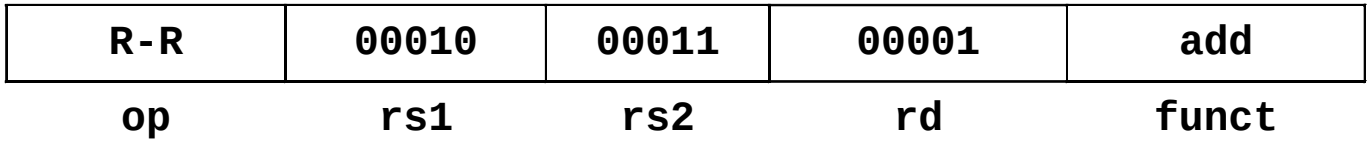
$$Reg[MEM/WB.IR_{11\text{-}15}] \leftarrow \begin{cases} MEM/WB.ALU_{OUT} & (ALU\text{-}I) \\ MEM/WB.LMD & (Load) \end{cases}$$

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

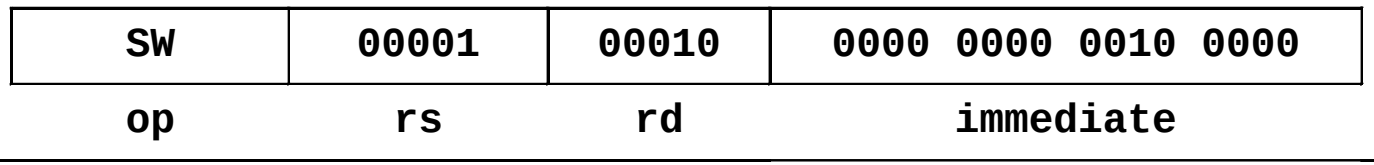
# Example Type-I ALU Instruction

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

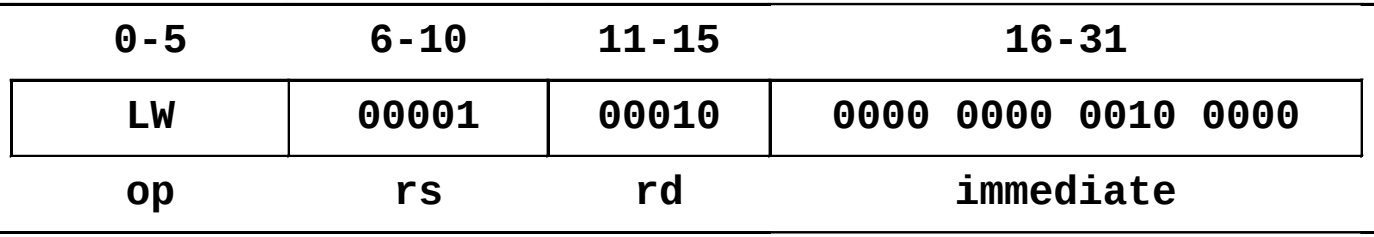
# Example Type-R ALU Instruction

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

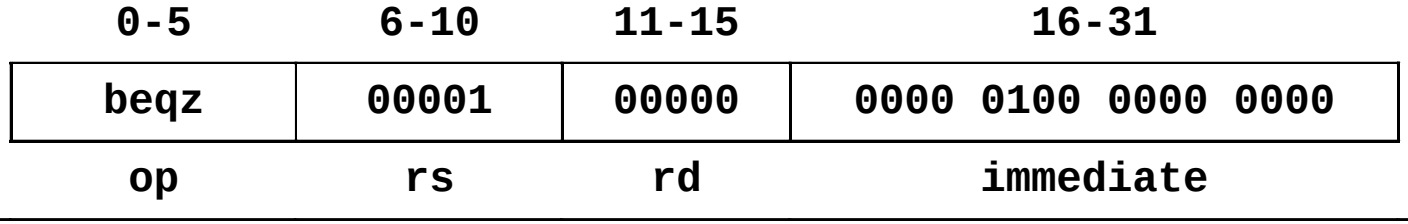
# Example Type-I Store Instruction

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

# Example Type-I Load Instruction

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

# Example Type-I Conditional Branch Instruction

| | |
|---|---|
| **Instruction** | `beqz R1, 1024` |
| **Operation** | `if (Reg[R1] == 0) PC ← NPC + 1024`<br>`else PC ← NPC` |
| **Encoding** |  |
| **Hardware Stage 1** | `IR ← Mem[PC]`<br>`NPC ← PC + 4` |
| **Hardware Stage 2** | $A \leftarrow Reg[IR_{6\text{-}10}]$ /* A ← Reg[R1] */<br>$B \leftarrow Reg[IR_{11\text{-}15}]$ /* B ← Reg[R0] */<br>$I \leftarrow (IR_{16})^{16}$ ## $IR_{16\text{-}31}$<br>`if (A == 0) cond = 1 else cond = 0`<br>`NNPC ← NPC + I` |
| **Hardware Stage 3** | `if (cond == 1) PC ←` $ALU_{out}$ `else PC ← NPC` |
| **Hardware Stage 4** | |
| **Hardware Stage 5** | |

# DLX Integer Pipeline Statistics

## Instruction distribution

Compile SPEC CINT → DLX instruction set

Sort object code into 4 groups

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

## Register dependencies

ALU instruction $I_N$

Destination operand ← ALU operation(source operands)

In 50% of ALU instructions

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

$I_{N-1}$ = ALU or load

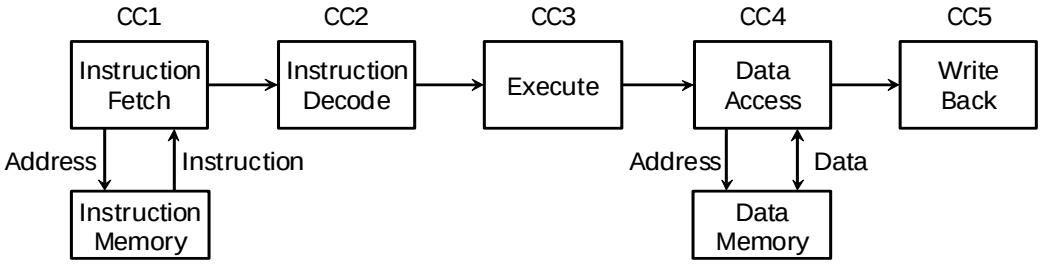
# Data Hazards in DLX Integer Pipeline

## RAW hazards

DLX registers updated in stage 5

Next instruction may read register in stage 2

Possible hazard to be avoided




## WAW hazards cannot occur

DLX writes in uniform order

- Memory updated in MEM
- Registers updated in WB

All updates performed in order of execution

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

## WAR hazards cannot occur

Loads performed in MEM and register reads in ID

Stores performed in MEM and registers updated in WB

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

# ALU-ALU RAW Dependencies

## Program with register-register dependencies

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

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

$I_2$ — $I_4$ have **R1** as **source**

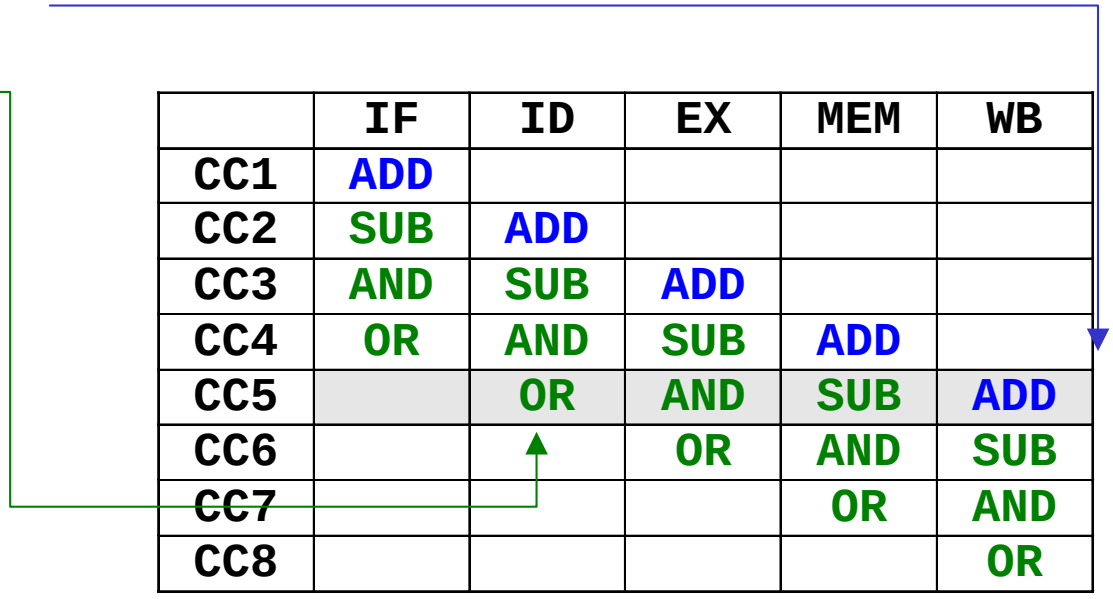
## Bad timing (uncorrected)

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

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

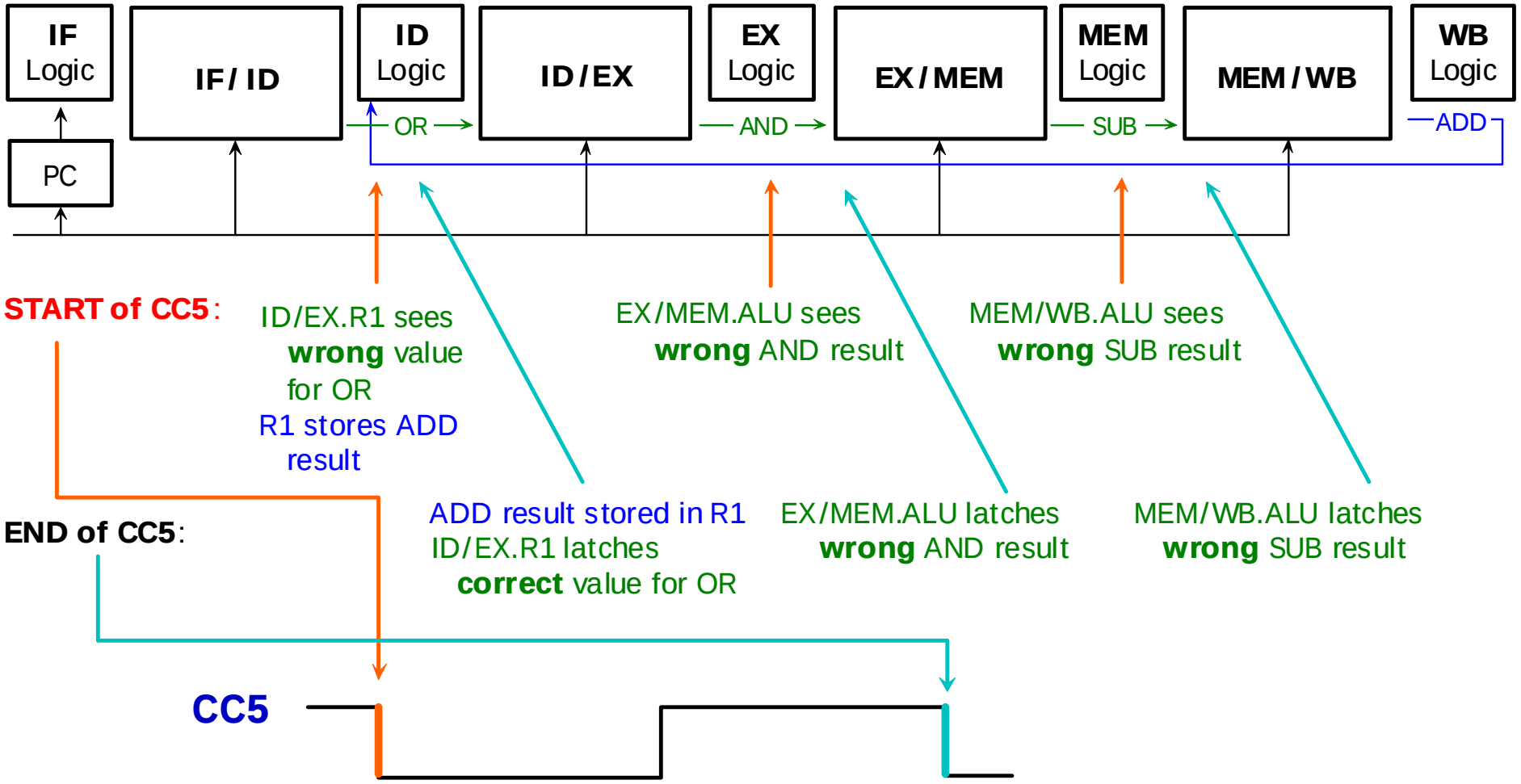
$I_3$ reads **R1** in **ID** during **CC4**

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



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

# Detailed View of CC5 (Uncorrected)




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

OR instruction reads correct value of R1

# Pipeline Stall to Avoid RAW Hazard

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

## Wait states

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

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

## Continuation

No hazard in **CC5**

**WB** operation performed at start of clock cycle

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

# Pipeline Stall in Instruction View

Clock Cycle →

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

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

## Performance degradation too large

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

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

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

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

# Forwarding or Bypass

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

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

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

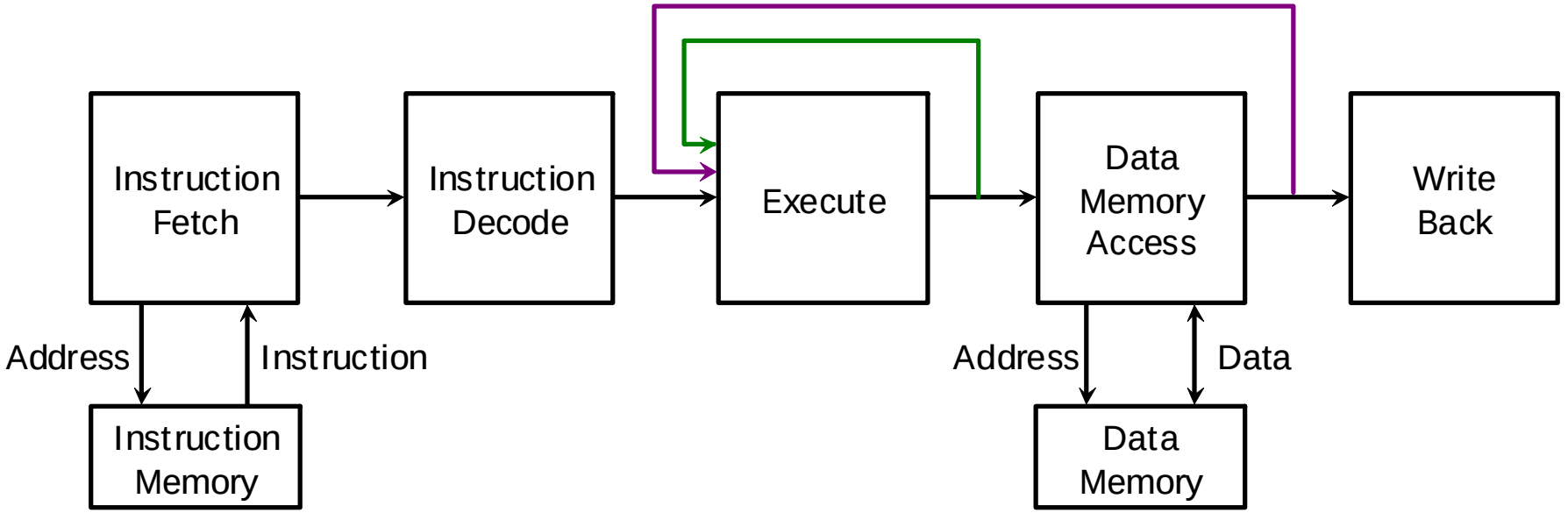
| | IF | ID | EX | MEM | WB |
|---|---|---|---|---|---|
| CC1 | ADD | | | | |
| CC2 | SUB | ADD | | | |
| CC3 | AND | SUB | ADD | | |
| CC4 | OR | AND | SUB | ADD | |
| CC5 | | OR | AND | SUB | ADD |
| CC6 | | | OR | AND | SUB |
| CC7 | | | | OR | AND |
| CC8 | | | | | OR |

## Trick to prevent stall

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

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

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



# Forwarding in Instruction View

Clock Cycle →

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

Processor moves state of **ADD** instruction from buffer to buffer

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

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

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

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

No stall cycles for Register-Register RAW hazard

$$CPI^{stall} = 0$$

# Load-ALU RAW Dependencies

## Program with register-load dependencies

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

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

$I_2$ — $I_4$ have **R1** as **source**

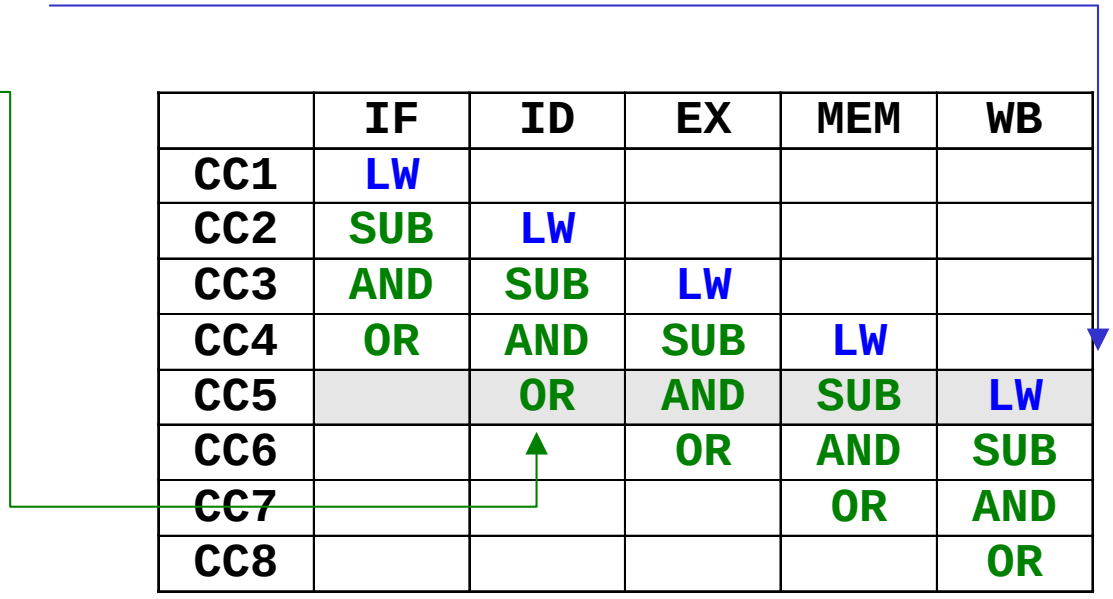
## Bad timing

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

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

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

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



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

# Memory Forwarding or Bypass

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

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

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

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

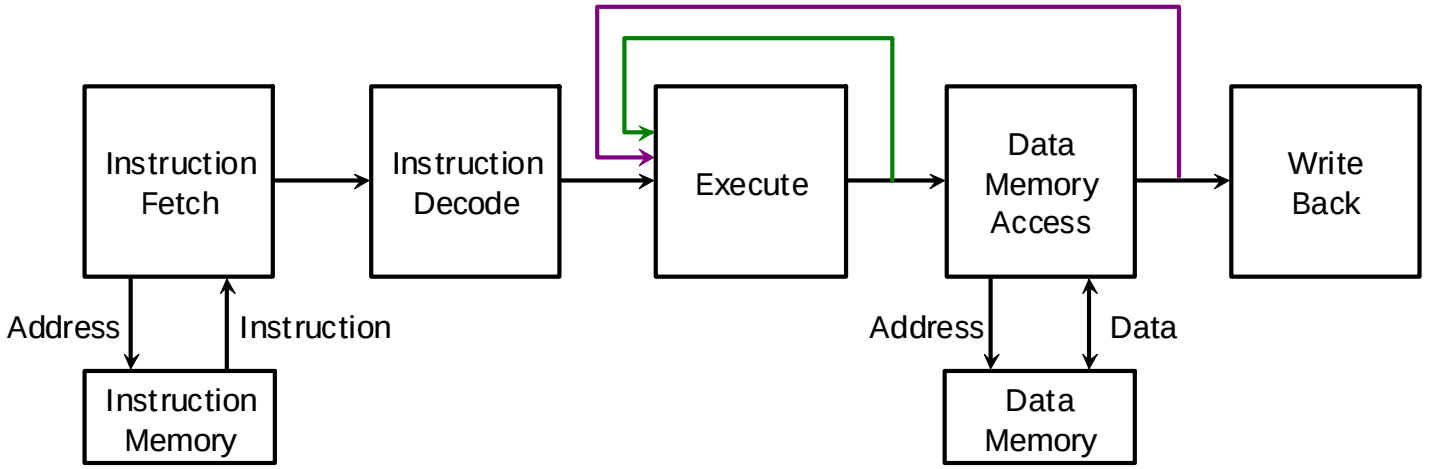
## Trick to minimize stall

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

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

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

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



# Forwarding in Instruction View

Clock Cycle →

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

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

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

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

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

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

# Register-Store RAW Dependencies

## Program with register-store dependency

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

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

$I_2$ has **R1** as **source**

## Bad timing

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

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

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

## Trick to prevent stall

**SW** reads incorrect value in **ID**

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

Clock Cycle →

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

# DLX Control Hazard

## Predict-Not Taken Policy

Flush stage IF on **BRANCH TAKEN**

Continue instruction in IF on **BRANCH NOT TAKEN**

Branch address and cond ready

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $I_1$ | BEQZ R1,$I_T$ | IF | ID | EX | MEM | WB | | | | |
| $I_{FT}$ | Fall-Through | | IF | ID | EX | MEM | WB | | | |
| $I_{FT+1}$ | | | | IF | | | | | | |
| ... | … | … | … | … | … | … | … | … | … | … |
| $I_T$ | Target | | | IF | ID | EX | MEM | WB | | |
| $I_{T+1}$ | | | | | IF | ID | EX | MEM | WB | |

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

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

# DLX Control Performance

## Predict-Not-Taken

Branch taken — Flush instruction in IF

Branch not taken — Continue instruction in IF

Better performance on not taken (no pipeline stall)

Ideal method if most branches are not taken

## Statistics from SPEC CINT

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

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

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

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

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

# Other Stalls

## Some instruction dependencies are not repaired by forwarding

Default handling — stall dependent instruction until source ready

## ALU – Branch Stall

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

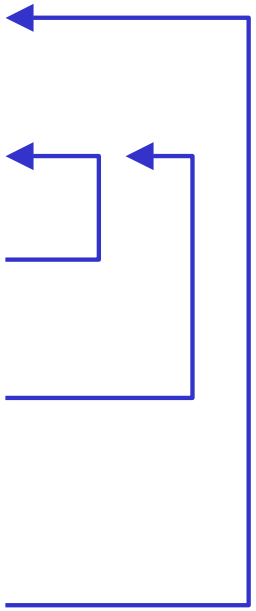
## ALU – Store

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

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

# Rescheduling

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



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

## Change to improve performance

Re-order instruction execution without affecting dependencies

Register renaming — remove false dependencies

Adjust address offsets

# Improvement by Re-Scheduling

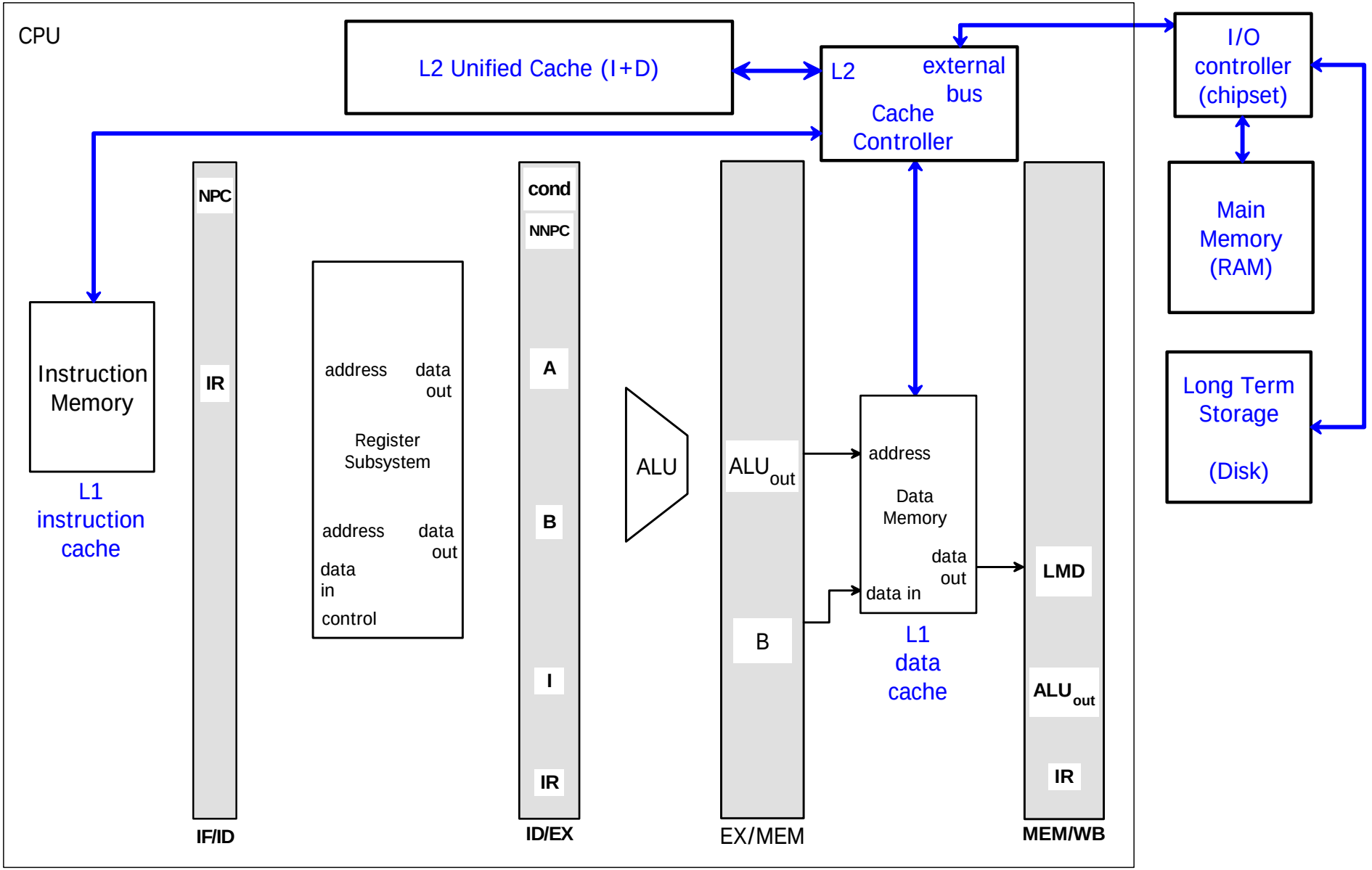
```
a[i] = a[i] + b[i] – c[i] + d[i]
```

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

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

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

# DLX Memory Hierarchy



# MIPS Architecture

## RISC Instruction Set Architecture (ISA)

Defines registers + instructions

## MIPS cores

Define device-dependent implementation details

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

## MIPS32

32-bit RISC ISA

**DLX = simplified MIPS32**

## MIPS64

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

Binary compatible with MIPS32

## Applications

Typically licensed to OEMs

Design implemented in embedded systems

MIPS-based PCs used in China

# MIPS32 ISA — 1

## Registers

32-bit integer registers

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

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

32-bit FP registers

**F0, F1, ... , F31**

Special registers

**HI, LO**

64-bit result of integer multiply

Quotient + remainder result of integer divide

## Instruction formats

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

# MIPS32 ISA — 2

## Coprocessors

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

## CP0

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

## CP1

- Interface to FPU

## CP2

- Available for device-specific implementations

## CP3

- Interface to FPU on MIPS64 and newer MIPS32

# MIPS32 Instruction Set — 3

## A few MIPS instructions not in DLX

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

# MIPS64 ISA

## Registers

64-bit integer registers

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

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

32-bit FP registers on 32-bit FPU

64-bit FP registers on 64-bit FPU

**`F0, F1, ... , F31`**

Special registers

**`HI, LO`**

128-bit result of integer multiply

Quotient + remainder result of integer divide

## Instruction formats

32-bit instruction length — binary compatible with MIPS32

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

MIPS64_double instructions act on full 64-bits in registers

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

# Directions for Improving Performance

# General Outlook

## Fundamental performance parameters

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

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

## Technological limitations

$IC$ grows with software complexity

$\tau$ 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 $CPI^{stall}_{branch\ penalty}$

# Processing Parallelism

## Instruction Level Parallelism (ILP)

Parallel execution of single program thread by **pipelining**

Superscalar processing

Multiple independent Execution Units (EU) on single CPU

EU = { ALU / FPU / Load Access / Store Access }

Instructions per cycle ≤ ALUs

## Thread Level Parallelism (TLP)

Parallel execution of single program by **multithreading**

Concurrency

Multiple threads defined in software to run independently

Interleaving

Threads execute by taking alternate turns on the same CPU core

Multiprocessing

Multiple independent cores running independent threads at same time

Processor cores = chips × (cores / chip)

# Flynn Taxonomy

| | Instruction | |
|---|---|---|
| **Data** | Single Instruction Single Data<br>SISD | Multiple Instruction Single Data<br>MISD |
| | Single Instruction Multiple Data<br>SIMD | Multiple Instruction Multiple Data<br>MIMD |

**SISD**

Standard single execution stream machine

**SIMD**

Single operation on multiple operands

Vector processor or processor array

**MISD**

Multiple operations on single operand

Few products — IBM Watson IA applies multiple algorithms to same data

**MIMD**

Multiple operations on multiple operands

Multiprocessor / multithreaded / cluster computer

Ref: M.J. Flynn, "Very High-Speed Computers", Proceedings of the IEEE, Dec. 1966.

# Single Instruction Multiple Data (SIMD)

## Single Instruction Single Data

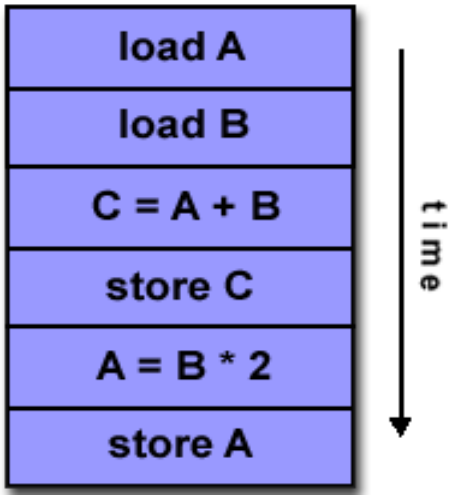
Single execution stream

## Single Instruction

One instruction stream

## Single Data

One input data stream




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

## Single Instruction Multiple Data

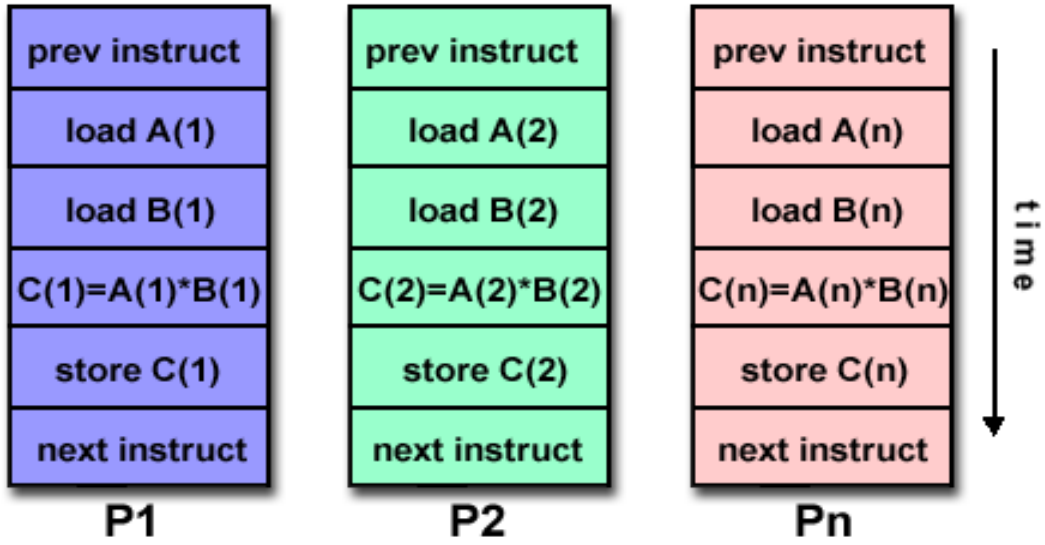
Multiple execution streams

## Single Instruction

One instruction stream

## Multiple Data

One instruction acts on multiple data



# Multiple Instruction Single Data (MISD)

## Single Instruction Single Data

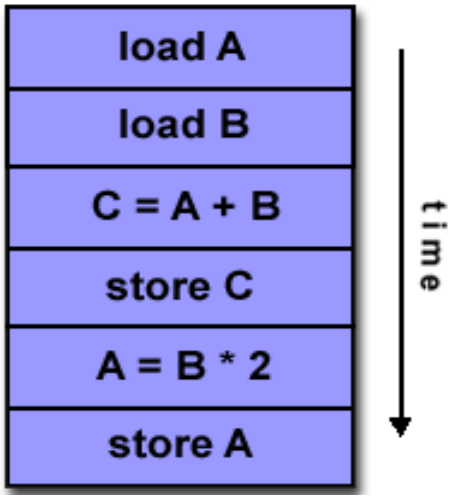
Single execution stream

## Single Instruction

One instruction stream

## Single Data

One input data stream




## Multiple Instruction Single Data

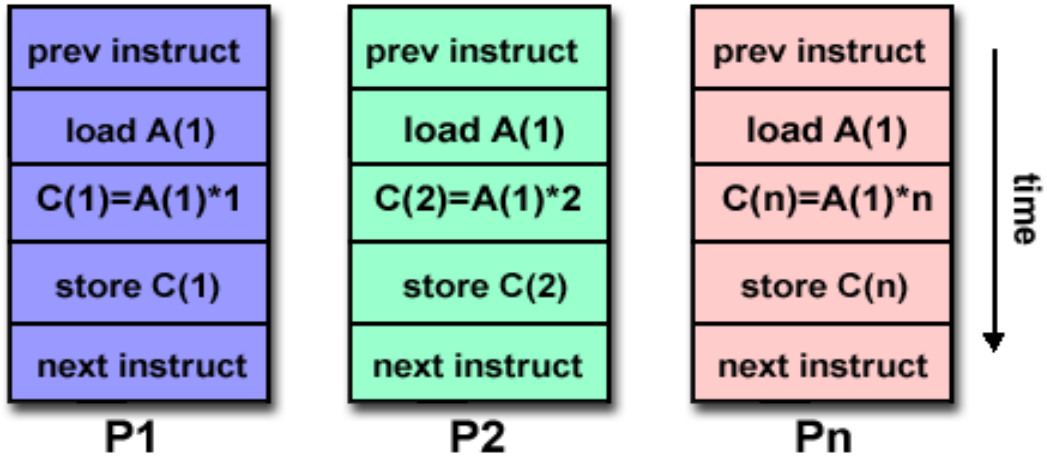
Multiple execution streams

## Multiple Instruction

Multiple instruction streams act on same data

## Single Data

One input data stream



# Multiple Instruction Multiple Data (MIMD)

## Single Instruction Single Data

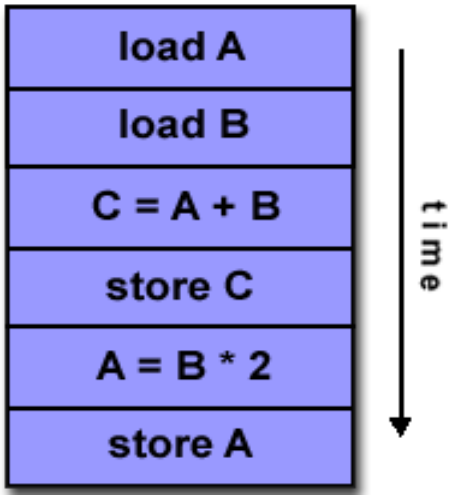
Single execution stream

## Single Instruction

One instruction stream

## Single Data

One input data stream




## Multiple Instruction Multiple Data

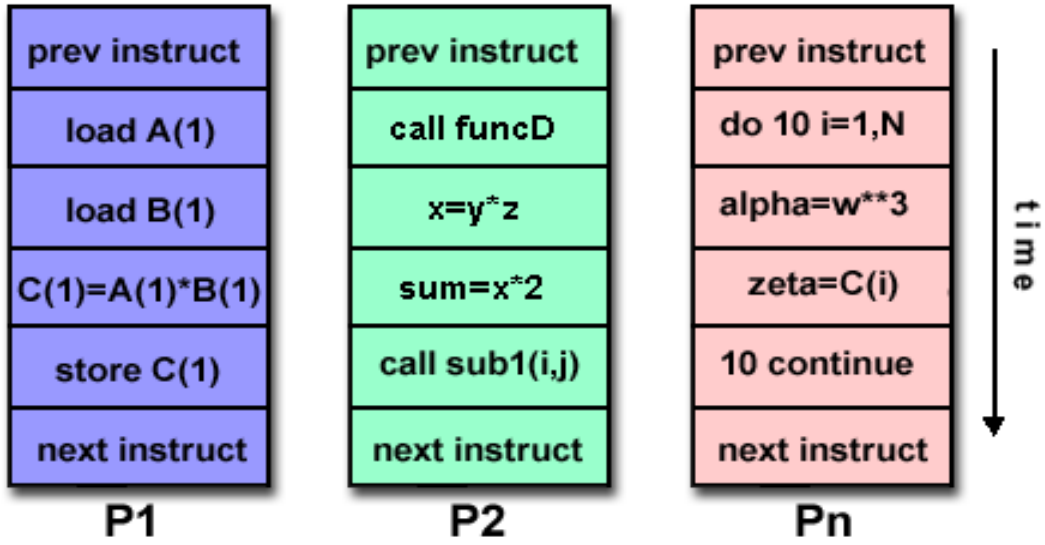
Multiple execution streams

## Multiple Instruction

Multiple instruction streams run independently

## Multiple Data

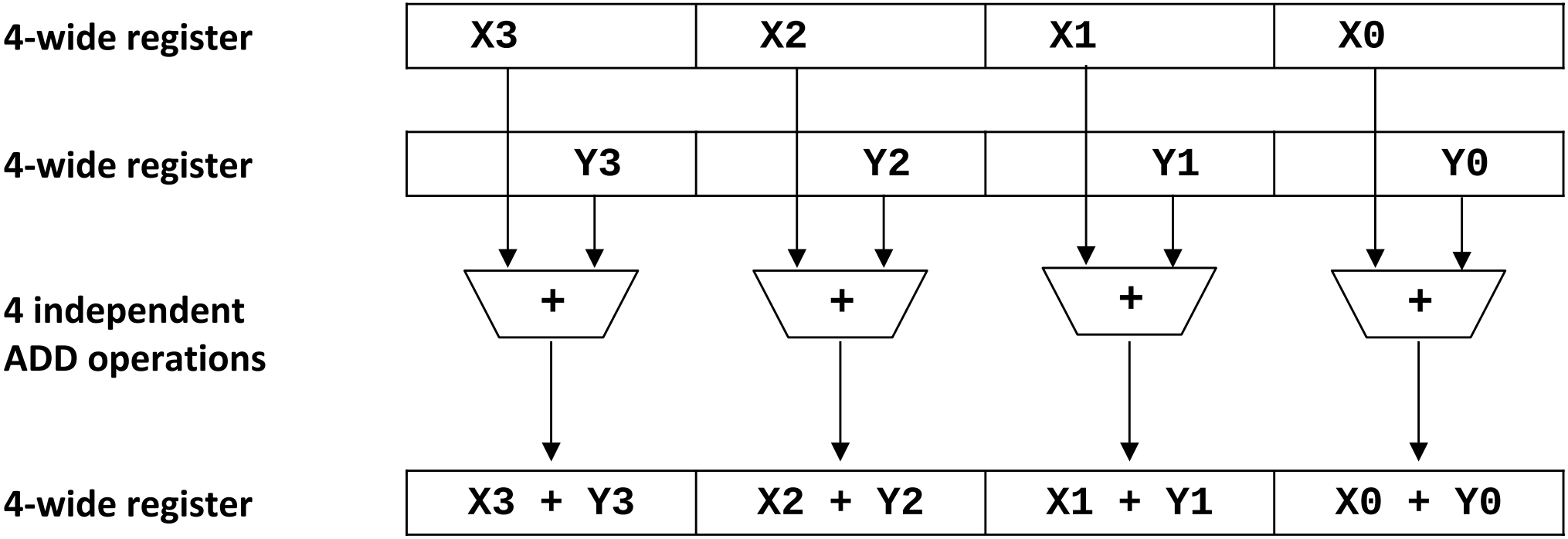
Each instruction stream acts on independent data



# SIMD — Vector ALU

## Example

`p_ADD = parallel add`

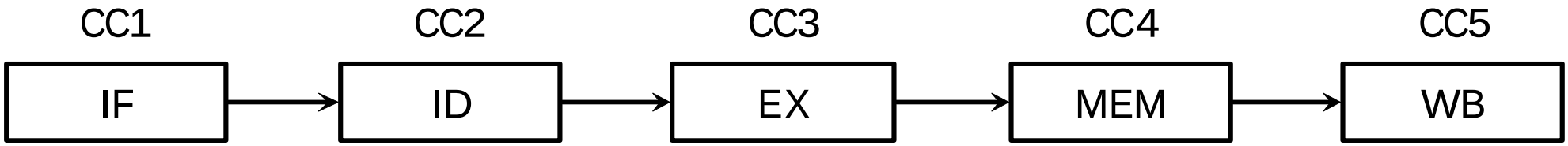



## Typical applications for vector operations

Data compression / decompression

Graphics / audio 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 / SSE / AVX

Vector integer and float operations

| | SSE | AVX |
|---|---|---|
| Register width | 128 bits = 16 bytes | 256 bits = 32 bytes |
| Integer | 4 | 8 |
| Single-precision FP | 4 | 8 |
| Double-precision FP | 2 | 4 |

## Power AltiVec vector processor

Similar to MMX integer operations

128-bit registers

## Compiler Support

Vector instructions in processor instruction set

Reasonable compiler supports vectorization

# Vectorization Example on Intel SSE

```
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 per loop iteration

---

```
int i;
float a[100], red;
…
red = 0;
ASM p_XOR xmm0,xmm0              /* zero accumulator */
for (i = 0; i < 25; i++) {
     ASM p_ADD xmm0,a[4*i]       /* four 32-bit additions */
}
ASM h_ADD xmm0                   /* add h_add X ⇒
                                    X3 ← X3 + X2 + X1 + X0 */
ASM p_MOV red, xmm0              /* red ← FP sum */
```

Loop runs **25** times

Operates on 4 array elements per loop iteration

# Optimizing Data Structures

## Data structure for 3D object

200 vertices in (x, y, z) coordinates

24-bit color (red, blue green) at each vertex

## Typical data structure — array of structures

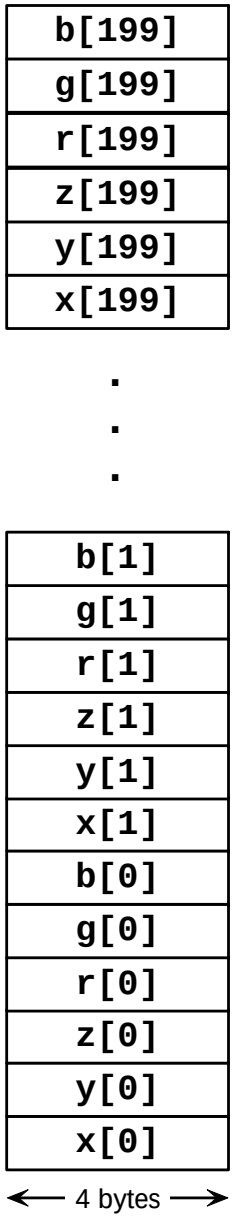
Struct = 200 consecutive groups of six 32-bit words

```
struct { float x, y, z, r, g, b
    } AoS_xyz_rgb[200];
```

SSE requires sequence

- 4 parallel **x** values
- 4 parallel **y** values
- 4 parallel **z** values
- 4 parallel **r** values
- 4 parallel **g** values
- 4 parallel **b** values

SSE cannot operate on AoS data structure



# Hybrid Data Structures

## Hybrid Structure

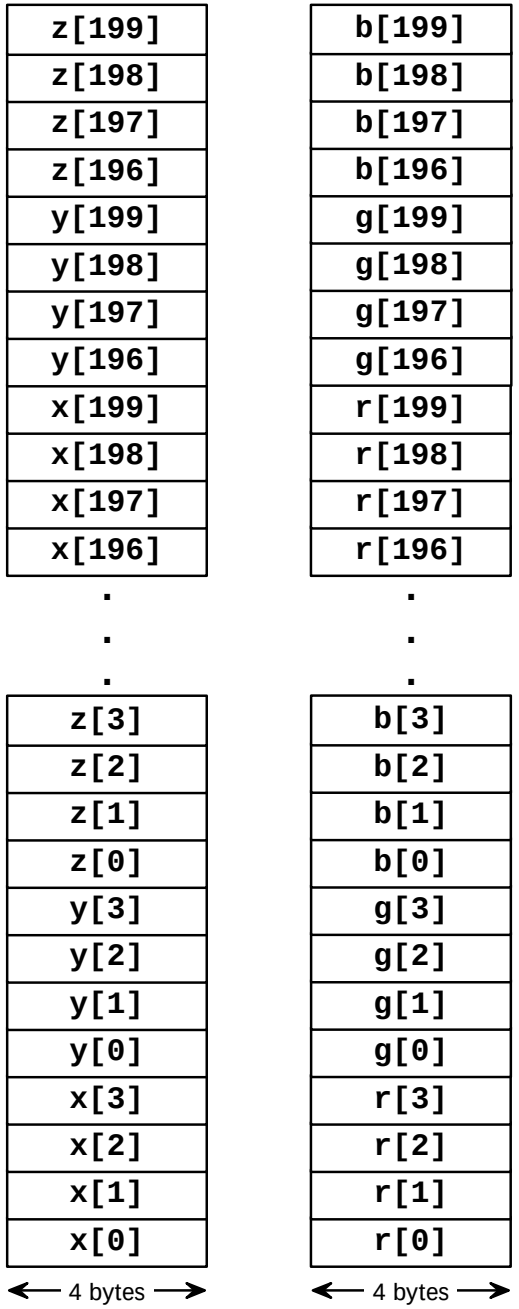
```
struct { float x[4], y[4], z[4] ;
} H_xyz[50] ;
struct {float r[4], g[4], b[4] ;
} H_rgb[50] ;
```

Vertex data and colors are loaded separately

SSE thread-1 loads 16 bytes in parallel

- Operates on vector of four 32-bit FP values
- Parallel operation on 4 **x** values
- Parallel operation on 4 **y** values
- Parallel operation on 4 **z** values

SSE thread-2 operates on rgb



# Graphics Processing Unit — Hardwired GPU

## Application of chip real estate (≈ 3 billion transistors)

CPU — efficient processing of sequential code

GPU — large number of execution cores

CPU offloads graphics operations to GPU as auxiliary processor

## Hardwired graphics pipeline

| Stage | Input | Output | |
|---|---|---|---|
| Geometry | 3D vertices | 2D triangles | Area enclosed by lines between vertices |
| Rasterization | Triangles | Fragments | Screen coordinates covered by triangle |
| Fragment | Fragments | Colored Fragments | Fragment colored by interpolation of vertices or from texture memory |
| Composition | Textures | Image | Pixel coordinates and colors |

## Theoretical SIMD performance

SSE on 3.7 GHz Pentium 4 (2000)

1 pipeline × 4-wide SIMD × 3.7 GHz = 14.8 GFLOPS

Nvidia GeForce 6800 Ultra (2005)

16 pipelines × 8-wide SIMD × 520 MHz =63 GFLOPS

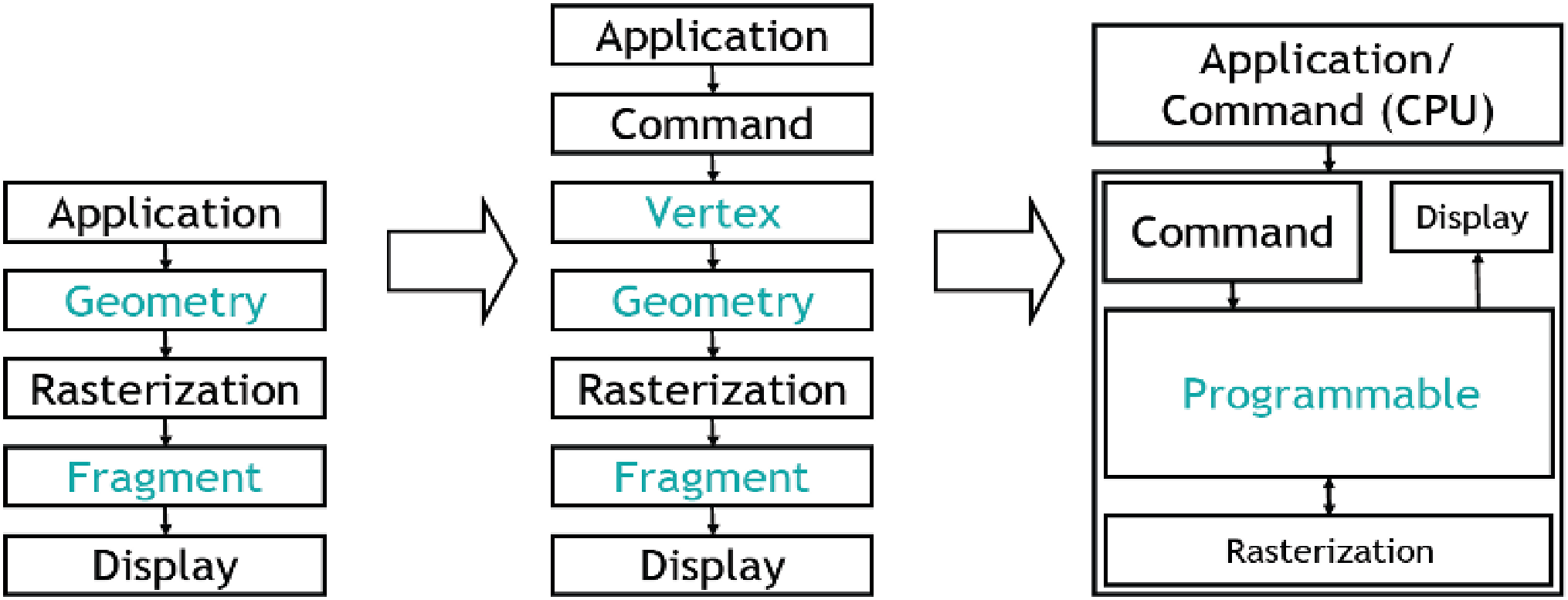
# Graphics Processing Unit — Programmable Pipeline

## GPU evolution

Hardwired pipeline stage → programmable pipeline stage

Hardwired pipeline organization → **SHADER**

- Programmable processor core
- Optimized to support graphics operations



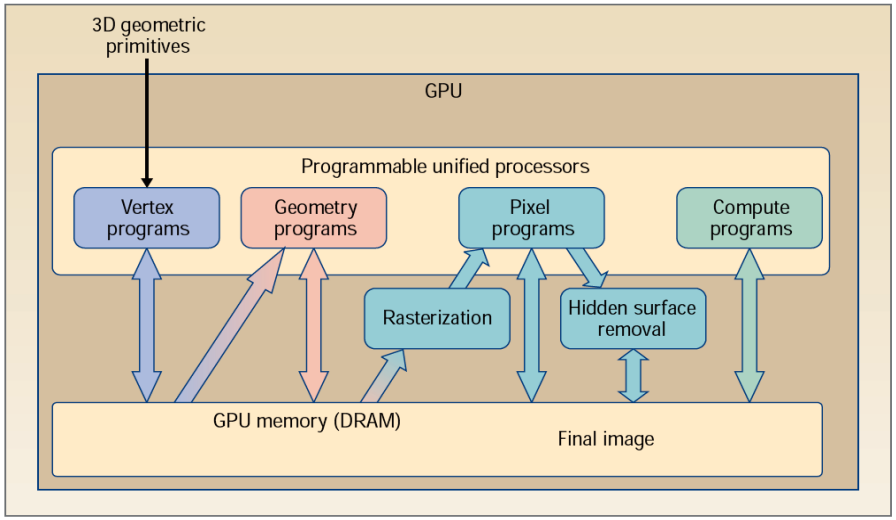

John Owens, *GPU Architecture Overview*, UC Davis, SIGGRAPH 2007

# Graphics Processing Unit — GPGPU

## General Purpose GPU (GPGPU)

Heterogeneous multi-core processor

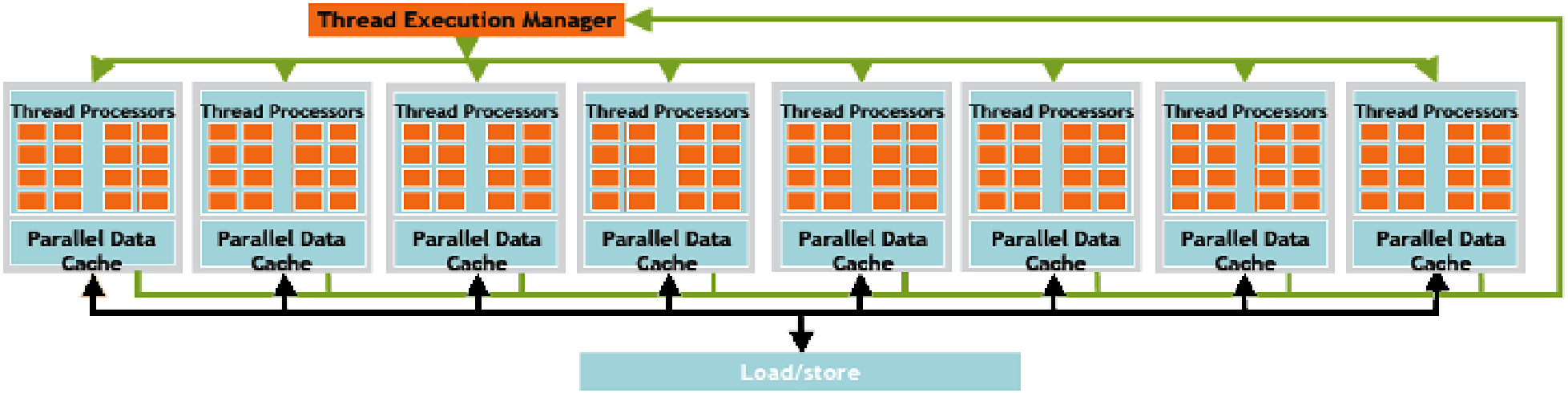
Grid of shaders




Nvidia shader

## Nvidia GeForce 8800 (2005)

8 thread processors × 16 shaders/processor = 128 shaders

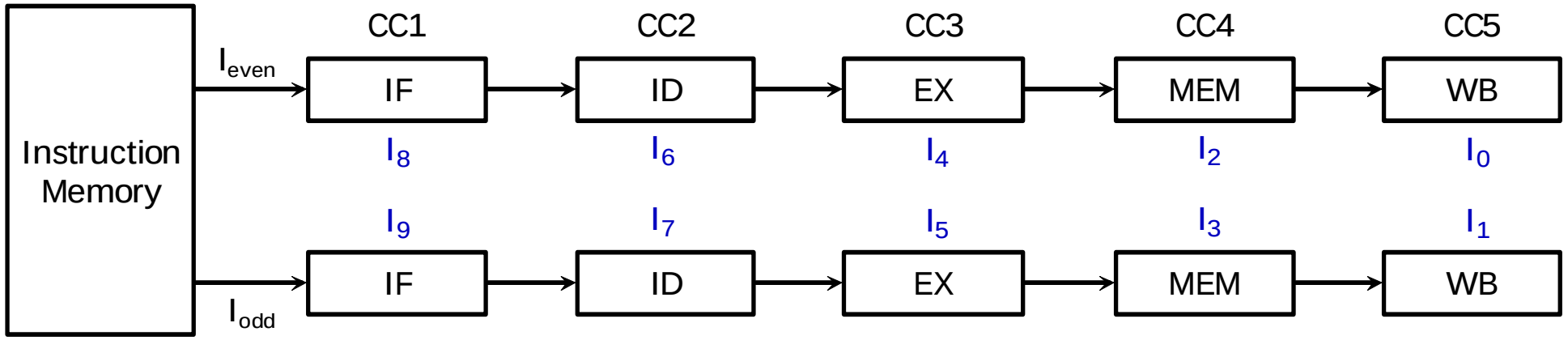



## GeForce RTX 3090 Ti (2022)

10752 shaders

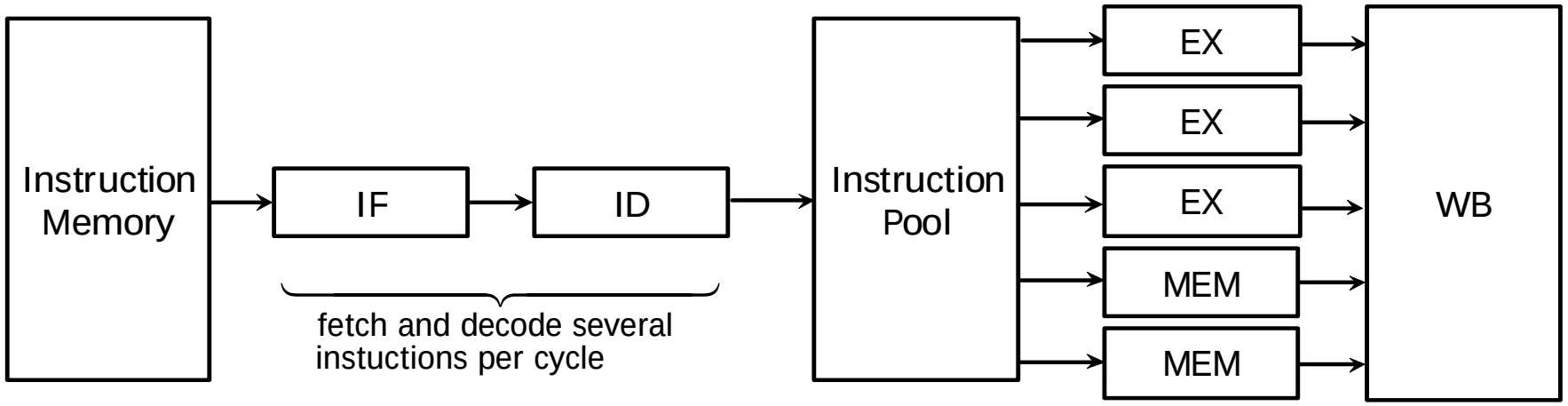
# Two Superscalar Pipeline Models for DLX

## Replicate entire pipeline




## Replicate EX and MEM execution units (EUs)

Instruction Pool holds decoded instructions



# Intel 486 — Single Pipeline Processor

## Instruction Fetch (IF)

## Stage-1 Decode (D1)

Identify operation

Read register sources

Calculate register part of effective address for memory source

## Stage-2 Decode (D2)

Calculate physical data address from logical address

Generate execution microcode

## Execution (EX)

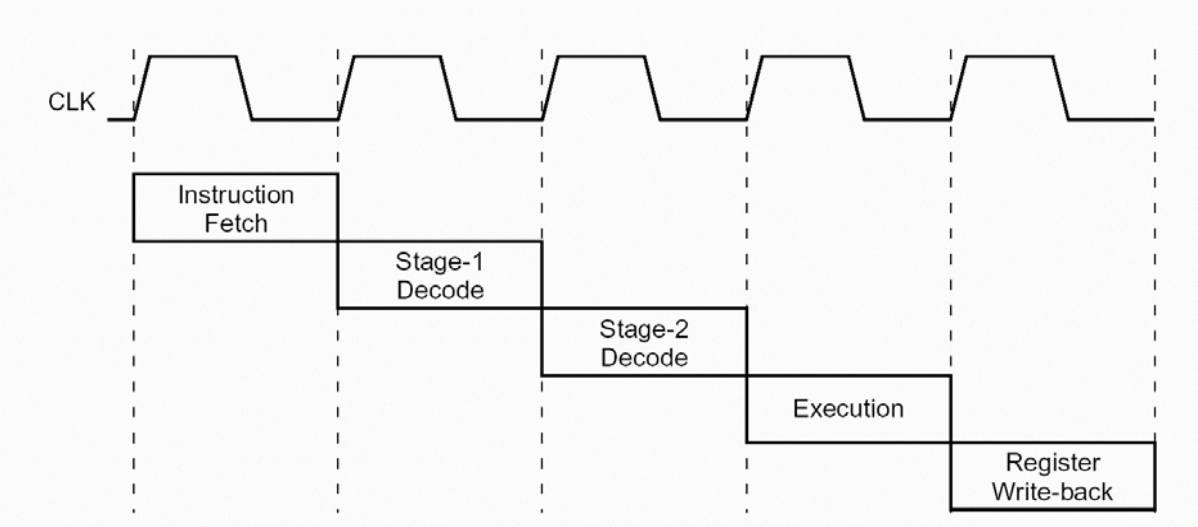
Integer ALU operations

FPU operations

Data memory accesses

## Register Writeback (WB)

Update destination registers




"Embedded Intel486™ Processor Hardware Reference Manual" (July 1997)

# Intel Pentium — Dual Pipeline Processor

## Ideal pipeline progress in Intel 486

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $I_0$ | PF | D1 | D2 | EX | WB | | | |
| $I_1$ | | PF | D1 | D2 | EX | WB | | |
| $I_2$ | | | PF | D1 | D2 | EX | WB | |
| $I_3$ | | | | PF | D1 | D2 | EX | WB |
| $I_4$ | | | | | PF | D1 | D2 | EX |
| $I_5$ | | | | | | PF | D1 | D2 |
| $I_6$ | | | | | | | PF | D1 |
| $I_7$ | | | | | | | | PF |

## Ideal pipeline progress in Intel Pentium

Pipeline stages as in 486

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $I_0$ | PF | D1 | D2 | EX | WB | | | |
| $I_1$ | PF | D1 | D2 | EX | WB | | | |
| $I_2$ | | PF | D1 | D2 | EX | WB | | |
| $I_3$ | | PF | D1 | D2 | EX | WB | | |
| $I_4$ | | | PF | D1 | D2 | EX | WB | |
| $I_5$ | | | PF | D1 | D2 | EX | WB | |
| $I_6$ | | | | PF | D1 | D2 | EX | WB |
| $I_7$ | | | | PF | D1 | D2 | EX | WB |

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

$$CPI = \frac{\text{clock cycles}}{IC} \Rightarrow CPI^{ideal} = \frac{N \text{ clock cycles}}{2 \times N} = 0.5 = \frac{CPI^{\text{one-pipeline ideal}}}{\text{number of pipelines}}$$

# Problems with Pipeline Replication

## Dependent instructions cannot be paired

$I_N$ and $I_{N+1}$ cannot be paired ⇒ run in usual pipeline sequence

## Paired instructions stall together

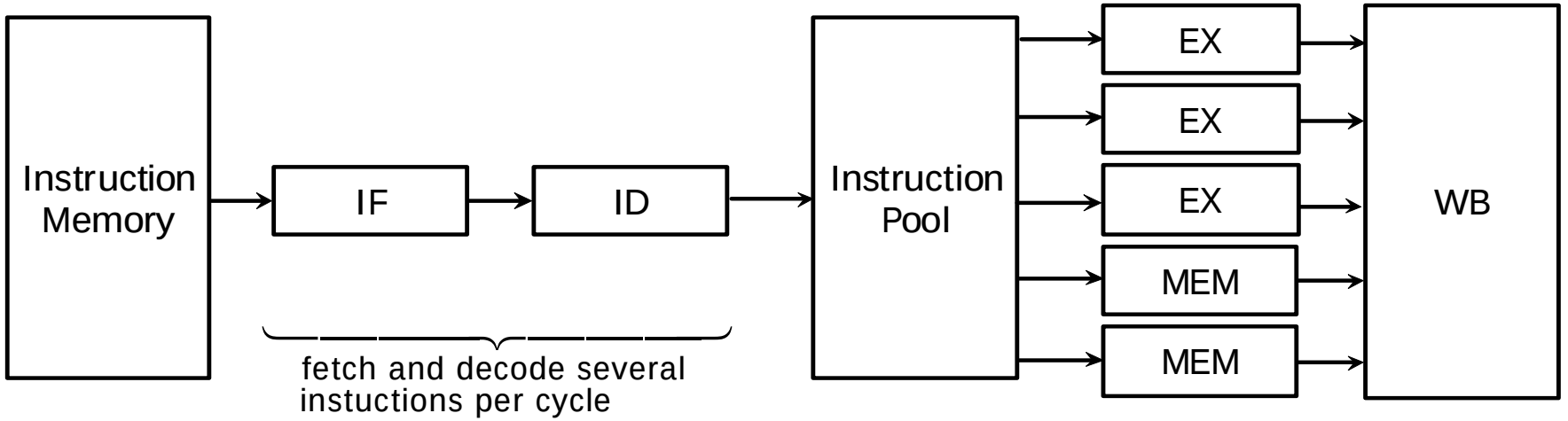
$I_S$ stalls ⇒ $I_{S+1}$ stalls

## Example

```
I0   MOV EAX, 100          ; I0 and I1 cannot be paired
I1   ADD ESP, EAX
I2   PUSH EAX              ; stack pointer ESP not forwarded
                           ; I2 waits for I1 write-back
I3   ADD ECX, EAX          ; I2 stalls ⇒ I3 stalls
```

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| $I_0$ | IF | D1 | D2 | EX | WB | | | | |
| $I_1$ | IF | D1 | D1 | D2 | EX | WB | | | |
| $I_2$ | | IF | IF | D1 | D1 | D1 | D2 | EX | WB |
| $I_3$ | | IF | IF | D1 | D1 | D1 | D2 | EX | WB |

# Instruction Pool Example




| | | | CC | Utilization |
|---|---|---|---|---|
| I1<br>I2<br>I3 | ADDI R1, R0, #10<br>ADDI R2, R0, #87<br>LW R6, 100(R4) | No dependencies<br>Run on 2 EX + 1 MEM | 1 | 3 |
| I4 | ADD R4, R2, R6 | Dependency — runs in EX | 2 | 1 |
| I5 | ADD R7, R4, R6 | Dependency — runs in EX | 3 | 1 |
| I6 | SW 100(R4), R7 | Dependency — runs in MEM | 4 | 1 |
| I7 | SW 200(R0), R8 | 1 store per CC — runs in MEM | 5 | 1 |

I7 has no dependency — could **RESCHEDULE** to CC1, CC2, CC3

# Instruction Rescheduling Strategies

## Instruction dependencies

RAW hazard — instruction $\mathbf{I_{k+j}}$ reads operand before $\mathbf{I_k}$ writes it

WAW hazard — instruction $\mathbf{I_{k+j}}$ writes operand before $\mathbf{I_k}$ writes it

WAR hazard — instruction $\mathbf{I_{k+j}}$ writes operand before $\mathbf{I_k}$ reads it

## Instruction dependency stall

Delay $\mathbf{I_{k+j}}$ until $\mathbf{I_k}$ completes

## Register renaming

Allocate separate register sets to independent instruction blocks

Eliminates dependencies between independent blocks using same registers

## Instruction rescheduling

Move independent instructions between $\mathbf{I_k}$ and $\mathbf{I_{k+j}}$

Increase instruction "distance" in pipeline to resolve dependence

**Compiler optimization**

Compiler reschedules object code (machine code listing) in second pass

**Dynamic rescheduling** (out-of-order execution)

Hardware executes instructions in most efficient order

Chooses instructions with no unresolved dependencies for execution

# Efficient Register Reuse

## Instruction block N

Allocate **R1**

Overwrite for use as destination operand

Instructions access **R1**

Access as source and destination operand

De-allocate (**RETIRE**) **R1**

After final source access in instruction block

## Instruction block N+1

Re-allocate **R1**

| Register Reuse | |
|---|---|
| `LW R1, 0(R2)` | `; allocate R1` |
| `ADDI R1, R1, #1` | `; R1 is source and destination` |
| `SW 0(R2), R1` | `; R1 is source — de-allocate R1` |
| `ADD R1, R2, R3` | `; re-allocate of R1` |

# Register Reuse versus Register Renaming

## Register renaming

- Instruction block N
  - Allocate **R1**
  - Instructions access **R1**
  - De-allocate (**RETIRE**) **R1**
- Instruction block N+1
  - Allocate new register instead of reuse

## Eliminates fictitious data dependence

- Instruction blocks use independent registers
- Enables rescheduling

## Example

| Reuse | Renaming | Rescheduling |
|---|---|---|
| `LW R1, 0(R2)` | `LW R1, 0(R2)` | `LW R1, 0(R2)` |
| `ADDI R1, R1, #1` | `ADDI R1, R1, #1` | `ADD R10, R2, R3` |
| `SW 0(R2), R1` | `SW 0(R2), R1` | `ADDI R1, R1, #1` |
| `ADD R1, R2, R3` | `ADD R10, R2, R3` | `SW 0(R2), R1` |

# Instruction Rescheduling for DLX

## Compiler reschedules instruction issue

- Respect real data flow dependencies
- Eliminate fictitious data dependencies by register renaming
- Enhance efficiency
  - Insert independent work between dependent instructions

| High Level Code | First Pass Assembly Code | Rescheduled Code for DLX |
|---|---|---|
| `X = X + 123` | `LW R2,[X]`<br>`ADD R2,R2,#123`<br>`SW [X],R2` | `LW R2,[X]`<br>`LW R3,[Y] ; Rename R2`<br>`LW R4,[Z] ; Rename R2`<br>`ADD R2,R2,#123`<br>`SUB R3,R3,#456`<br>`SUB R4,R4,#789`<br>`SW [X],R2`<br>`SW [Y],R3`<br>`SW [Z],R4` |
| `Y = Y – 456` | `LW R2,[Y] ; Reuse R2`<br>`SUB R2,R2,#456`<br>`SW [Y],R2` | |
| `Z = Z - 789` | `LW R2,[Z] ; Reuse R2`<br>`SUB R2,R2,#789`<br>`SW [Z],R2` | |

# Advantage of Rescheduled Code

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LW R2,[X] | F | D | X | M | W | | | | | | | | | | | |
| ADD R2,R2,#123 | | F | D | D | X | M | W | | | | | | | | | |
| SW [X],R2 | | | F | F | D | X | M | W | | | | | | | | |
| LW R3,[Y] | | | | | F | D | X | M | W | | | | | | | |
| SUB R3,R3,#456 | | | | | | F | D | D | X | M | W | | | | | |
| SW [Y],R3 | | | | | | | F | F | D | X | M | W | | | | |
| LW R4,[Z] | | | | | | | | | F | D | X | M | W | | | |
| SUB R4,R4,#789 | | | | | | | | | | F | D | D | X | M | W | |
| SW [Z],R4 | | | | | | | | | | | F | F | D | X | M | W |

| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LW R2,[X] | F | D | X | M | **W** | | | | | | | | | | | |
| LW R3,[Y] | | F | D | X | M | **W** | | | | | | | | | | |
| LW R4,[Z] | | | F | D | X | M | **W** | | | | | | | | | |
| ADD R2,R2,#123 | | | | F | **D** | X | M | **W** | | | | | | | | |
| SUB R3,R3,#456 | | | | | F | **D** | X | M | **W** | | | | | | | |
| SUB R4,R4,#789 | | | | | | F | **D** | X | M | **W** | | | | | | |
| SW [X],R2 | | | | | | | F | **D** | X | M | W | | | | | |
| SW [Y],R3 | | | | | | | | F | **D** | X | M | W | | | | |
| SW [Z],R4 | | | | | | | | | F | **D** | X | M | W | | | |

# Scheduling Algorithm — 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 Renamed DLX Code

## Instruction status after EX clock cycle

| 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 Renamed DLX Code

## Instruction status after EX clock cycle

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

**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 |

# Virtual Register Architecture

## Architectural register set (AR)

Defined in Instruction Set Architecture (ISA) of CPU

**DLX: R0, R1, … , R31, F0, …, F31**

**x86: EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP, EIP**

Visible to programmer

Atomic read / write at level of ISA semantics

## Virtual register set (VR)

Physical hardware registers

**VR0, VR1, VR2, …, VRn-1**

Physical register set > architectural register set

Pentium 4: n = 128

Not visible to programmer

VRs allocated to ARs per instruction block

Instruction blocks identified by block ID

# VR Allocation — Automatic Register Renaming in CPU

## CPU recodes each machine instruction

- Destination AR ← new VR
- Stable source AR ← literal value stored in AR
- Variable source AR ← last VR assigned to AR

## Recoded instruction

- Stored in instruction pool
- Executed in some specified order

## Commit to state (RETIREMENT)

- Copy AR ← VR in original order
- De-allocate VR

## Example

| Machine Code | Recoded | Execution CC |
|---|---|---|
| `LW R1, 0(R2)` | `LW VR1, 0(reg[R2])` | 1 |
| `ADDI R1, R1, #1` | `ADDI VR2, VR1, #1` | 2 |
| `SW 0(R2), R1` | `SW 0(reg[R2]), VR2` | 3 |
| `ADD R1, R2, R3` | `ADD VR3, reg[R2], reg[R3]` | 1 |

VRs retired in order (R1 ← VR1, R1 ← VR2, R1 ← VR3)

# Bookkeeping — Register Allocation Table (RAT)

## Cross-reference method

- Index
  - VR name
- Entry
  - AR allocated to VR
  - Instruction block ID
  - Status (allocated / not allocated)

## AR look-up method

- Index
  - AR name
- Entry
  - Allocated VR
  - Sequence ID to record VR allocation ordering
  - Status (stable / allocated)
- Method used in Intel x86

Ref: Hinton, et. al., *The Microarchitecture of the Pentium® 4 Processor*, Intel Technology Journal Q1, 2001

# Intel RAT Bookkeeping — Decoding and Retirement

## Instruction decoding

- Stable source AR ← value stored in AR
- Allocated source AR ← VR with highest sequence in RAT (for AR)
- Destination AR ← new VR allocation
  - New RAT entry written
  - Available VRs found by searching RAT for unallocated VRs
  - Decoder allocates available VRs sequentially mod n
  - Maximize code distance between VR reuse
- Recoded instruction (with VRs) placed in instruction pool
  - Instructions executed from pool by reading and writing VRs

## VR is retired if either:

- Instruction that overwrites allocated AR is marked finished
- VR does not appear as source for any unfinished instructions in pool
  - VR (with highest sequence number) copied to AR (commitment)
  - AR marked stable in RAT
  - Next use of AR as a source will access stable AR value

# Virtual Register Example

| | Instruction | AR | VR | Recoded Instruction |
|---|---|---|---|---|
| I1 | LW R1, 0(R20) | R1 | VR1 | LW VR1, 0(R20) |
| I2 | ADDI R2, R1, #1 | R2 | VR2 | ADDI VR2, VR1, #1 |
| I3 | SW 0(R20), R2 | | | SW 0(R20), VR2 |
| I4 | MUL R2, R3, R4 | R2 | VR3 | MUL VR3, Regs[R3], Regs[R4] |
| MUL overwrites R2 ⇒ retire VR2 | | | | |
| I5 | SUBI R3, R2, #3 | R3 | VR4 | SUBI VR4, VR3, #3 |
| I6 | XOR R2, R3, R4 | R2 | VR5 | XOR VR5, VR4, Regs[R4] |
| XOR overwrites R2 ⇒ retire VR3 | | | | |
| I1 and I2 complete ⇒ retire VR1<br>Commit Reg[R1] ← Reg[VR1] ⇒ R1 marked stable | | | | |
| I7 | AND R4, R1, R2 | R4 | VR6 | AND VR6, Regs[R1], VR5 |

# WAR and WAW Pseudo-Dependencies

## WAR and WAW data dependencies

```
I1   ADD R2, R1, R3      ; WAR hazard if I1 placed after I2 or I3
I2   ADD R1, R3, R4      ; WAW hazard if I2 placed after I3
I3   ADD R1, R3, R5
```

## Remove data dependencies

```
I1   ADD R2, R1, R3      ; allocate VR1 to R2
                         ; read stable values from R1, R3
I2   ADD R1, R3, R4      ; allocate VR2 to R1
                         ; read stable values from R3, R4
I3   ADD R1, R3, R5      ; allocate VR3 to R1
                         ; read stable values from R3, R5
```

## Prevent hazards

1. Read source values and allocate destination VRs in program order
2. Execute $I_1$, $I_2$, $I_3$ in any order
3. Retire **VR2** and **VR3** in program order

# Removing False Register Dependencies

## Virtual registers on superscalar CPU

| | Sequential code | Allocate | Rescheduled Code with Virtual Registers |
|---|---|---|---|
| CC1 | LW R1, [0x4] | VR1 ← R1 | VR1 (R1) ← [0x4] |
| CC2 | MUL R1, R1, R2 | VR2 ← R1 | VR2 (R1) ← VR1 (R1) * regs[R2]<br>VR4 (R1) ← [0x1234]<br>VR5 (R2) ← 22 |
| CC3 | SUBI R1, R1, 11 | VR3 ← R1 | VR3 (R1) ← VR2 (R1) - 11<br>VR6 (R1) ← VR4 (R1) AND 33 |
| CC4 | SW [0x4], R1 | | [0x4] ← VR3 (R1)<br>VR7 (R2) ← VR5 (R2) + VR6 (R1) |
| CC5 | LW R1, [0x1234]<br>ADDI R2, R0, 22 | VR4 ← R1<br>VR5 ← R2 | Retire in order<br>R1 ← VR3<br>R1 ← VR6<br>R2 ← VR7 |
| CC6 | AND R1, R1, 33 | VR6 ← R1 | |
| CC7 | ADD R2, R2, R1 | VR7 ← R2 | |

# Dynamic Scheduling

## Hardware performs rescheduling (out-of-order execution)

- Execute program in efficient order
  - Not in program listing order
- Evolution of Tomasulo algorithm (IBM 1967)

## Fast hardware decoder

- Decodes many instructions into an instruction queue
- Automatic register renaming with virtual registers

## Mark instructions with status fields

**NR** — Not Ready / **R** — Ready / **F** — Finished

## Execution

- Issue **R** instructions from instruction pool
- Return **F** instructions to instruction pool and update destination **VR**

## Retirement

- Update architectural state in original program order

# Tomasulo Algorithm

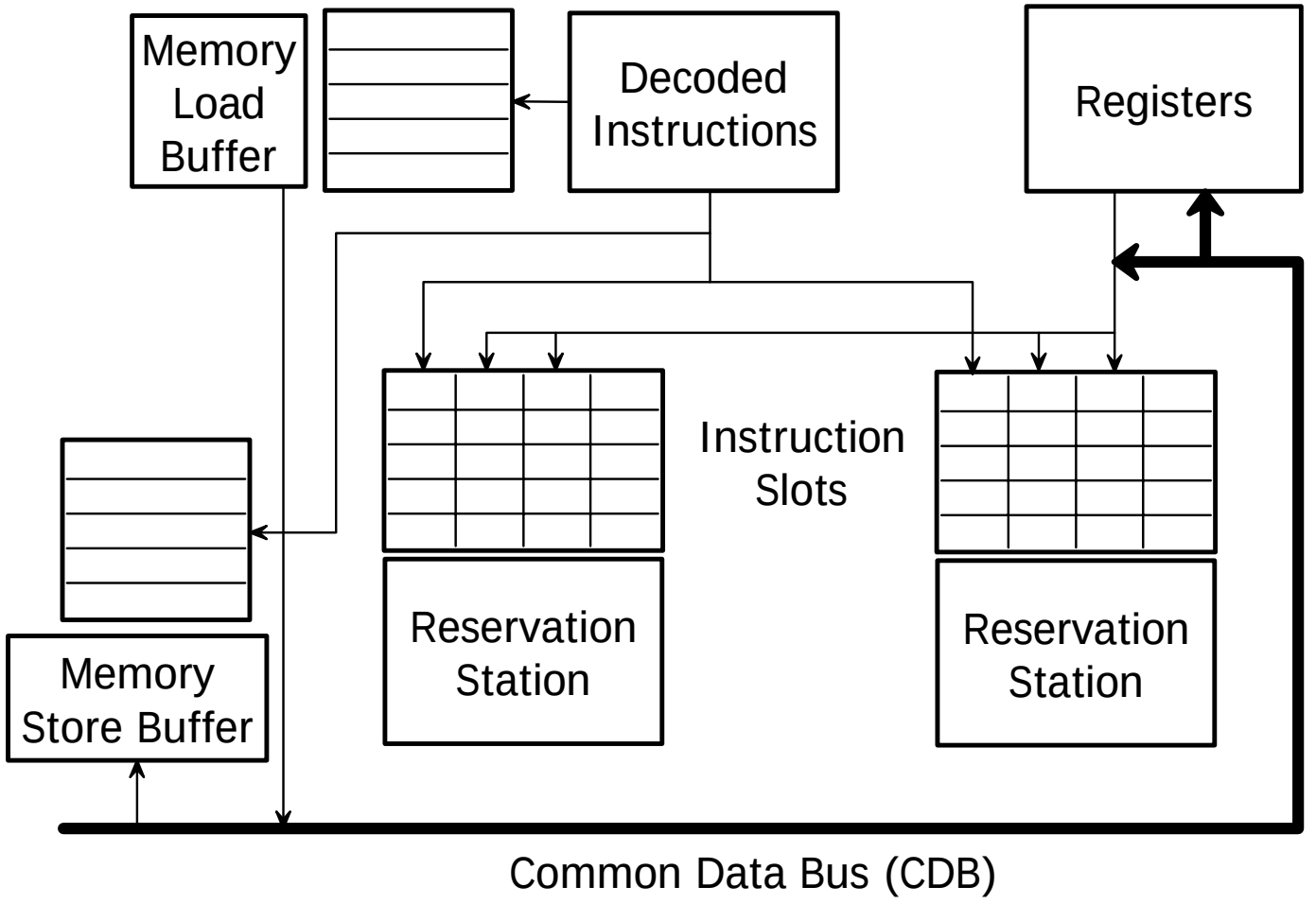
## Original dynamic rescheduling algorithm

Virtual register architecture

Automatic register renaming and out-of-order execution

Distributed control mechanism

Used in IBM 360/91 floating point unit (1969)




Ref: Robert Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, (1967) 11.

# Tomasulo Algorithm — General Description

## Reservation station (RS)

- Execution unit (EU) for particular operation type
- Virtual registers (VR)
- Load/store buffers — read/write reservation stations

## Decoder

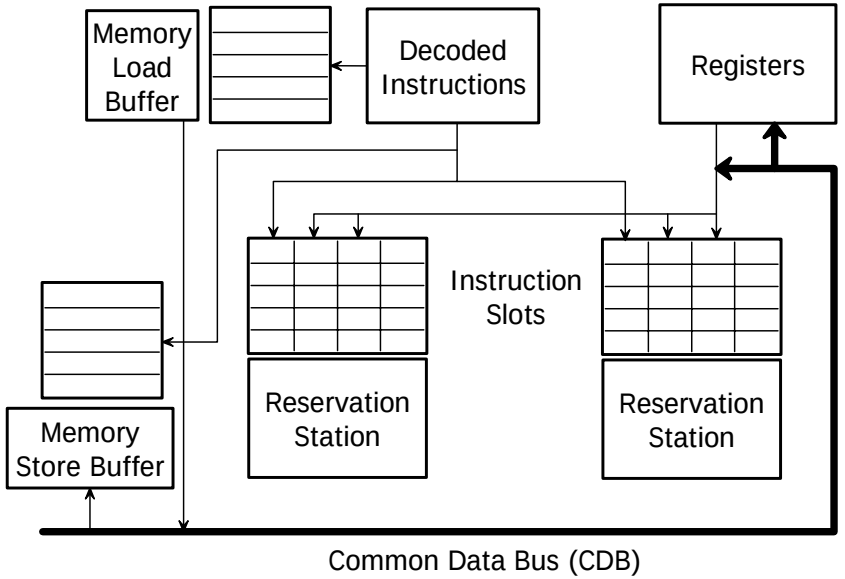
- Distributes instructions to Instruction Slot for reservation station
- Allocates VR to all destination operands (no RAT)
- Source operand
  - Ready at decode time $\Rightarrow$ operand = source value
  - Not ready $\Rightarrow$ operand = destination VR of unfinished instruction
    - VR = pointer to instruction in particular RS

## Execution order

- RS executes ready instructions
- Broadcasts VR pointers and result values on data bus (CDB)
- Instruction Slots update VR values from CDB

# Tomasulo Algorithm — Example

| Program | Decode | Allocation / Issue |
|---|---|---|
| `LW R1,[0x4]` | `VR1 (R1) ← [0x4]` | **Assign value at allocation** |
| `MUL R1, R1, R2` | `VR2 (R1) ← VR1 (R1) * regs[R2]` | Issue with values VR1, R2 |
| `SUBI R1, R1,0x22` | `VR3 (R1) ← VR2 (R1) - 0x22` | Issue with pointer VR2, imm |
| `SW [0x4],R1` | `[0x4] ← VR3 (R1)` | Issue with pointer VR3 |
| `LW R1,[0x33]` | `VR4 (R1) ← [0x33]` | **Assign value at allocation** |
| `ADDI R2, R0, 0x44` | `VR5 (R2) ← 0x44` | **Assign value at allocation** |
| `AND R1, R1, 0x7F` | `VR6 (R1) ← VR4 (R1) AND 0x7F` | Issue with value VR4, imm |
| `ADD R2, R2, R1` | `VR7 (R2) ← VR5 (R2) + VR6 (R1)` | Issue with pointers VR5, VR6 |



# Tomasulo Algorithm — Example

| Program | Decoding | CC-1 |
|---|---|---|
| `LW R1,[0x4]` | `VR1 (R1) ← [0x4]` | F |
| `MUL R1, R1, R2` | `VR2 (R1) ← VR1 (R1) * regs[R2]` | R |
| `SUBI R1, R1,0x22` | `VR3 (R1) ← VR2 (R1) – 0x22` | NR |
| `SW [0x4],R1` | `[0x4] ← VR3 (R1)` | NR |
| `LW R1,[0x33]` | `VR4 (R1) ← [0x33]` | F |
| `ADDI R2, R0, 0x44` | `VR5 (R2) ← 0x44` | F |
| `AND R1, R1, 0x7F` | `VR6 (R1) ← VR4 (R1) AND 0x7F` | R |
| `ADD R2, R2, R1` | `VR7 (R2) ← VR5 (R2) + VR6 (R1)` | NR |

## Assigned values

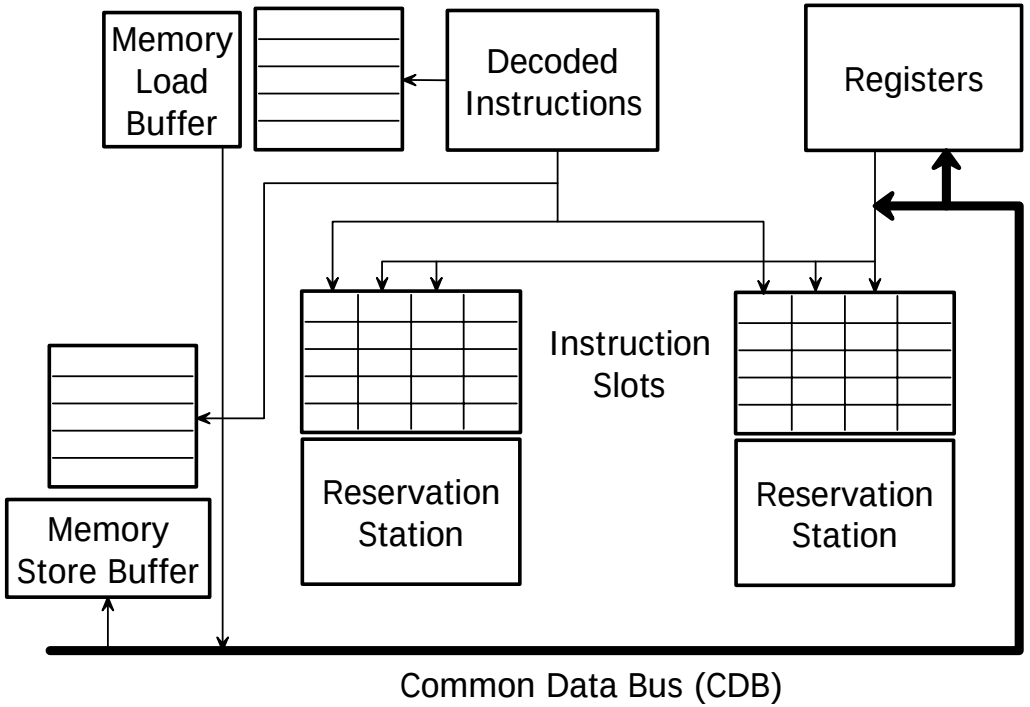
`VR1, VR4, VR5`

## Issued with values

`MUL, AND`

## Issued with pointers

`SUBI, SW, ADD`



# Tomasulo Algorithm — Example

| Program | Decoding | CC-2 |
|---|---|---|
| `LW R1,[0x4]` | `VR1 (R1) ← [0x4]` | F |
| `MUL R1, R1, R2` | `VR2 (R1) ← VR1 (R1) * regs[R2]` | F |
| `SUBI R1, R1,0x22` | `VR3 (R1) ← VR2 (R1) – 0x22` | R |
| `SW [0x4],R1` | `[0x4] ← VR3 (R1)` | NR |
| `LW R1,[0x33]` | `VR4 (R1) ← [0x33]` | F |
| `ADDI R2, R0, 0x44` | `VR5 (R2) ← 0x44` | F |
| `AND R1, R1, 0x7F` | `VR6 (R1) ← VR4 (R1) AND 0x7F` | F |
| `ADD R2, R2, R1` | `VR7 (R2) ← VR5 (R2) + VR6 (R1)` | R |

## Execute

`MUL, AND`

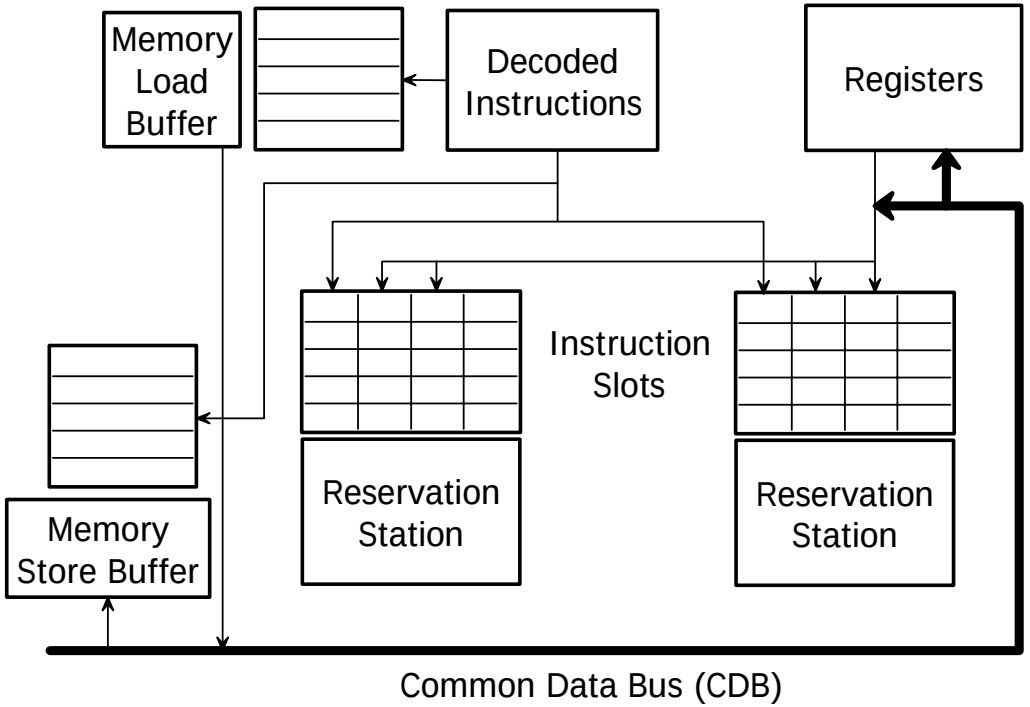
## Broadcast updates on CDB

`VR2, VR6`

## Replace pointers with values

`SUBI – VR2`

`ADD – VR6`



# Tomasulo Algorithm — Example

| Program | Decoding | CC-3 |
|---|---|---|
| `LW R1,[0x4]` | `VR1 (R1) ← [0x4]` | F |
| `MUL R1, R1, R2` | `VR2 (R1) ← VR1 (R1) * regs[R2]` | F |
| `SUBI R1, R1,0x22` | `VR3 (R1) ← VR2 (R1) – 0x22` | F |
| `SW [0x4],R1` | `[0x4] ← VR3 (R1)` | F |
| `LW R1,[0x33]` | `VR4 (R1) ← [0x33]` | F |
| `ADDI R2, R0, 0x44` | `VR5 (R2) ← 0x44` | F |
| `AND R1, R1, 0x7F` | `VR6 (R1) ← VR4 (R1) AND 0x7F` | F |
| `ADD R2, R2, R1` | `VR7 (R2) ← VR5 (R2) + VR6 (R1)` | F |

## Execute

**SUBI, ADD, SW**

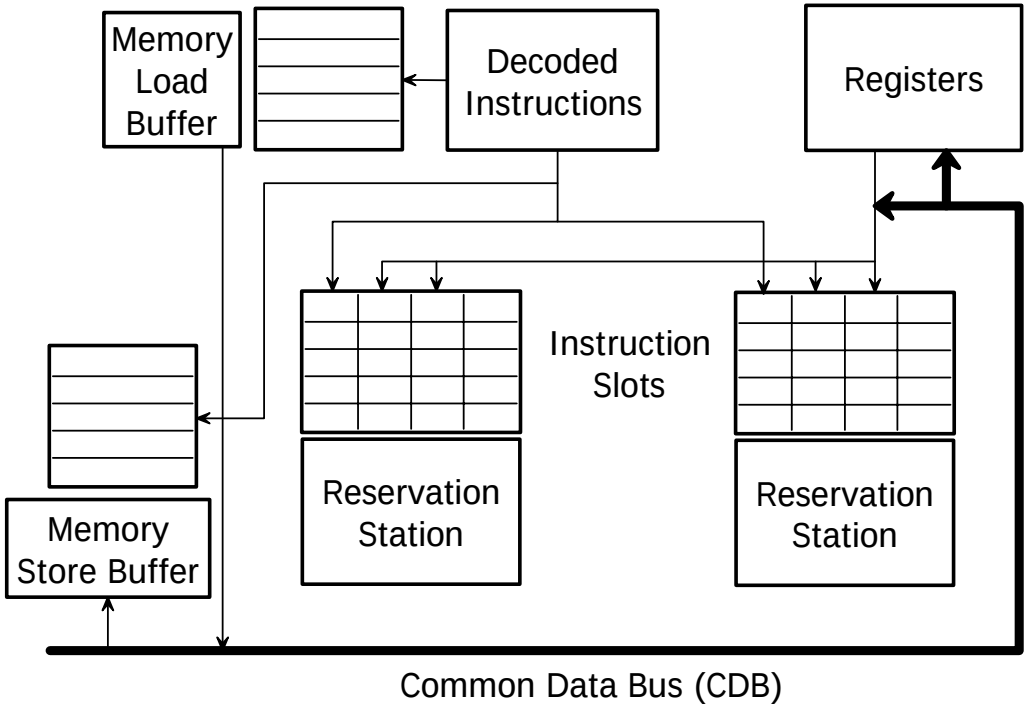
## Broadcast updates on CDB

**VR3, VR7**

## Replace pointers with values

**[0x4] – VR3**

**R2 – VR7**



# Issues in Tomasulo Algorithm

## Scheduling similar to scoreboarding

- Instructions execute when source operands are ready

## Distributed control simplifies implementation

- Automatic register renaming
- Decoder keeps list of allocated ARs
- No centralized register allocation table (RAT)
  - RAT records all VR allocations
- No central scoreboard controller

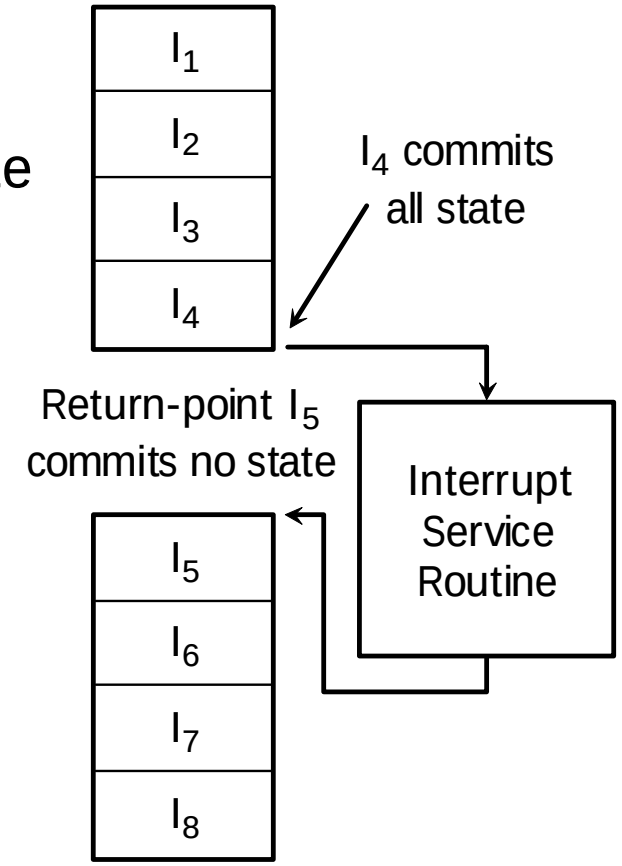
## Difficulties with Tomasulo algorithm

- Common Data Bus $\rightarrow$ communication bottleneck
- Instructions complete out-of-order
  - Instructions following conditional branch — **NR**
    - No way to handle branch mispredictions $\Rightarrow$ no branch prediction
  - No well-defined return-point between instructions
    - No precise breakpoint for exceptions and interrupts

# No Precise Exceptions in Tomasulo Machine

## Return-point

- Follows atomic operation
- Previous operations commit all results to state
- No following operations commit any results to state




## Precise exception

- Exception with well-defined return-point
- Service exception following atomic operation
- Restart execution at return point without error

## In Tomasulo algorithm

- PC advances after fetching instruction
- Complete in order of execution (out-of-order)
  - Broadcast finished results on CDB
- No return-point
  - PC points past instructions that have committed to state
  - Some instructions have committed state before prior instructions that have not

# In-Order Retirement

## Set retirement order at decode time

- Retirement order = program order (PC)
  - Instructions commit to system state in retirement order
- Program progress index = retirement order
  - Enables speculative execution — issue beyond undetermined branch
  - Enables precise exceptions

## Retirement queue indexed by retirement order

- Instruction → retirement queue when
  - Marked as finished
  - Verified as valid — no errors / no exceptions / no mispredictions

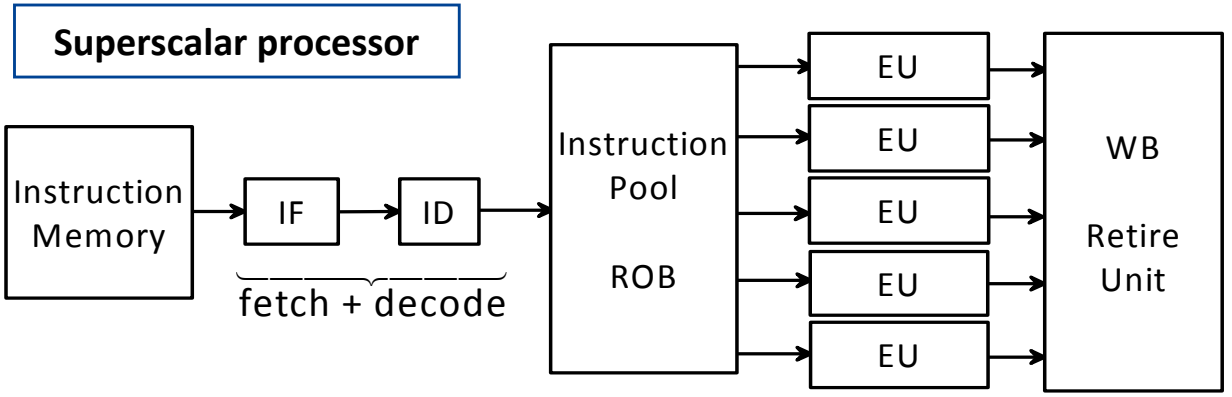
## On exception / misprediction

- Identify last valid instruction
- Cancel instructions with later retirement order
- Exception
  - Branch to exception handler
  - Return point = next instruction following last valid

# Dynamic Scheduling — Tomasulo + In-Order Retirement

## Reorder Buffer (ROB)

Instruction pool



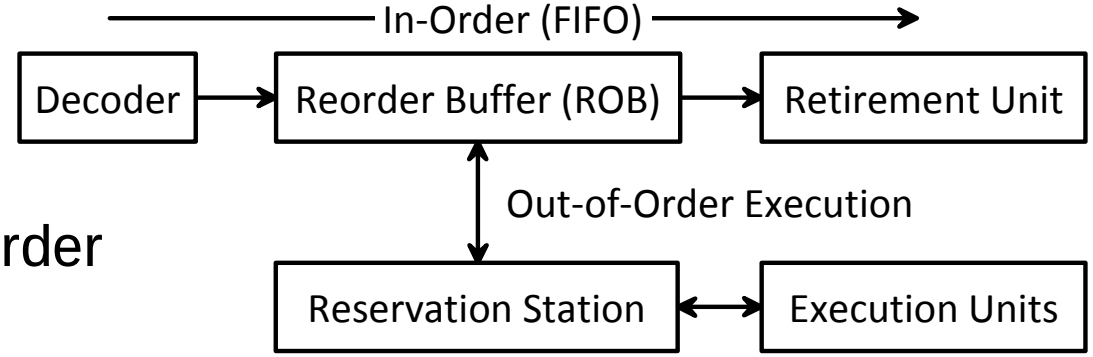

## ROB entry

| Instruction PC | Operation | Status | VR pointers + stable values |
|---|---|---|---|

## In-Order (FIFO) path

Decode → ROB → Retire

Retirement Unit commits in PC order




## Out-of-Order execution path

ROB ↔ Execution Units

RS issues Ready instructions to EU

EU updates VR pointers + status fields in ROB entry

# In-Order Completion Issues

## Reorder buffer performs data forwarding

```
I_k   ADD R1, R2, R3
I_k+j ADD R4, R5, R1
```

| $I_k$ condition at $I_{k+j}$ decode time | Decoded **R1** value for $I_{k+j}$ entry in ROB |
|---|---|
| Retired | Value of **R1** from architectural register |
| Finished but not retired | Value of $I_k$ destination entry |
| Not finished | source **VR** in $I_{k+j}$ entry = destination **VR** in $I_k$ entry |

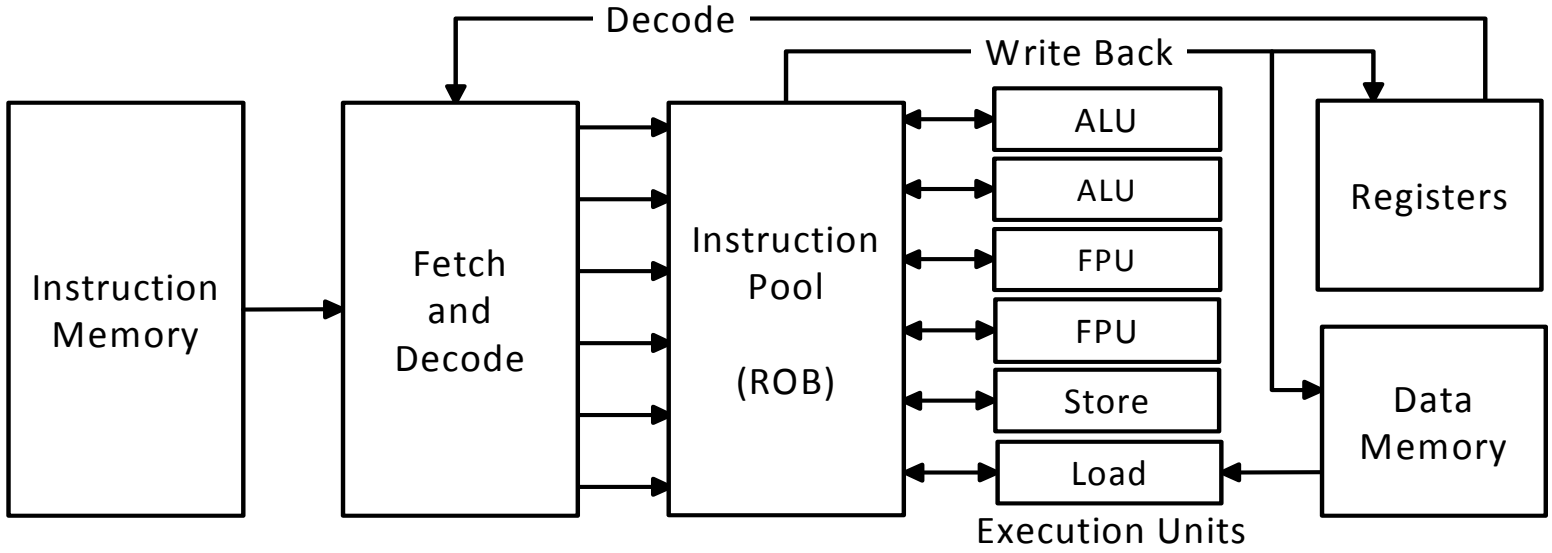
## On exception

Cancel invalid instructions in ROB

Finished instructions following exception

New decoded instructions moved into ROB

# Intel x86 with Dynamic Scheduling




**Intel meta-architecture for Pentium II, III, 4, Multicore, ...**

RISC machine with IA-32 CISC adaptation layer

**Fetch + Decode**

Converts IA-32 instructions to 1 – 6 RISC micro-ops per CC

**Dynamic scheduling (out-if-order execution)**

ROB holds micro-ops until ready for execution

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

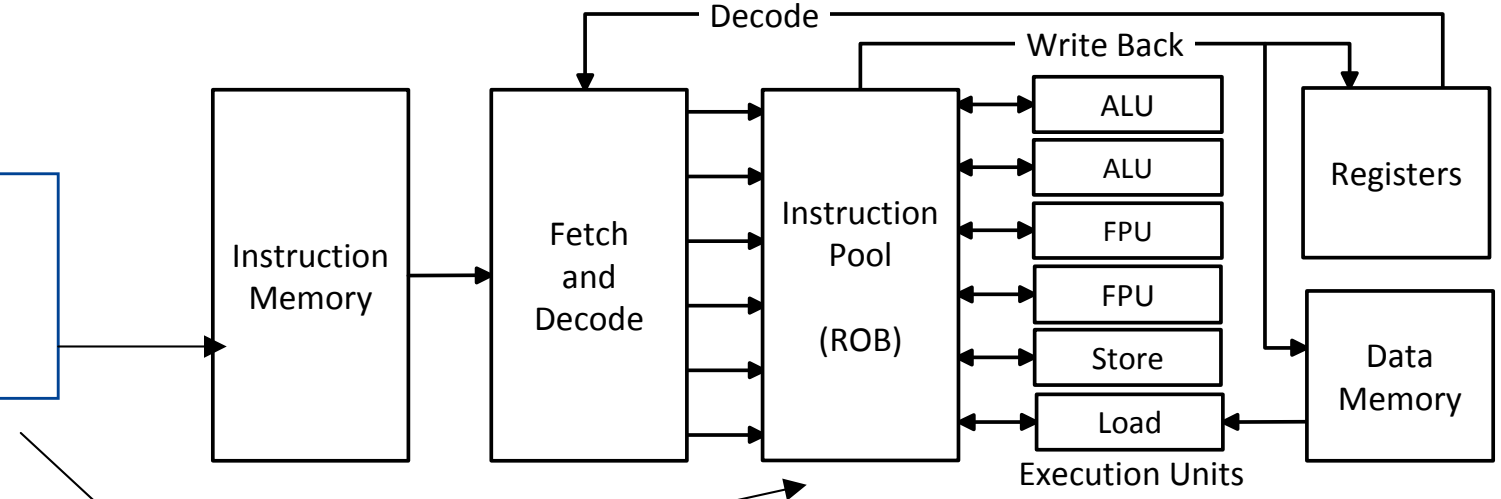
Finished micro-ops update execution results to instruction pool

**Retirement (in-order write back)**

Finished micro-ops commit in original program order

# Program Execution in x86 with Dynamic Scheduling

```
ADD [X],123
SUB [Y],456
SUB [Z],789
```




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

```
LW   VR2,[X]
ADD  VR3,VR2,#123
SW   [X],VR3
LW   VR4,[Y]
SUB  VR5,VR4,#567
SW   [Y],VR5
LW   VR6,[Z]
SUB  VR7,VR6,#789
SW   [Z],VR7
```

Dynamic scheduling

| | | |
|---|---|---|
| `LW VR2,[X]` | Load | CC1 |
| `ADD VR3,VR2,#123`<br>`LW VR4,[Y]` | ALU<br>Load | CC2 |
| `SW [X],R3`<br>`SUB VR5,VR4,#567`<br>`LW VR6,[Z]` | Store<br>ALU<br>Load | CC3 |
| `SW [Y],VR5`<br>`SUB VR7,VR6,#789` | Store<br>ALU | CC4 |
| `SW [Z],VR7` | Store | CC5 |

# Scoreboard Example x86 with Dynamic Scheduling

## Execution condition after each CC (ignoring fetch, decode, retire)

| Instruction | Execution Unit | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 |
|---|---|---|---|---|---|---|---|
| `LW VR2,[X]` | Load | **F** | F | F | F | F | |
| `ADD VR3,VR2,#123` | ALU | R | **F** | F | F | F | |
| `SW [X],VR3` | Store | NR | R | **F** | F | F | |
| `LW VR4,[Y]` | Load | R | **F** | F | F | F | |
| `SUB VR5,VR4,#456` | ALU | NR | R | **F** | F | F | |
| `SW [Y],VR5` | Store | NR | NR | R | **F** | F | |
| `LW VR6,[Z]` | Load | R | R | **F** | F | F | |
| `SUB VR7,VR6,#789` | ALU | NR | NR | R | **F** | F | |
| `SW [Z],VR7` | Store | NR | NR | NR | R | **F** | |

## Scheduling rules

Issued only ready instructions

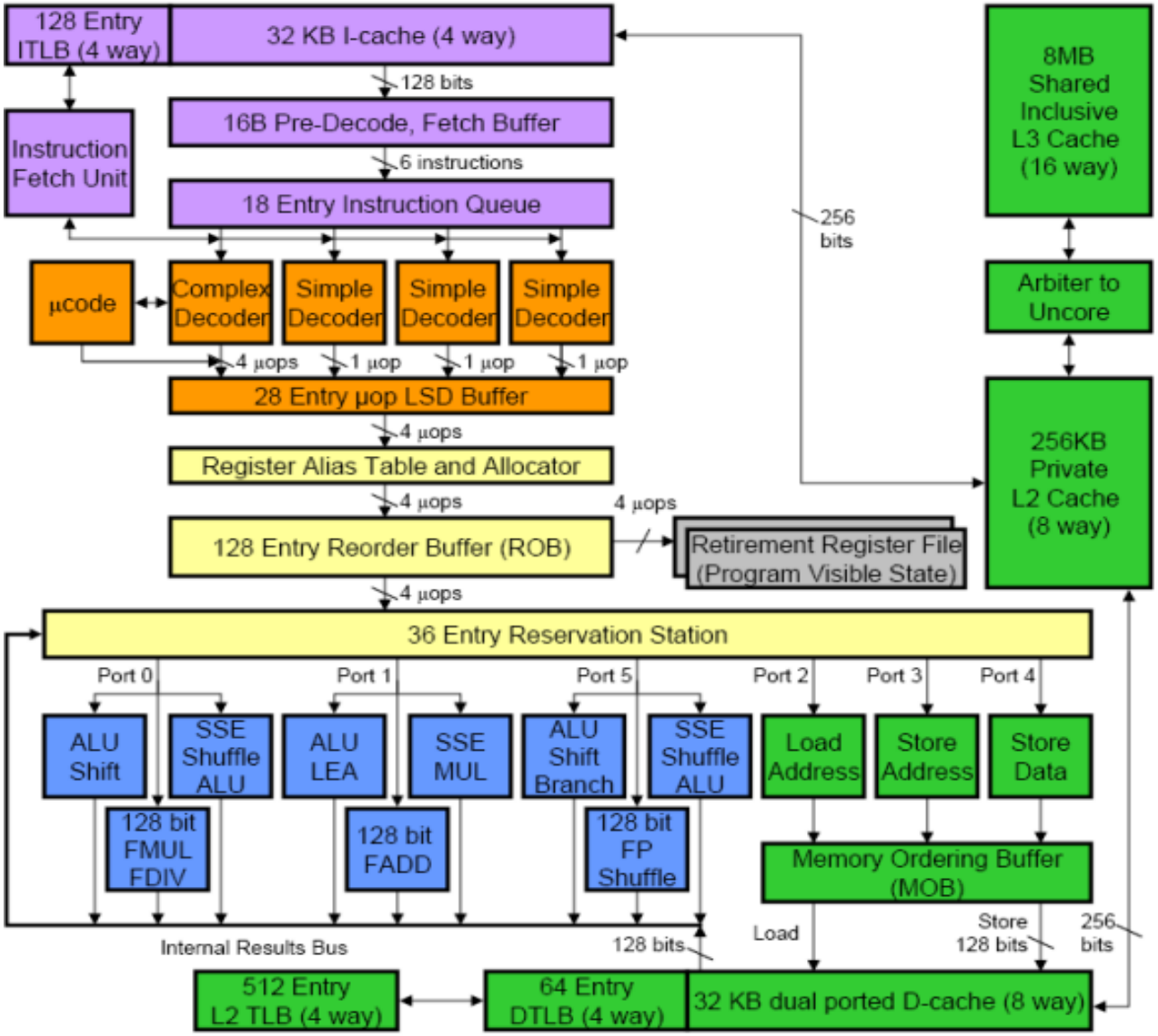
Among ready instructions, maintain program list order

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 |

# Intel Nehalem Micro-Architecture




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

# Deep Superpipeline

## Each pipeline stage → 2 smaller stages

New stage does ½ work in ½ time

New stage finishes in ½ time ⇒ clock speed → 2 × clock speed

## Deep superpipeline for DLX

| | 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 |

$$2 \times \text{clock speed} \Rightarrow \tau_{\text{superpipeline}} = \frac{\tau_{\text{pipeline}}}{2}$$

$$CPI^{\text{ideal}}_{\text{superpipeline}} = \frac{IC + 9}{IC} \xrightarrow[IC \to \text{large}]{} 1$$

$$T^{\text{ideal}}_{\text{superpipeline}} = CPI^{\text{ideal}}_{\text{superpipeline}} \times IC \times \frac{\tau_{\text{pipeline}}}{2} \xrightarrow[IC \to \text{large}]{} IC \times \frac{\tau_{\text{pipeline}}}{2} = \frac{1}{2} T \times^{\text{ideal}}_{\text{pipeline}}$$

## Problems with deep superpipeline

Cannot effectively split all instructions

Some operations do not scale — require fixed time

Page fault, branch penalty, cache update, etc

Faster clock ⇒ more stall cycles ⇒ larger $CPI^{\text{stall}}$

# Pentium 4 Superpipeline

## Pentium III

10 stage pipeline — clock speed ≤ 1.5 GHz

## Pentium 4

20 stage pipeline — clock speed ≤ 4.0 GHz

## Expect

1.5 GHz processor is 50% faster than same processor at 1.0 GHz

## Measurement on SPEC CINT2000

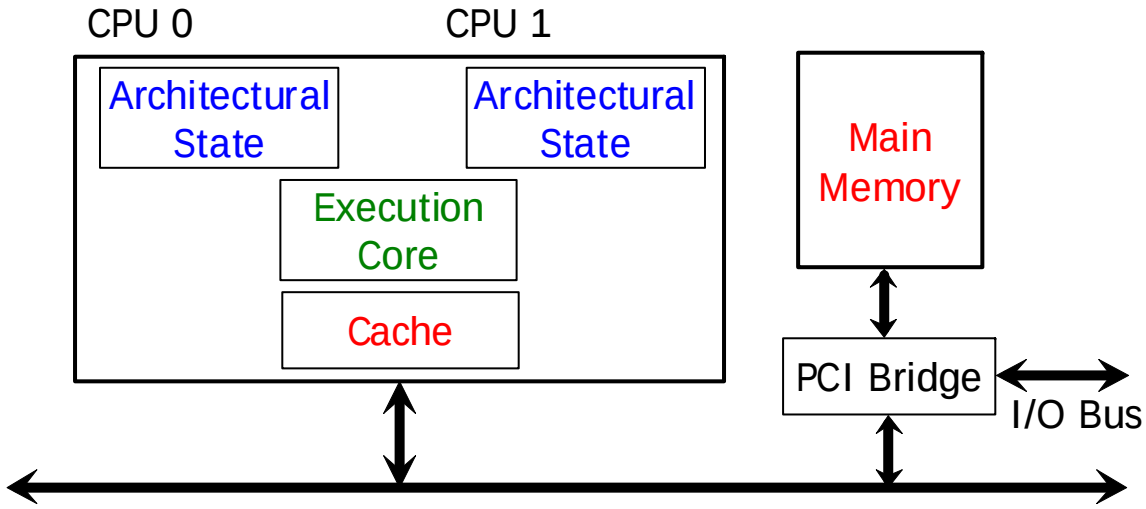
1.5 GHz Pentium-4 is 20% faster than 1.0 GHz Pentium-III

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

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

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

# 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 registers sets — 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 data hazard in either thread
  - CPU 0 and CPU 1 issue instructions on alternate clock cycles
- Data hazard 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

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

$CPS = $ cycles per stall

## With hyper - threading

$$CPI^{\text{stall HT}} = \frac{\text{stall cycles}}{\text{stall}} \times P(\text{2 simultaneous stalls}) \approx CPS \times [P_s]^2$$

## Speedup

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

## Take for Pentium - 4

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

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

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

Measured improvement $= 1.2$

Intel, "Hyper-Threading Technology Architecture and Microarchitecture"

# Hardware Utilization in x86 Example

| Unit | CC1 | CC2 | CC3 | CC4 | CC5 |
|---|---|---|---|---|---|
| ALU | IDLE | ADD R3,R2,#123 | SUB R5,R4,#567 | SUB R7,R6,#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 R4,[Y] | LW R6,[Z] | IDLE | IDLE |
| Store | IDLE | IDLE | SW [X],R3 | SW [Y],R5 | SW [Y],R7 |

**6 EUs in x86 example**

2 integer ALUs, 2 FPUs, 1 load unit, 1 store unit

**Most units idle in most clock cycles**

CC1 — 1 EU active

CC2 — 2 EUs active

CC3 — 3 EUs active

CC4 — 2 EUs active

CC5 — 1 EU active

**Increased utilization ⇒ more parallel instructions**

# Overhead in Performance Calculations

## Typical program structure

| | | |
|---|---|---|
| **Head** | setup instructions | run-once |
| **Main functions** | program loop instructions | large number of iterations |
| **Tail** | shutdown instructions | run-once |

## Dynamic Scheduling Superscalar with u Execution Units

| `Fetch + Decode stages` | `Execute in EU_1` | `Retire stages` |
|---|---|---|
| Overhead cycles | `Execute in EU_2` | Overhead cycles |
| Fills Instruction Pool | … | Retires Instructions |
| Independent of IC | `Execute in EU_u` | Independent of IC |

## Pipelining overlaps fetch / decode / retire with execution in EUs

$$N = N_{IF+ID} + N_{program} + N_{retirement} = N_{IF+ID} + N_{head} + N_{main\ loops} + N_{tail} + N_{retirement}$$

$$= N_{overhead} + N_{main\ loops}$$

where $N_{overhead} = N_{IF+ID} + N_{head} + N_{tail} + N_{retirement}$

# CPI Loop Approximation in Superscalar

## General approximation

$$CPI = \frac{N_{\text{overhead}} + n_{\text{iterations}} \times N_{\text{one-loop}}}{IC_{\text{overhead}} + n_{\text{iterations}} \times IC_{\text{one-loop}}} = \frac{\frac{N_{\text{overhead}}}{n_{\text{iterations}}} + N_{\text{one-loop}}}{\frac{IC_{\text{overhead}}}{n_{\text{iterations}}} + IC_{\text{one-loop}}}$$

$$CPI \xrightarrow[n_{\text{iterations}} \to \text{large}]{} \frac{N_{\text{one-loop}}}{IC_{\text{one-loop}}} = CPI_{\text{one-loop}}$$

## Ideal case

$$N_{\text{one-loop}} = \frac{N_{\text{one-loop}}^{u=1}}{u}$$

$$CPI = \frac{1}{u}\frac{N_{\text{one-loop}}^{u=1}}{IC_{\text{one-loop}}} = \frac{1}{u}CPI_{\text{ideal}}^{u=1} = \frac{1}{u}$$

# Loop Performance Example

## Loop runs 0x1000/4 = 0x400 = 1024 iterations

| | | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|
| ADDI R1, R0, #1000 | EX | | | | | |
| L1: SUBI R1, R1, #4 | | EX | | | | 1 |
| LW R2, 0(R1) | | | EX | | | 2 |
| ADDI R2, R2, #1 | | | | EX | | 3 |
| SW 0(R1), R2 | | | | | EX | 4 |
| BNEZ R1, L1 | | | | | EX | 5 |
| overhead = fetch + decode + 1 CC + retire | | loop = 4 CC | | | | |

$$CPI = \frac{overhead + 4 \times 1024}{1 + 5 \times 1024} \rightarrow 0.80$$

# Loop Unrolling

## Iterations of loop instructions create data dependencies

| Successive iterations use same registers<br>Cannot run in parallel | Replicate instructions over shorter loop<br>Register renaming<br>Scheduled for parallel execution |
|---|---|
| `ADDI R1, R0, #1000`<br>`L1: SUBI R1, R1, #4`<br>`LW R2, 0(R1)`<br>`ADDI R2, R2, #1`<br>`SW 0(R1), R2`<br>`BNEZ R1, L1` | `ADDI R1, R0, #1000`<br>`L1: SUBI R1, R1, #10`<br>`LW R2, 0(R1)`<br>`LW R3, 4(R1)`<br>`LW R4, 8(R1)`<br>`LW R5, C(R1)`<br>`ADDI R2, R2, #1`<br>`ADDI R3, R3, #1`<br>`ADDI R4, R4, #1`<br>`ADDI R5, R5, #1`<br>`SW 0(R1), R2`<br>`SW 4(R1), R3`<br>`SW 8(R1), R4`<br>`SW C(R1), R5`<br>`BNEZ R1, L1` |

# Loop Unrolling Performance — 1

## EUs: 2 ALUs + 1 Read + 1 Write

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|---|
| ADDI R1, R0, #1000 | F | | | | | | | | |
| SUBI R1, R1, #10 | R | F | F | F | F | F | F | F | 1 |
| LW R2, 0(R1) | NR | R | F | F | F | F | F | F | 2 |
| LW R3, 4(R1) | NR | R | R | F | F | F | F | F | 3 |
| LW R4, 8(R1) | NR | R | R | R | F | F | F | F | 4 |
| LW R5, C(R1) | NR | R | R | R | R | F | F | F | 5 |
| ADDI R2, R2, #1 | NR | NR | R | F | F | F | F | F | 6 |
| ADDI R3, R3, #1 | NR | NR | NR | R | F | F | F | F | 7 |
| ADDI R4, R4, #1 | NR | NR | NR | NR | R | F | F | F | 8 |
| ADDI R5, R5, #1 | NR | NR | NR | NR | NR | R | F | F | 9 |
| SW 0(R1), R2 | NR | NR | NR | R | F | F | F | F | 10 |
| SW 4(R1), R3 | NR | NR | NR | NR | R | F | F | F | 11 |
| SW 8(R1), R4 | NR | NR | NR | NR | NR | R | F | F | 12 |
| SW C(R1), R5 | NR | NR | NR | NR | NR | NR | R | F | 13 |
| BNEZ R1, L1 | NR | R | R | R | R | R | R | F | 14 |

$$CPI = \frac{\text{overhead} + N_{\text{1-loop}} \times N_{\text{loop iterations}}}{IC_{\text{head}} + IC_{\text{1-loop}} \times N_{\text{loop iterations}} + IC_{\text{tail}}} = \frac{\text{overhead} + 7 \times 256}{1 + 14 \times 256 + 0} = 0.50$$

# Loop Unrolling Performance — 2

## EUs: 4 ALUs + 4 Read + 4 Write

| | | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|
| ADDI R1, R0, #1000 | F | | | | | |
| SUBI R1, R1, #10 | R | F | F | F | F | 1 |
| LW R2, 0(R1) | NR | R | F | F | F | 2 |
| LW R3, 4(R1) | NR | R | F | F | F | 3 |
| LW R4, 8(R1) | NR | R | F | F | F | 4 |
| LW R5, C(R1) | NR | R | F | F | R | 5 |
| ADDI R2, R2, #1 | NR | NR | R | F | F | 6 |
| ADDI R3, R3, #1 | NR | NR | R | F | F | 7 |
| ADDI R4, R4, #1 | NR | NR | R | F | R | 8 |
| ADDI R5, R5, #1 | NR | NR | R | F | NR | 9 |
| SW 0(R1), R2 | NR | NR | NR | R | F | 10 |
| SW 4(R1), R3 | NR | NR | NR | R | F | 11 |
| SW 8(R1), R4 | NR | NR | NR | R | F | 12 |
| SW C(R1), R5 | NR | NR | NR | R | F | 13 |
| BNEZ R1, L1 | NR | R | R | R | F | 14 |

$$CPI = \frac{\text{overhead} + N_{\text{1-loop}} \times N_{\text{loop iterations}}}{IC_{\text{head}} + IC_{\text{1-loop}} \times N_{\text{loop iterations}} + IC_{\text{tail}}} = \frac{\text{overhead} + 4 \times 256}{1 + 14 \times 256 + 0} = 0.29$$

# Loop Unrolling with x86 VR Renaming — 1

## Simple x86 code fragment

Sets counter , loads word to register, updates register from table, stores word in register, loops while counter > 0

```
     MOV ECX, 400h          ; reg[ECX] ← 400h = 1K = 1024
L1:  LODSD                  ; reg[EAX] ← mem[reg[ESI]]
                            ; reg[ESI] ← reg[ESI] + 4
     ADD EAX,[EBX + ESI]    ; reg[EAX] ← reg[EAX] + table
                            ; entry at mem[reg[EBX]+reg[ESI]]
     STOSD                  ; mem[EDI] ← reg[EAX]
                            ; reg[EDI] ← reg[EDI] + 4
     LOOP L1                ; reg[ECX] ← reg[ECX] - 1
                            ; branch to L1 if reg[ECX] > 0
```

# Loop Unrolling with x86 VR Renaming — 2

| x86 Machine Instruction | VR Allocation | Recoded Instruction |
|---|---|---|
| ECX ← 400h = 1K = 1024 | VR1 ← ECX | VR1 ← 400h |
| L1: EAX ← (ESI)<br>ESI ← ESI + 4 | VR2 ← EAX<br>VR3 ← ESI | VR2 ← (reg[ESI])<br>VR3 ← reg[ESI] + 4 |
| EAX ← EAX + (EBX+ESI) | VR4 ← EAX | VR4 ← VR2 + (reg[EBX]+VR3) |
| (EDI) ← EAX<br>EDI ← EDI + 4 | VR5 ← EDI | (reg[EDI]) ← VR4<br>VR5 ← reg[EDI] + 4 |
| | | ESI ← VR3 // commit VR3<br>EDI ← VR5 // commit VR5 |
| ECX ← ECX – 1<br>branch L1 if ECX > 0 | VR9 ← ECX | VR9 ← VR1 – 1<br>branch L1 if VR9 > 0 |

# Loop Unrolling with x86 VR Renaming — 3

## Loop unrolling

Define two parallel pointers for read pointer

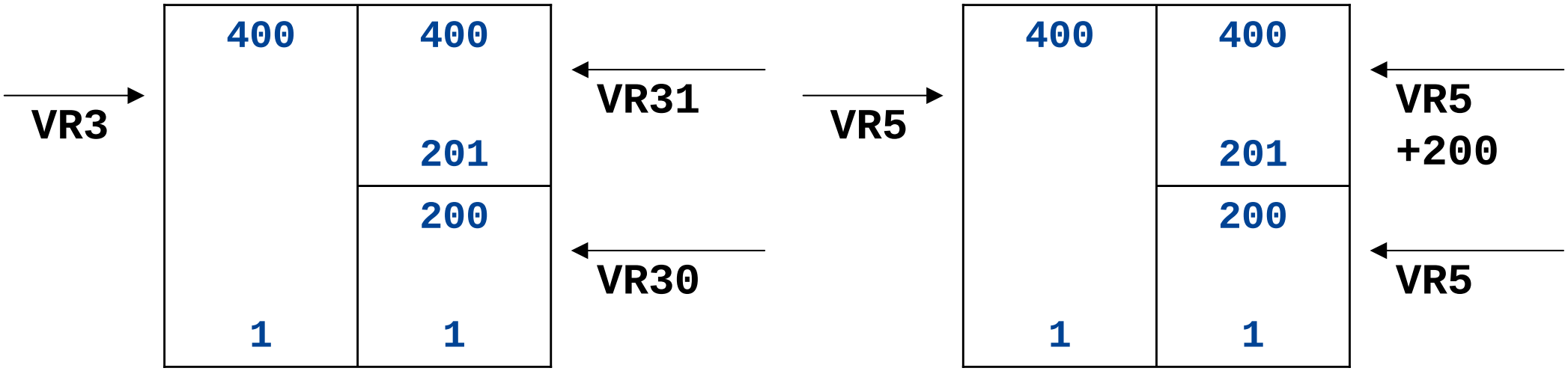
**VR3 ← ESI ⇒ VR30 ← ESI  VR31 ← ESI + 200**

Perform two ALU operations per loop

Enhance EU utilization

Half number of loop iterations

Program runs twice as fast



# Loop Unrolling with x86 VR Renaming — 4

| | |
|---|---|
| VR1 ← 400h | VR1 ← 200h |
| L1: VR2 ← (reg[ESI]) | L1: VR20 ← (reg[ESI]) |
| | VR21 ← (reg[ESI]+200) |
| VR3 ← reg[ESI] + 4 | VR30 ← reg[ESI] + 4 |
| | VR31 ← reg[ESI] + 204 |
| VR4 ← VR2 + (reg[EBX]+VR3) | VR40 ← VR20 + (reg[EBX]+VR30) |
| | VR41 ← VR21 + (reg[EBX]+VR31) |
| (reg[EDI]) ← VR4 | (reg[EDI]) ← VR40 |
| | (reg[EDI]+200) ← VR41 |
| VR5 ← reg[EDI] + 4 | VR5 ← reg[EDI] + 4 |
| ESI ← VR3 // commit VR3<br>EDI ← VR5 // commit VR5 | ESI ← VR30 // commit VR30<br>EDI ← VR5 // commit VR5 |
| VR9 ← VR1 – 1<br>branch L1 if VR9 > 0 | VR9 ← VR1 – 1<br>branch L1 if VR9 > 0 |

# Loop Unrolling with Automatic VR Renaming in x86 — 3

## Loop before unrolling

4 CC per loop iteration

1024 / 4 = 256 iterations

## EUs: 4 ALUs + 4 Read + 4 Write

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| L1: VR2 ← (reg[ESI]) | F | F | F | F |
| VR3 ← reg[ESI] + 4 | F | F | F | F |
| VR4 ← VR2 + (reg[EBX]+VR3) | R | F | F | F |
| (reg[EDI]) ← VR4 | NR | R | F | F |
| VR5 ← reg[EDI] + 4 | NR | R | F | F |
| ESI ← VR3 // commit VR3<br>EDI ← VR5 // commit VR5 | NR | NR | R | F |
| VR9 ← VR1 – 1<br>branch L1 if VR9 > 0 | NR | NR | R | F |

# Loop Unrolling with Automatic VR Renaming in x86 — 4

## Loop before unrolling

4 CC per loop iteration

512 / 4 = 128 iterations ⇒ half the run time

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| L1: VR20 ← (reg[ESI]) | F | F | F | F |
| VR21 ← (reg[ESI]+200) | F | F | F | F |
| VR30 ← reg[ESI] + 4 | F | F | F | F |
| VR31 ← reg[ESI] + 204 | F | F | F | F |
| VR40 ← VR20 + (reg[EBX]+VR30) | R | F | F | F |
| VR41 ← VR21 + (reg[EBX]+VR31) | R | F | F | F |
| (reg[EDI]) ← VR40 | NR | R | F | F |
| (reg[EDI]+200) ← VR41 | NR | R | F | F |
| VR5 ← reg[EDI] + 4 | NR | R | F | F |
| ESI ← VR30 // commit VR3<br>EDI ← VR50 // commit VR5 | NR | NR | R | F |
| VR9 ← VR1 – 1<br>branch L1 if VR9 > 0 | NR | NR | R | F |

# Software Pipelining

## Simple sequential loop

```
R2 ← mem[index]
R3 ← ALU(R2, literal)
mem[index] ← R3
index ← index - offset
```

## Pipelined version

Header

```
Initialize old_R2, old_R3, index
```

Loop

```
mem[index + offset] ← old_R3
new_R2 ← mem[index - offset]
new_R3 ← ALU(old_R2, literal)
new_index ← ALU(index, literal)
```

All loop instructions run in parallel

During loop iteration N

- All source reads were written during iteration N-1
- All destinations are read only during iteration N+1

## Parallel read / write

Reads performed at start of clock cycle

Writes performed at end of clock cycle

Reads/Writes in same clock cycle are independent

# Software Pipelining Example — 1

| Sequential | Pipelined |
|---|---|
| `ADDI R1, R0, #1000 ;;` | `head: ADDI R1, R0, #1000 ;;` |
| `L1: SUBI R1, R1, #4 ;;` | `SUBI R1, R1, #4 ;;` |
| `LW R2, 0(R1) ;;` | `LW R2, 0(R1) ;;` |
| `ADDI R3, R2, #1 ;;` | `ADDI R3, R2, #1`<br>`LW R2, -4(R1)`<br>`SUBI R1, R1, #4 ;;` |
| `SW 0(R1), R3 ;;` | |
| `BNEZ R1, L1 ;;` | // first loop iteration uses source values set in head<br>// iteration N+1 uses source values set in iteration N |
| Software pipelining requires explicit parallelization support in CPU hardware<br>Compiler marks groups of instructions to run in parallel<br>symbol `;;` marks end of parallel instruction group | `L1: SUBI R1, R1, #4`<br>`SW 4(R1), R3`<br>`ADDI R3, R2, #1`<br>`LW R2, -4(R1)`<br>`BNEZ R1, L1 ;;` |
| | `tail: SW 0(R1), R3` |

# Software Pipelining Example — 2

| Pipelined | | R1 | R2 | R3 | Read | Write |
|---|---|---|---|---|---|---|
| head: | ADDI R1, R0, #1000 | 1000 | — | — | — | — |
| | SUBI R1, R1, #4 | 996 | — | — | — | — |
| | LW R2, 0(R1) | | (996) | — | 996 | — |
| | ADDI R3, R2, #1 | | | (996)+1 | | — |
| | LW R2, -4(R1) | | (992) | | 992 | — |
| | SUBI R1, R1, #4 | 992 | | | | |
| L1: | SUBI R1, R1, #4 | 988 | | | | |
| | SW 4(R1), R3 | | | | | 996 |
| | ADDI R3, R2, #1 | | | (992)+1 | | |
| | LW R2, -4(R1) | | (988) | | | |
| | BNEZ R1, L1 | | | | | |
| L1: | SUBI R1, R1, #4 | 984 | | | | |
| | SW 4(R1), R3 | | | | | 992 |
| | ADDI R3, R2, #1 | | | (988)+1 | | |
| | LW R2, -4(R1) | | (984) | | | |
| | BNEZ R1, L1 | | | | | |

# Software Pipelining Example — 3

| Pipelined | | Action |
|---|---|---|
| head: | ADDI R1, R0, #1000 | R1 ← 1000 |
| | SUBI R1, R1, #4 | R1 ← 1000-4 = 996 |
| | LW R2, 0(R1) | R2 ← (996) |
| | ADDI R3, R2, #1 | R3 ← (996)+1 |
| | LW R2, -4(R1) | R2 ← (996-4) = (992) |
| | SUBI R1, R1, #4 | R1 ← 996-4 = 992 |
| L1: | SUBI R1, R1, #4 | R1 ← 992-4 = 988 |
| | SW 4(R1), R3 | (992+4) = (996) ← (996)+1 |
| | ADDI R3, R2, #1 | R3 ← (992)+1 |
| | LW R2, -4(R1) | R2 ← (992-4) = (988) |
| | BNEZ R1, L1 | BNEZ 992, L1 |
| L1: | SUBI R1, R1, #4 | R1 ← 988-4 = 984 |
| | SW 4(R1), R3 | (988+4) = (992) ← (992)+1 |
| | ADDI R3, R2, #1 | R3 ← (988)+1 |
| | LW R2, -4(R1) | R2 ← (992-4) = (988) |
| | BNEZ R1, L1 | BNEZ 988, L1 |

# Software Pipelining Example — 4

| Pipelined | | Action |
|---|---|---|
| head: | ADDI R1, R0, #12 | R1 ← 12 |
| | SUBI R1, R1, #4 | R1 ← 12-4 = 8 |
| | LW R2, 0(R1) | R2 ← (8) |
| | ADDI R3, R2, #1 | R3 ← (8)+1 |
| | LW R2, -4(R1) | R2 ← (8-4) = (4) |
| | SUBI R1, R1, #4 | R1 ← 8-4 = 4 |
| L1: | SUBI R1, R1, #4 | R1 ← 4-4 = 0 |
| | SW 4(R1), R3 | (4+4) = (8) ← (8)+1 |
| | ADDI R3, R2, #1 | R3 ← (4)+1 |
| | LW R2, -4(R1) | R2 ← (0-4) = (0) |
| | BNEZ R1, L1 | BNEZ 4, L1 |
| L1: | SUBI R1, R1, #4 | R1 ← 0-4 = -4 |
| | SW 4(R1), R3 | (0+4) = (4) ← (4)+1 |
| | ADDI R3, R2, #1 | R3 ← (0)+1 |
| | LW R2, -4(R1) | R2 ← (0-4) = (-4) |
| | BNEZ R1, L1 | BNEZ 0, L1 // last iteration |
| tail: | SW 0(R1), R3 | (0+0) = (0) ← (0)+1 |

# Loop Unrolling + Software Pipelining

## For unrolled program on slide 61

Requires 5 ALUs + 4 Reads + 4 Writes + Branch Unit for efficient execution

```
head: ADDI R1, R0, #1000 ;;
      SUBI R1, R1, #10 ;;
      LW R2, 0(R1)
      LW R3, 4(R1)
      LW R4, 8(R1)
      LW R5, C(R1) ;;
      ADDI R6, R2, #1
      ADDI R7, R3, #1
      ADDI R8, R4, #1
      ADDI R9, R5, #1
      LW R2, -10(R1)
      LW R3, -0C(R1)
      LW R4, -08(R1)
      LW R5, -04(R1)
      SUBI R1, R1, #10 ;;
```

```
L1:   SUBI R1, R1, #10
      SW 10(R1), R6
      SW 14(R1), R7
      SW 18(R1), R8
      SW 1C(R1), R9
      ADDI R6, R2, #1
      ADDI R7, R3, #1
      ADDI R8, R4, #1
      ADDI R9, R5, #1
      LW R2, -10(R1)
      LW R3, -0C(R1)
      LW R4, -08(R1)
      LW R5, -04(R1)
      BNEZ R1, L1 ;;
```

```
tail: SW 0(R1), R6
      SW 4(R1), R7
      SW 8(R1), R8
      SW C(R1), R9
```

**Parallel reads/writes**

Independent operations

Example:

**ADDI R6, R2, #1**

reads **OLD R2**

at **START** of CC

**LW R2, 0(R1)**

writes **NEW R2**

at **END** of CC

# Software Pipelining Performance

| | | | | |
|---|---|---|---|---|
| | | | 1 | |
| head | 15 instructions in 5 CC | | | |
| SUBI R1, R1, #10 | | R | F | 1 |
| SW 10(R1), R6 | | R | F | 2 |
| SW 14(R1), R7 | | R | F | 3 |
| SW 18(R1), R8 | | R | F | 4 |
| SW 1C(R1), R9 | | R | F | 5 |
| ADDI R6, R2, #1 | | R | F | 6 |
| ADDI R7, R3, #1 | | R | F | 7 |
| ADDI R8, R4, #1 | | R | F | 8 |
| ADDI R9, R5, #1 | | R | F | 9 |
| LW R2, -10(R1) | | R | F | 10 |
| LW R3, -0C(R1) | | R | F | 11 |
| LW R4, -08(R1) | | R | F | 12 |
| LW R5, -04(R1) | | R | F | 13 |
| BNEZ R1, L1 ;; | | R | F | 14 |
| … | | | | |
| tail | 4 sequential instructions | | | |

$$CPI = \frac{\text{overhead} + N_{\text{1-loop}} \times N_{\text{loop iterations}}}{IC_{\text{head}} + IC_{\text{tail}} + IC_{\text{1-loop}} \times N_{\text{loop iterations}}} = \frac{\text{overhead} + 1 \times 256}{7 + 8 + 14 \times 256} \simeq 0.07$$

# Improving x86 Hardware Utilization — New Directions

| 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 |

## Hardware utilization

Most execution units idle in most clock cycles

## Higher ILP ⇒ higher utilization of execution units

Higher utilization ⇒ more independent instructions to choose from

## Speculation

Issue some instructions beyond undetermined conditional branch

Keep CPU busy — accurate prediction performs work before needed

# Speculative Execution

# Speculative Execution

## Instruction stream speculation

- Predict program behavior before execution flow determined
- Execute **PREDICTED** program flow
- Verify program results before commitment to state
- Recover from **MISPREDICTION**

## Branch prediction

- Predict target address and action (taken/not-taken)

## Instruction / data cache prefetch

- Load instructions and data to L1 cache before cache miss

## Trace cache

- Cache frequent instruction flow sequences
- Span multiple control blocks

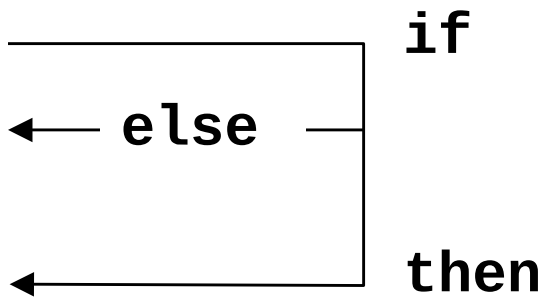
## Branch predication

- Execute multiple branch flow alternatives
- Commit only verified instruction flow

# Branch Behavior

**`if-then-else`**

| PC | Instruction |
|---|---|
| $PC_{BEQZ}$ | `BEQZ Reg, offset` |
| $PC_{BEQZ}$ + 4 | `fall-through` |
| | `...` |
| $PC_{BEQZ}$ + 4 + offset | `target` |
| | `continue` |




## Speculation in dynamic scheduling

At **PC < $PC_{BEQZ}$**

- Prediction → branch taken
- Issue instructions target, continue, ...
- Cannot retire until **PC > $PC_{BEQZ}$**

At **PC = $PC_{BEQZ}$**

- Verify **BEQZ**
- Continue or correct misprediction

# Basic Branch Prediction

## Branch predictor in fetch + decode stage

Branch History Register (BHR)

| Index | Source identifier (hash of instruction PC) |
|---|---|
| Entry | **Branch outcome history**<br>**Prediction (target address, taken/not-taken) by some algorithm** |

Branch Predictor

PC → BHR → prediction + target address

Predict taken ⇒ PC ← target address

Predict not taken or miss in BHR ⇒ PC ← fall-through address

## Branch validation (verification)

Complete detailed branch execution

Compare with prediction

Correct prediction ⇒ continue program flow

Misprediction ⇒ flush pipeline → restart with correct target address

Update history table

# 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

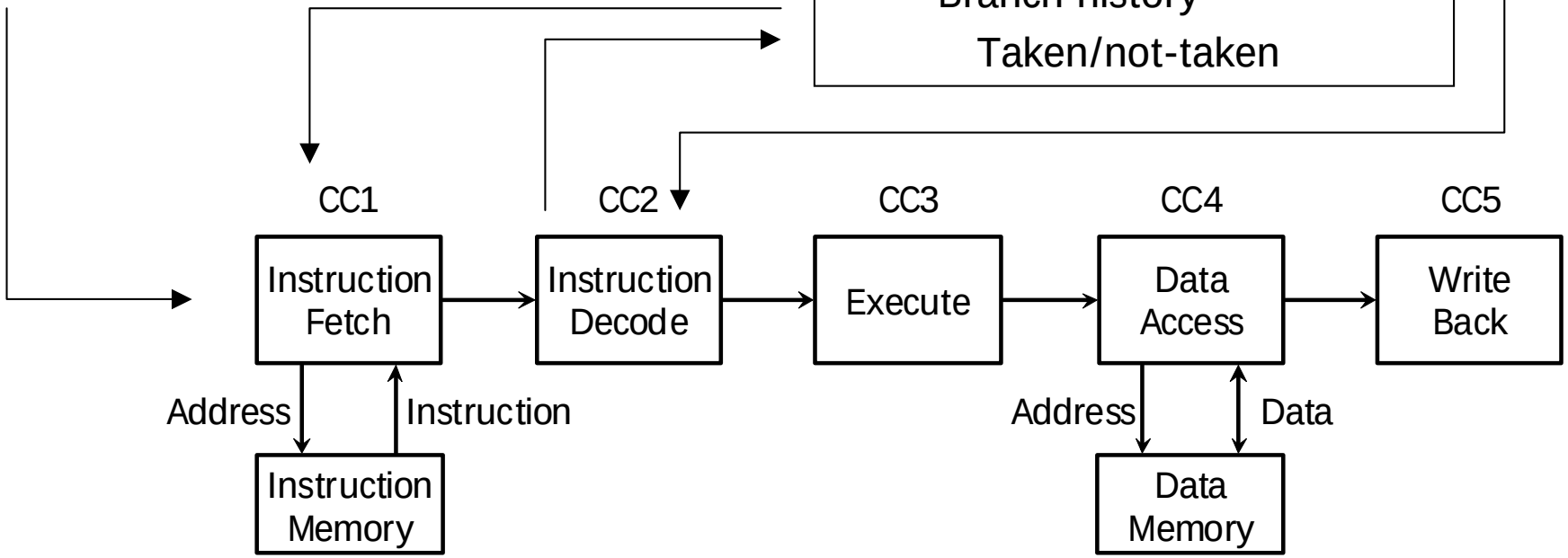
**3. After validation**

Update branch predictor

Target address

Branch history

Taken/not-taken



# Simple Branch Prediction Performance for Loop

**Simple static loop**

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | ADDI R1, R0, # N | IF | ID | EX | MEM | WB | | |
| L1: | SUBI R1, R1, #1 | | 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: | SUBI R1, R1, #1 | | | IF | ID | EX | MEM | WB |
| | < B-2 lines of ALU code > | | | | | | | |
| | BNEZ R1, L1 | IF | ID | EX | MEM | WB | | |
| L1: | SUBI R1, R1, #1 | | IF | ID | EX | MEM | WB | |
| | ... | | | | | | | |
| | < B-2 lines of ALU code > | | | | | | | |
| | BNEZ R1, L1 | IF | ID | EX | MEM | WB | | |
| L1: | SUBI R1, R1, #1 | | IF | ID | ϕ | ϕ | ϕ | |
| | $I_{fall-through}$ | | | IF | ID | EX | MEM | WB |

**R1 = N - 1**

**R1 = N - 2**

**R1 = 0**

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

# Importance of Branch Hazards

## Branch instructions ⇒ program control flow

Loop instructions account for most run time

## Branch statistics

Branch ≈ 20% — 25% of instructions

≈ 2/3 of conditional branches are taken

$$\frac{1 \text{ branch}}{4 \text{ instructions}} \times \frac{2 \text{ taken branches}}{3 \text{ branch}} = \frac{1 \text{ taken branch}}{6 \text{ instructions}}$$

## Branch verification (correct execution) = N clock cycles

N wrong instructions → pipeline before branch verification

N wrong instructions must be flushed

N wasted clock cycles

$$CPI^{stall} = \frac{N \text{ wasted clock cycles}}{\text{taken branches}} \times \frac{1 \text{ taken branch}}{6 \text{ instructions}} = \frac{N \text{ wasted clock cycles}}{6 \text{ instructions}}$$

N ≥ 1 for DLX ⇒ $CPI^{stall} \geq 1/6$

# Dynamic Branch Prediction

## Past behavior predicts future branch behavior

Assumes repetitive/cyclical behavior in branches

## Possible approaches

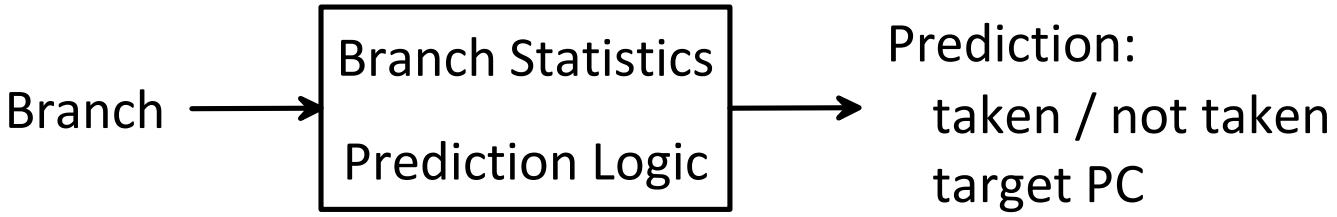
Global prediction

Predict from recent execution pattern of all recent branches

Local prediction

Predict from recent execution pattern of individual branch

Combination of global and local prediction



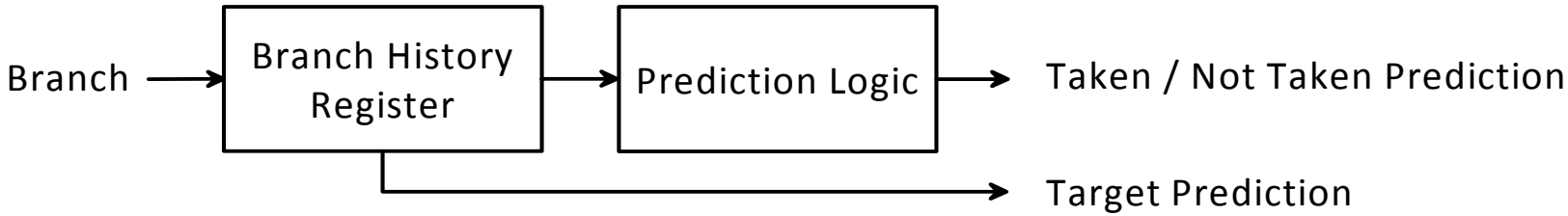

## Limitations

Complex hardware

Large branch history register storage

Long prediction time

# One-Level Prediction




## Branch History Register (BHR)

Index — hash function of branch PC

N bit entry (profile)

N most recent branch outcomes

Target — PC of last taken branch

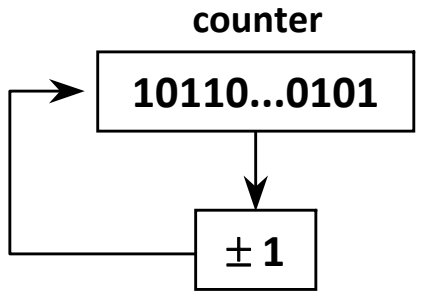
## Prediction Logic

n-bit counter ⇒ $2^n$ levels

Operates on n-bit BHR entry

**000…0** = Strongly Not Taken

**111…1** = Strongly Taken




## Prediction

**0xxxx...x** = not taken

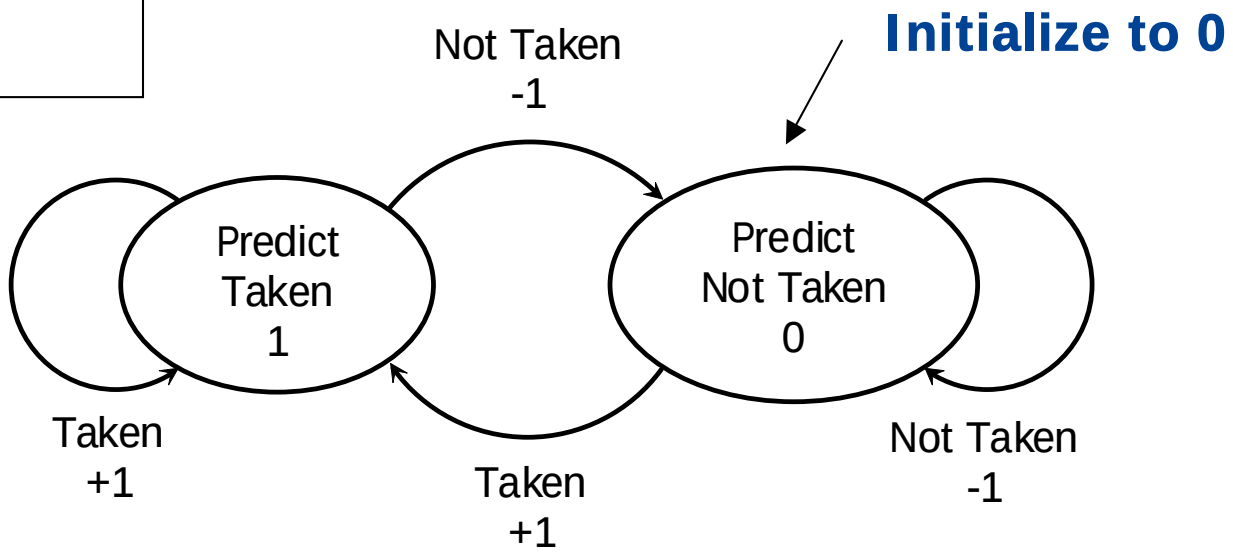
**1xxxx...x** = taken

| | |
|---|---|
| Initialize | `counter ← 0` |
| Taken | `counter++` |
| Not Taken | `counter--` |

# 1-Bit Saturating Up/Down Counter

**1-bit confidence level**

Remembers last action only




## After validation of branch action

Taken ⇒ **`counter++`**

Not Taken ⇒ **`counter--`**

## Prediction based on last action

**`Counter = 0`** ⇒ not taken

**`Counter = 1`** ⇒ taken

Predicts single static loops with accuracy of 1 / (loop iterations)

# 2-Bit Saturating Up/Down Counter

**2-bit confidence level**

Remembers up to 3 actions

## After branch validation

Taken ⇒ **`counter++`**

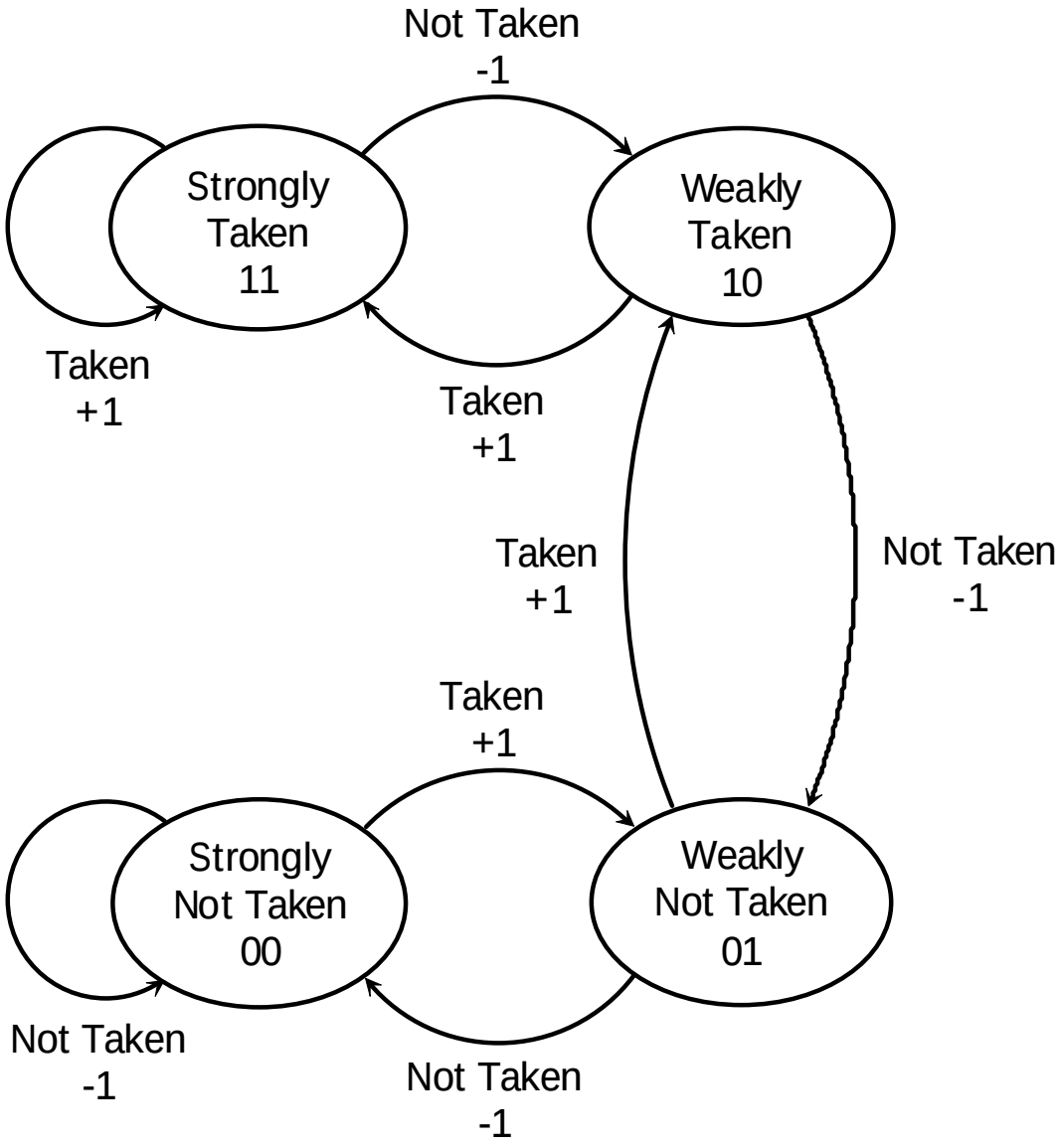
Not Taken ⇒ **`counter--`**

## Prediction

**`Counter = 0x`** ⇒ not taken

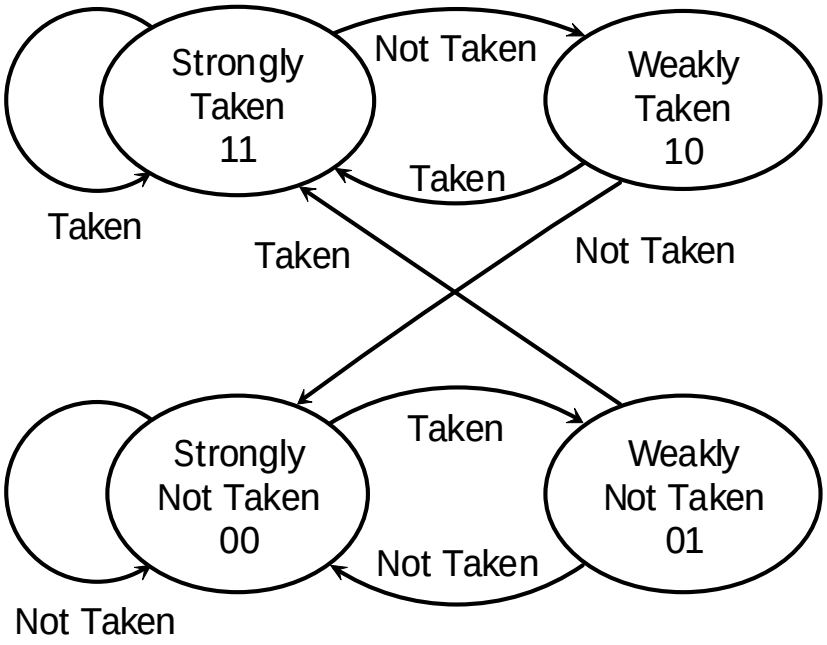
**`Counter = 1x`** ⇒ taken

Predicts 2 nested loops




**Initialize to 00**

# Alternative 2-Bit Counters

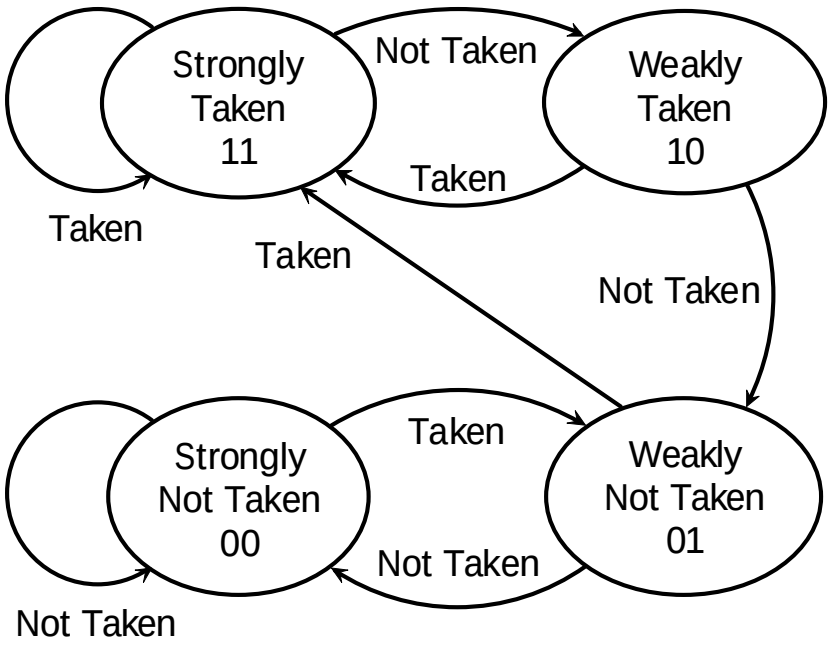



**2 consecutive taken branches**

Predict strongly taken

**2 consecutive not taken branches**

Predict strongly not taken




**2 consecutive taken branches**

Predict strongly taken

Preference to taken branches

# Short Nested Loop

```
    ADDI R1, R0, #Len
L1: ADDI R2, R0, #Len
L2: SUBI R2, R2, #1
    BNEZ R2, L2
    SUBI R1, R1, #1
    BNEZ R1, L1
```

## Example

Len = 3 global profile

## N-bit BHR entries

## Branch profile — global branch actions

1 = taken

0 = not taken

11…1011…1011…101…11…1011…100

1 2 3 Len-1 Len

123456789ABC

1101 1101 1100

1 2 3

| Branch | 1-bit | 2-bit | 3-bit | 4-bit |
|---|---|---|---|---|
| init | 0 | 00 | 000 | 0000 |
| 1 | 1 | 01 | 001 | 0001 |
| 2 | 1 | 11 | 011 | 0011 |
| 3 | 0 | 10 | 110 | 0110 |
| 4 | 1 | 01 | 101 | 1101 |
| 5 | 1 | 11 | 011 | 1011 |
| 6 | 1 | 11 | 111 | 0111 |

| Branch | 1-bit | 2-bit | 3-bit | 4-bit |
|---|---|---|---|---|
| 7 | 0 | 10 | 110 | 1110 |
| 8 | 1 | 01 | 101 | 1101 |
| 9 | 1 | 11 | 011 | 1011 |
| A | 1 | 11 | 111 | 0111 |
| B | 0 | 10 | 101 | 1110 |
| C | 0 | 00 | 100 | 1100 |

# Short Nested Loop

```
     ADDI R1, R0, #L
L1:  ADDI R2, R0, #L
L2:  SUBI R2, R2, #1
     BNEZ R2, L2
     SUBI R1, R1, #1
     BNEZ R1, L1
```

## 1-bit mispredictions

$1 + (L - 2) \times 2 + 1$

## 2-bit mispredictions

$2 + (L - 2) \times 1 + 2$

## Large L

mispredictions

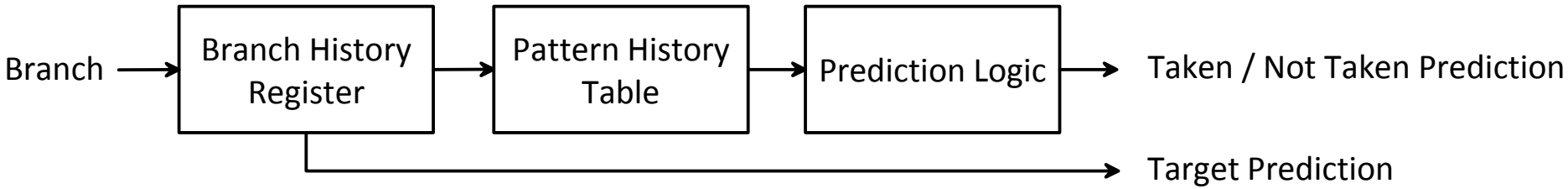
~ L for 2-bit

~ 2L for 1-bit

**Counters at predict time**
**Based on previous Action**

| R1 (at L2) | R2 (at L2) | Action | 1-bit | 2-bit |
|---|---|---|---|---|
| L | L | T | 0 | 00 |
| L | L – 1 | T | 1 | 01 |
| L | L – 2 … 2 | T | 1 | 10 |
| L | 1 | NT – T | 1 – 0 | 11 – 10 |
| L – 1 | L | T | 1 | 10 |
| L – 1 | L – 1 | T | 1 | 11 |
| L – 1 | L – 2 … 2 | T | 1 | 11 |
| L – 1 | 1 | NT – T | 1 – 0 | 11 – 10 |
| L – 2 … 2 | L | T | 1 | 10 |
| L – 2 … 2 | L – 1 | T | 1 | 11 |
| L – 2 … 2 | L – 2 … 2 | T | 1 | 11 |
| L – 2 … 2 | 1 | NT – T | 1 – 0 | 11 – 10 |
| 1 | L | T | 1 | 10 |
| 1 | L – 1 | T | 1 | 11 |
| 1 | L – 2 … 2 | T | 1 | 11 |
| 1 | 1 | NT – NT | 1 – 0 | 11 – 10 |

# Yeh-Patt Two-Level Adaptive Prediction




## Branch History Register (BHR)

Profile — sequence of observed branch actions

Target — PC of last taken branch

Characterizes history of branch behavior

## Pattern History Table (PHT)

Pattern — observed next branch action for specific profile

Characterizes pattern of branch behavior given particular history

## Prediction

Confidence level = n-bit saturating up/down counter

Yeh and Patt, Alternative Implementations of Two-Level Adaptive Branch Prediction, 1992

# Yeh-Patt — BHR

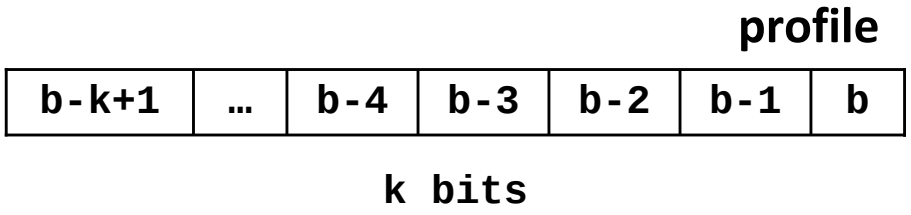
## Index

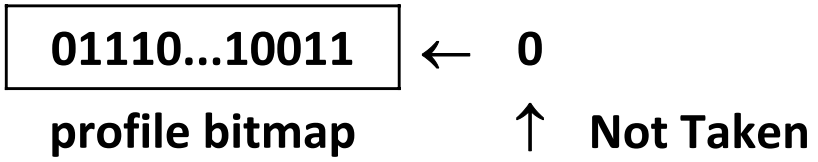
- Global table
  - 1 entry
  - No index
- Local table
  - 1 entry per branch
  - Index = hash function of branch source PC
  - Source address aliasing ⇒ instruction misidentified ⇒ misprediction




## Entry

- Profile
  - k-bit bitmap = last k actions of b last branches
  - Taken = 1 / Not Taken = 0
  - Shift in newest action from right
- Target — PC of last taken branch



# Yeh-Patt — PHT + Prediction Logic

## Index

k-bit profile

Branch action sequence



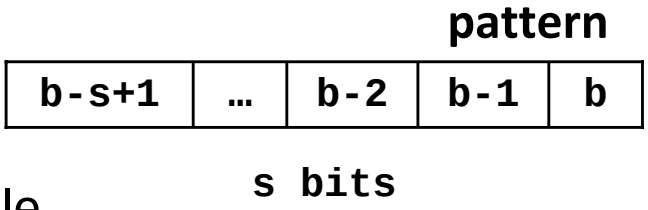

## Entry

Pattern

s-bit bitmap = last s (out of b) next actions for profile

Taken = 1 / Not Taken = 0

Shift in newest action from right




## Prediction Logic

n-bit saturating up/down counter

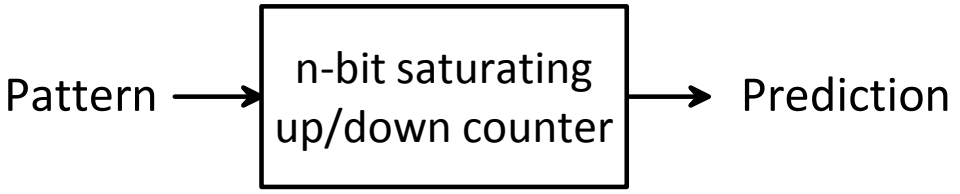
Counts s-bit pattern from PHT (from left to right)

**000…0** = Strongly Not Taken —— **111…1** = Strongly Taken

Prediction

**0xxxx...x** = not taken

**1xxxx...x** = taken



# Yeh-Patt — Simple Example

## Profile for short nested loop with Len = 3

k = 2 branch actions (PHR)

s = 1 next action (BHT)

1-bit counter (prediction = last action for pattern)

```
     ADDI R1, R0, #L
L1:  ADDI R2, R0, #L
L2:  SUBI R2, R2, #1
     BNEZ R2, L2
     SUBI R1, R1, #1
     BNEZ R1, L1
```

## Actual branch sequence — 110111011100

| Branch | | Global Profile m | | Observed Profiles | | | | Actual Next Action | | | | Predicted Next Action (last action) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| start | — | 0 | 0 | 00 | | | | 1 | | | | — | | | |
| BNEZ R2 | 1 | 0 | 1 | 00 | 01 | | | 1 | 1 | | | 1 | — | | |
| BNEZ R2 | 1 | 1 | 1 | 00 | 01 | 11 | | 1 | 1 | 0 | | 1 | 1 | — | |
| BNEZ R2 | 0 | 1 | 0 | 00 | 01 | 11 | 10 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | — |
| BNEZ R1 | 1 | 0 | 1 | 00 | 01 | 11 | 10 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| BNEZ R2 | 1 | 1 | 1 | 00 | 01 | 11 | 10 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| BNEZ R2 | 1 | 1 | 1 | 00 | 01 | 11 | 10 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| BNEZ R2 | 0 | 1 | 0 | 00 | 01 | 11 | 10 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| BNEZ R1 | 1 | 0 | 0 | | | | | | | | | | | | |

First observation  Repeat  = misprediction

# Yeh-Patt — Building BHR and PHT

## Branch actions for some branch instruction

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|

5-bit BHR (starting from 5th branch)

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | BHR Entry | next |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | | | | | | | | 10101 | 0 |
| | 0 | 1 | 0 | 1 | 0 | | | | | | | 01010 | 1 |
| | | 1 | 0 | 1 | 0 | 1 | | | | | | 10101 | 0 |
| | | | 0 | 1 | 0 | 1 | 0 | | | | | 01010 | 1 |
| | | | | 1 | 0 | 1 | 0 | 1 | | | | 10101 | 0 |
| | | | | | 0 | 1 | 0 | 1 | 0 | | | 01010 | 1 |

3-bit patterns (PHT)

| | | | |
|---|---|---|---|
| 10101 | 0 | 0 | 0 |
| 01010 | 1 | 1 | 1 |

PC → BHR profile = last 5 branches → PHT entry = 3 Next Actions after profile

PHT entry → Counter → Prediction

# Example — Alternating Branches in One-Level Prediction

```
     ADD R1, R0, R0
L1:  ADDI R1, R1, #L
     ANDI R2, R1, #L
     BNEZ R2, L1
     BEQZ R2, L1
```

Misprediction rate for **BNEZ**

1-bit counter — 100%

2-bit counter — 50%

**Counting 2-bit BHR entries**

Entry 01: counter 00 —> 00 —> 01

Entry 10: counter 00 —> 01 —> 00

Counter **never** counts up to 10

Counter **always** predicts 0

**Separate Counters for BNEZ and BEQZ**

| State | Action | | counters at prediction time | | | |
|---|---|---|---|---|---|---|
| | | | 1-bit BHR (last action) | | 2-bit counter 2-bit BHR | |
| | BNEZ | BEQZ | BNEZ | BEQZ | BNEZ | BEQZ |
| 1 | 1 | | 0 | | 00 | |
| 2 | 0 | 1 | 1 | 0 | 01 | 00 |
| 3 | 1 | | 0 | | 00 | |
| 4 | 0 | 1 | 1 | 1 | 01 | 01 |
| 5 | 1 | | 0 | | 00 | |
| 6 | 0 | 1 | 1 | 1 | 01 | 10 |
| 7 | 1 | | 0 | | 00 | |
| 8 | 0 | 1 | 1 | 1 | 01 | 11 |
| 9 | 1 | | 0 | | 00 | |
| 10 | 0 | 1 | 1 | 1 | 01 | 11 |
| 11 | 1 | | 0 | | 00 | |
| 12 | 0 | 1 | 1 | 1 | 01 | 11 |

For **BNEZ** 0 is correct half the time

**Prediction**

1x = taken

0x = not taken

# Example — Alternating Branches with Y-P

```
        ADD R1, R0, R0
L1:     ADDI R1, R1, #L
        ANDI R2, R1, #L
        BNEZ R2, L1
        BEQZ R2, L1
```

4-bit BHR

2-bit PHT for each BHR pattern

| | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | BHR | PHT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Learning For BNEZ | init | | | | | | | | | | | | 0000 | 00 |
| | 1 | | | | | | | | | | | | 0001 | 00 |
| | 1 | 0 | | | | | | | | | | | 0010 | 00 |
| | 1 | 0 | 1 | | | | | | | | | | 0101 | 00 |
| | 1 | 0 | 1 | 0 | | | | | | | | | 1010 | 00 |
| | | 0 | 1 | 0 | 1 | | | | | | | | 0101 | 00 |
| | | | 1 | 0 | 1 | 0 | | | | | | | 1010 | 01 |
| Steady State For BNEZ | | | | 0 | 1 | 0 | 1 | | | | | | 0101 | 00 |
| | | | | | 1 | 0 | 1 | 0 | | | | | 1010 | 11 |
| | | | | | | 0 | 1 | 0 | 1 | | | | 0101 | 00 |
| | | | | | | | 1 | 0 | 1 | 0 | | | 1010 | 11 |
| | | | | | | | | 0 | 1 | 0 | 1 | | 0101 | 00 |

# Example — Alternating Branches with Y-P

```
        ADD R1, R0, R0
L1:     ADDI R1, R1, #L
        ANDI R2, R1, #L
        BNEZ R2, L1
        BEQZ R2, L1
```

k = 4-bit BHR

s = 2-bit PHT for each BHR pattern

Prediction function

Output of 2-bit counter on PHT bits

00 ⇒ 00 → 00 → 00

01 ⇒ 00 → 00 → 01

10 ⇒ 00 → 01 → 00

11 ⇒ 00 → 01 → 10

0x = Not Taken

1x = Taken

Correctly predicts every branch from 8th branch run

Steady state sequence for **BNEZ**

BHR:PHT Alternate 0101:00 / 1010:11

**Data at Prediction Time**

| State | Action | | BHR | PHT | Y-P |
|---|---|---|---|---|---|
| | BNEZ | BEQZ | | | BNEZ |
| 1 | 1 | | 0000 | 00 | – |
| 2 | 0 | 1 | 0001 | 00 | – |
| 3 | 1 | | 0010 | 00 | – |
| 4 | 0 | 1 | 0101 | 00 | – |
| 5 | 1 | | 1010 | 00 | - |
| 6 | 0 | 1 | 0101 | 00 | 0 |
| 7 | 1 | | 1010 | 01 | 0 |
| 8 | 0 | 1 | 0101 | 00 | 0 |
| 9 | 1 | | 1010 | 11 | 1 |
| 10 | 0 | 1 | 0101 | 00 | 0 |
| 11 | 1 | | 1010 | 11 | 1 |
| 12 | 0 | 1 | 0101 | 00 | 0 |

Steady state Y-P misprediction rate for **BNEZ** — 0%

# Another Example

## Branch actions for some branch instruction

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

## k = 5-bit BHR (from 5th branch run)

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | | | | | | | | | | | | |
| | 0 | 1 | 0 | 1 | 0 | | | | | | | | | | | |
| | | 1 | 0 | 1 | 0 | 1 | | | | | | | | | | |
| | | | 0 | 1 | 0 | 1 | 1 | | | | | | | | | |
| | | | | 1 | 0 | 1 | 1 | 1 | | | | | | | | |
| | | | | | 0 | 1 | 1 | 1 | 0 | | | | | | | |
| | | | | | | 1 | 1 | 1 | 0 | 1 | | | | | | |
| | | | | | | | 1 | 1 | 0 | 1 | 0 | | | | | |
| | | | | | | | | 1 | 0 | 1 | 0 | 1 | | | | |
| | | | | | | | | | 0 | 1 | 0 | 1 | 0 | | | |
| | | | | | | | | | | 1 | 0 | 1 | 0 | 1 | | |
| | | | | | | | | | | | 0 | 1 | 0 | 1 | 1 | |
| | | | | | | | | | | | | 1 | 0 | 1 | 1 | 1 |

## s = 3-bit PHT

| | | | |
|---|---|---|---|
| 10101 | 1 | 0 | 1 |
| 01010 | 1 | 1 | 1 |
| 01011 | 1 | 1 | 1 |
| 10111 | 0 | 0 | 0 |
| 01110 | 1 | 1 | 1 |
| 11101 | 0 | 0 | 0 |
| 11010 | 1 | 1 | 1 |

**Misprediction rate**

(Repeating profile 10101011)

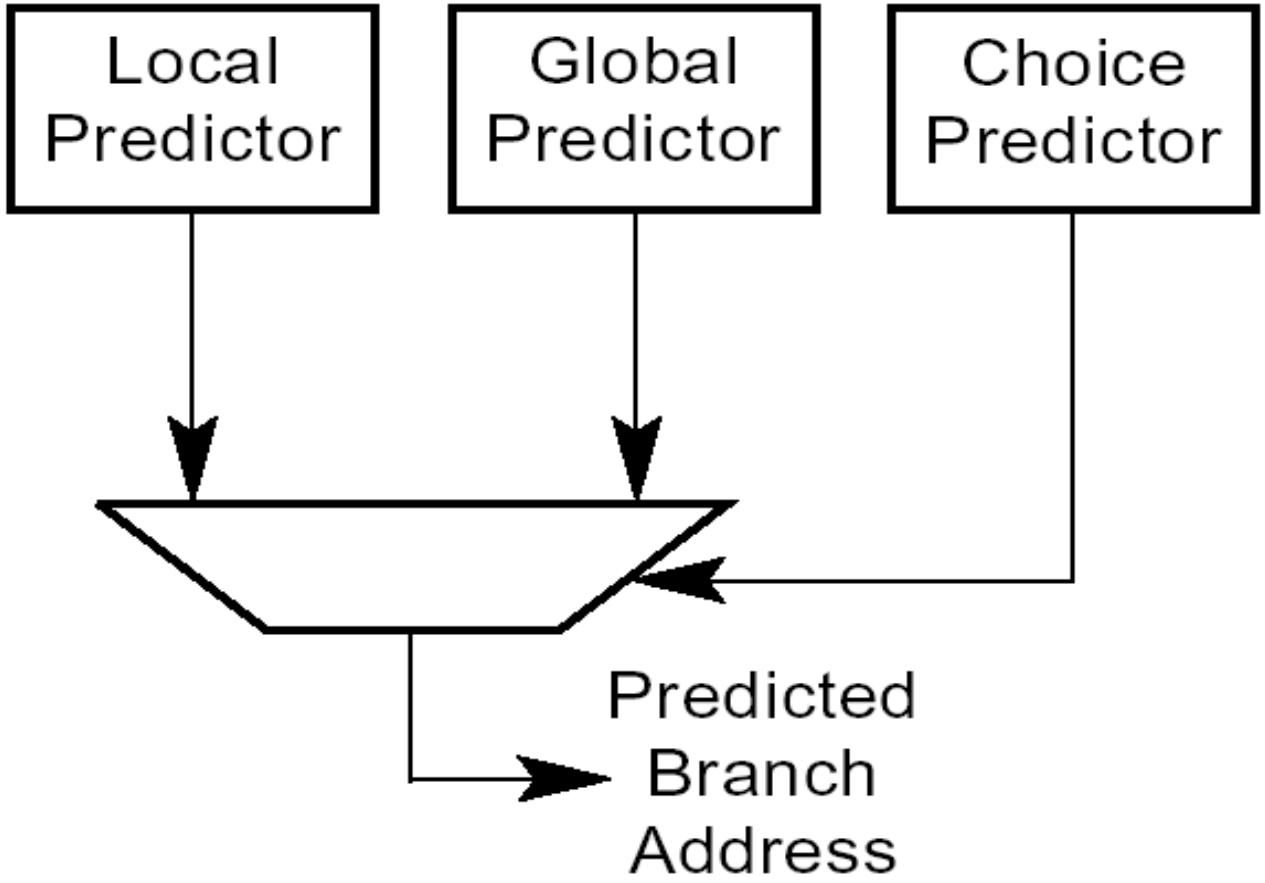
10101 50% (always predicts 0)

other 0%

PHT entry for 10101 is either 101 or 010

Counter (101) = 00—>01—>00—>01 Counter (010) = 00—>00—>01—>00

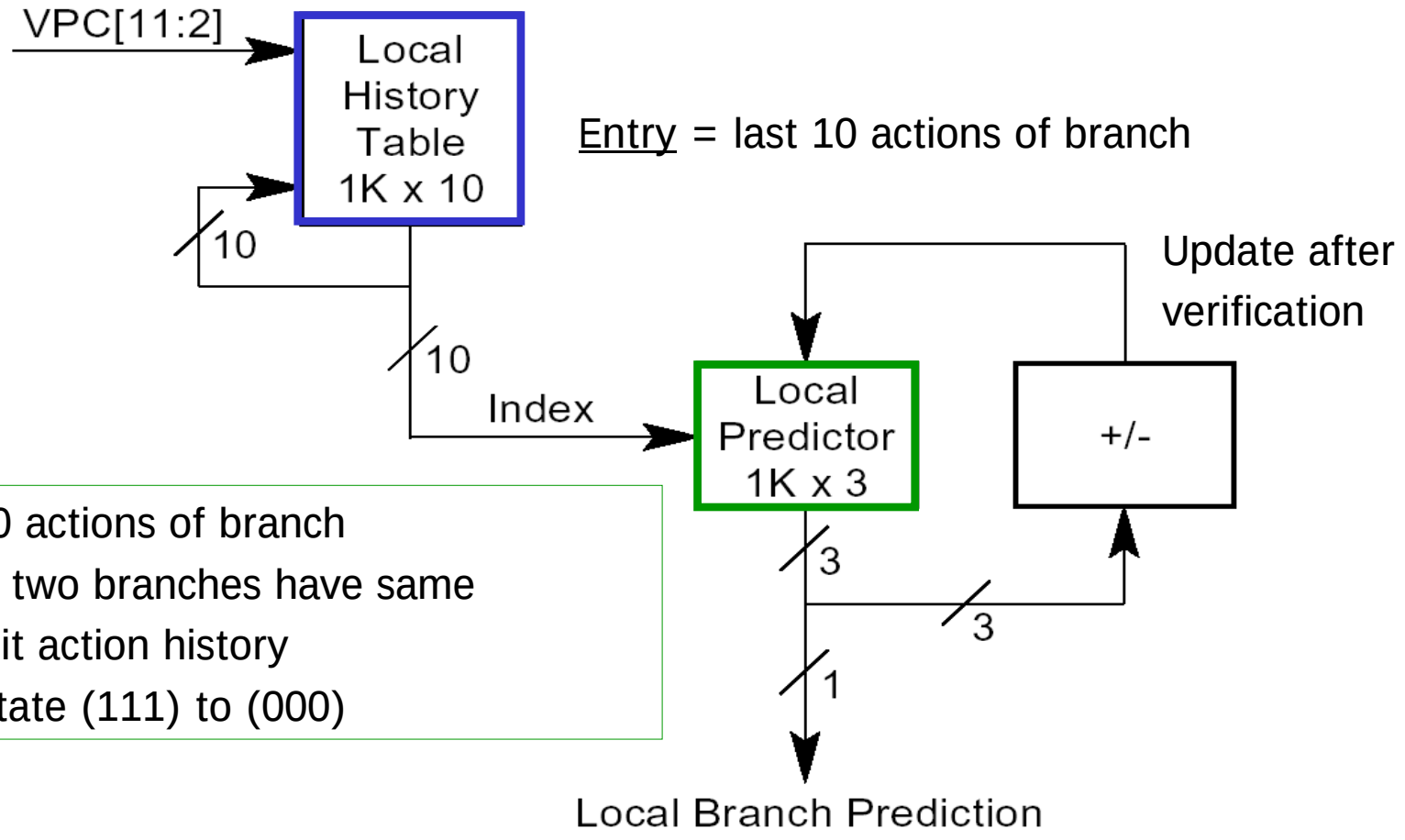
# DEC Alpha Branch Predictor — 1




Ref: 21264/EV68A Microprocessor Hardware Reference Manual, 1992

# Two-Level Local Predictor — 2

Index: Branch source address bits 2 to 11 (10 bits ⇒ $2^{10}$ = 1024 entries)
Instruction addresses aligned on 4-byte (2-bit) boundaries




Entry = last 10 actions of branch

Index: Last 10 actions of branch
Alias if two branches have same 10-bit action history

Entry: 3-bit state (111) to (000)

Predictor = (0xx) ⇒ not-taken
Predictor = (1xx) ⇒ taken

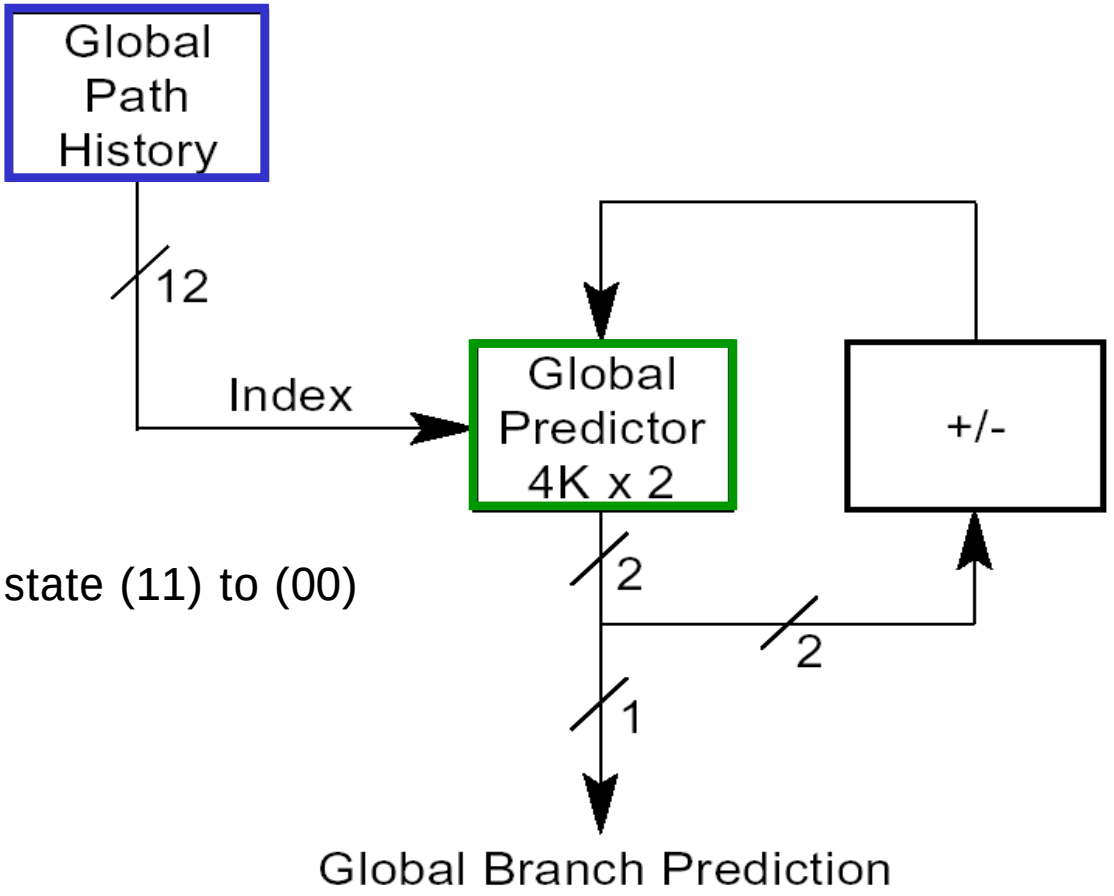
# Global Predictor — 3

Last 12 actions of all branches

$2^{12}$ = 4096 possible action histories

Index: Last 12 actions of all branches

Entry: 2-bit state (11) to (00)



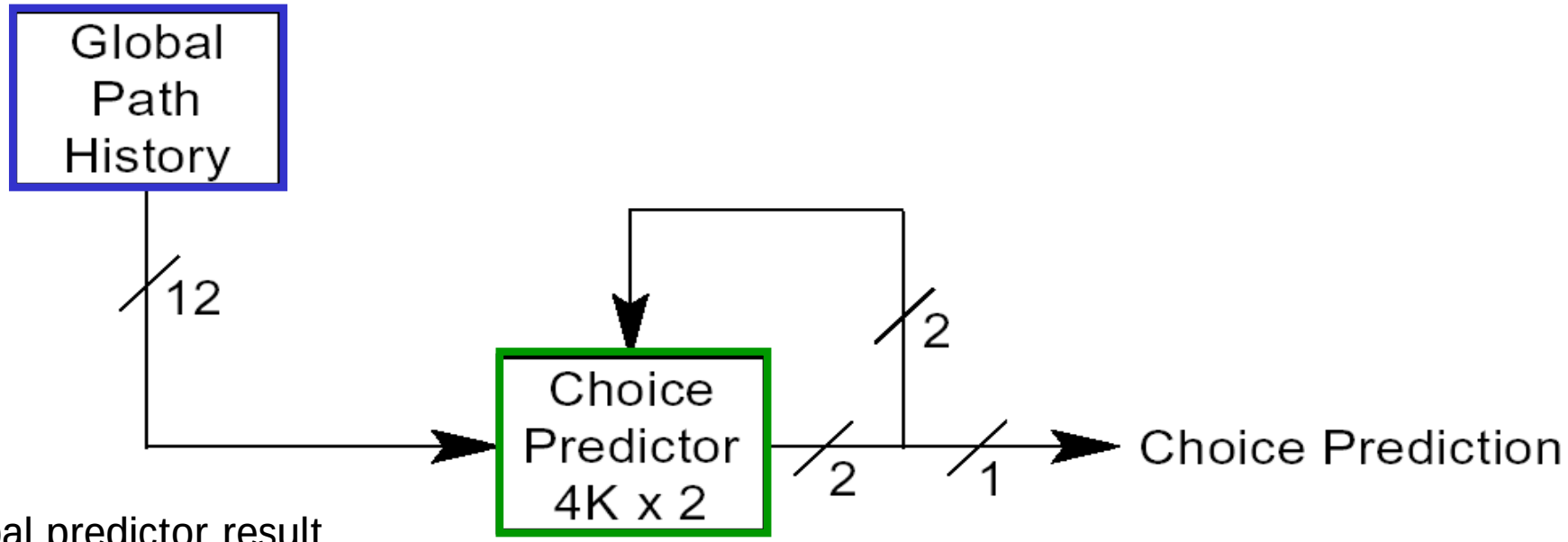

Prediction: Predictor = (0x) ⇒ not-taken

Predictor = (1x) ⇒ taken

# Choice Predictor — 4

## Last 12 actions of all branches

$2^{12}$ = 4096 possible action histories




Global predictor result

1 = correct

0 = misprediction

Prediction: (0x) chooses Local Predictor

(1x) chooses Global Predictor

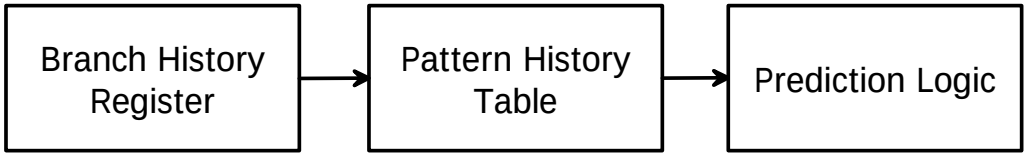
# Yeh-Patt Versus Alpha

## Yeh-Patt model

Stage 1 — store k-bit profile (taken / not taken history) in BHR

Stage 2 — store s-bit pattern (next action history) in PHT

Stage 3 — count over stored pattern




## Alpha model

Local

Stage 1 — store k = 10 bit profile

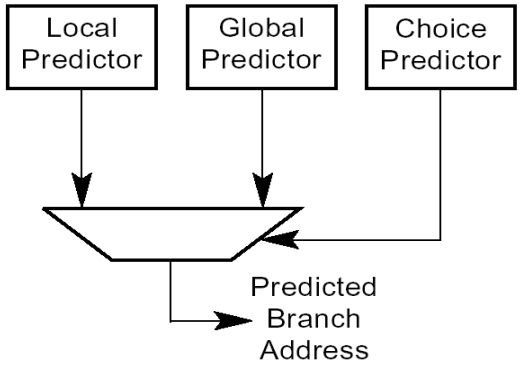
Stage 2 — count over all next actions for profile

Global

Stage 1 — store k = 12 bit profile

Stage 2 — count over all next actions for profile

Choice predictor — local / global



# Event Horizon in Counter

## Event horizon

Prediction on s-bit pattern from PHT ≠ counting all next actions

## Example

Branch history = **11 11110 11110 11110 11110...**

Next actions for profile **11**

| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | | | | | | | | | | | | | | |
| | 1 | 1 | 1 | | | | | | | | | | | | | |
| | | 1 | 1 | 1 | | | | | | | | | | | | |
| | | | 1 | 1 | 1 | | | | | | | | | | | |
| | | | | 1 | 1 | 0 | | | | | | | | | | |
| | | | | | | | 1 | 1 | 1 | | | | | | | |
| | | | | | | | | 1 | 1 | 1 | | | | | | |
| | | | | | | | | | 1 | 1 | 0 | | | | | |
| | | | | | | | | | | | | 1 | 1 | 1 | | |
| | | | | | | | | | | | | | 1 | 1 | 1 | |
| | | | | | | | | | | | | | | 1 | 1 | 0 |

## Compare

2-bit counter on all next actions

2-bit counter on 2-bit pattern

| nexts | | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 00 | 01 | 10 | 11 | 11 | 10 | 11 | 11 | 10 | 11 | 11 | 10 | 11 | 11 |
| patterns | 00 | 01 | 11 | 11 | 11 | 10 | 01 | 11 | 10 | 01 | 11 | 10 | 01 | 11 |
| count | 00 | 01 | 10 | 10 | 10 | 01 | 01 | 10 | 01 | 01 | 10 | 01 | 01 | 10 |
| outcome | | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

# Address Prediction

## Branch Target Buffer (BTB)

- Organized as cache with LRU eviction
- Integrated with action predictor or separate table

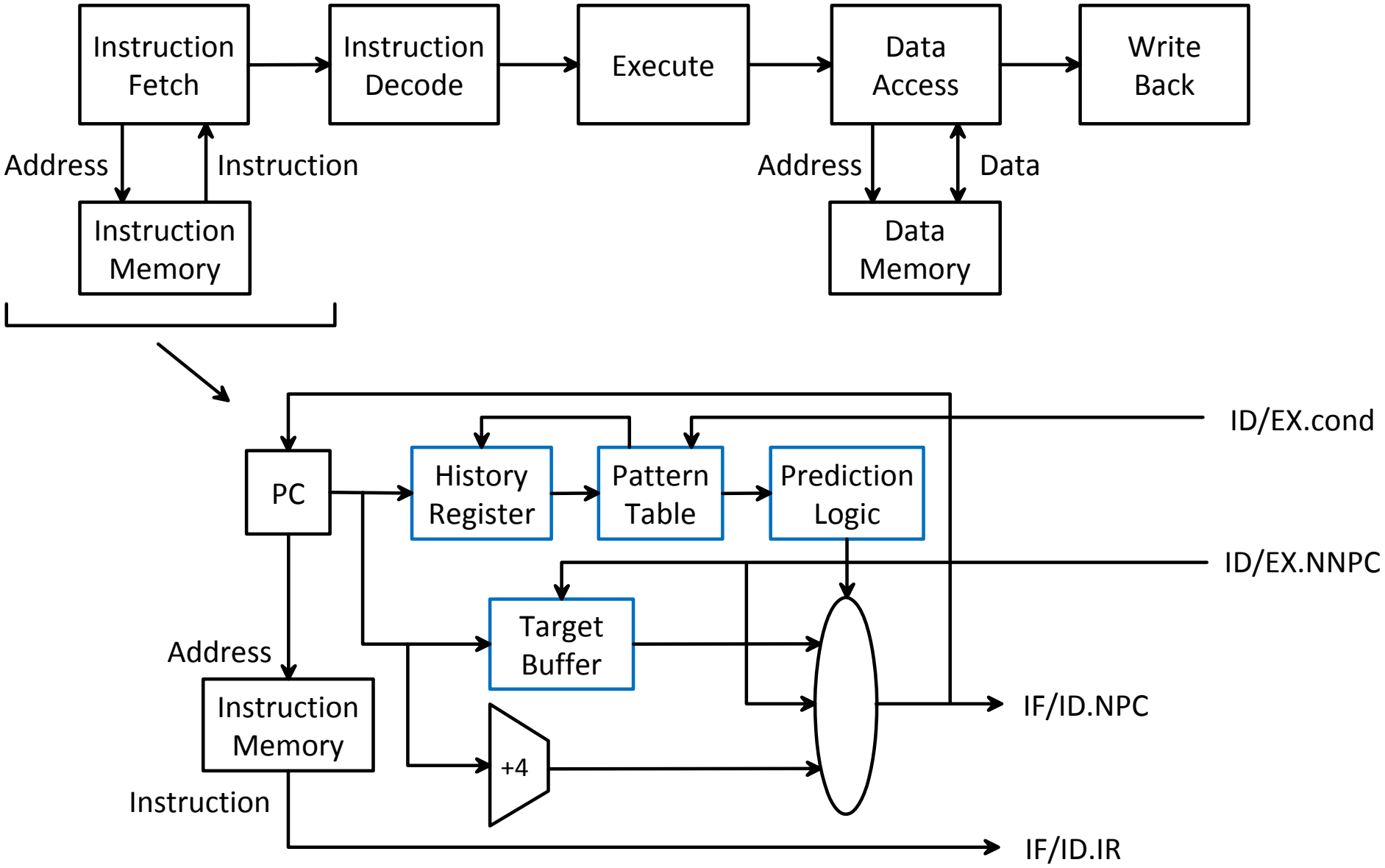
## Index

- Branch PC

## Entry

- Last verified target address
- Valid
  - Initialized to 0
  - Changed when entry is written
- Special
  - Indicates low confidence in predicted address

# DLX Pipeline with Y-P Branch Prediction



# Low Confidence Address Predictions

## Indirect branches

| | | |
|---|---|---|
| x86 | **JMP [ESI]** | **EIP ← Mem[Regs[ESI]]** |
| DLX | **JR R1** | **PC ← Regs[R1]** |

Pentium M caches target for indirect branches

Target prediction made from cache of register instead of last jump

## Return from subroutine

Function call, trap, interrupt

Lowest confidence branch addresses

Enter subroutine from multiple departure points

Each subroutine call has private return point

| | | |
|---|---|---|
| x86 | **CALL [ESI]** | **PUSH EIP**<br>**EIP ← Mem[Regs[ESI]]** |
| | **RET** | **POP EIP** |
| DLX | **JALR R2, offset** | **Regs[R2] ← PC** |
| | **JR R2** | **PC ← Regs[R2]** |

# Predicting Subroutine Return Addresses

## x86 Call/Return instructions

**CALL** pushes return address onto stack

First run of **RETURN** misses in branch history cache

Store entry after verification with Special Field

Subsequent runs of **RETURN** pre-load stack entry for target address

## RISC JALR/JR instructions

Recognize subroutine call

Encounter instruction **JR R2**

**R2** points to instruction immediately below **JALR R2**, **`offset`**

Mark **JALR** instructions as Probable Call in BTB entry

Create BTB entry for **JALR** target (from subroutine)

Mark entry as Subroutine Entry Point (SEP)

Target address = pointer to **JR** return instruction

On **JALR** hit at PC

Target field of **JR** pointed to by SEP ← PC + 4

Returning **JR** has correct target address

# Exotic Branch Mispredictions

## Exotic branch conditions

Simple branch prediction has accuracy ≈ 90%

Typically

- 80% of branches ~ 100% accuracy
- 20% of branches ~ 50% accuracy

## Least predictable branches — on-time test of single operand

x86 — **`CMP EAX, immediate`** followed by **`JZ offset`**

DLX — **`BEQZ R1, offset`**

## Special treatment for tested operands (proposed)

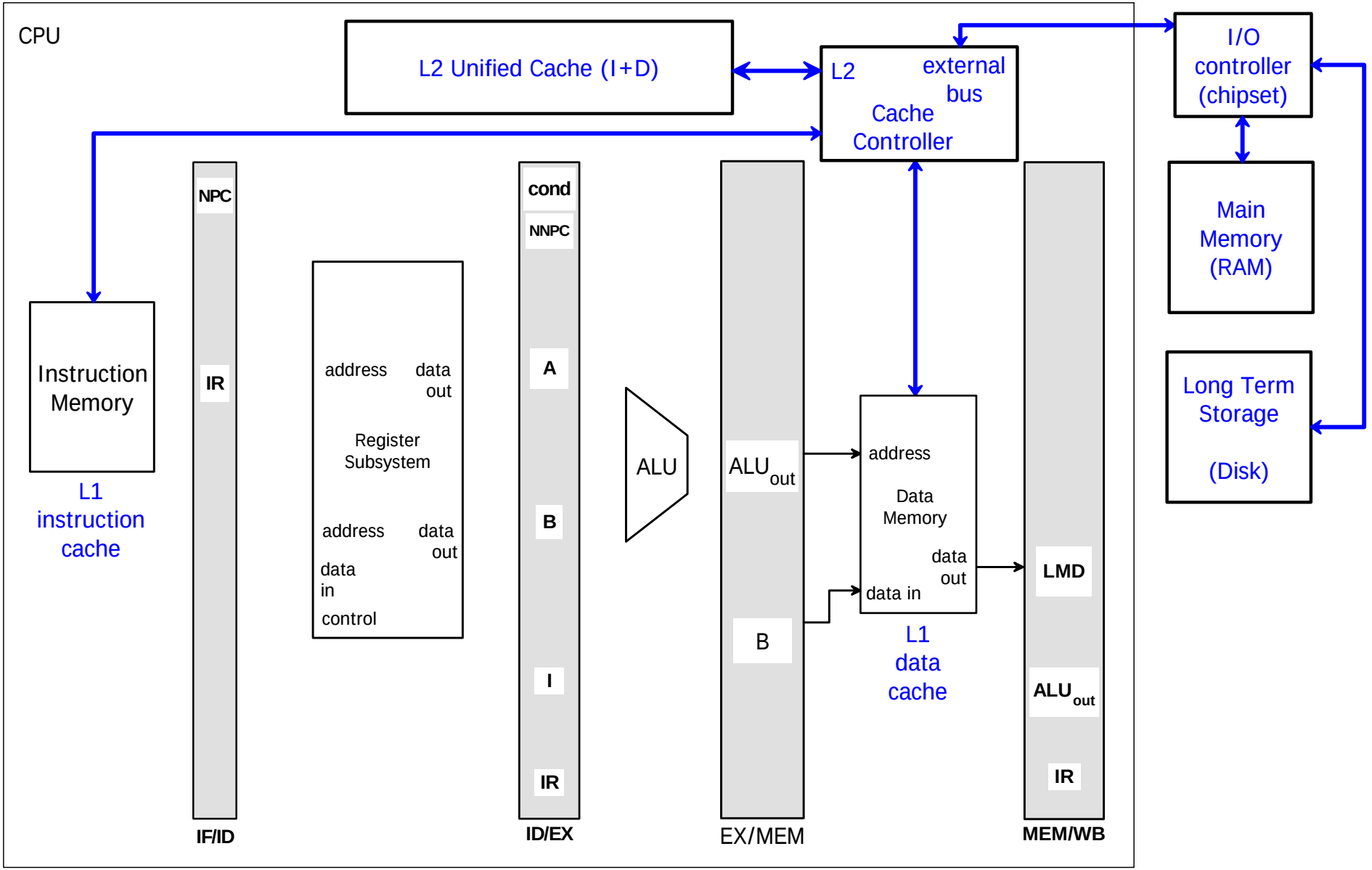
Maintain cache of tested operands and test

- Entry for **`BEQZ R1`** caches value of **`R1`** and output of **`(R1 == 0)`**

On update to operand that hits in table

- Update table entry
- Update action prediction for associated branch instruction

# DLX Memory Hierarchy



# CPU and Memory Hierarchy

## CPU controller accesses L1 cache

if (address in L1 cache) {access performed in 1 clock cycle}

else {

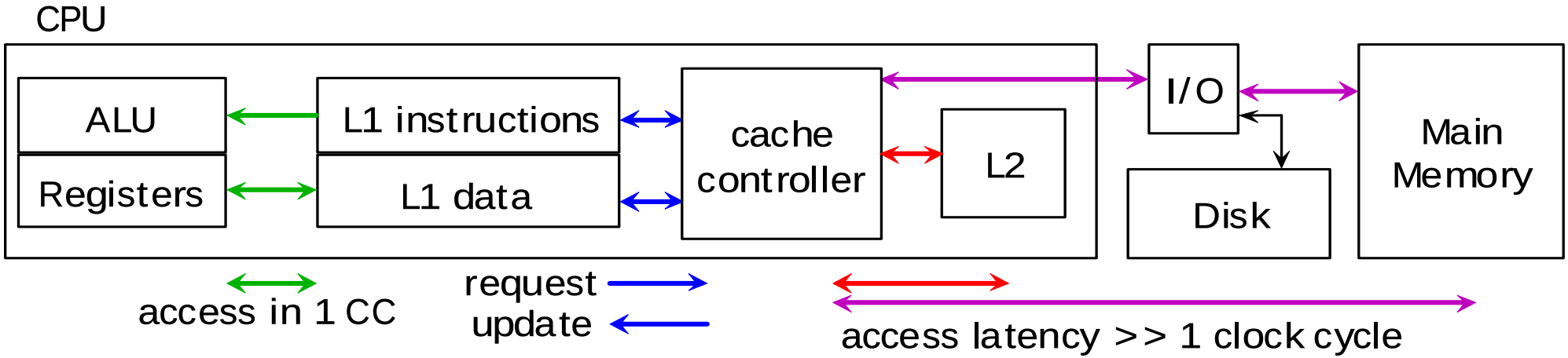
L1 cache accesses cache controller

cache controller initiates access to L2 and main memory

if (address in L2 cache) {controller copies contents to L1 from L2}

else {controller copies location to L1 from main memory}

}




## Cache miss penalty

Address not in L1 ⇒ delay in memory access

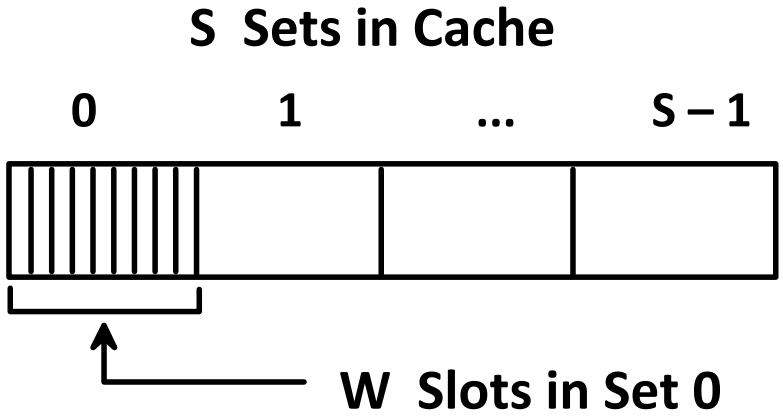
# Cache Organization — 1

## Copy memory data to cache

How much data to copy?

Which data is in cache?

How to find data in cache?




## Copy DATA LINE (block) to cache

Line = **B** bytes

Page size (virtual memory) = integer × block size

## Sets and Slots

Cache = **S** sets

Set = **W** slots

Slot = **1** data line

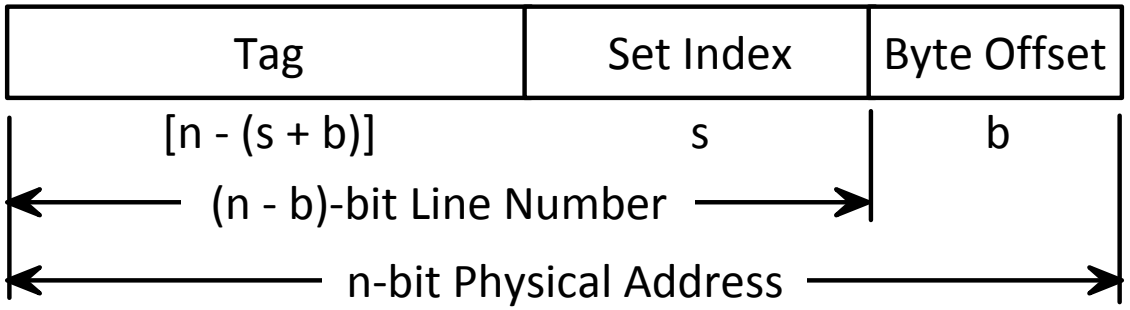
Copy memory line to:

- Specific Set
- Any Slot in Set

# Cache Organization — 2

## Physical address space

N = $2^n$ bytes




## Lines

B = $2^b$ bytes / line

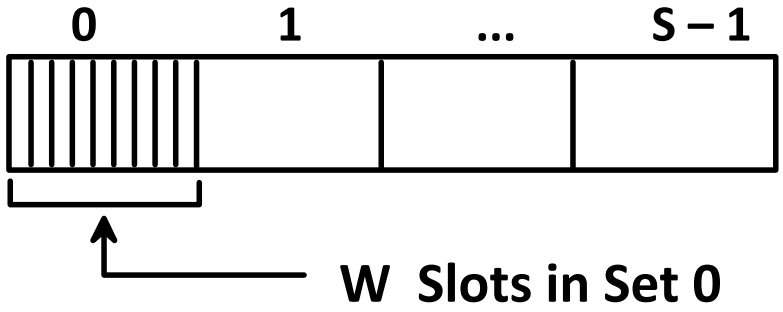
Line number = int (Address / B)

## Sets

S = $2^s$ sets

Set index = s bits

Line → line number % S




## W-way associativity

Line → any slot in set

Identify line by Tag field in cache directory

## Total Cache Size

$$\text{total cache size} = S \times W \times B = \frac{\text{sets}}{\text{cache}} \times \frac{\text{blocks}}{\text{set}} \times \frac{\text{bytes}}{\text{block}} = \frac{\text{bytes}}{\text{cache}}$$

# Cache Definitions and Policies

**Cache hit**

CPU accesses line in cache

**Cache miss**

CPU accesses line not in cache

**Read miss**

Load line to cache

**Write miss**

Write allocate — load line to cache

No write allocate — write to main memory (no load to cache)

**Swapping out cache line**

Need new line in full set

Remove least recently used (LRU)

**WRITE BACK** — update RAM on line swap-out

**WRITE THROUGH** — update RAM on every write to cache

# Cache Performance Issues

## L1 Cache hit

CPU memory access = 1 clock cycle

## L1 Cache Miss

CPU stalls — L1 loads missing line

## Miss Rate

Cache misses per cache accesses

Instruction fetch, load, store

Depends on cache size and organization

## Miss Penalty

Cache update latency

Stall cycles while cache ← missing line

# Performance of Two-Level Unified Cache

## Parameters

$M_{L1}^{I}$ = Instruction Miss Rate at L1

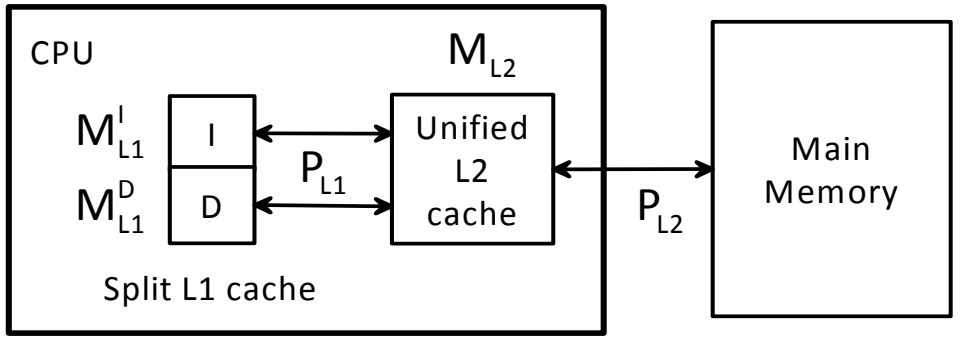
$1 - M_{L1}^{I}$ = Instruction Hit Rate at L1

$M_{L1}^{D}$ = Data Miss Rate at L1

$1 - M_{L1}^{D}$ = Data Hit Rate at L1

$M_{L2}$ = Miss Rate at L2

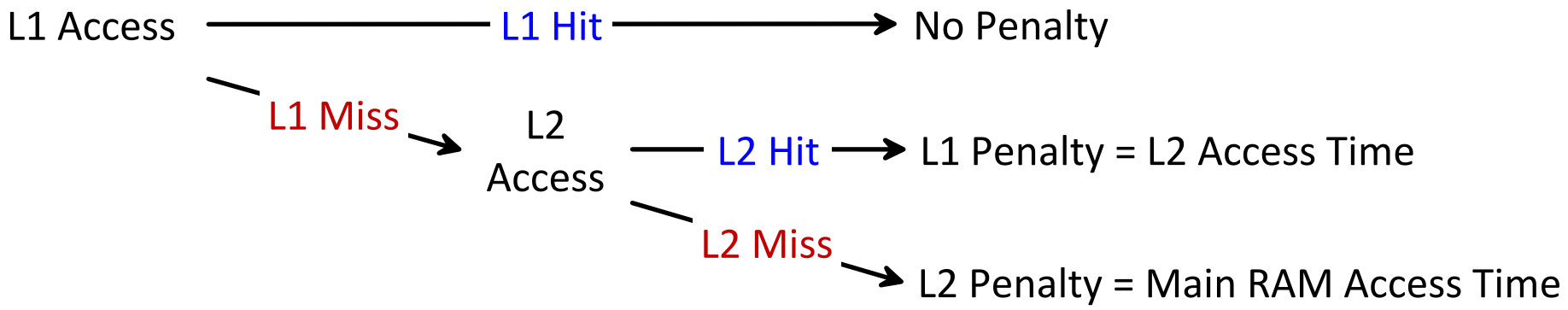
$1\text{-}M_{L2}$ = Hit Rate at L2




$P_{L1}$ = Miss Penalty at L1 $\qquad$ $P_{L2}$ = Miss Penalty at L2

L1 Instruction Access per Instruction = 1

$$\text{L1 Data Access per Instruction} = \frac{IC^{\text{data access}}}{IC}$$




$$CPI_{\text{cache}}^{\text{stall}} = \frac{\text{cache miss stall cycles}}{\text{instruction}} = \left[ M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{\text{data access}}}{IC} \right] \times \left( P_{L1} + P_{L2} \times M_{L2} \right)$$

# 1-Level Cache versus 2-Level Cache

## 1 level cache

$$\mathrm{CPI}^{\text{stall}}_{\text{1-level}} = \left[ M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^{\text{data access}}}{IC} \right] \times \underbrace{\text{Penalty}^{\text{L1-to-RAM}}}_{\text{50 clock cycles}}$$

## 2 level cache

$$\mathrm{CPI}^{\text{stall}}_{\text{2-level}} = \left[ M^{I}_{L1} + M^{D}_{L1} \times \frac{IC^{\text{data access}}}{IC} \right] \times \underbrace{\left( \underbrace{\text{Penalty}^{\text{L1-to-L2}}}_{\text{5 clock cycles}} + \underbrace{M_{L2}}_{<0.01} \times \underbrace{\text{Penalty}^{\text{L2-to-RAM}}}_{\text{50 clock cycles}} \right)}_{<5.5 \text{ clock cycles}}$$

# Issues Affecting Miss Rate

## Compulsory miss

- Demand access update
  - Cache ← line on first access
  - First access to line ⇒ cache miss
- Not affected by cache size or associativity

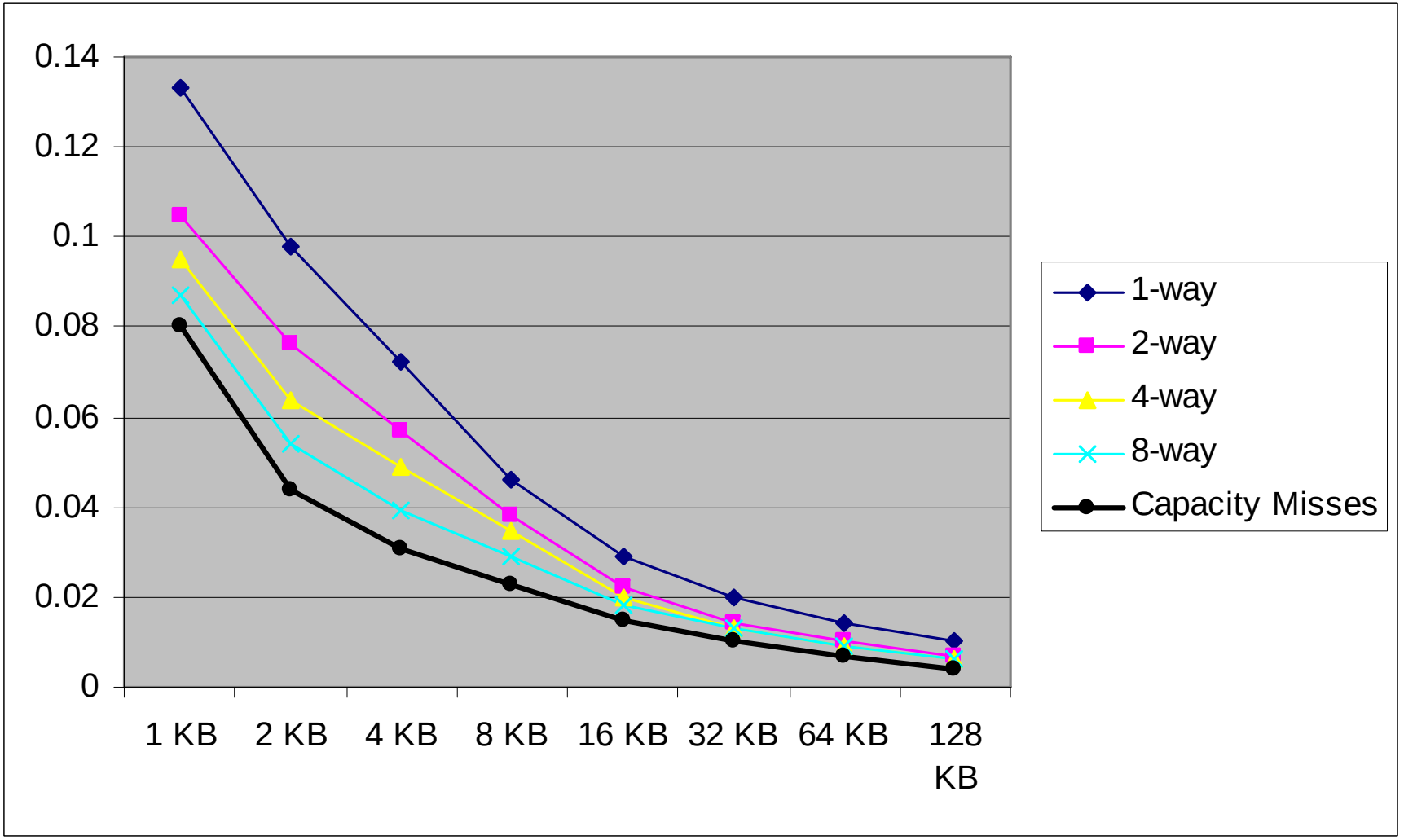
## Capacity miss

- Swap out lines for required load ⇒ miss
- Cache < main memory
- Larger cache ⇒ lower capacity miss rate

## Conflict miss

- Swap out lines for required load ⇒ miss
- Line ← specific set
  - Spare capacity in other sets
  - Example — address aliasing
- Higher associativity ⇒ lower conflict miss rate

# Miss Rate




**Total miss rate drops as capacity or associativity increases**

# Associativity Trade-Off

## Any slot in set ← line

Find line ⇒ search all tags

Larger associativity ⇒ more blocks per set ⇒ longer search time

## For fixed cache capacity = S × W × B

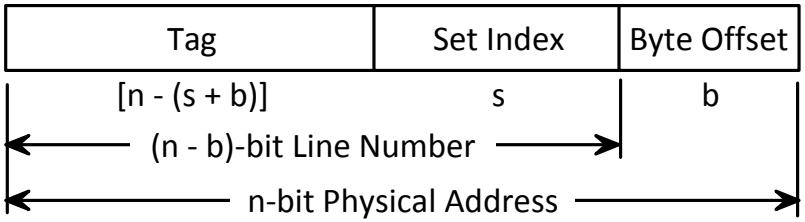
Larger associativity W

⇒ fewer sets S

⇒ smaller set index s

⇒ larger tag size n – (s + b)

⇒ longer tag search

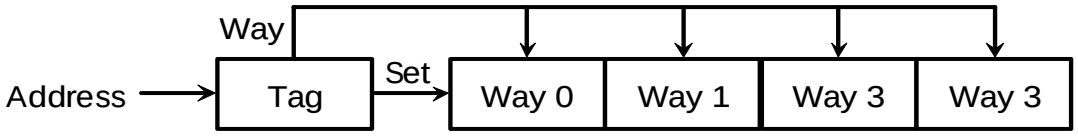



Small advantage beyond 4-way associativity

| | s | n – (s + b) | Tag Search Length |
|---|---|---|---|
| **Direct mapped** | n – b | 0 | $2^0 = 1$ |
| **Fully associative** | 0 | n – b | $2^{n-b}$ |

# Tag Search Alternatives

## Serial access




Address → tag controller

Hit ⇒ memory ← set number / way number

Prepare tagged line for access

Longer latency at lower power

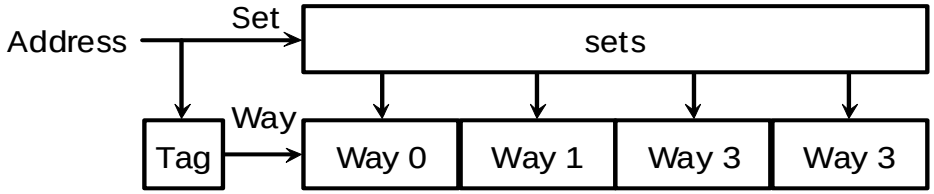
## Parallel access

Address → tag controller + set controller

Prepare set for access

Hit ⇒ memory ← way number

Shorter latency at higher power




## Way prediction

Address fed to tag controller and way predictor

Hit ⇒ way predictor feeds set number and way number to memory

Misprediction ⇒ tag controller feeds correct way number on next CC

# Cache Prefetching

## Prefetching "hides memory latency"

- Predict cache update before demand miss
- Initiate cache update request
- Bring memory block to cache before needed
- Memory latency does not cause delay

## Data cache prefetching

- Useful for sequential accesses to large data structures
- Software prefetch
  - Assembly language prefetch instructions for cache update
  - Inserted by programmer or compiler
- Hardware — stride buffer predicts sequential accesses

## Instruction cache prefetching

- Hardware-based prefetch mechanisms
- Predict instruction cache updates based on program structure

# Data Cache Latency Example

## Loop over large data structure

Accesses 1024 × 1024 SSE (16-byte vector) operands = 16 MB

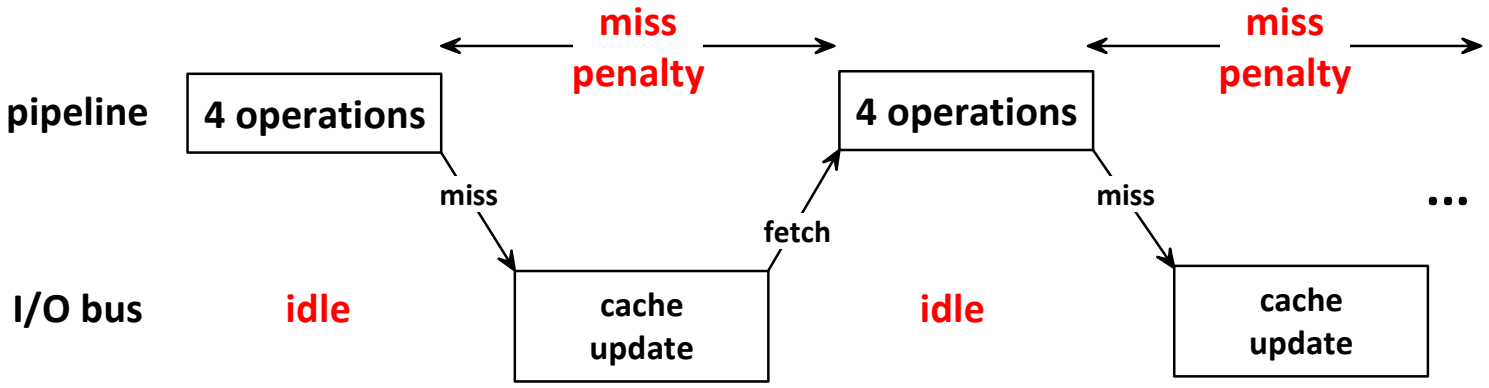
Sequential vector reads — no repeated access to same data

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

## Pentium 4 with 16 KB L1 cache and 64-byte cache line

Data miss every 64 bytes = 4 SSE operands

Data miss in L1 1024 × 1024 / 4 = 256K compulsory misses



# Data Cache Prefetch Instruction

## Software prefetch loads 128 bytes (2 cache lines)

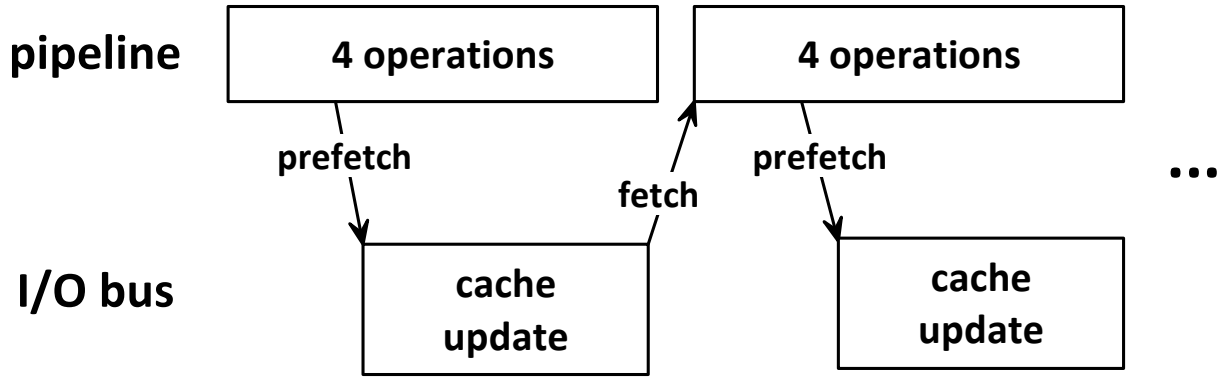
128 bytes = (128/16) = 8 SSE operands

Prefetch 8 operands forward

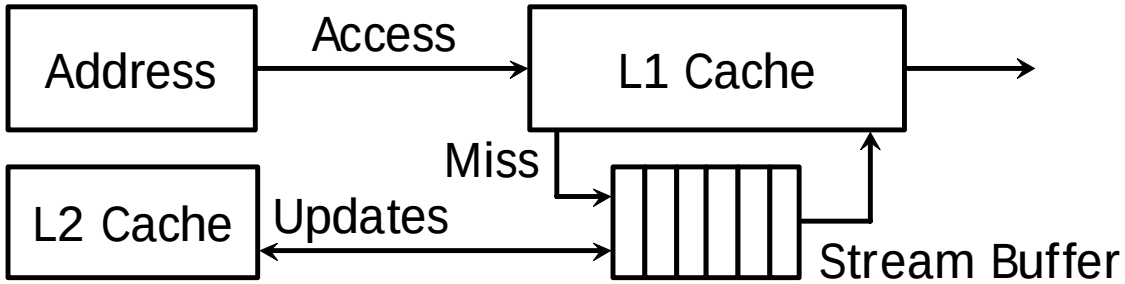
```
for (i = 0; i < 1024; i++) {
     for (j = 0; j < 1024; j++) {
          prefetch a[i][j+8];
          SSE_vector_operation a[i][j];
     }
}
```

## Prefetch filtering

NOP on prefetch of cache hit



# General Stream Buffer




## Stream

Successive accesses to sequential cache lines

## Stream buffer

FIFO buffer

Generates + buffers update requests for successive cache lines

Request buffer implementation

Requested line → cache from higher memory level

Line buffer implementation

Requested line → stream buffer until demand miss

## Request filtering

Stream buffer skips lines already in cache to minimize prefetches

# Stream Buffer Operation

## Cache miss to data line L on CC k

| CC | Request Buffer | Line Buffer |
|---|---|---|
| k | Stream buffer allocated to missed line **L**<br>Buffer generates, buffers, and issues cache request for line L<br>Pipeline suffers miss penalty for line **L** | |
| k + 1, 2, 3, … | Stream buffer generates, FIFO-buffers, and issues cache request for lines **L + 1, 2, 3, …** | |
| k + miss penalty | Line **L** brought to cache | Line **L** brought to cache |
| k + miss penalty + 1 | Line **L + 1** brought to cache | Line **L + 1** brought to stream buffer<br>Copied to cache on demand |
| k + miss penalty + 2 | Line **L + 2** brought to cache | Line **L + 2** brought to stream buffer<br>Copied to cache on demand |
| No further miss penalties<br>Stream buffer issues sequential requests until full or re-allocated to new miss | | |

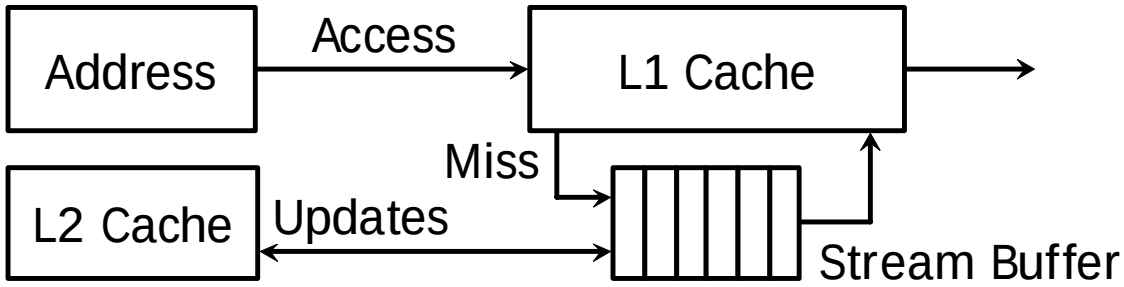
# Stream Buffer Example

## Lines in cache

| 0 | 1 | 2 | 3 | 256 | 257 | 6 | 15 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|

| CC | Line Accessed | Cache | Stream Buffer Slots | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 2 | Hit | | | | | | |
| 1 | 3 | Hit | | | | | | |
| 2 | 4 | Miss | 4 | | | | | |
| 3 | 4 | Miss | 4 | 5 | | | | |
| 4 | 4 | Miss | 4 | 5 | 7 | | Skip 6 | |
| 5 | 4 | Miss | 4 | 5 | 7 | 9 | Skip 8 | |
| 6 | 4 | Miss | 4 | 5 | 7 | 9 | 10 | |
| ... | | | | | | | | |
| 10 | 4 | Hit | 4 | 5 | 7 | 9 | 10 | 11 |
| 11 | 5 | Hit | 4 | 5 | 7 | 9 | 10 | 11 |

# Stride Buffer




## Stride

Sequential accesses to lines L, L+s, L+2s, L+3s, ...

Access stride = s

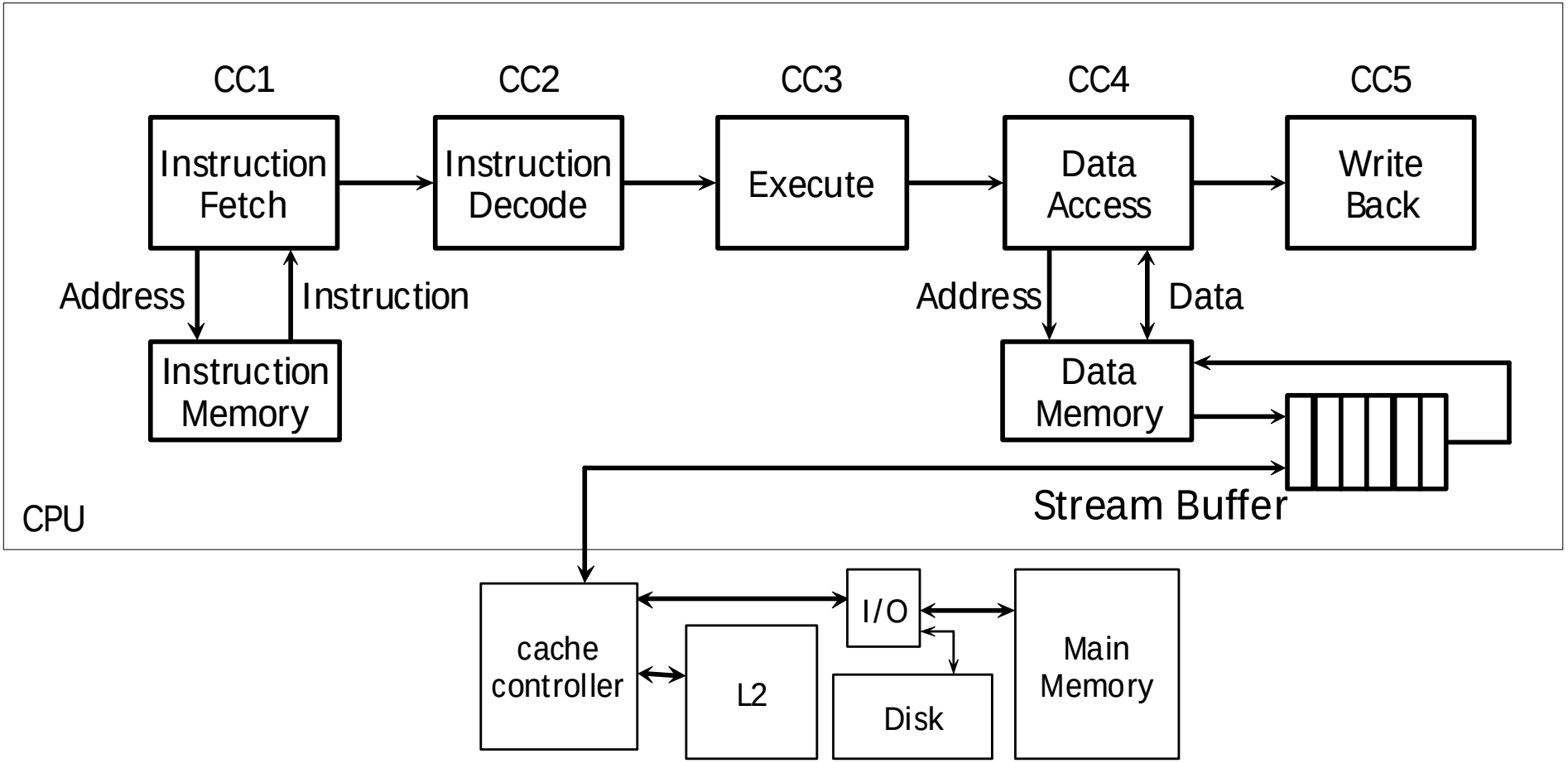
## Stride buffer

Variation of stream buffer

Prefetches lines according to stride s

Stream buffer allocated following 2 sequential cache misses

# Stream Buffer for DLX Data Cache

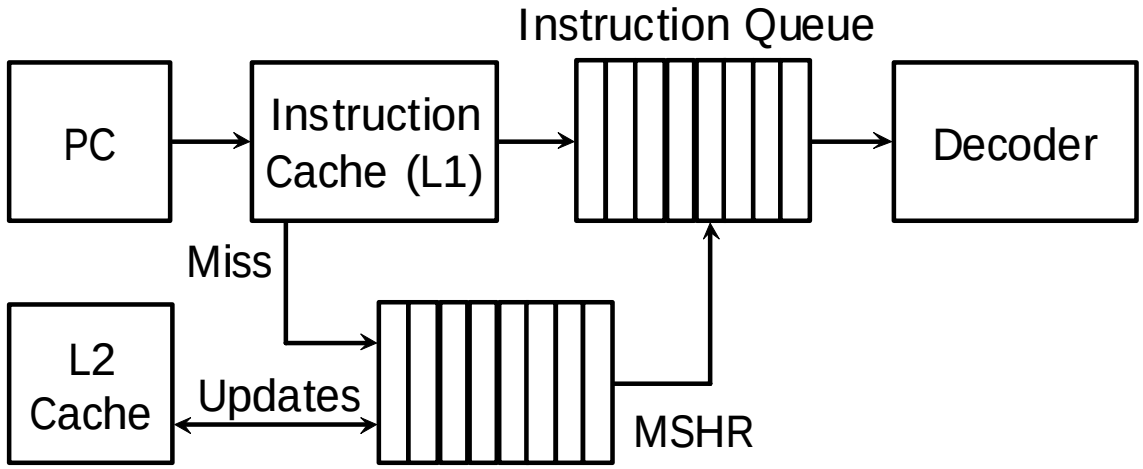


# Out-of-Order Fetch

## Miss Information Status History Register (MSHR)

Variation of stream buffer for instruction cache

Additional random access buffer for instruction blocks




## On cache miss

Allocate MSHR slot to missed address

- Requests cache update for missed address
- Writes pointer for missed addresses in Instruction Queue

PC advances

- L1 hits → Instruction Queue
- L1 misses → Instruction Queue on update

# MSHR Example

## Blocks in cache

| 0 | 1 | 2 | 3 | 256 | 257 | 6 | 15 | 8 | ... |
|---|---|---|---|---|---|---|---|---|---|

| CC | Block In PC | Cache | MSHR Slots | | | Instruction Queue | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 2 | Hit | | | | 2 | | | | | | |
| 1 | 3 | Hit | | | | 3 | | | | | | |
| 2 | 4 | Miss | 4 | | | P4 | | | | | | |
| 3 | 5 | Miss | 4 | 5 | | P4 | P5 | | | | | |
| 4 | 6 | Hit | 4 | 5 | | P4 | P5 | 6 | | | | |
| 5 | 7 | Miss | 4 | 5 | 7 | P4 | P5 | 6 | P7 | | | |
| 6 | 8 | Hit | 4 | 5 | 7 | P4 | P5 | 6 | P7 | 8 | ... | ... |
| ... | | | | | | | | | | | | |
| 50 | | | 4 | 5 | 7 | 4 | P5 | 6 | P7 | 8 | ... | ... |
| 51 | | | 4 | 5 | 7 | 5 | 6 | P7 | 8 | ... | ... | ... |

# Fetch Directed Instruction Prefetching

**Branch prediction → cache prefetch**

FIFO between branch predictor and L1 instruction cache

**Branch detected at address PC**

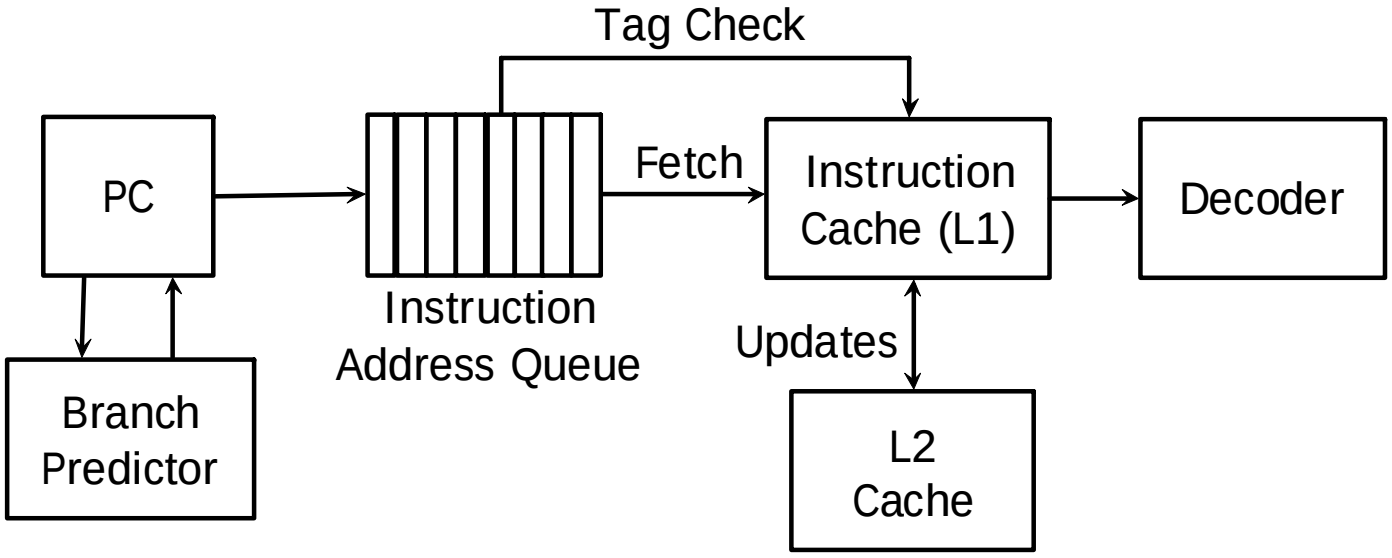
Branch prediction

Predicted next PC → address queue

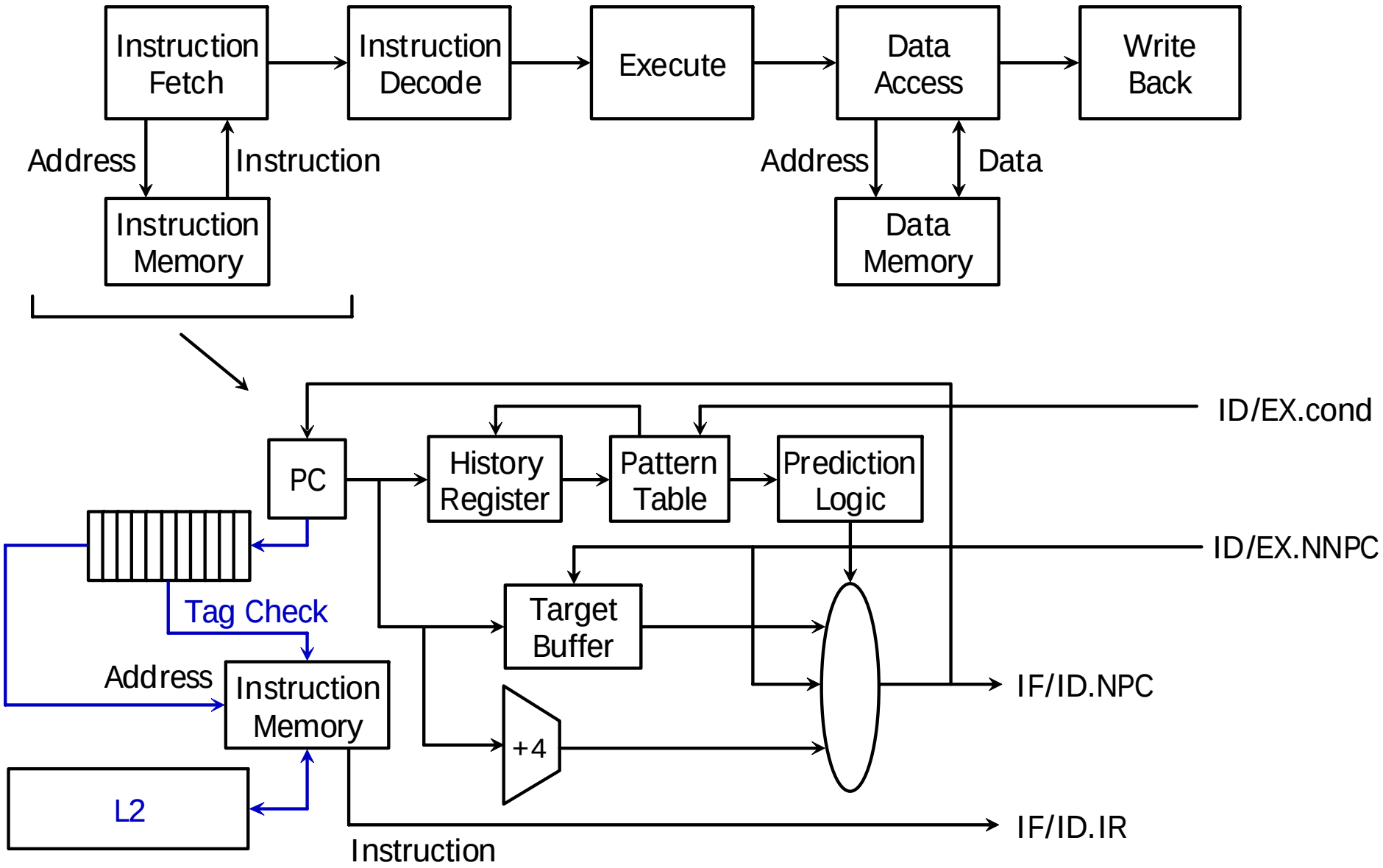
Address queue initiates tag check

Cache update on miss

L1 prefetches missed line before access to fetch address



# Fetch Directed Prefetching for DLX



# Advanced Techniques for Avoiding Misprediction

# Branch Mispredictions Limit ILP

## Advanced superscalar techniques

- Fast decoding
- Branch prediction
- Cache prefetch

## Long speculative instruction sequences in cache

- Misprediction $\Rightarrow$ long recovery process
  - Cancel large block of executed instructions
  - Evict and replace many instruction cache lines
  - Decode/reschedule many instructions for instruction pool

## Misprediction $\rightarrow$ predict-prefetch-decode bottleneck

- Predict-prefetch-decode
  - Stop advance
  - Switch to recovery
- EUs starve (stall)
- Instruction pool fill disrupted

# Trace Cache

## Trace

- Long instruction sequence in dynamic execution order
  - Multiple branch actions (taken/not-taken)
  - Dynamic scheduling (actual execution order in CPU)

## Trace cache

- Cache of instruction traces
- Constructed dynamically on first run of trace
  - Fetch → branch predict → decode → reschedule → execute
  - Instructions come from multiple instruction cache lines

## Rapid recovery

- Misprediction → hit in trace cache
  - Copy trace to instruction pool from cache
  - Skips fetch → branch predict → decode → reschedule

# Trace Cache Construction

## Trace generation

- Load cache lines → fetch instructions
- Predict branches
- Decode instructions → instruction pool
- Instructions dynamically scheduled for execution in EUs

## Trace cache entry

- Start on target or fall-through of branch
- Trace embeds next few branch actions
  - Validated taken / not-taken execution paths

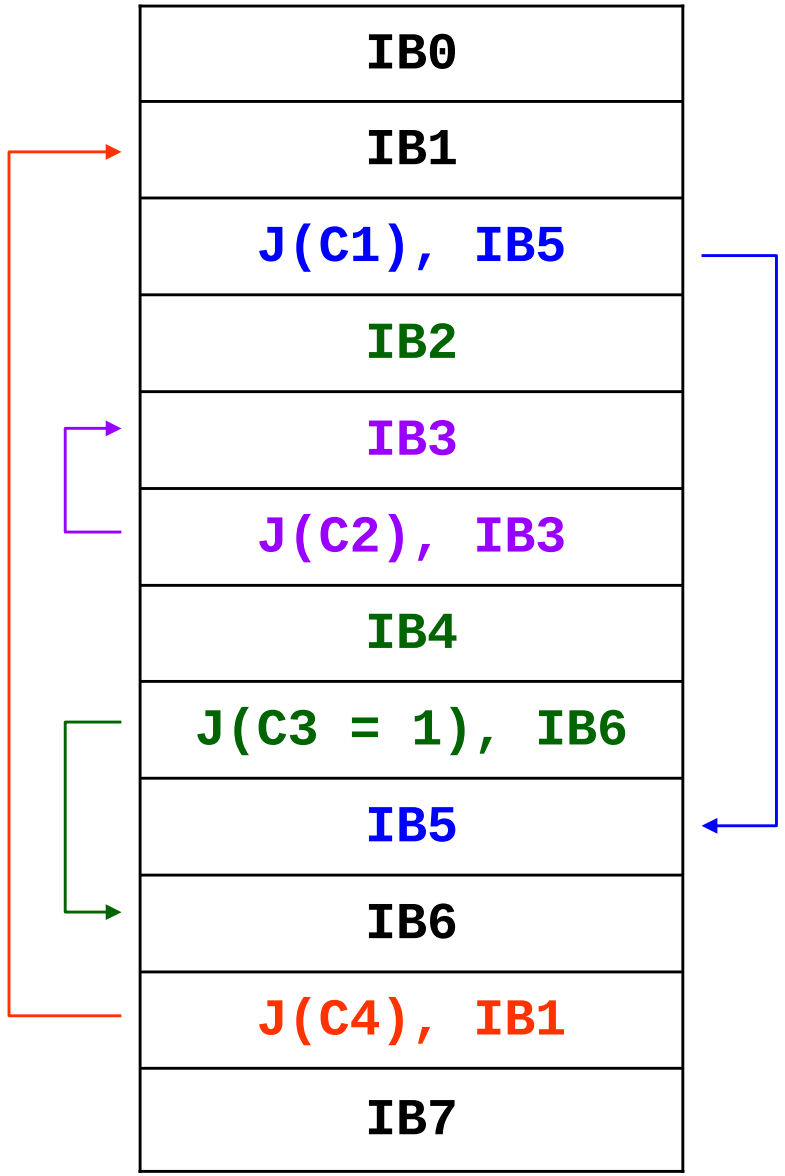
## Trace cache entry ID

- PC of first instruction
- Sequence of taken / not-taken actions embedded in trace

# Program Fragment

## IB = instruction block

```
IB0(...);
while (C4) {
      IB1(...);
      if (C1)
         IB5(...);
      else {
         IB2(...);
         while (C2) IB3(...);
         IB4(...);
      }
      IB6(...);
}
IB7(...);
```



# Trace Construction Example — 1

## Program fragment

32 instructions

$I_0 - I_{31}$

3 conditional branches

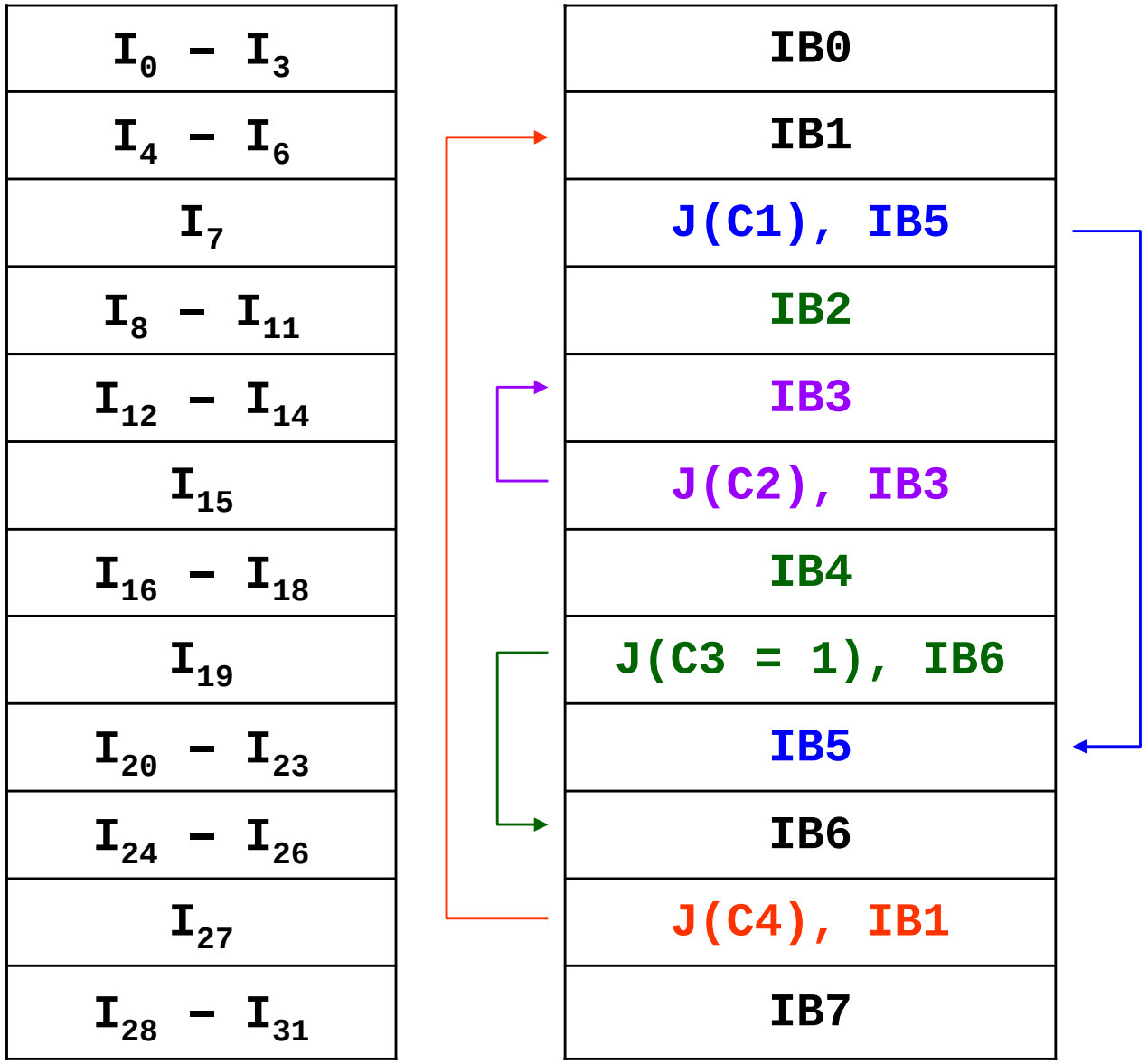
$I_7$ , $I_{15}$ , $I_{27}$

1 absolute branch

$I_{19}$

## Trace cache

8 instructions

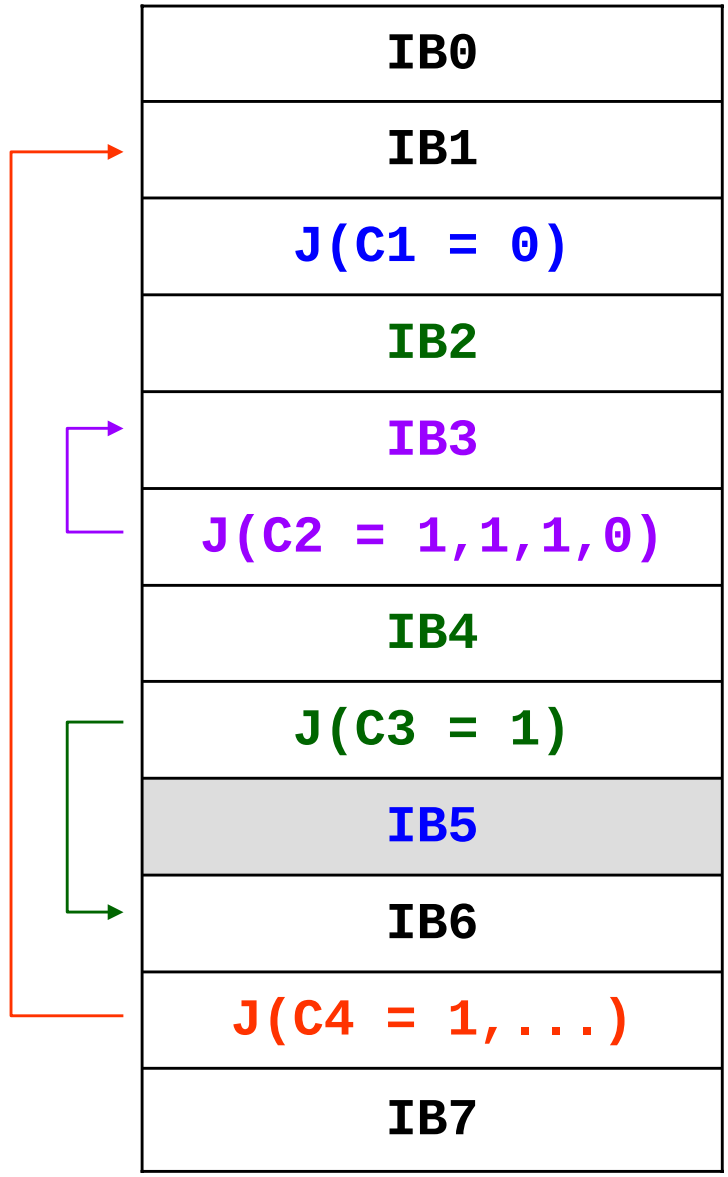


# Trace Construction Example — 2

## Runtime sequence

$IB_0$
$IB_1$
J(C1 = 0)
$IB_2$
$IB_3$
J(C2 = 1)
$IB_3$
J(C2 = 1)
$IB_3$
J(C2 = 1)
$IB_3$
J(C2 = 0)
$IB_4$
J(C3 = 1)
$IB_6$
J(C4 = 1)
$IB_1$ ...

| |
|---|
| $I_0$ – $I_3$ |
| $I_4$ – $I_6$ |
| $I_7$ |
| $I_8$ – $I_{11}$ |
| $I_{12}$ – $I_{14}$ |
| $I_{15}$ |
| $I_{16}$ – $I_{18}$ |
| $I_{19}$ |
| $I_{20}$ – $I_{23}$ |
| $I_{24}$ – $I_{26}$ |
| $I_{27}$ |
| $I_{28}$ – $I_{31}$ |



# Trace Construction Example — 3

## Runtime sequence

| | | |
|---|---|---|
| $IB_0$ | $I_0$ – $I_3$ | |
| $IB_1$ | $I_4$ – $I_6$ | T4 |
| J(C1 = 0) | $I_7$ | |
| $IB_2$ | $I_8$ – $I_{11}$ | T1 |
| $IB_3$ | $I_{12}$ – $I_{14}$ | |
| J(C2 = 1) | $I_{15}$ | |
| $IB_3$ | $I_{12}$ – $I_{14}$ | T2 |
| J(C2 = 1) | $I_{15}$ | |
| $IB_3$ | $I_{12}$ – $I_{14}$ | |
| J(C2 = 1) | $I_{15}$ | |
| $IB_3$ | $I_{12}$ – $I_{14}$ | T3 |
| J(C2 = 0) | $I_{15}$ | |
| $IB_4$ | $I_{16}$ – $I_{18}$ | |
| J(C3 = 1) | $I_{19}$ | |
| $IB_6$ | $I_{24}$ – $I_{26}$ | T4 |
| J(C4 = 1) | $I_{27}$ | |

## First Traces in Cache

(start after first branch)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| T1 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| T2 | 12 | 13 | 14 | 15 | 12 | 13 | 14 | 15 |
| T3 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| T4 | 24 | 25 | 26 | 27 | 4 | 5 | 6 | 7 |

## Typical iteration of main loop

T1 T2 T3 T4

## Initial misprediction

Fourth J(C2)

Predict C2 = 1 ⇒ T2

Validate C2 = 0

Cancel T2

Load T3

# Trace Cache Header

| header | | | | | | trace instructions | | | |
|---|---|---|---|---|---|---|---|---|---|
| start PC | flag | mask | end type | fall through | target | | | … | |
| PC-1 | 0111 | 1111 | conditional | PC-2 | PC-3 | | | … | |
| PC-2 | 110x | 1110 | absolute | — | PC-4 | | | … | |
| PC-3 | 0xxx | 1000 | no branch | PC-4 | — | | | | |

**Start PC**

Address of 1st instruction in trace

Low bits = index to trace cache

High bits = tag

**Flag**

Sequence of branch actions in trace

0/1 = not taken / not taken

x = not valid (don't care)

**Mask**

1 = valid flag bit in position

Sum of mask = branches in trace

**End type**

Type of last instruction in trace

Conditional branch

Absolute branch

No branch

**Fall-through / target**

PC address for next instruction fetch

Points to trace or instruction

Target for flag = 1 (taken)

Fall-through for flag = 0 (not-taken)

# Trace Cache Header Example

| header | | | | | | trace instructions | | | |
|---|---|---|---|---|---|---|---|---|---|
| start PC | flag | mask | end type | fall through | target | | | … | |
| PC-1 | 0111 | 1111 | Conditional (C) | PC-2 | PC-3 | | | … | |
| PC-2 | 110x | 1110 | Absolute (A) | — | PC-4 | | | … | |
| PC-3 | 0xxx | 1000 | no branch (N) | PC-4 | — | | | | |

| | start PC | flag | mask | end type | fall through | target | trace instructions | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **T1** | **8** | **10** | **1x** | **C** | **16** | **12** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **T2** | **12** | **11** | **11** | **C** | **16** | **12** | **12** | **13** | **14** | **15** | **12** | **13** | **14** | **15** |
| **T3** | **12** | **01** | **11** | **A** | **—** | **24** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** |
| **T4** | **24** | **10** | **11** | **C** | **8** | **20** | **24** | **25** | **26** | **27** | **4** | **5** | **6** | **7** |

# Trace Cache Line Length

## Trace length = instruction cache line length

Longer ⇒ multiple L1 accesses ⇒ higher miss rate

Shorter ⇒ blocks poorly with L1 ⇒ inefficient trace

## Maximum branches per trace

Limit = branch predictor throughput

Predictions per CC

Terminate trace at subroutine returns

Return address varies ⇒ correct embedding difficult

Trace termination

Fill trace to end with NOPs (not scheduled in instruction pool)

Example

| 0 | 1 | 2 | … | E | F |
|---|---|---|---|---|---|
| Trace length = cache line length = 16 bytes | | | | | |

| Instructions | | | | Terminated | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | … | B | NOP | NOP | NOP | NOP |

# Instruction Fetch Unit

## Multiple Branch Predictor (MBP)

Predicts PC for traces

Access to trace cache and L1 instruction cache in parallel

## PC misses in trace cache

Fetch+decode from L1 cache to instruction pool

Next PC logic predicts next branch action

Target predictor → next PC

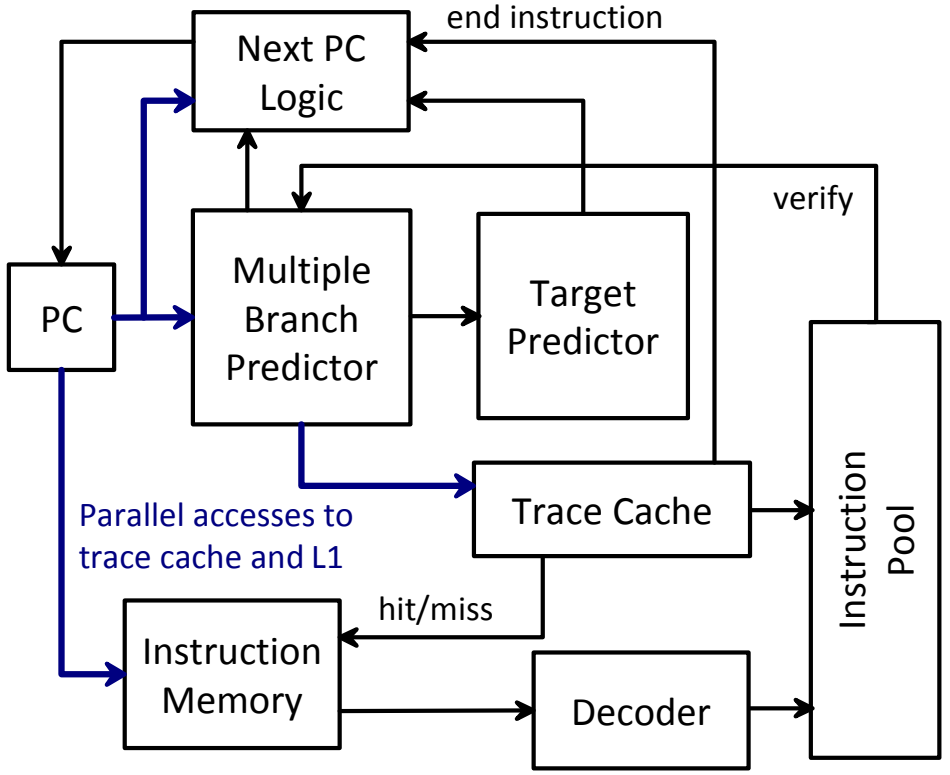
## PC hits in trace cache

Cancel L1 access

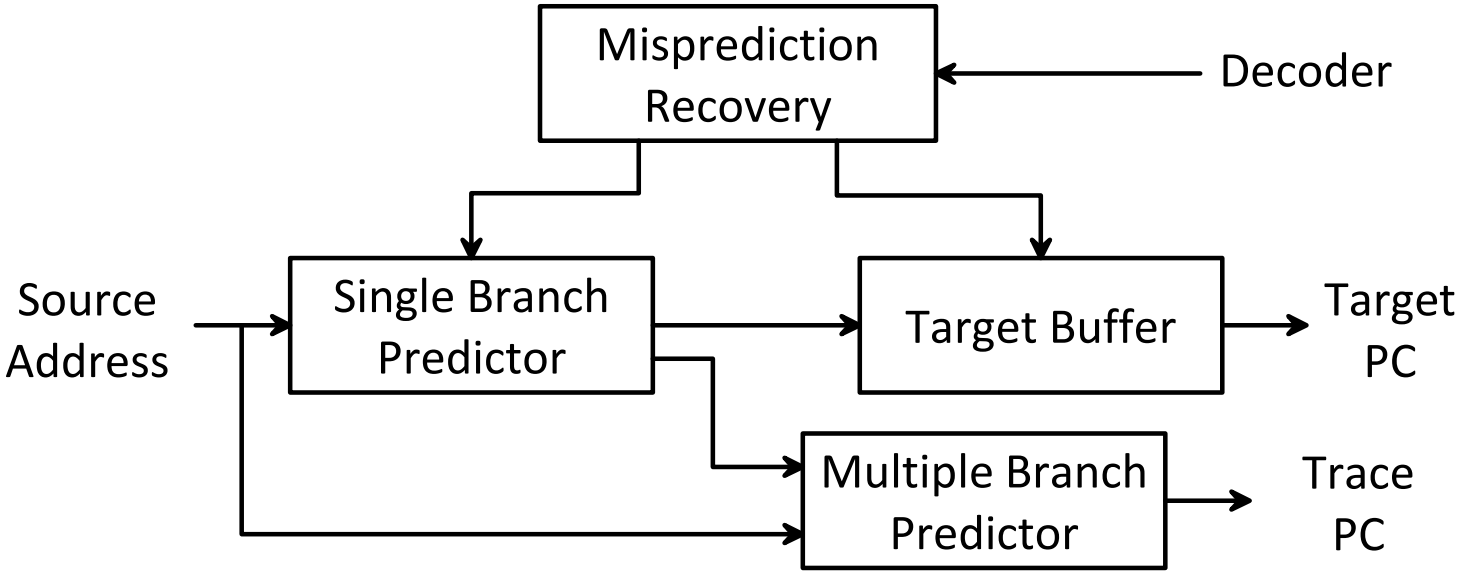
Fetch trace to instruction pool

MBP predicts embedded branches

Target predictor not used



# Multiple Branch Predictor




## Single branch predictor

Standard taken / not taken prediction (1-level or 2- level)

## Multiple branch predictor

Predicts m embedded branches in trace

m-bit bitmap (0 = NT or 1 = T)

Bit = last prediction for branch from Single Branch Predictor

# Multiple Branch Predictor Example

## Instruction sequences

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| T1 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| T2 | 12 | 13 | 14 | 15 | 12 | 13 | 14 | 15 |
| T3 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| T4 | 24 | 25 | 26 | 27 | 4 | 5 | 6 | 7 |

## Single Branch History Register

| | profile | | | | target |
|---|---|---|---|---|---|
| $I_7$ | 0 | 0 | 0 | 0 | $I_{20}$ |
| $I_{15}$ | 1 | 1 | 1 | 0 | $I_{12}$ |
| $I_{19}$ | 1 | 1 | 1 | 1 | $I_{24}$ |
| $I_{27}$ | 1 | 1 | 1 | 1 | $I_4$ |

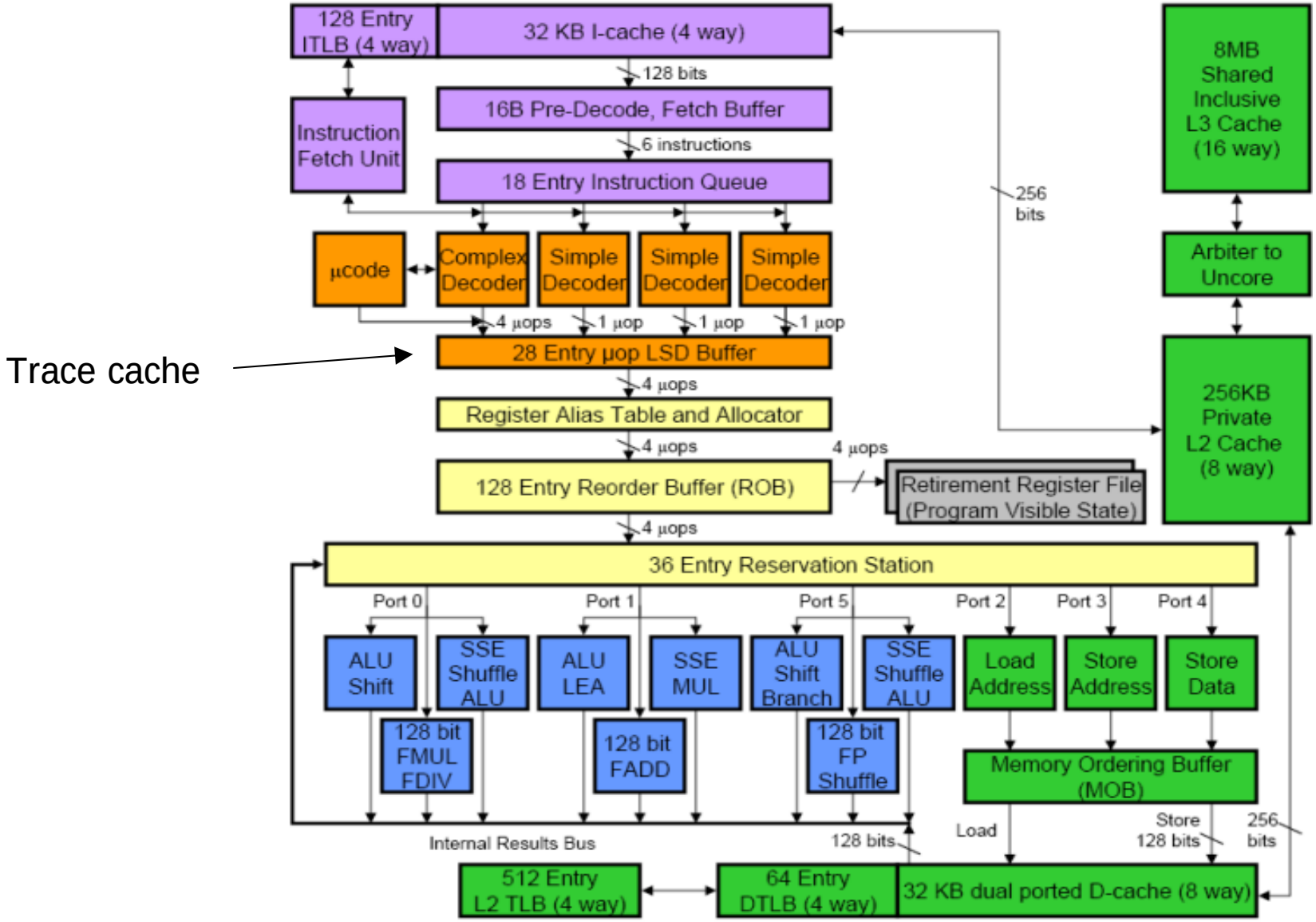
## Single Branch Pattern Table

| profile | pattern | |
|---|---|---|
| 0000 | 0 | 0 |
| 1111 | 1 | 1 |
| 1110 | 1 | 1 |
| 1101 | 1 | 1 |
| 1011 | 1 | 1 |
| 0111 | 0 | 0 |

## Multiple Branch Predictor (MBP)

| Target | Trace | Predictions | | Branch |
|---|---|---|---|---|
| $I_8$ | T1 | 1 | – | 15 |
| $I_{12}$ | T2 | 1 | 1 | 15-15 (for profile != 0111) |
| | T3 | 0 | 1 | 15-19 (for profile == 0111) |
| $I_{24}$ | T4 | 1 | 0 | 27-7 |

# 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

# Branch Predication

# Another Look at Branches

## Two basic branch types

| Branch type | loop | if-condition |
|---|---|---|
| **Branch direction** | Jump backward<br>target PC < fall-through PC | Jump forward<br>target PC > fall-through PC |
| **Control type** | code reuse on data streams | forward path of program flow |
| **Predictability** | high<br>repetitive iteration pattern | low<br>exotic data dependent statistics |
| **Branch density** | long loops ⇒ low density | high<br>run 20% to 25% of instructions |
| **Branch prediction and trace caching** | good performance | poor performance<br>expensive to implement<br>8 parallel instructions ⇒ 2 branches per clock cycle<br>difficult to update branch history |

# Predication

## Predication

Predication registers **`p0, p1, …, pN`**

**`<p1> Instruction`**

**`p1 = 1`** ⇒ CPU executes **`Instruction`**

**`p1 = 0`** ⇒ CPU executes **`NOP`**

Generalization of **`compare`** → **`conditional branch`** pairs

## Alternative to if-condition branches

**`Test_and_set`** instructions for predication registers

Predicate standard instruction set (if-conversion)

Trade-off — eliminate branch stall at cost of canceling extra work

## Example

```
    BNEZ R1, L1
    SUBI R1, R1, #1
L1: ADD R2, R2, R1
```

```
     p1 = CMP R1, 0
<p1> SUBI R1, R1, #1
     ADD R2, R2, R1
```

# Predication in IA-64

## IA-64 (Intel 64-bit architecture)

ISA for Intel Itanium series processors

## Static predicate registers `p0` — `p15`

1-bit width written by compare instructions

**`Regs[p0]`** = **`1`**

## Rotating predicate registers

48 renamed virtual copies of **`p0`** — **`p15`**

Allocated dynamically by hardware for parallel loop iterations

## General instruction syntax

**`[(qp)] mnemonic[.comp1][.comp2] dests = srcs`**

**`(qp)`** — optional predication of instruction

**`.comp`** — optional comparison type

**`dest`** — destination

**`dest = p1`** ⇒ **`p1`** ← **`1`** on true and **`p1`** ← **`0`** on false

# IA-64 Predication Example

```
switch (type)
{
  case 'a':
    type=type+10;
    break;
  case 'b':
    type=type+20;
    break;
  default:
    break;
}
```

Compile for IA-32

```
     MOV  EAX, [type]
     CMP EAX, 'a'
     JNE L1
     ADD EAX, 10
     JMP L2
L1:  CMP EAX, 'b'
     JNE L2
     ADD EAX, 20
L2:  MOV [type], EAX
```

Compile for IA-64

```
mov   r1 = type
ld4   r2 = [r1]
cmp.eq   p1 = 'a',r2
cmp.eq   p3 = 'b',r2
(p1)   add r2 = 10,r2
(p3)   add r2 = 20,r2
st4   [r1] = r2
```

Scheduling

| | |
|---|---|
| CC1 | mov r1 = type<br>ld4 r2 = [r1] |
| CC2 | cmp.eq p1 = 'a',r2<br>cmp.eq p3 = 'b',r2 |
| CC3 | (p1) add r2 = 10,r2<br>(p3) add r2 = 20,r2 |
| CC4 | st4 [r1] = r2 |

Ref: "Introducing the Intel Itanium Architecture"

# Branching versus Predication

```
     MOV  EAX, [type]
     CMP EAX, 'a'
     JNE L1
     ADD EAX, 10
     JMP L2
L1:  CMP EAX, 'b'
     JNE L2
     ADD EAX, 20
L2:  MOV [type], EAX
```

| | |
|---|---|
| CC1 | `mov  r1 = type`<br>`ld4  r2 = [r1]` |
| CC2 | `cmp.eq  p1 = 'a',r2`<br>`cmp.eq  p3 = 'b',r2` |
| CC3 | `(p1)  add r2 = 10,r2`<br>`(p3)  add r2 = 20,r2` |
| CC4 | `st4  [r1] = r2` |

## Branching

- 9 instructions
- 3 run-once branch instructions
  - Branches predicted not-taken
  - Type = 'a' ⇒ 1 mispredictions
  - Type = 'b' ⇒ 1 mispredictions
  - Default ⇒ 2 mispredictions

## Predication

- 7 instructions
- No branch instructions
  - No mispredictions
- Type = default ⇒ CC3 cancelled
  - Equivalent to stall
- Removes 1 stall for each type

# Predication in ARM

## Flag set suffix S

ALU instructions set status flags

**N — Negative** **Z — Zero**

**C — Carry** **V — oVerflow**

## Conditional execution suffix

Execute instructions if flag combination is true

## Example — operate-compare-if-else

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

# Predication in PIC

## Family of microcontrollers for embedded systems

Hundreds of versions — 8, 16, 32 bit data word

| | | |
|---|---|---|
| `BTFSC f, b` | `Cancel next instruction if bit b in word f = 0` | |
| `BTFSS f, b` | `Cancel next instruction if bit b in word f = 1` | |
| `INCFSZ f, d` | `d ← f + 1` | `Cancel next instruction if result = 0` |
| `DECFSZ f, d` | `d ← f – 1` | `Cancel next instruction one if result = 0` |

**Switch Implementation in PIC Assembly language**

```
BTFSC PIR1, TMR1IF     ; Cancel next if (PIR1<TMR1IF> == 0)
GOTO T1_INT            ; branch to Timer1 ISR
BTFSC PIR1, ADIF       ; Cancel next if (PIR1<ADIF> == 0)
GOTO AD_INT            ; branch to A/D ISR
BTFSC PIR1, LCDIF      ; Cancel next if (PIR1<LCDIF> == 0)
GOTO LCD_INT           ; branch to LCD ISR
BTFSC PIR1, RBIF       ; Cancel next if (PIR1<RBIF> == 0)
GOTO PORTB_INT         ; branch to PortB ISR
GOTO INT_ERROR_LP1     ; default ISR
```

# Generalized Predication Model

## 1-bit width predicate registers

**p0 — pN**

**Regs[p0] = 1**

## Predication

Every standard instruction can be predicated (guarded)

## Predicate comparison instructions

Standard integer, unsigned, float, double comparisons of ISA

Can take 1 or 2 predicate register destinations

Syntax

**$p_1$ = CMP.type1 $src_1$,$src_2$ <p>**

**$p_1$,$p_2$ = CMP.type1.type2 $src_1$,$src_2$ <p>**

**$p_1$,$p_2$** — 1 or 2 destinations of specified types

**C = CMP.type1.type2** — 1 or 2 compare conditions

**<p>** — instruction canceled if **p = 0**

# Predicate Types

## Predicate registers assigned according to deposit types

| | | unconditional | | conditional | | OR | | AND | | ∨ | | ∧ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| p | C | T | F | T | F | T | F | T | F | T | F | T | F |
| 0 | 0 | 0 | 0 | - | - | - | - | - | - | - | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | - | - | - | - | - | - | 1 | - | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | - | 1 | 0 | - | 1 | 1 | 0 | - |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | - | - | 0 | 1 | 1 | - | 0 |

| | Assignment | | Condition for Assignment |
|---|---|---|---|
| | T | F | |
| unconditional | (p && C) | (p && C') | — |
| conditional | C | C' | (p = 1) |
| OR | 1 | 1 | (p && C,C') = 1 |
| AND | 0 | 0 | (p && !(C,C')) = 1 |
| ∨ | 1 | 1 | (p \|\| C,C') = 1 |
| ∧ | 0 | 0 | (p && C,C') = 0 |

# Direct Predicate Access

## Predicate clearing and setting

**`pclear`** and **`pset`** clear (assign 0) and set (assign 1)

Act on a predicate register or group of predicate registers

## Integer register to predicate register move

Copy 64-bit integer register to 64 predicate registers

Copy 64 predicate registers to 64-bit integer register

## Predicate saving and restoring

Predicate registers are essential part of architectural state

Saved before procedure call, context switch, etc.

Restored after procedure call, context switch, etc.

**`pspill`** and **`pfill`** push and pop predicate registers to/from stack

# Predicate Concatenation Examples

## Sequence of predicate instructions

| | | |
|---|---|---|
| p1 = C1.UT | p1 ← condition C1 | – |
| p1 = C2.OT | if C2 = 1<br>  p1 ← 1<br>else<br>  no change in p1 | (1 && C2) = 1 |
| p1 = C3.AT | if C3 = 0<br>  p1 ← 0<br>else<br>  no change in p1 | (1 && !C3) = 1 |
| p2 = 1 | p2 ← 1 | – |
| p2 = C4.∧T  <p1> | if (C4 = 0 OR p1 = 0)<br>  p2 ← 0<br>else<br>  no change in p2 | ($p_1$ && C4) = 0 |

# Predication Example — 1

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

gcc output for IA-32 (Linux)

```
 mov [x], 0x0
     jmp L2
     ; miss on 1st
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jle L1
     ; miss on 1st and last
```

Predicated version

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     mov eax, [x]  <p1>
     mov [y], eax  <p1>
     add [x], 0x1  <p1>
     mov eax, [y]  <p1>
     add eax, [x]  <p1>
     mov [z], eax  <p1>
     jmp L1        <p1> ; miss on 1st
```

# Predication Example — 2

```
main()
{
   int x = 0, y, z;
   while (x < 10 && y < 5){
      y = x++;
      z = x + y;
   }
}
```

gcc output for IA-32 (linux)

```
     mov [x], 0x0
     jmp L2
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jg end
     cmp [y], 0x4
     jle L1
end:
```

Predicated version

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.AT [y], 0x4
     mov eax, [x]  <p1>
     mov [y], eax  <p1>
     add [x], 0x1  <p1>
     mov eax, [y]  <p1>
     add eax, [x]  <p1>
     mov [z], eax  <p1>
     jmp L1        <p1>
```

```
; p1 = 0 if !C = 1
```

# Predication Example — 3

```
main()
{
   int x = 0, y, z;
   while (x < 10 || y < 5){
      y = x++;
      z = x + y;
   }
}
```

gcc output for IA-32 (linux)

```
     mov [x], 0x0
     jmp L2
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jle L1
     cmp [y], 0x4
     jle L1
```

Predicated version

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.OT [y], 0x4
     mov eax, [x]  <p1>
     mov [y], eax  <p1>
     add [x], 0x1  <p1>
     mov eax, [y]  <p1>
     add eax, [x]  <p1>
     mov [z], eax  <p1>
     jmp L1        <p1>
```

```
; p1 = 1 if C = 1
```

# Predication Example — 4

```
main()
{
   int x = 0, y, z;
   while (x < 10 && y < 5 || z < 20){
      y = x++;
      z = x + y;
   }
}
```

gcc →

```
     mov [x], 0x0
     jmp L2
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [y]
     add eax, [x]
     mov [z], eax
L2:  cmp [x], 0x9
     jg L3
     cmp [y], 0x4
     jle L1
L3:  cmp [z], 0x13
     jle L1
```

Predicated version ←

```
; p1 = 0 if !C = 1
; p1 = 1 if C = 1
```

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.AT [y], 0x4
     p1 = cmp.le.OT [z], 0x13
     mov eax, [x]   <p1>
     mov [y], eax   <p1>
     add [x], 0x1   <p1>
     mov eax, [y]   <p1>
     add eax, [x]   <p1>
     mov [z], eax   <p1>
     jmp L1         <p1>
```

# Predication Example — 5a

```
main()
{
   unsigned x = 0, y, z;
   while (x < 10 && y < 5){
      y = x++;
      if (x % 2 == 0)
         z = x + y;
      else
         z = x - y;
   }
}
```

gcc →

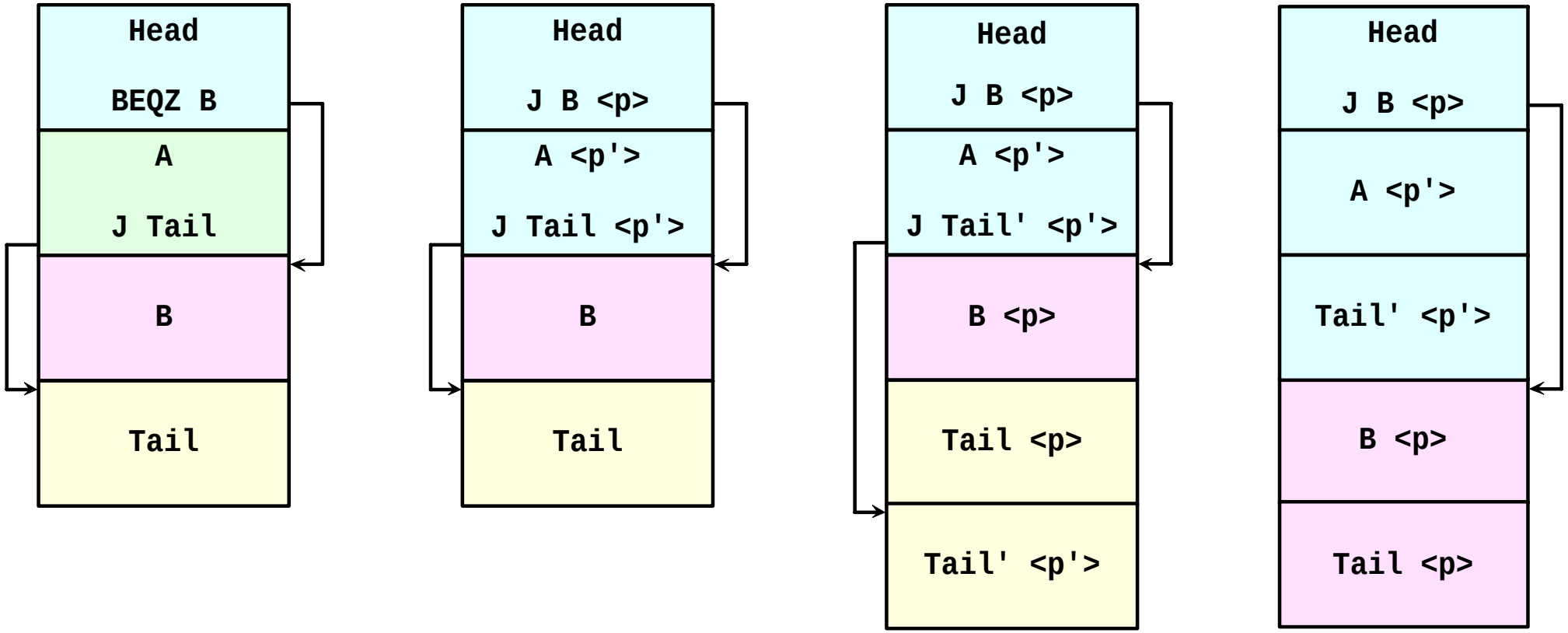
```
     mov [x], 0x0
     jmp L3
L1:  mov eax, [x]
     mov [y], eax
     add [x], 0x1
     mov eax, [x]
     and eax, 0x1
     test eax, eax
     jne L2
     mov eax, [y]
     add eax, [x]
     mov [z], eax
     jmp L3
L2:  mov eax, [y]
     sub eax, [x]
     mov [z], eax
L3:  cmp [x], 0x9
     ja end
     cmp [y], 0x4
     jna L1
end
```

# Predication Example — 5b

```
main()
{
   unsigned x = 0, y, z;
   while (x < 10 && y < 5){
      y = x++;
      if (x % 2 == 0)
         z = x + y;
      else
         z = x - y;
   }
}
```

```
     mov [x], 0x0
L1:  p1 = cmp.le.UT [x], 0x9
     p1 = cmp.le.AT [y], 0x4
     mov eax, [x]                          <p1>
     mov [y], eax                          <p1>
     add [x], 0x1                          <p1>
     mov eax, [x]                          <p1>
     and eax, 0x1                          <p1>
     p2,p3 = cmp.eq.UT.UF eax, 0x0         <p1>
     mov eax, [y]                          <p2>
     add eax, [x]                          <p2>
     mov [z], eax                          <p2>
     mov eax, [x]                          <p3>
     sub eax, [y]                          <p3>
     mov [z], eax                          <p3>
     jmp L1                                <p1>
```

# Hyperblock Formation




## General form of if-conversion

- Height reduction of control dependencies
  - Replace conditional branch (A or B) with predication
  - Fewer branch decisions ⇒ greater ILP and fewer distinct traces in trace cache
- Duplicate Tail code to avoid unconditional branch
- Compromise between branching and predication

# Compiler Issues

## Predication introduced as compiler optimization

- First pass — standard compilation with branches
- Optimization pass
  - if-conversion of branches to predication
  - Requires high level view of large sections of code
  - Generally start from innermost if-then-else sections

## Predication not yet fulfilling its potential for improvement

- Complex issues in compiler theory
  - Exposure of inherent ILP
  - Branch removal versus excess fetching for cancelled instructions
  - Interactions with other compiler optimizations
    - Dead code removal, loop-invariant code motion, sub-expression elimination
- Small market
  - Not implemented in mass market PC processors
  - Important in smartphone, mainframe, and server sectors

# Multipath Execution — Example 1

## Predicated code

```
I0 p1,p2 = comp.eq.UT.UF r1,r2
I1 <p1>
I2 <p2>
I3 <p1>
I4 <p2>
I5 <p2>
I6 <p1>
```

Predicate comparison $I_0$ must be executed before $I_1$ – $I_6$

Either $I_1, I_3, I_6$ or $I_2, I_4, I_5$ must be cancelled

## Multipath execution

Execute all of $I_1$ – $I_6$ without cancellations

Execute predicate comparison $I_0$ in any order

Cancel $I_1, I_3, I_6$ or $I_2, I_4, I_5$

- After $I_0$ is complete
- Before retirement of $I_1$ – $I_6$

# Multipath versus Branch — Example 2

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SEQ R3, R1, R2 | F | D | X | M | W | | | | | | |
| | BNEZ R3, $I_2$ | | F | D | X | M | W | | | | | |
| $I_1$ | | | | F | D | X | M | W | R1 = R2 | | | |
| | J $I_3$ | | | F | D | X | M | W | | | | |
| $I_2$ | | | | F | F | D | X | M | W | R1 ≠ R2 | | |
| $I_3$ | | | | | F | D | X | M | W | | | |

**8 clock cycles**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| p1,p2 = comp.eq.UT.UF r1,r2 | F | D | X | M | W | | | | | |
| <p1> $I_1$ | F | D | X | M | W | Cancel if R1 ≠ R2 | | | | |
| <p2> $I_2$ | F | D | X | M | W | Cancel if R1 = R2 | | | | |
| $I_3$ | F | D | X | M | W | | | | | |

**5 clock cycles**

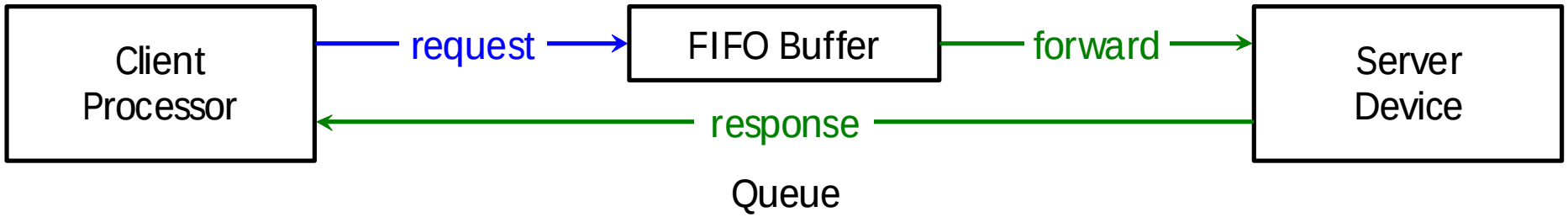
$$S \sim \frac{8}{5} = 1.6$$

# Queuing Theory:

## Memory Buffer

## Limits on Superscalar Processing

# Client/Server Model of I/O




**Fast CPU is client for slower I/O services**

Buffer stores client requests and forwards at slower server response rate

**Latency**

Time between client request and buffer server response

**Throughput**

Number of services provided per unit time

**Bandwidth**

Maximum data transfer rate of I/O channel (including buffer)

**Capacity**

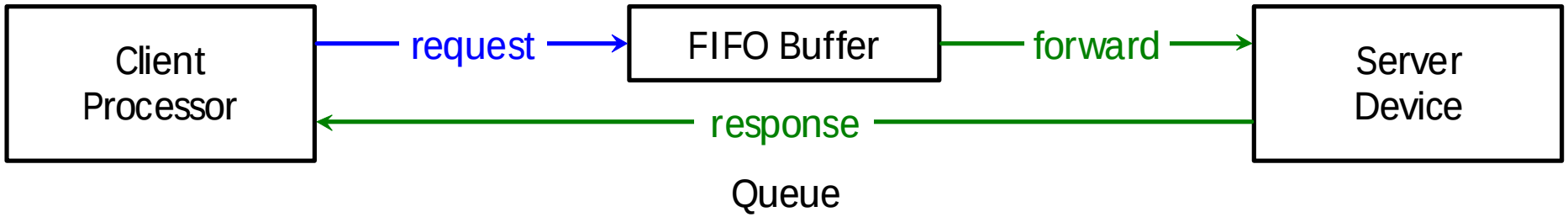
Maximum throughput of server through buffer

Depends on bandwidth and service rate (server speed)

**Utilization**

Request rate as proportion of capacity

# Buffer Operation




## Client requests

Intermittent high speed requests (**bursty**)

**Peak** client request rate >> **Average** client request rate

## Service responses

**Peak** client rate > Server rate > **Average** client rate

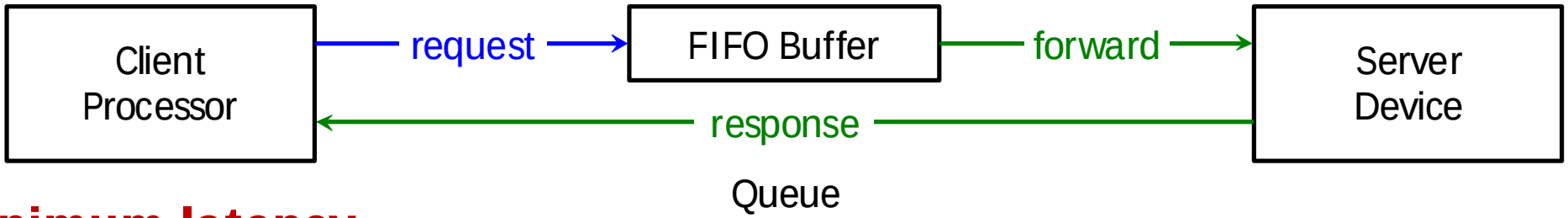
## FIFO buffers requests in order of arrival

Stores requests arriving at higher client request rate

Forwards requests to server at lower server response rate

Request forwarding rate = server response rate

# Buffer Latency and Overflow




## Minimum latency

Determined by maximum service response rate

## Buffer overflow

Buffer fill rate = Client Request Rate – Service Response Rate

Buffer fills continuously for too long ⇒ buffer overflow

## Example

Peak CPU disk read rate = 1 read/cycle = $10^9$ read requests/second

Disk can provide = $10^7$ responses per second = 100 CPU cycles/read

CPU sees minimum latency of about 100 CPU cycles

Buffer can hold 1000 requests

Continuous requests ⇒ overflow in 1000/($10^9$ – $10^7$) ~ $10^{-6}$ seconds

# Utilization — Latency Trade-Off

## Utilization

Higher client request rate

More services per second ⇒ Higher utilization

Server cannot work faster (service rate is fixed)

More requests are buffered ⇒ longer queue length (higher buffer level)

Total latency for one request = server latency + queuing time

More requests in buffer queue ⇒ longer queuing time

Higher utilization ⇒ Higher total latency

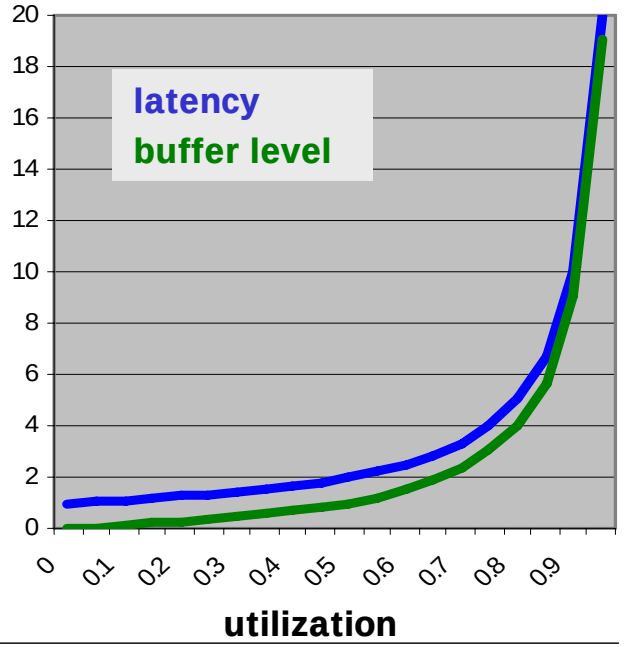
## Buffer overflow

Average request rate > average response rate

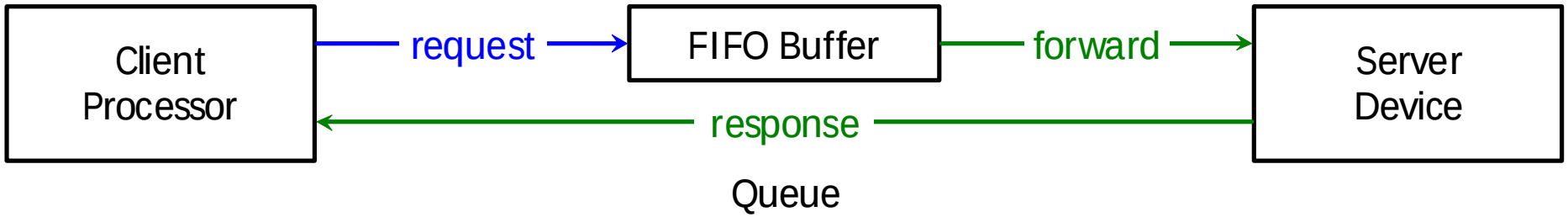
More requests enter buffer than leave

Buffer level rises

After long time, buffer overflows



# Queuing Theory




## Assumptions

- Client Requests
  - Arrive independently (Poisson statistics)
  - Have random length (bytes to transfer)
- Buffer
  - Stores requests and forwards in order of arrival (FIFO) at service rate
  - Reaches an average "steady state" level of stored requests
- Server
  - Provides services to each request independently (Poisson statistics)

## Results

Average latency = 1 / (average service rate – average arrival rate)

Average buffer level = average arrival rate × average buffer latency

(Little's Law)

# Poisson Statistics

## Random events

Occur independently

Average total number of events grows linearly with time

$$P_t(k) = \text{probability}(k \text{ events in time } t) = \frac{(\alpha t)^k}{k!} e^{-\alpha t}$$

$$p(t) = \text{probability}(\text{time interval between 2 events is } t) = \alpha e^{-\alpha t}$$

$$\langle t \rangle = \text{Average time between events} = \int_0^\infty t\left[\alpha e^{-\alpha t}\right] dt = \frac{1}{\alpha}$$

$\alpha =$ average rate of events

$$\langle k \rangle = \text{Average number of events} = \sum_{k=0}^{\infty} k\, P_t(k) = \alpha t$$

$\alpha t =$ average event rate $\times$ time interval

# Request and Response Statistics

$\lambda =$ average request rate

Probability of $k$ requests in time $t = \frac{(\lambda t)^k}{k!} e^{-\lambda t}$

Distribution of times between requests $= \lambda e^{-\lambda t}$

$\frac{1}{\mu} =$ average volume (bytes) per response

$C =$ server capacity (maximum response volume per second)

$\mu C = \frac{\text{server volume / second}}{\text{average volume / response}} =$ average response rate

Probability of $k$ responses in time $t = \frac{(\mu C t)^k}{k!} e^{-\mu C t}$

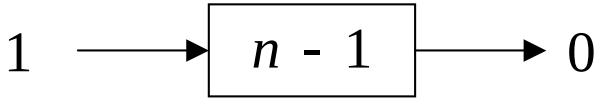
Distribution of server response times $= \mu C e^{-\mu C t}$

# Buffer State

$B(t+\Delta t)$ = requests in buffer at time $t+\Delta t$

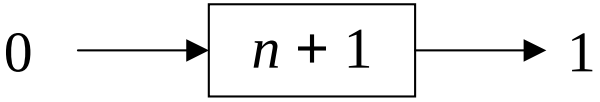
Possible cases for $B(t+\Delta t)=n$

1. $B(t)=n-1$



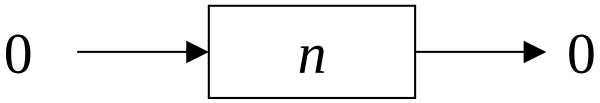

During $[t,t+\Delta t]$ $1$ new request and $0$ finished responses

2. $B(t)=n+1$



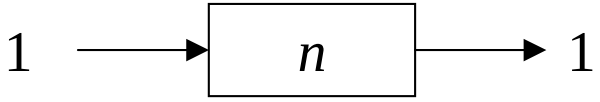

During $[t,t+\Delta t]$ $0$ new requests and $1$ finished response

3. $B(t)=n$




During $[t,t+\Delta t]$ $0$ new request and $0$ finished responses

4. $B(t)=n$




During $[t,t+\Delta t]$ $1$ new request and $1$ finished response

# Buffer State Transition

$$P_n(t+\Delta t) = \text{probability}\left(B(t+\Delta t) = n\right)$$

$$= \text{probability}\left\{\begin{array}{l} B(t) = n-1 \textbf{ AND } 1 \text{ request } \textbf{AND } 0 \text{ finish} \\ \textbf{OR } B(t) = n+1 \textbf{ AND } 0 \text{ request } \textbf{AND } 1 \text{ finish} \\ \textbf{OR } B(t) = n \textbf{ AND } 0 \text{ request } \textbf{AND } 0 \text{ finish} \\ \textbf{OR } B(t) = n \textbf{ AND } 1 \text{ request } \textbf{AND } 1 \text{ finish} \end{array}\right\}$$

$$\begin{aligned} = {} & P_{n-1}(t)\,P_{\Delta t}(1 \text{ request})\,P_{\Delta t}(0 \text{ finish}) \\ & + P_{n+1}(t)\,P_{\Delta t}(0 \text{ requests})\,P_{\Delta t}(1 \text{ finish}) \\ & + P_n(t)\,P_{\Delta t}(0 \text{ requests})\,P_{\Delta t}(0 \text{ finish}) \\ & + P_n(t)\,P_{\Delta t}(1 \text{ request})\,P_{\Delta t}(1 \text{ finish}) \end{aligned}$$

# Buffer State Transition

$$P(1 \text{ request in } \Delta t) = \frac{(\lambda \Delta t)^1}{1!} e^{-\lambda \Delta t} = \lambda \Delta t + o\left((\Delta t)^2\right)$$

$$P(0 \text{ request in } \Delta t) = \frac{(\lambda \Delta t)^0}{0!} e^{-\lambda \Delta t} = 1 - \lambda \Delta t + o\left((\Delta t)^2\right)$$

$$P(1 \text{ finish in } \Delta t) = \frac{(\mu C \Delta t)^1}{1!} e^{-\mu C \Delta t} = \mu C \Delta t + o\left((\Delta t)^2\right)$$

$$P(0 \text{ finish in } \Delta t) = \frac{(\mu C \Delta t)^0}{0!} e^{-\mu C \Delta t} = 1 - \mu C \Delta t + o\left((\Delta t)^2\right)$$

$$\begin{aligned} P_n(t+\Delta t) &= P_{n-1}(t) P(\lambda \Delta t)(1-\mu C \Delta t) + P_{n+1}(t)(1-\lambda \Delta t)(\mu C \Delta t) \\ &\quad + P_n(t)\left[(1-\lambda \Delta t)(1-\mu C \Delta t) + (\lambda \Delta t)(\mu C \Delta t)\right] + o\left((\Delta t)^2\right) \\ &= P_n(t) + \left\{(-\lambda - \mu C) P_n(t) + \lambda P_{n-1}(t) + \mu C P_{n+1}(t)\right\} \Delta t + o\left((\Delta t)^2\right) \end{aligned}$$

# Steady State Assumption

For large $t$, buffer level reaches "steady state" (almost constant)

$$B(t) \xrightarrow[t\to\infty]{} n \quad \text{and} \quad P_n(t) \xrightarrow[t\to\infty]{} P_n$$

New requests balance finished responses

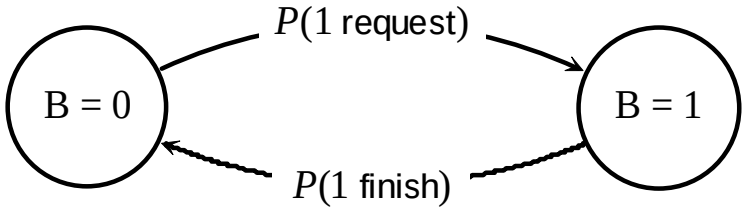
## Almost Empty Buffer

$$P\left[B(t)=0 \textbf{ AND } 1 \text{ request}\right] = P\left[B(t)=1 \textbf{ AND } 1 \text{ finish}\right]$$

$$P_0 \times P_{\Delta t}(1 \text{ request}) = P_1 \times P_{\Delta t}(1 \text{ finish})$$

$$P_0 \times \lambda \Delta t = P_1 \times \mu C \Delta t$$

$$P_1 = \frac{\lambda \Delta t}{\mu C \Delta t} P_0 = \frac{\lambda}{\mu C} P_0 = \rho P_0$$




$$\rho = \frac{\lambda}{\mu C} = \frac{\text{average request rate}}{\text{average service rate}} = \textbf{utilization}$$

# General Steady State

$$P_n(t) \to P_n \qquad \text{(constant)}$$

$$0 = \frac{d}{dt} P_n(t) = \lim_{\Delta t \to 0} \frac{P_n(t+\Delta t) - P_n(t)}{\Delta t}$$

$$= \lim_{\Delta t \to 0} \frac{\left[(-\lambda - \mu C) P_n(t) + \mu C P_{n+1}(t) + \lambda P_{n-1}(t)\right]\Delta t + o\left((\Delta t)^2\right)}{\Delta t}$$

$$= -P_n(t)(\lambda + \mu C) + \mu C P_{n+1}(t) + \lambda P_{n-1}(t)$$

$$P_{n+1} = \frac{1}{\mu C}\left[(\lambda + \mu C) P_n - \lambda P_{n-1}\right] = \left[\left(\frac{\lambda}{\mu C} + 1\right) P_n - \frac{\lambda}{\mu C} P_{n-1}\right]$$

$$= (1+\rho) P_n - \rho P_{n-1}$$

$$\rho = \frac{\lambda}{\mu C}$$

# Average Buffer Level

From recursion relation,

$$P_{n+1} = (1+\rho)P_n - \rho P_{n-1}$$

$$P_2 = (1+\rho)P_1 - \rho P_0 = (1+\rho)\rho P_0 - \rho P_0 = \rho^2 P_0$$

$$P_3 = (1+\rho)P_2 - \rho P_1 = (1+\rho)\rho^2 P_0 - \rho P_0 = \rho^3 P_0$$

$$\vdots$$

$$P_n = \rho^n P_0$$

sum of probabilities $= 1 = \sum_{n=0}^{\infty} P_n = \sum_{n=0}^{\infty} \rho^n P_0 = P_0 \frac{1}{1-\rho} \Rightarrow \quad P_0 = 1-\rho = 1-\frac{\lambda}{\mu C}$

**Probability of Buffer Level** $n$ $\qquad P_n = \left(\frac{\lambda}{\mu C}\right)^n \left(1-\frac{\lambda}{\mu C}\right) = \rho^n (1-\rho)$

**Average Buffer Level** $\qquad \langle n \rangle = \sum_{n=0}^{\infty} n\rho^n (1-\rho) = \frac{\rho}{1-\rho} = \frac{\lambda/\mu C}{1-\lambda/\mu C} = \frac{\lambda}{\mu C - \lambda}$

# Total Latency and Buffer Level

$$\langle T \rangle = \langle \text{service time for one request} \rangle + \langle \text{buffer latency} \rangle$$

$$= \langle \text{service time for one request} \rangle + \langle \text{time to service FIFO queue} \rangle$$

$$= \frac{1}{\mu C} + \langle n \rangle \frac{1}{\mu C} = \frac{1}{\mu C} + \left[ \frac{\lambda / \mu C}{1 - \lambda / \mu C} \right] \frac{1}{\mu C}$$

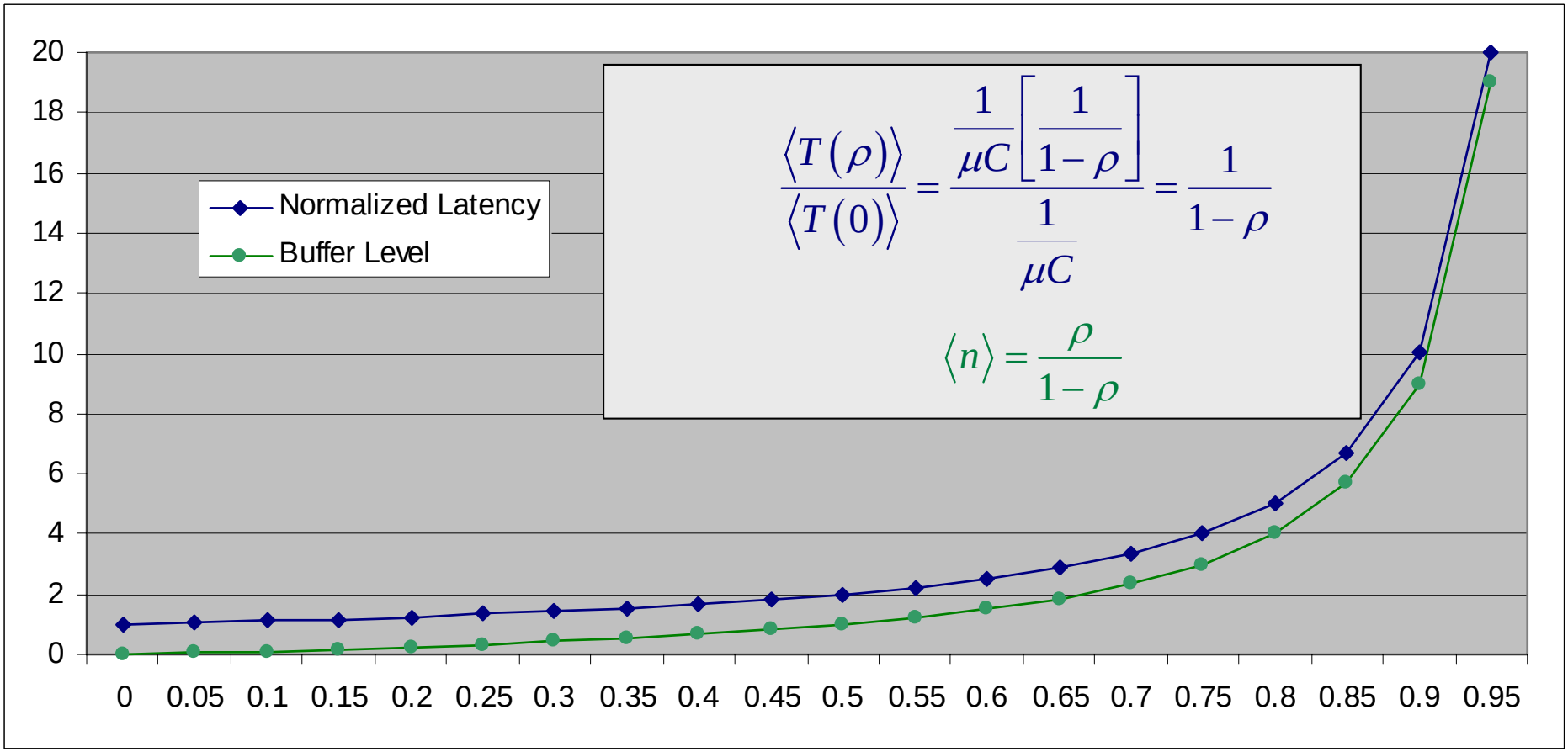
$$= \frac{1}{\mu C} \left[ 1 + \frac{\lambda / \mu C}{1 - \lambda / \mu C} \right] = \frac{1}{\mu C} \left[ \frac{1}{1 - \lambda / \mu C} \right] = \frac{1}{\mu C} \left[ \frac{1}{1 - \rho} \right]$$

$$= \frac{1}{\mu C - \lambda} = \frac{1}{\text{average service rate} - \text{average request rate}}$$

## Little's Law

$$\langle n \rangle = \frac{\rho}{1 - \rho} = \frac{\lambda}{\mu C} \left[ \frac{1}{1 - \rho} \right] = \lambda \cdot \langle T \rangle = \left( \text{average request rate} \right) \times \left( \text{average latency} \right)$$

# Latency and Buffer Level versus Utilization

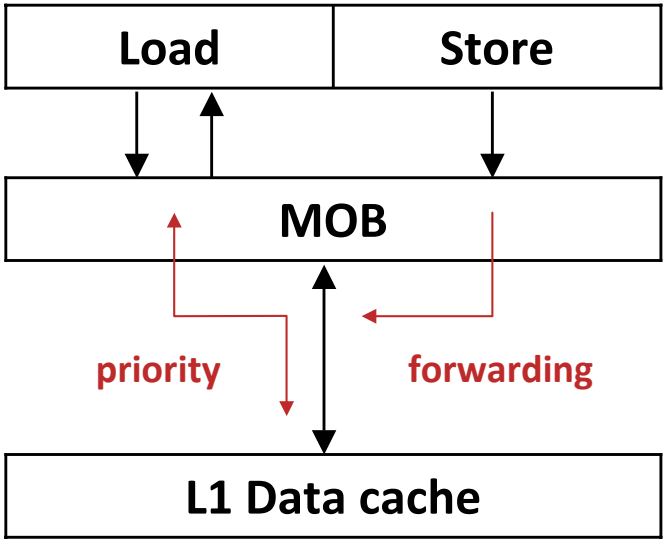



$$\text{Normalized Latency} = \frac{\text{Latency (seconds)}}{\text{Latency with zero utilization}}$$

# Memory Queuing

## Memory Ordering Buffer (MOB)

- Between Load/Store EU and data cache
  - Buffers read/write operations
  - Enables "simultaneous" reads and writes
- Issues loads before stores
  - Enhances throughput of speculative reads
- Forwards queued stores to read requests

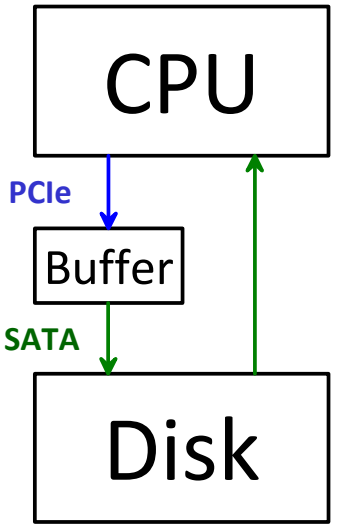



## Hard drive

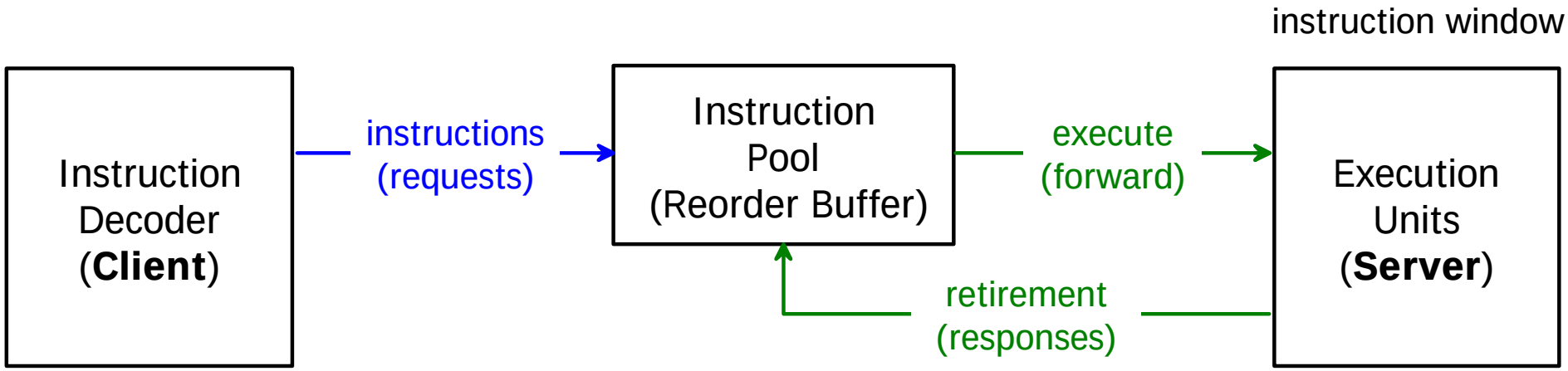
- SATA 3.0 sustained throughput = 6 Gbps < 600 MB/s
- PCI Express 4.0 sustained throughput < 16 GB/sec
- Typical disk write buffer = 64 MB
- Overflow time = 64 MB / [(16 – 0.6) GB/s] = 4.16 ms

## Safe removal of external drive

- Dump buffer to file
- Mark file closed in file system
- Unsafe removal ⇒ possible data loss



# Client/Server Model of CPU




## Client

Decoder sends $\lambda$ instructions per cycle to ROB

## Server

EUs can operate on $\mu C$ instructions (one instruction window) per cycle

## Goal

Maximize utilization of EUs $\rho = \frac{\lambda}{\mu C}$ without ROB overflow

For fixed instruction window $\mu C \Rightarrow$ maximize $\lambda$

No overflow $\Rightarrow$ average decoder rate < instruction window

# General Superscalar Model

## Execution units (EUs) operate in parallel

EU stages ≥ 1

## Ideal case

Every stage of every EU working on every clock cycle

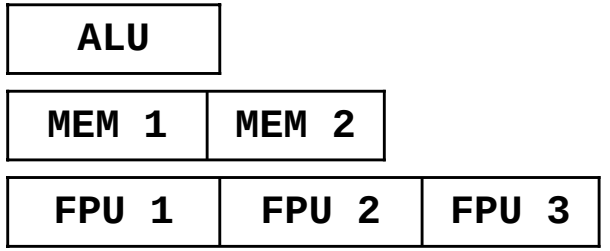
Multiple instructions pipelined through EU stages

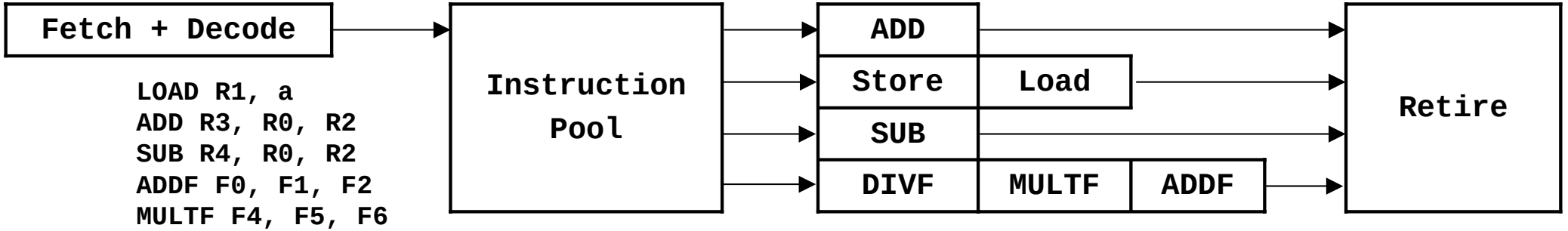
## Example

2 ALUs — 1 cycle per instruction

1 Load + Store — 2 cycles per instruction

1 FPU — 3 cycles per instruction




**7 instructions in various stages of execution**

# Detailed Analysis of ILP

## Pipeline structure

$u_i =$ execution units (EU) of type $i$

$u = \sum_i u_i =$ total execution units (EU) in CPU

$s_i =$ pipeline stages in EU of type $i$

$u_i \times s_i =$ pipeline stages of type $i$

$IC_{\text{EU}} = \sum_i u_i \times s_i =$ total pipeline stages in CPU

$=$ instructions executing in all EUs

$=$ size of instruction window

$=$ instructions executing in parallel (ILP)

$$s = \sum_i \frac{u_i}{u} \times s_i = \frac{\sum_i u_i \times s_i}{u} = \frac{IC_{\text{EU}}}{u} = \text{average pipeline stages in EUs}$$

instruction window $= IC_{\text{EU}} = u \times s$

# Queuing Theory and Decoder Rate

## Instructions Per Clock Cycle

$$IPC_i = \frac{1}{CPI_i} = u_i \times IPC_i^{scalar} = \frac{u_i}{CPI_i^{scalar}} = \frac{u_i}{1 + CPI_i^{stall}}$$

$$IPC = \sum_i IPC_i = \sum_i \frac{u_i}{1 + CPI_i^{stall}} = \mu C = \text{average response rate}$$

$$\text{Average buffer (ROB) level} = \langle n \rangle = \frac{\lambda}{\mu C - \lambda} = \frac{\lambda}{IPC - \lambda}$$

## Maximize utilization $\Rightarrow$ ROB level = instruction window

$$\langle n \rangle = \frac{\lambda}{IPC - \lambda} = IC_{EU} \Rightarrow \lambda = \frac{IC_{EU} \times IPC}{1 + IC_{EU}} = \frac{s \times u \times IPC}{1 + s \times u}$$

$$CPI^{stall} = 0 \Rightarrow IPC^{ideal} = \sum_i u_i = u$$

$$\lambda^{ideal} = \frac{s \times u^2}{1 + s \times u} = \text{ instructions attempting to execute per cycle}$$

# Example of Instruction Window

## Execution units

2 integer ALUs — 1 CC per execution

2 FPUs — 3 CC per execution

2 load/store units — 2 CC per execution

## Instruction window

$$u = \sum_i u_i = 6 = IPC^{ideal}$$

$$s = \sum_i \frac{u_i}{u} \times s_i = \frac{2}{6} \times 1 + \frac{2}{6} \times 3 + \frac{2}{6} \times 2 = \frac{12}{6} = 2$$

$IC_{EU} = u \times s = 12$ instructions executing in parallel

## Decoder rate (ideal)

Maximize $\rho = \frac{\lambda}{\mu C} \Rightarrow \lambda = \lambda^{ideal}$ without overflow (ROB level $= IC_{EU}$)

$$IPC^{ideal} = \mu C = 6 > \lambda = \lambda^{ideal} = \frac{s \times u^2}{1 + s \times u} = \frac{12 \times 6}{1 + 12} = 5.54$$

# Implications for ILP Scalability

## Scaling instruction window and decoder rate

execution units $u_i \rightarrow u_i' = \alpha_u u_i$

pipeline stages $s_i \rightarrow s_i' = \beta_s s_i$

instruction window $IC_{EU} \rightarrow IC_{EU}' = \alpha_u \beta_s IC_{EU}$

$$\lambda^{ideal} \rightarrow \lambda^{ideal\,\prime} = \frac{\left(\beta_s \alpha_u^2\right) s \times u^2}{1 + \left(\beta_s \alpha_u\right) s \times u}$$

## Scaling $6 \rightarrow 15$ EUs with $2 \rightarrow 8$ superpipelined stages

$$\alpha_u = \frac{15}{6} \quad \beta_s = \frac{8}{2} \quad \Rightarrow \quad \alpha_u \times \beta_s = 10$$

$IC_{EU} = 120$ instructions executing in parallel

$15 > \lambda^{ideal} \geq 14.9$ instructions decoded per CC

## Difficulties

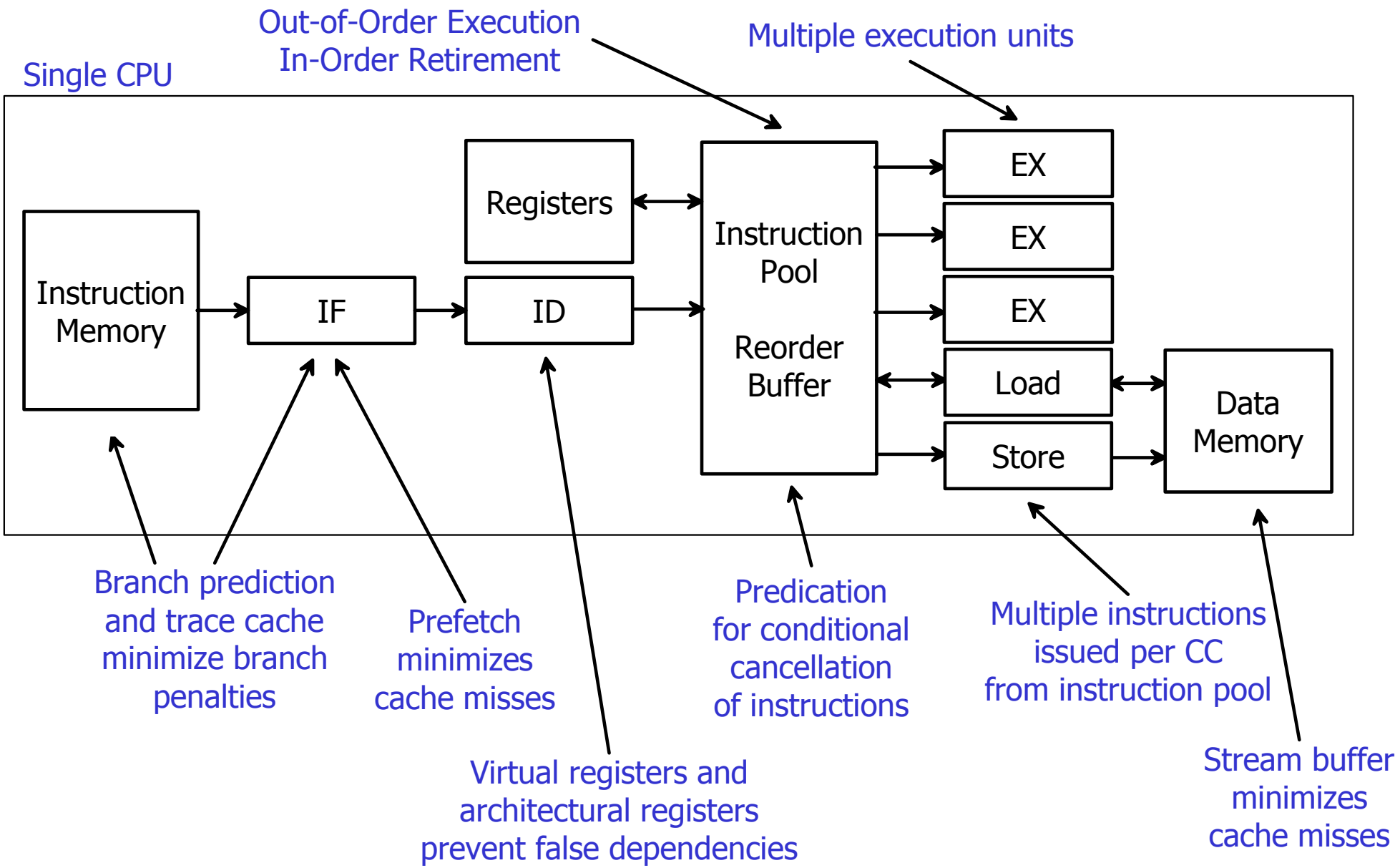
Decode 15 instructions per CC with cache misses, mispredictions, ...

Maintain window of 120 independent instructions

25 – 30 branch instructions in window $\Rightarrow$ large misprediction probability

# Beyond Instruction Level Parallelism

# Summary of Superscalar Processing



# ILP Scalability Limit

## Scaling instruction window and decoder rate

execution units $u_i \rightarrow u_i' = \alpha_u u_i$

pipeline stages $s_i \rightarrow s_i' = \beta_s s_i$

instruction window $IC_{EU} \rightarrow IC_{EU}' = \alpha_u \beta_s IC_{EU}$

$$\lambda^{ideal} \rightarrow \lambda^{ideal\,\prime} = \frac{\left(\beta_s \alpha_u^2\right) s \times u^2}{1 + \left(\beta_s \alpha_u\right) s \times u}$$

## Scaling 6 → 15 EUs with 2 → 8 superpipelined stages

$$\alpha_u = \frac{15}{6} \quad \beta_s = \frac{8}{2} \quad \Rightarrow \quad \alpha_u \times \beta_s = 10$$

$IC_{EU} = 120$ instructions executing in parallel

$15 > \lambda^{ideal} \geq 14.9$ instructions decoded per CC

## Difficulties

Decode 15 instructions per CC

- Despite cache misses, mispredictions, …

Maintain window of 120 independent instructions

- Branches ≈ 20% of instructions
- 25 – 30 branches in window ⇒ large misprediction probability

## Require larger source of independent instructions

Exploit inherent parallelism in software operations

# Sequential and Parallel Operations

## Programs combine parallel + sequential constructs

- High-level job → model-dependent sections
  - Processes
  - Threads
  - Classes
  - Procedures
  - Control blocks
- Sections compiled → ISA = low level CPU operations
  - Data transfers
  - Arithmetic/logic operations
  - Control operations

## High-level job → execution

- Machine instructions — small sequential operations
  - Local information on 2 or 3 operands
- CPU cannot recognize abstract model-dependent structures
- Information about inherent parallelism lost in translation to CPU

# Parallelism in Sequential Jobs

## Concurrency in high-level job

Two or more independent activities in defined to execute at same time

- Parallel — execute simultaneously on multiple copies of hardware
- Interleave — single hardware unit alternates between activities

Example

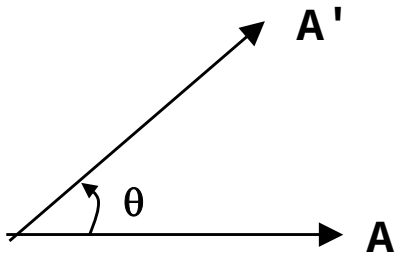
- Respond to mouse events
- Respond to keyboard input
- Accept network message

## Functional concurrency

Procedure maps $\mathbf{A' = R(\theta) \times A}$

Code performs sequential operations

$$A_x' = A_x \cos\theta + A_y \sin\theta$$
$$A_y' = -A_x \sin\theta + A_y \cos\theta$$

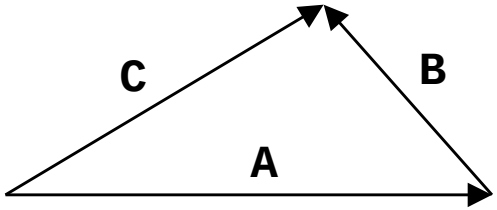



## Data concurrency

Procedure maps `C = A + B`

Code performs sequential operations

```
for (i = 0, i < n, i++) C[i] = A[i] + B[i]
```



# Extracting Concurrency in Sequential Programming

## Programmer

- Codes in high level language
- Code reflects abstract programming models
  - Procedural, object oriented, frameworks, structures, system calls, ...

## Compiler

- Converts high level code to sequential list
  - Localized CPU instructions and operands
- Information about inherent parallelism lost in translation

## Hardware applies heuristics

- Partially recover concurrency as ILP

| Technique | Concurrency Identified / Reconstructed |
|---|---|
| Pipelining | Parallelism in single instruction execution |
| Dynamic scheduling superscalar | Operation independence |
| Branch and trace prediction | Control blocks |
| Predication | Decision trees |

# Extracting Parallelism in Parallel Programming

## Programmer

- Identifies inherently parallel operations in high level job
  - Functional concurrency
  - Data concurrency
- Translates parallel algorithm into source code
- Specifies parallel operations to compiler
  - Parallel threads for functional decomposition
  - Parallel threads for data decomposition

## Hardware

- Receives deterministic instructions reflecting inherent parallelism
  - Code + threading instructions
- Disperses instructions to multiple processors or execution units
  - Vectorized operations
  - Pre-grouped independent operations
  - Thread Level Parallelism

# The "Old" Parallel Processing

**1958 — research at IBM on parallelism in arithmetic operations**

**1960 – 1980**

- Mainframe SMP machines with N = 4 to 24 CPUs
- OS dispatches process from shared ready queue to idle processor

**1980 – 1995**

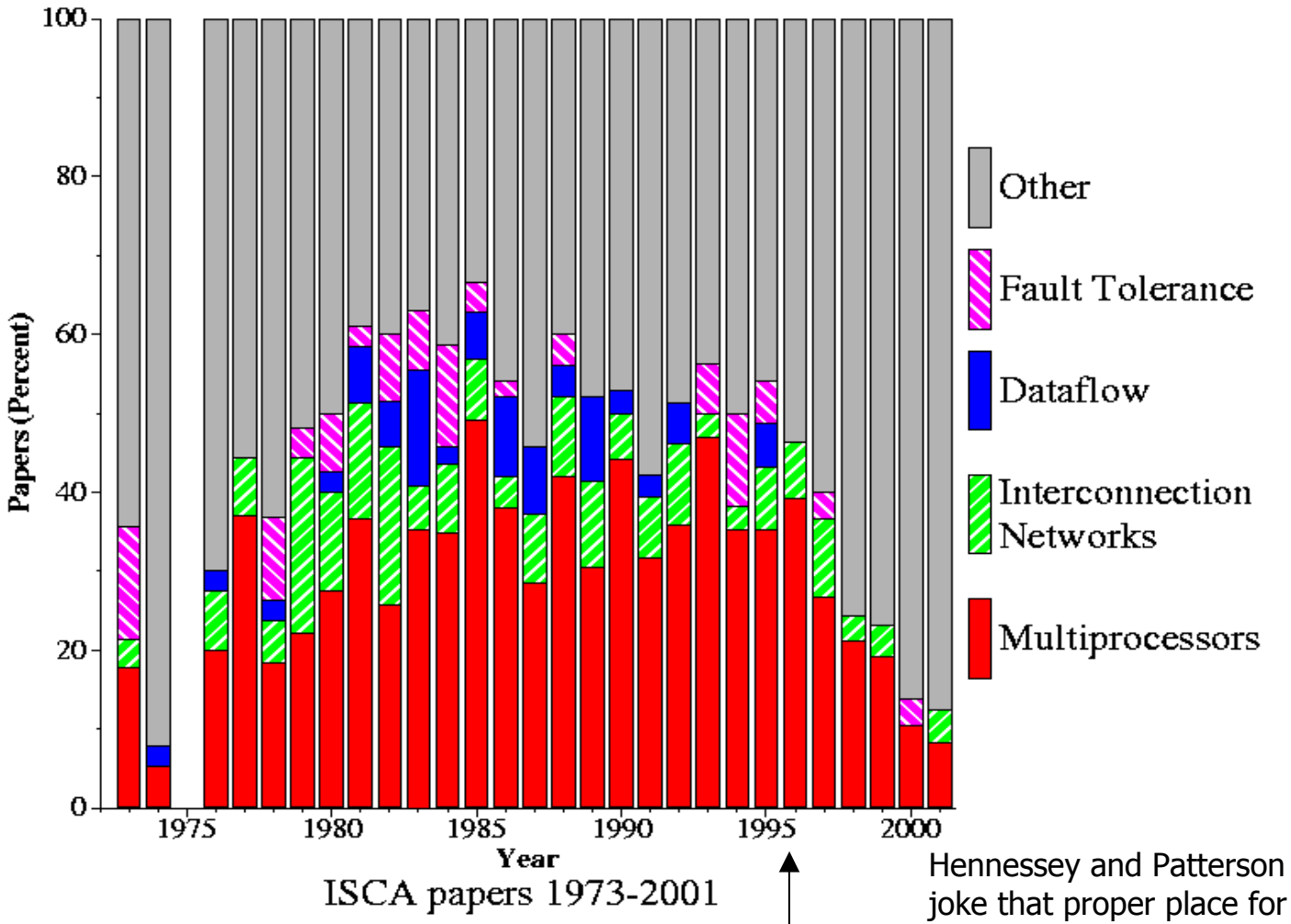
- Research boom
- Automated parallelization by compiler
  - Limited success — compilers cannot identify inherent parallelism
- Parallel constructs in high level languages
  - Long learning curve — parallel programmers are typically specialists
- Inherent complexities
  - Processing and communication overhead
    - Inter-process message passing — spawning/assembling with many CPUs
    - Synchronization to prevent race conditions (data hazards)
  - Data structures
    - Shared memory model
    - Good blocking to cache organization

**1999 — fashionable to consider parallel processing a dead end**

# 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

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

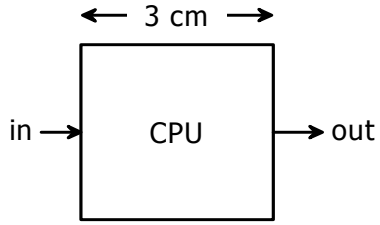
ISCA — International Symposium on Computer Architecture

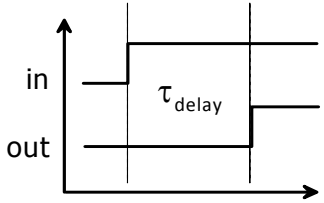
Ref: Mark D. Hill and Ravi Rajwar, "The Rise and Fall of Multiprocessor Papers in the International Symposium on Computer Architecture (ISCA)", http://pages.cs.wisc.edu/~markhill/mp2001.html

# It's Back — the "New" Parallel Processing

## Crisis rebranded as opportunity

Processor clock speed near physical limit (speed of light = $3 \times 10^{10}$ cm/s)




$$\tau_{delay} > \frac{3 \text{ cm}}{3 \times 10^{10} \text{ cm/sec}} = 10^{-10} \text{ sec}$$

$$\tau_{clock} \approx 10^{-10} \text{ sec} \Rightarrow R_{max} < 10^{10} \text{ Hz} \approx 10 \text{ GHz}$$

Heating

- Clock rate ↑ ⇒ heat output ↑
- CPU power ↑ ⇒ chip size ↑ ⇒ heat transfer rate ↓ ⇒ CPU overheats

Superscalar ILP cannot rise significantly

- Instruction window ~ 100 independent instructions

## "Old" parallel processing is not sufficient

Some interesting possibilities

- Multicore processors cheaper and easier to manufacture
- User level thread management
- Multithreaded OS kernels and OS level thread scheduling
- Compiler support for thread management APIs
- New debugging tools

# Processes and Threads

## Process

One instance of an independently executable program

Basic unit of OS kernel scheduling (on traditional kernel)

Entry in process control block (PCB) defines resources

- ID, state, PC, register values, stack+memory space, I/O descriptors, …

Process context switch → high volume transfer operation

Organized into one or more owned threads

## Thread

One instance of independently executable instruction sequence

Not organized into smaller multitasked units

Limited private resources — PC, stack, and register values

- Other resources shared with other threads owned by process

Scheduled by kernel or threaded user code

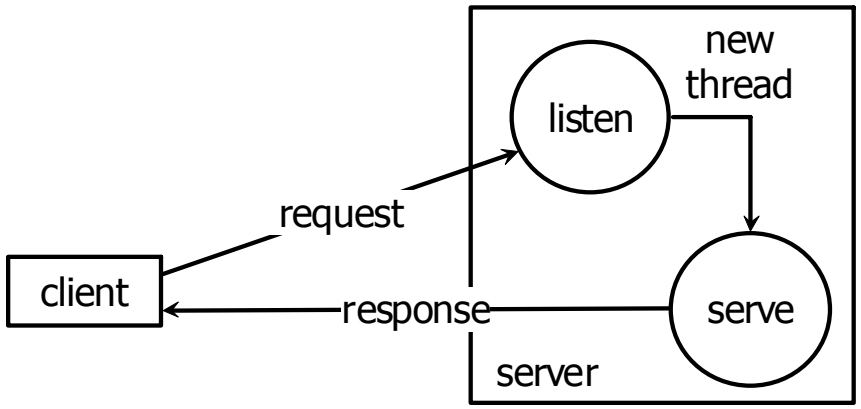
Thread switch → low volume transfer operation

# Multithreaded Software

## Threaded OS kernel

- Process = one or more threads
- Multithreaded application
  - Organized as more than one thread
  - Threads scheduled by OS or application code
  - Not specific to parallel algorithms

## Classic multithreading example




- Multithreaded web server
  - Serves multiple clients
  - Creates thread per client
- Server process creates **`listen`** thread
  - **`Listen`** thread blocks — waits for service **`request`**
  - Service **`request`** → **`listen`** thread creates new **`serve`** thread
- **`Serve`** thread handles web service **`request`**
- **`Listen`** thread returns to blocking

# Decomposing Work

## Decomposition

Break down program into basic activities

Identify dependencies between activities

"Chunking" — choose size parameters for coded activities

## Functional Decomposition

Each thread assigned different activity

Example — 3D game

- Thread 1 updates ground
- Thread 2 updates sky
- Thread 3 updates character

## Data Decomposition

Each thread runs same code on separate block of data

Example — 3D game

- Divide sky into n sections
- Threads 1 — n update section of sky

# Hardware Implementation of Multithreading

## No special hardware requirements

Multithreaded code runs on single / multiple CPU system

Run-time efficiency depends on hardware/software interaction

## Coarse-grained multithreading

Single CPU swaps among threads on long stall

## Fine-grained multithreading

Single CPU swaps among threads on each clock cycle

## Simultaneous multithreading (SMT)

Superscalar CPU pools instructions from multiple threads

Enlarges instruction window

## Hyper-Threading

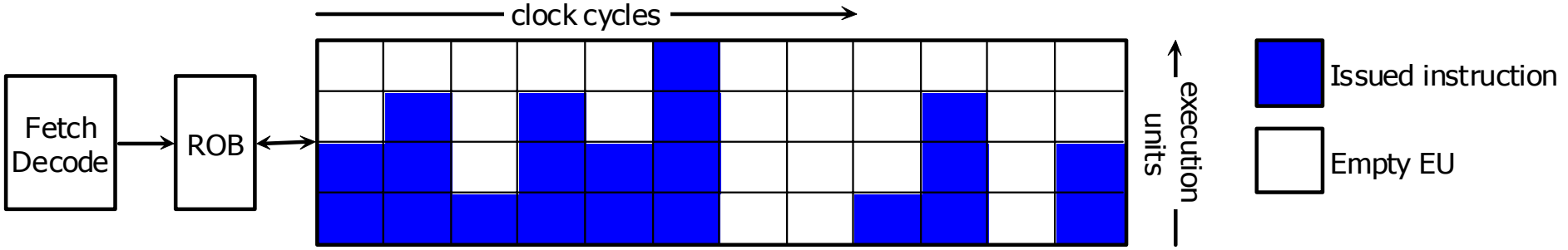
Intel technology combining fine-grained multithreading and SMT

## Multiprocessing

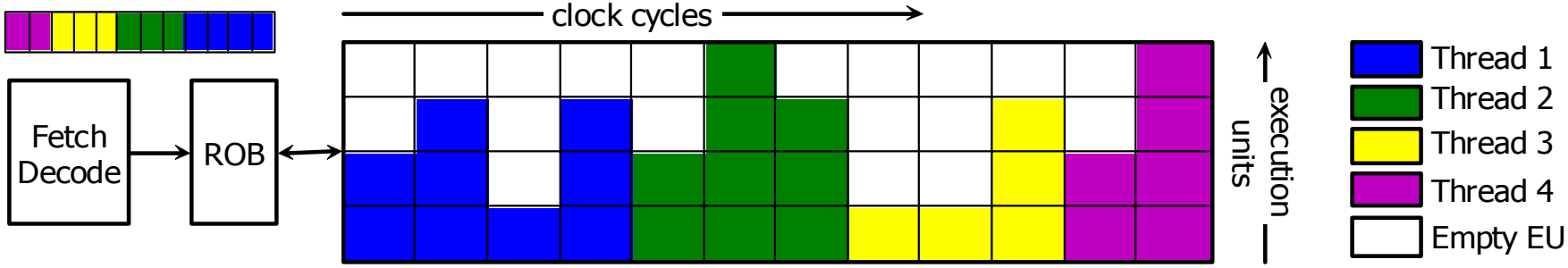
Dispatches threads to CPUs

# Superscalar CPU Multithreading

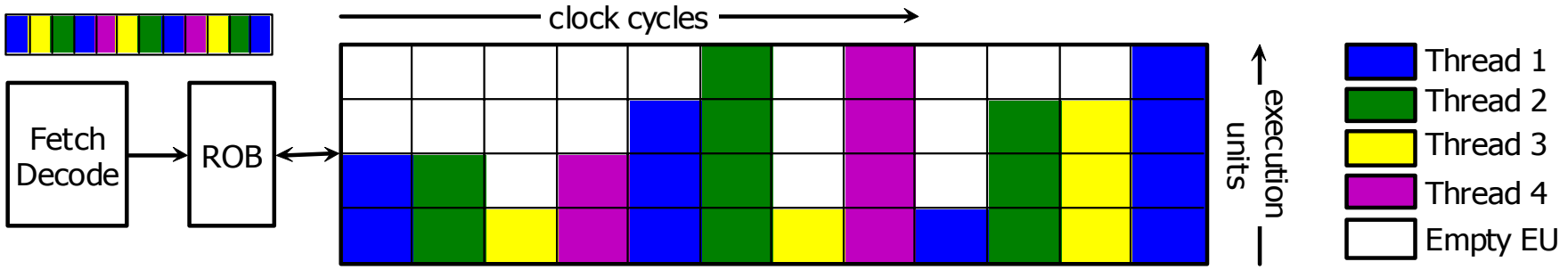
## Single thread on superscalar



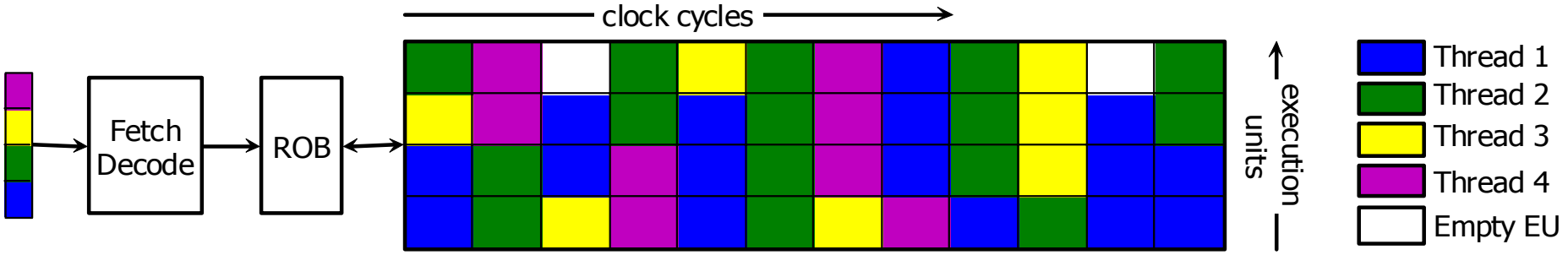

## Course grained multithreading on superscalar




## Fine grained multithreading on superscalar



# Simultaneous Multithreading



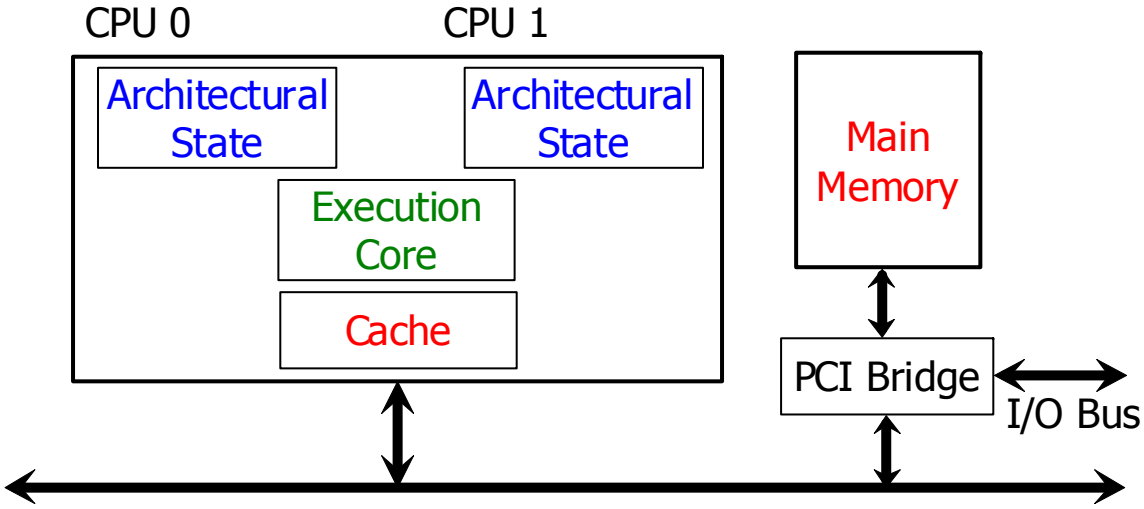

## Simultaneous multithreading on superscalar

- Pool instructions from multiple threads
- Instructions labeled in reorder buffer (ROB)
  - PC
  - Thread number
  - Operands
  - Status
- Large instruction window

## Advantage on mispredictions

- Only thread with misprediction is cancelled
  - Other threads continue to execute
- Cancellation rate from mispredictions → ¼ single-thread cancellation rate

# 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**

- Fine grained N = 2 multithreading
- Interleaves threads on In-Order fetch/decode/retire units

**Issue instructions to shared Out-of-Order execution core**

- Simultaneous N = 2 multithreading (SMT)
- Executes instructions from shared instruction pool (ROB)

**Stall in one thread ⇒ other thread continues**

- Both CPUs keep working on most clock cycles
- Advantage of course-grained N = 2 multithreading

# Thread Coexistence

## Multiprocessor code

- Provides source of independent instructions
- Permits high processor utilization

## Independent applications running in parallel

- Unrelated instructions with no data dependencies
- Independence can create resource conflicts
  - Require different data blocks in cache
  - Use different branch prediction cache and trace cache

## Parallel threads of single application

- Different pieces of same program
- Run in coordinated fashion
  - Communicate, synchronize, exchange data
- Stall in thread can stall related threads
  - Cache miss, page fault, branch misprediction, ...

# Helper Thread Model

## Performs no committed work

- Do not change any program result
- Results not committed to memory
- Require no additional hardware support

## Performs loads and branches that appear in work thread

- Encounter cache misses before work thread
- Prepares caches
- Prevents costly misses

# Helper Thread Example

## Example

```
L:   MUL R4, R6, R8
     ADD R4, R6, R9
     ADD R1, R2, R3
     SUB R3, R4, R5
     LW R6, 0(R1)       ; cache miss
     ADD R6, R3, R2
     BEZQ R6, L         ; misprediction
```

| Work Thread | Helper Thread |
|---|---|
| `L: MUL R4, R6, R8` | `L: ADD R1, R2, R3` |
| `ADD R4, R6, R9` | `LW R6, 0(R1)` ; cache miss |
| `ADD R1, R2, R3` | ; cache update |
| `SUB R3, R4, R5` | `BEQZ R6, L` ; misprediction |
| `LW R6, 0(R1)` ; no cache miss | ; update predictor |
| `ADD R6, R3, R2` | |
| `BEZQ R6, L` ; no misprediction | |

# Flynn Taxonomy for CPU Architectures

| | Instruction | |
|---|---|---|
| **Data** | Single Instruction Single Data<br>**SISD** | Multiple Instruction Single Data<br>**MISD** |
| | Single Instruction Multiple Data<br>**SIMD** | Multiple Instruction Multiple Data<br>**MIMD** |

## SISD

Standard single CPU machine with single or multiple pipelines

## SIMD

Vector processor or processor array

Performs one operation on data set on each CC

## MISD

Perform multiple operations on one data set each CC

Few products — IBM Watson IA applies multiple algorithms to same data

## MIMD

Multiprocessor or cluster computer

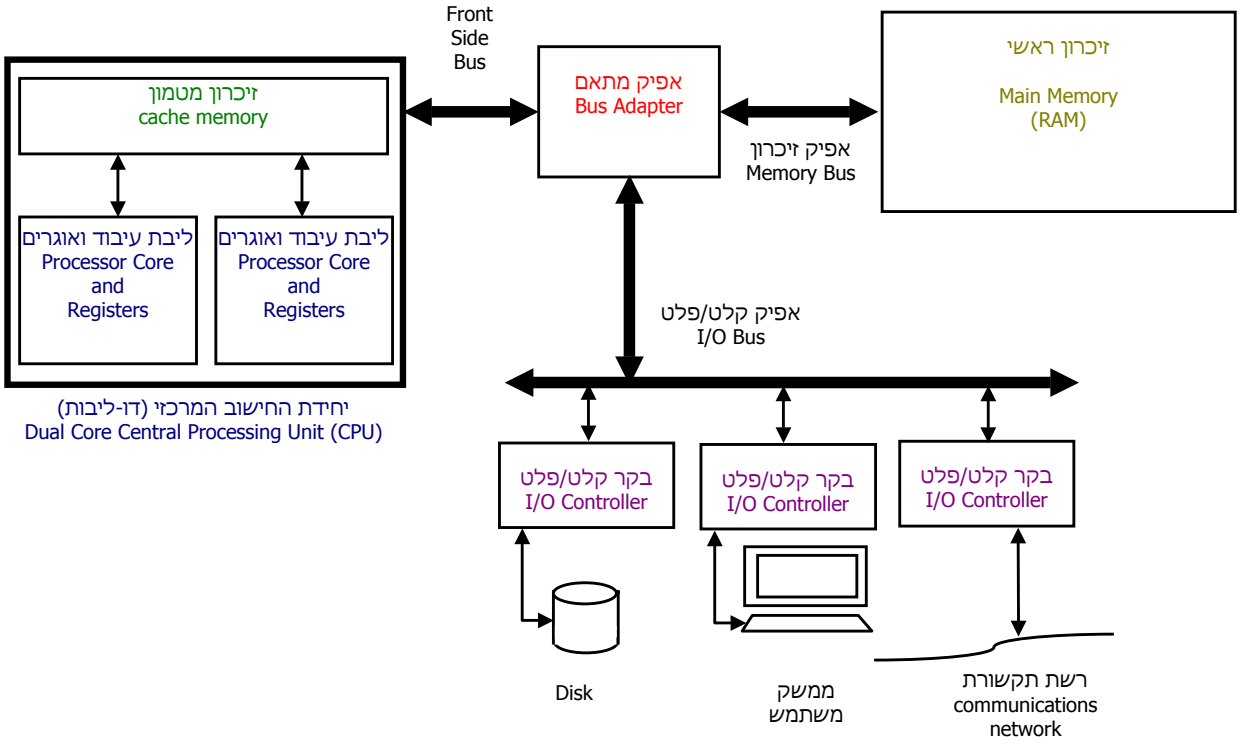
Perform multiple operations on multiple data sets on each CC

Ref: M.J. Flynn, "Very High-Speed Computers", Proceedings of the IEEE, Dec. 1966.

# Multiprocessor Architecture

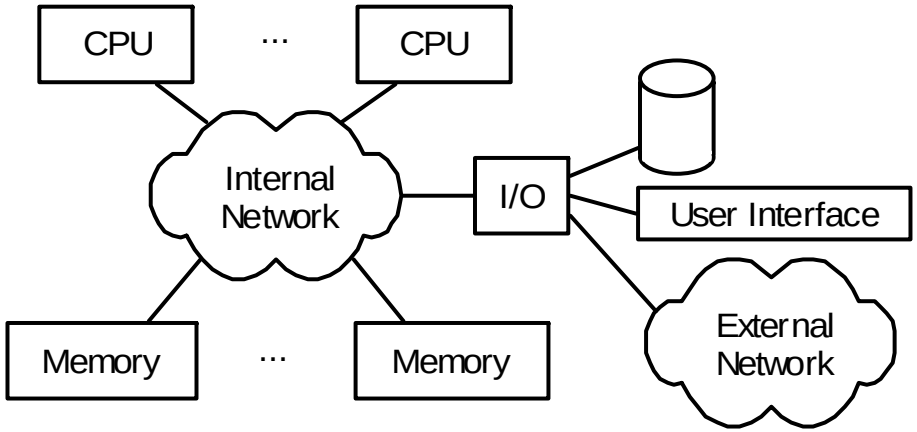
## SISD/SIMD workstation

- Dual core CPU
  - Architectural registers
  - Cache
  - Execution units
- I/O system
  - Long-term storage
  - Peripheral devices
  - System support functions
- Main memory
- Internal network system




## MIMD multiprocessor

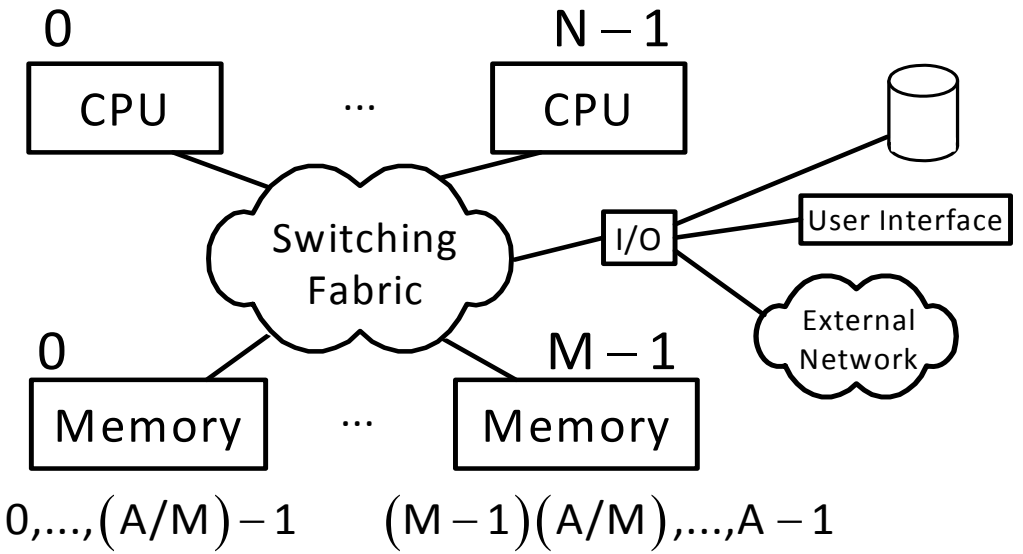
- Multiple CPUs
- I/O system
- Main memory
  - Unified or partitioned
- Internal network
  - From simple bus to complex mesh



# Network Topology → Parallelization Model

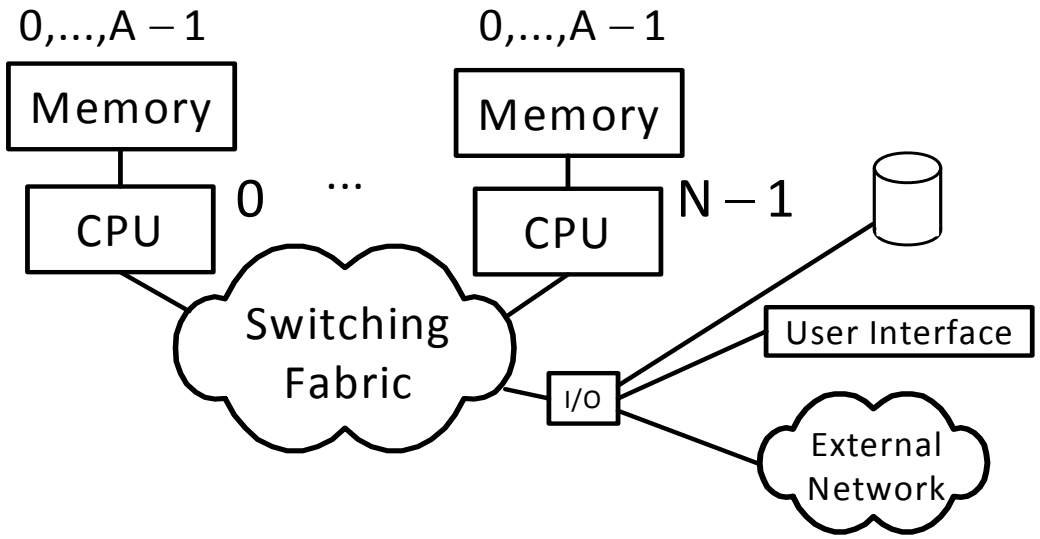
## Shared Memory System

- Global memory space *A* physically partitioned into *M* blocks
- *N* processors access full memory space via internal network
- Processors communicate by write/read to shared addresses
- Synchronize memory accesses to prevent data hazards




## Message Passing System

- *N* nodes — processors with private address space *A*
- Processors communicate by passing messages over internal network
- Messages combine data and memory synchronization



# Flynn-Johnson Taxonomy

<table>
<tr><td></td><td>Single Instruction</td><td colspan="3">Multiple Instruction</td></tr>
<tr><td>Single Data</td><td>SISD</td><td colspan="3">MISD</td></tr>
<tr><td rowspan="3">Multiple Data</td><td rowspan="3">SIMD</td><td>GMSM</td><td>GMMP</td><td>Global Memory</td></tr>
<tr><td colspan="2">MIMD</td><td></td></tr>
<tr><td>DMSM</td><td>DMMP</td><td>Distributed Memory</td></tr>
<tr><td></td><td>Shared Memory</td><td>Message Passing</td><td></td></tr>
</table>

Ref: E. E. Johnson, "Completing an MIMD Multiprocessor Taxonomy", Computer Architecture News, June 1988.

# Shared Memory versus Message Passing

| | Shared Memory | Message Passing |
|---|---|---|
| **Interprocess communication** | Multiple CPUs access shared addresses in common address space | Multiple CPUs exchange messages |
| **Communication overhead** | Cache / RAM updates<br>Cache coherency | Message formulation<br>Message distribution<br>Network overhead |
| **Scalability** | Limited by complexity of CPU access to shared memory | Independent of number of CPUs<br>Limited by network capacity |
| **Applicability** | Fine grain parallelism<br>Light parallel threads<br>Short code length<br>Small data volume | Course grain parallelism<br>Heavy parallel threads<br>Long code length<br>Large data volume |
| **API** | OpenMP | Message Passing Interface (MPI) |

# Amdahl's Law for Multiprocessors

## Parallelization

Divide work among $N$ processors

$F_P$ = fraction of program that can be parallelized $= \frac{IC_P}{IC} \Rightarrow IC_P = F_P \times IC$

For parallel work $CPI \rightarrow CPI_{\text{parallel}} = CPI / N$

$$S = \frac{CPI \times IC \times \tau}{CPI' \times IC' \times \tau'} = \frac{CPI \times IC}{\left[ CPI \times \left( IC - IC_P \right) + \frac{CPI}{N} \times IC_P \right]}$$

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

With contemporary technology, for most applications, $F_P \approx 80\%$

$$S = \frac{1}{\left(1 - 0.8\right) + \frac{0.8}{N}} \xrightarrow[N \to \infty]{} 5 \qquad CPI^{ideal} = \left(1 - 0.8\right) \times 1 + 0.8 \times \frac{1}{N} \xrightarrow[N \to \infty]{} = 0.2$$

# MP and HT Performance Enhancements

## MP Without Hyper Threading

| CPUs | S | S/CPU |
|---|---|---|
| 2 | 1.7 | 0.85 |
| 4 | 2.6 | 0.65 |

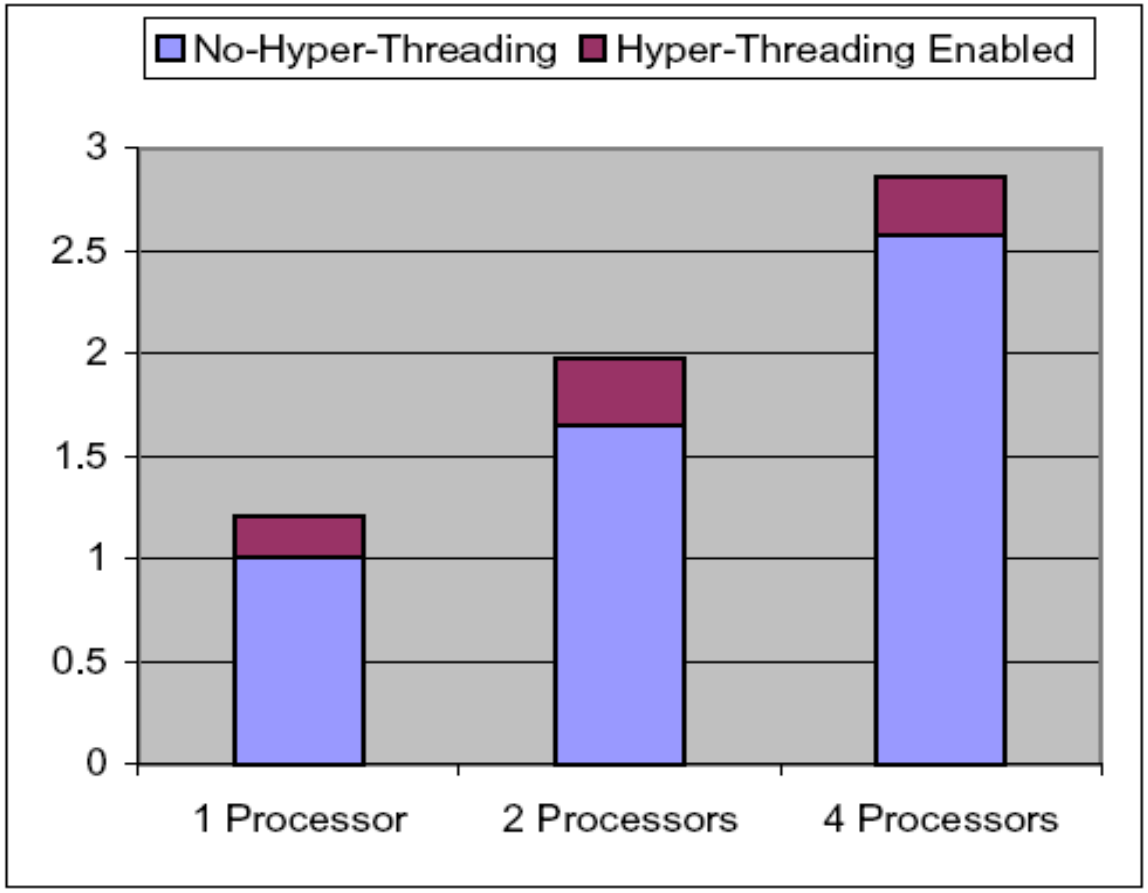
$$1.7 = \frac{1}{(1 - F_P) + \frac{F_P}{2}}$$

$$2.6 = \frac{1}{(1 - F_P) + \frac{F_P}{4}}$$

$$F_P \approx 0.8$$

## Hyper Threading Without MP

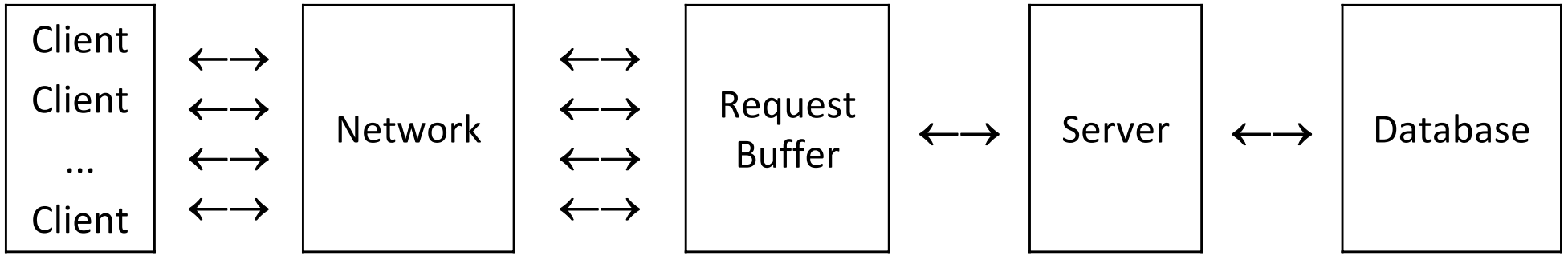
| CPUs | S | S/CPU |
|---|---|---|
| 1 | 1.2 | 0.60 |




Speed-up for On Line Transaction Processing (OLTP)

# On Line Transaction Processing (OLTP)

## Model




## Transactions

Client requests to server + database

- Banking, order processing, inventory management, student info system

Independent work — inherently multithreaded

- 1 thread per request
- Server sees large batch of small parallel threads

Short sequential code

- SQL transactions — short accesses to multiple tables

Complex (DB) access ⇒ memory latency ⇒ CPU stalls per thread

- $CPI^{OLTP}$ = 1.27 on 8-pipeline dynamic scheduling superscalar
- $CPI^{SPEC}$ = 0.31 on same hardware

# Memory Access Complexities in OLTP

## SQL thread

- Access multiple tables
- Example
  - Order processing ⇒ customer account, inventory, shipping, ...
- Tables in separate areas of memory
  - Cache conflicts
- Generates multiple memory latencies per thread

## Multiple threads

- Threads access same tables
- Requires atomic SQL transaction
- Requires thread synchronization
- Synchronization ⇒ locks on parallel threads ⇒ memory latencies

## SMT advantage

- Process many threads to hide memory latency

# Multiprocessor Efficiency

## Ideal speedup

$$S_{F_P=1} = \left.\frac{1}{(1-F_P)+\frac{F_P}{N}}\right|_{F_P=1} = N$$

## Efficiency

Actual speedup relative to ideal (linear) speedup

Speedup per processor

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

## Efficiency of large system

$$E \xrightarrow[N\to\infty]{} 0$$

# Grosch's Law versus Amdahl's Law

## Computers enjoy economies of scale

Claim formulated by Herbert R. J. Grosch at IBM in 1953

Performance-to-price ratio rises as price rises

$$\text{performance} = k_G \times C^s \qquad k_G = \text{constant},\ C = \text{cost},\ s = \text{constant} \sim 2$$

$$\text{performance / cost} \sim k_G \times C$$

## If cost of multiprocessor system is linear in unit price of CPU

$$Cost(N) = \alpha \times N$$

$$\text{performance}(N) = k_G \times (\alpha \times N)^2 = k_G \alpha^2 N^2 \Rightarrow S = \frac{\text{performance}(N)}{\text{performance}(1)} = N^2$$

## Amdahl's law implies

$$\text{performance}(N) = \frac{k_{Amdahl}}{(1-F_P)+\frac{F_P}{N}} = \frac{k_{Amdahl}}{(1-F_P)+\frac{F_P}{Cost(N)/\alpha}} = \frac{Cost(N)}{(1-F_P)Cost(N)+\frac{\alpha}{k_{Amdahl}}F_P}$$

$$\frac{\text{performance}(N)}{Cost(N)} = \frac{1}{(1-F_P)Cost(N)+\frac{\alpha}{k_{Amdahl}}F_P},$$ for some constant $k_{Amdahl}$

# Claims Against Amdahl's Law

## Assumption in Amdahl's law

FP = constant

## Suppose instead

$$F_P = F_P(N) \text{ with } F_P(N) \xrightarrow[N\to\infty]{} 1$$

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

$$E = \frac{S}{N} \xrightarrow[N\to\infty]{} 1$$

## Gustafson-Barsis Law

Parallel part of large problem can scale with problem size

run time in serial execution $= s + p \times n$ , $n =$ size of problem

speedup compared to serial execution $= \frac{s + p \times n}{s + p} \xrightarrow[n \text{ large}]{} n$

# Interconnection Network Types

| | | | |
|---|---|---|---|
| **Static** | Permanent point-to-point connections between end nodes | | |
| | Full connectivity | | Requires N × (N-1) point-to-point connections |
| | Limited connectivity | | Requires multiple hops between end nodes |
| **Dynamic** | **Bus** | Nodes perform arbitration for bus access | |
| | | **Single** | Simplest implementation with standard I/O bus types VME, SCSI, PCI, datakit, etc |
| | | **Multiple** | End nodes connect to N identical buses in parallel |
| | **Switch** | Switch elements configured specifically for each transfer | |
| | | **Single Stage** | N × N switch with limited connectivity<br>Data makes multiple node-to-node hops between end nodes (source to destination) |
| | | **Multistage** | Full connectivity switch assembled from multiple single stage switches<br>Not simultaneously non-blocking |
| | | **Crossbar** | N × N simultaneous non-blocking connections |

# Communication Overhead and Amdahl's Law

## Parallelization with overhead

$F_P$ = fraction of program that can be parallelized $\Rightarrow IC_P = F_P \times IC$

Ideally $CPI \rightarrow CPI_{\text{parallel}} = CPI / N$

Including communication overhead $T^{comm}$ in speedup

$$T^{comm} = CPI^{comm} \times IC_P \times \tau = CPI^{comm} \times F_P \times IC \times \tau$$

$CPI^{comm}$ = processor clock cycles devoted to communication per instruction executed in parallel

$F_{overhead}$ = overhead factor = $CPI^{comm} / CPI$

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

$$= \frac{CPI}{CPI \times (1 - F_P) + \frac{CPI}{N} \times F_P + CPI^{comm} \times F_P} = \frac{1}{(1 - F_P) + F_P \left( \frac{1}{N} + F_{overhead} \right)}$$

# Large Communication Overhead

## Parallelization with large overhead

$$S = \frac{1}{(1-F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)}$$

$$F_{overhead} = \text{overhead factor} = \frac{CPI^{comm}}{CPI} = \frac{\text{communication activity}}{\text{processing activity}}$$

$$S_{\max} = \lim_{N\to\infty} \frac{1}{(1-F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)} = \frac{1}{(1-F_P) + F_P F_{overhead}}$$

$$= \frac{1}{1 - F_P(1 - F_{overhead})}$$

$$S_{\max} \xrightarrow[F_{overhead}\to 1]{} 1$$

**Communication overhead can eliminate benefits of parallelization**

# Shared Memory Architecture

# Shared Memory Multiprocessor

## Single global memory space 0, … , A – 1

Physically partitioned into M physical devices

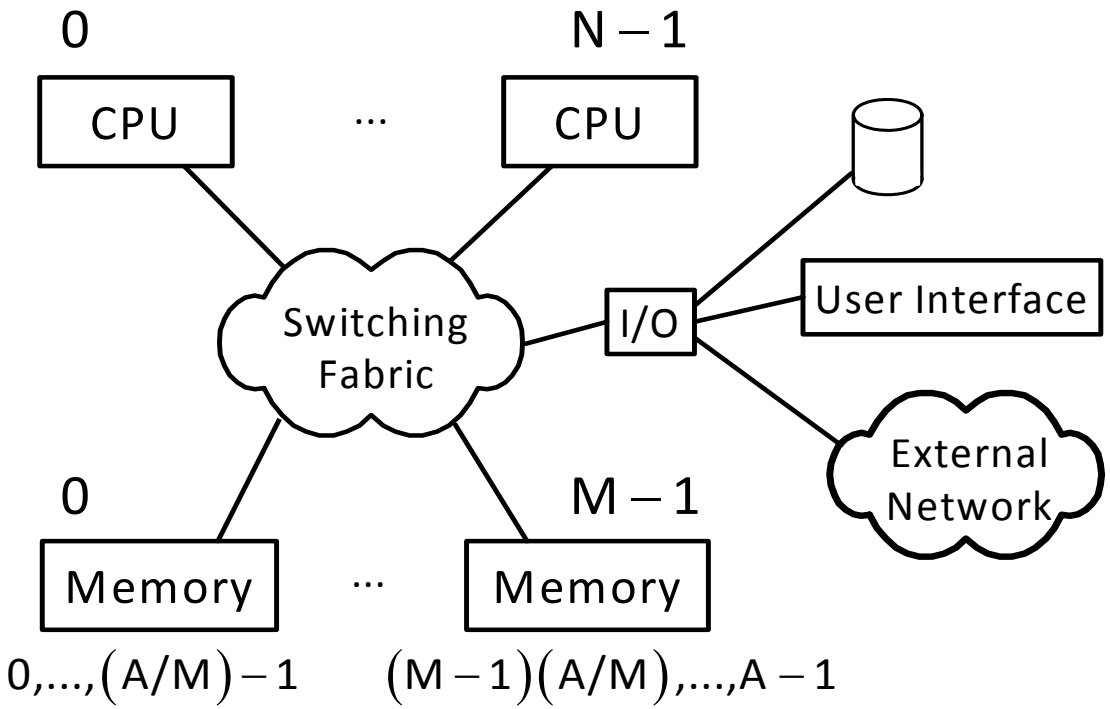
All CPUs access full memory space via interconnection network

## CPUs communicate via shared memory addresses

Standard write / read transactions to shared addresses

## Synchronize memory access

Prevent data hazards



# Shared Memory System Subtypes

## Uniform memory access (UMA)

- Shared memory accessible by all processors
- Access time to any memory location uniform for all processors
- Symmetric multiprocessor
  - All processors are of equivalent type
  - All processors have symmetric relationship to memory
  - Typically fast bus-based network (SCSI or PCI)

## Non-uniform memory access (NUMA)

- Memory distributed among processors
  - Each processor physically attached to part of shared memory
  - Cost-effective scaling to large systems
- Access time to memory address depends on distance to processor
  - Most memory accesses are local and fast

## Cache-only memory architecture (COMA)

- Each processor has part of shared memory as private cache
- Requires that data be migrated to processor requesting it

# Contemporary Trend

## Symmetric Multiprocessor (SMP)

N equivalent microprocessors

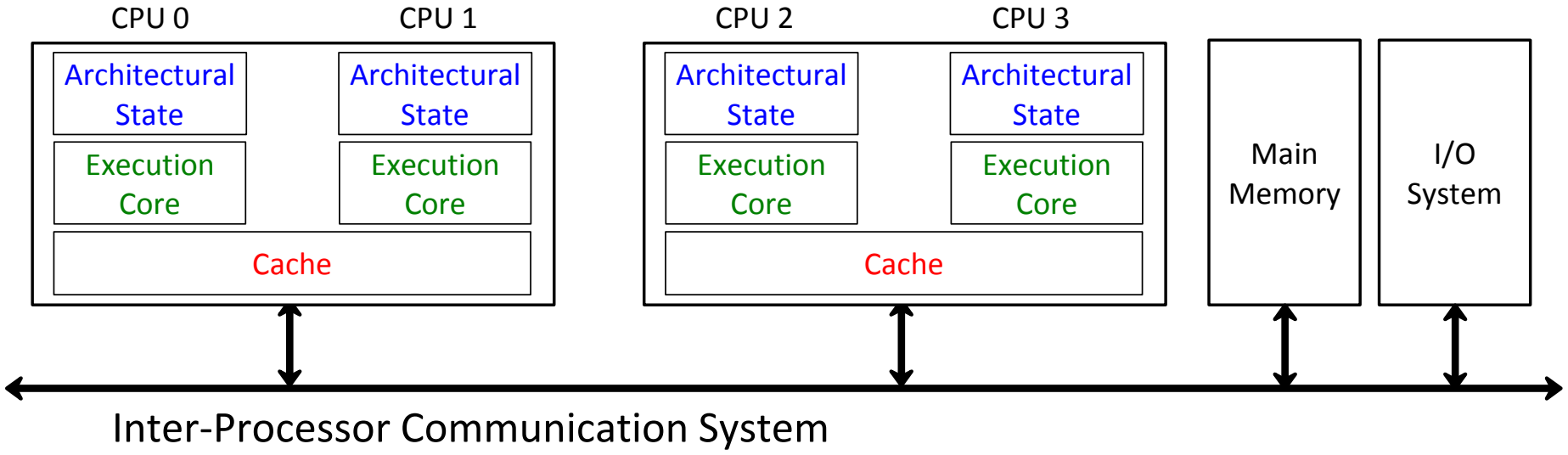
Multiple processor cores on single integrated circuit

Communication network between processors

## Thread Level Parallelism (TLP)

Operating system runs in one processor

OS assigns threads to processors by some scheduling algorithm



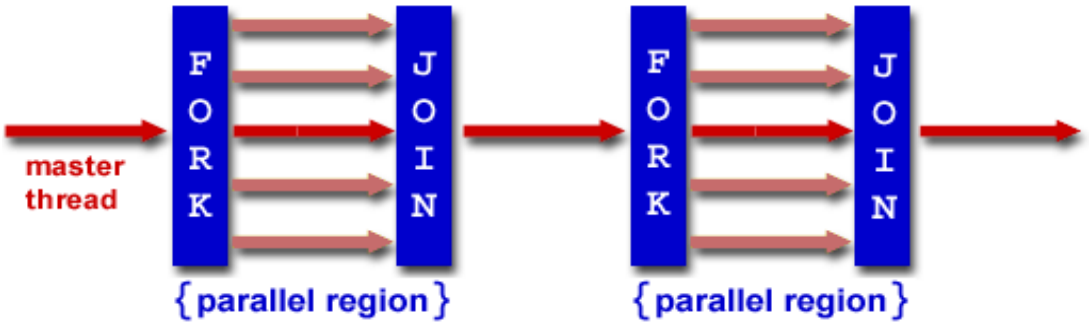
# Open MP for Shared Memory Systems

## Application Program Interface (API) for multiprocessing

- Supports shared memory applications in C/C++ and Fortran
- 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 threads
  - Master thread creates team of parallel threads
  - Program statements in parallel construct execute in parallel
- Join producer threads
  - Producer 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

# Selected Options for Parallel Pragma

**if (scalar_expression)**

Performs conditional multithreading

**private (list)**

Creates copy of variable types for each thread

**shared (list)**

Single shared copy of variables in list

Requires synchronization of memory accesses to avoid hazards

**reduction (operator: list)**

Performs join operation on list of private variables

**num_threads (integer-expression)**

Specifies number of threads to create

# "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 Directive

```
#pragma omp for
```

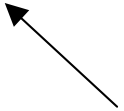
## Options

- schedule (type [,chunk])
  - static
    - Loop iterations divided into chunk pieces size
    - Statically assigned to threads
    - Chunk is not specified — iterations are evenly divided
  - dynamic
    - Loop iterations dynamically scheduled
    - When thread finishes chunk it is dynamically assigned another
- ordered
  - iterations executed as in a serial program
- nowait
  - No synchronization at end of parallel loop
  - Threads do not wait for other threads to finish at end of private code

# Parallel For Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
main ()
{
int i, chunk;
float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section */
}
```

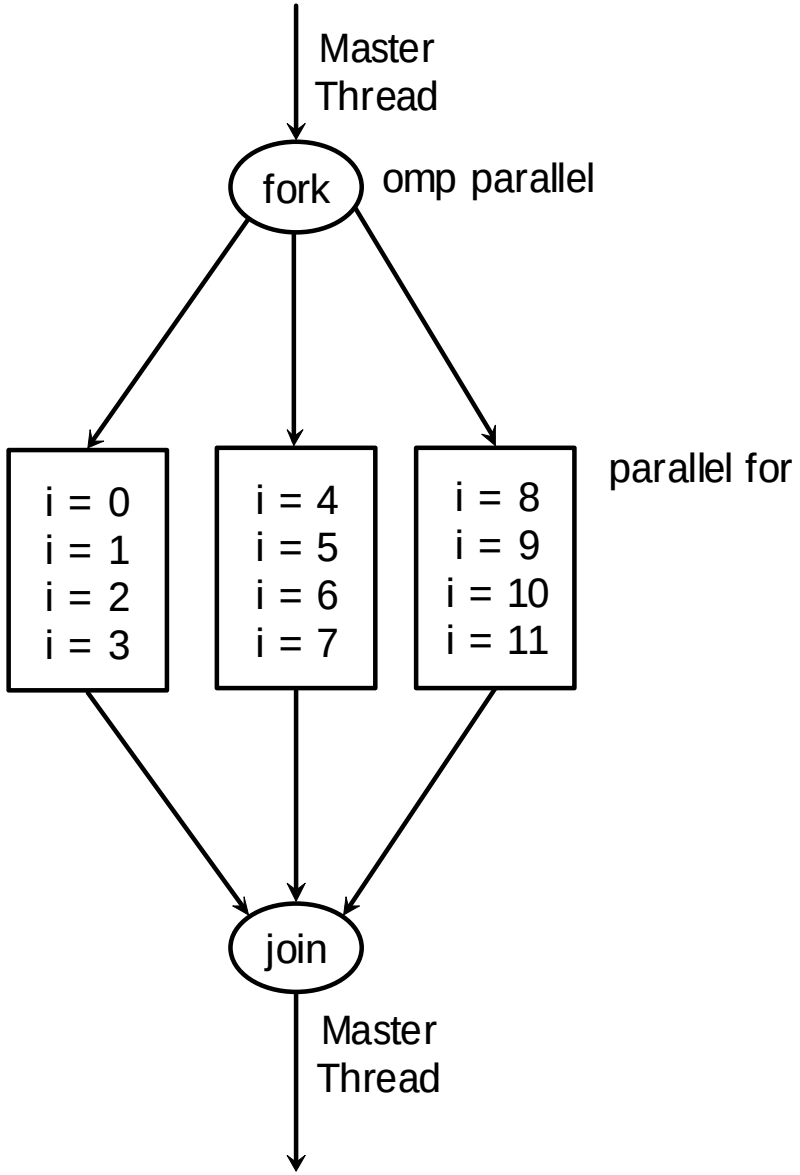
Data decomposition
- Arrays A, B, C, and variable N shared
- Variable i private
  - Each thread has unique copy
- Each loop iterates on chunk sized piece
- Threads do not synchronize (NOWAIT)



threads = N / chunk = 10

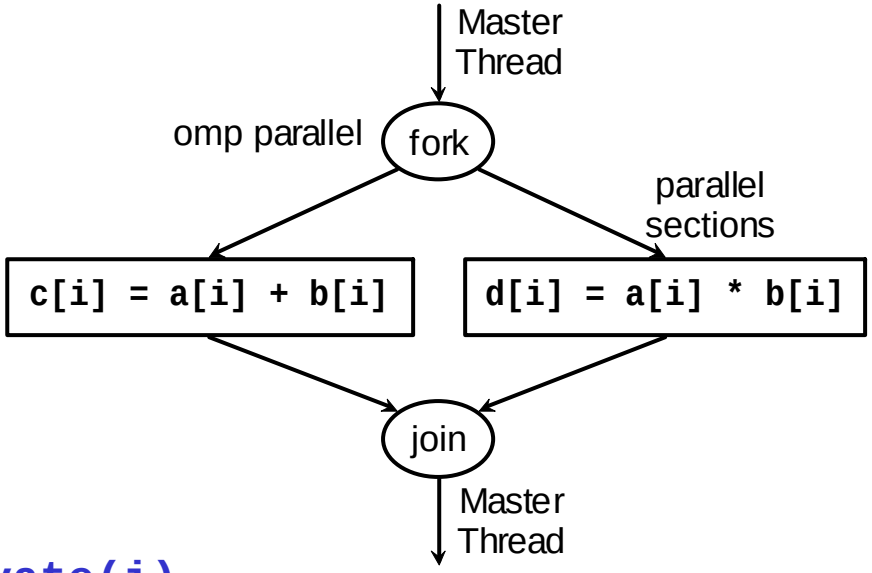
# Running Parallel For

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



# Sections Directive

```
#include <omp.h>
#define N     1000
main ()
{
int i;
float a[N], b[N], c[N], d[N];
for (i=0; i < N; i++){
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }
#pragma omp parallel shared(a,b,c,d) private(i)
  {
  #pragma omp sections nowait
    {
    #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

# Race Conditions

## Race condition

Data hazard caused by parallel access to shared memory

## Example

```
#pragma omp parallel shared(x) num_threads(2)
    {
      x = x + 1;
    }
```

Two threads should increment x independently: **x ← x + 2**

Interleaved execution sequence for **x = 5**

```
Thread 1 – R1 ← x          ; CPU1 loads copy of x = 5
Thread 2 – R1 ← x          ; CPU2 loads copy of x = 5
Thread 1 – R1 ← R1 + 1     ; CPU1 updates R1 ← 6
Thread 2 – R1 ← R1 + 1     ; CPU2 updates R1 ← 6
Thread 1 – x ← R1          ; CPU1 writes x ← 6
Thread 2 – x ← R1          ; CPU2 writes x ← 6
```

Program completes with result **x ← 6** instead of **x ← 7**

# Synchronization

## Directives to control access to shared data among threads

**`#pragma omp master`**

Only master thread (thread 0) performs following **`{block}`**

**`#pragma omp critical`**

Only one thread can execute following **`{block}`** at a time

Other threads wait for thread to leave critical section before entering

**`#pragma omp barrier`**

Each thread reaching barrier waits until all threads reach barrier

**`#pragma omp atomic`**

Memory update in next statement must be completed atomically

Mini-critical section for memory write

# Preventing Race Condition

**Example**

```
#pragma omp parallel shared(x) num_threads(2)
    {
      #pragma omp critical
      x = x + 1;
    }
```

Execution sequence with critical section for **x = 5**

```
Thread 1 – R1 ← x            ; CPU1 loads copy of x = 5
 ; Thread 2 blocks until thread 1 completes
Thread 1 – R1 ← R1 + 1       ; CPU1 updates R1 ← 6
Thread 1 – x ← R1            ; CPU1 writes x ← 6
 ; Thread 1 completes ⇒ thread 2 unblocked
Thread 2 – R1 ← x            ; CPU2 loads copy of x = 6
Thread 2 – R1 ← R1 + 1       ; CPU2 updates R1 ← 7
Thread 2 – x ← R1            ; CPU2 writes x ← 7
```

Program completes with result **x ← x + 2**

Performance implications — critical section runs sequentially

# Reduction

## Reduction (operator: list)

Performs join operation on list of private variables

Operation can be **`+ * - & ^ | && ||`**

## Example

```
#pragma omp parallel for reduction(+:sum)
for (i=0; i < N; i++)
{
     sum+ = a[i] * b[i];
}
```

Each thread has private copy of variable sum

On join (end of parallel construct)

- Private copies of sum combined by addition (+)
- Result copied into master thread copy of sum

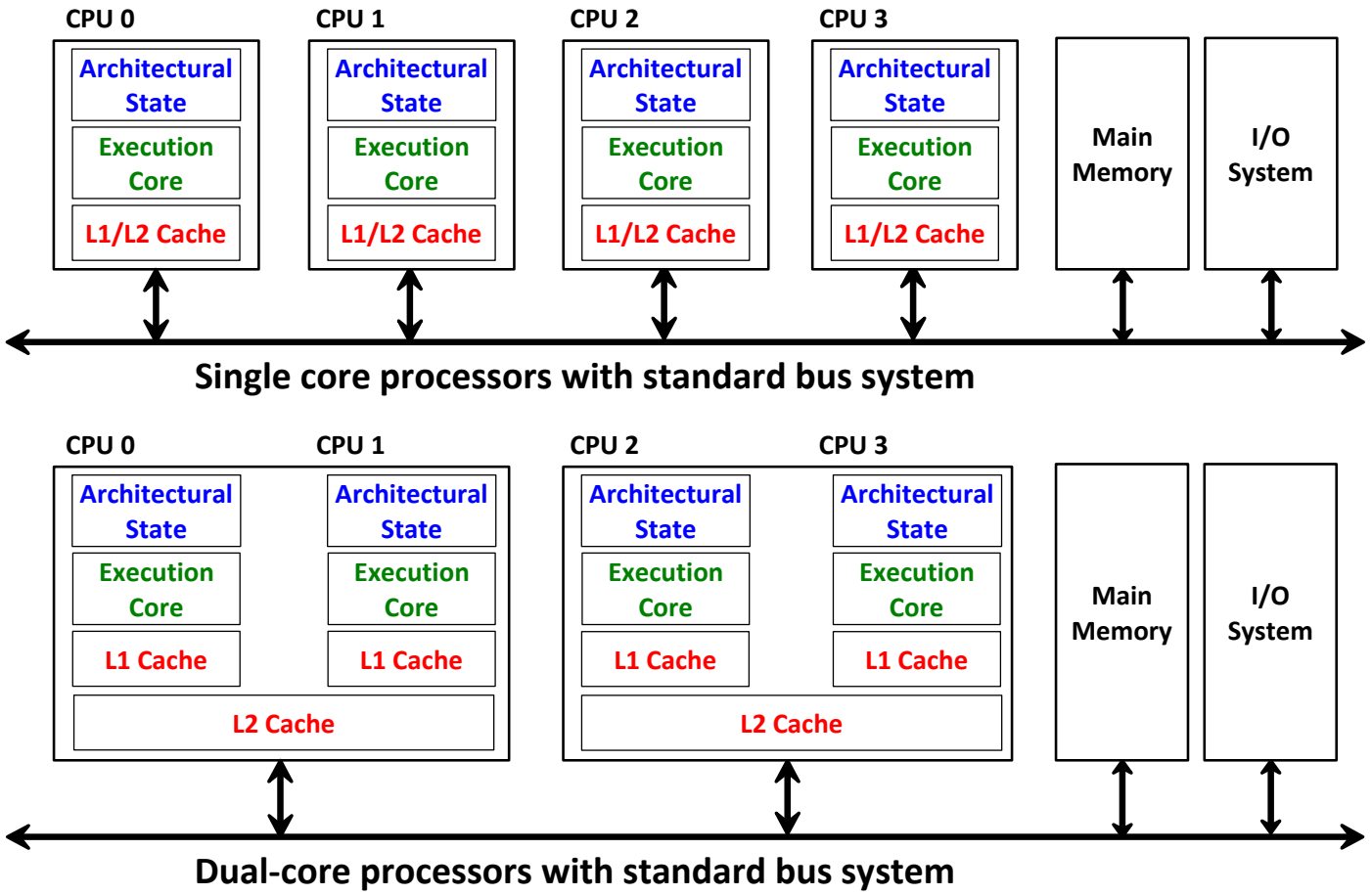
# Bus-Based Symmetric Multiprocessor (SMP)

## Bootstrap Processor (BSP)

Chosen by BIOS to perform system startup and run OS

## Application Processors (AP)

In halt state unless allocated application threads by OS



# Cache Organization for Dual Core Processors

## Pentium D

Dual core processor

Each core has private L1 D+I cache and L2 D+I cache

## Core Duo, Core 2, i3, i5, i7, ...

Dual core processor

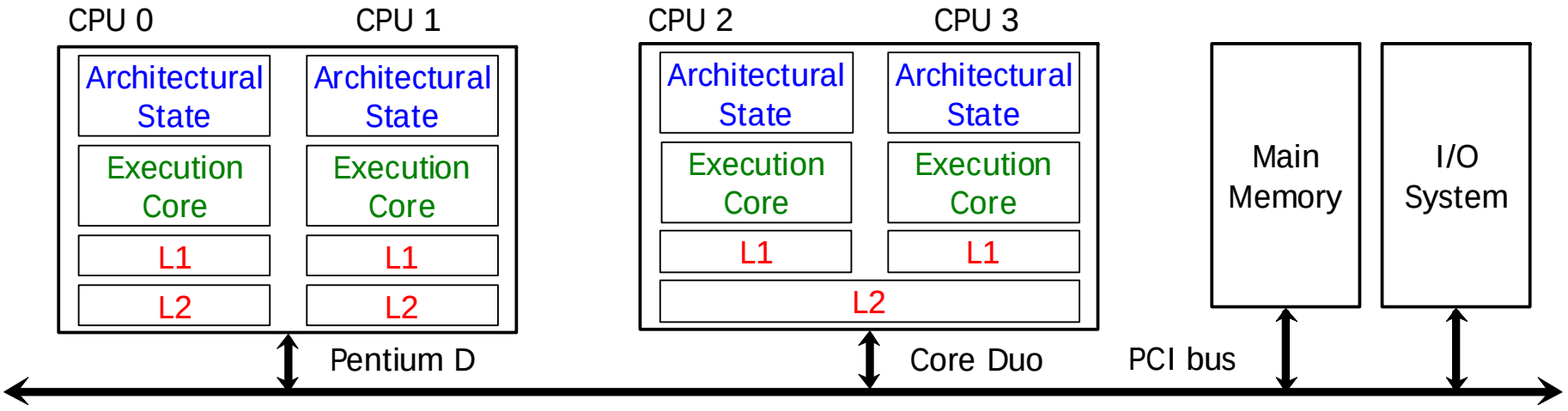
Each core has private L1 data cache

Cores share L2 cache

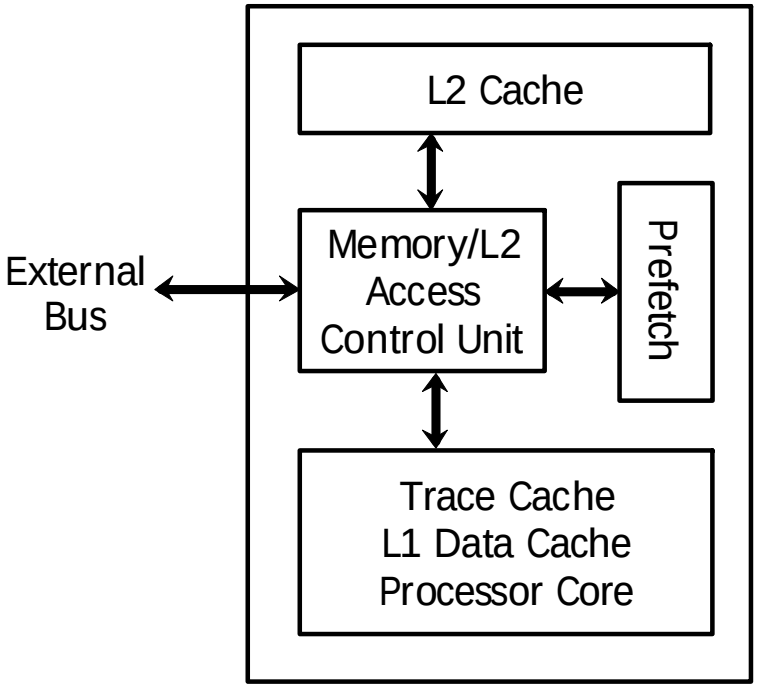
No L1 instruction cache

Instructions fetched directly from L2

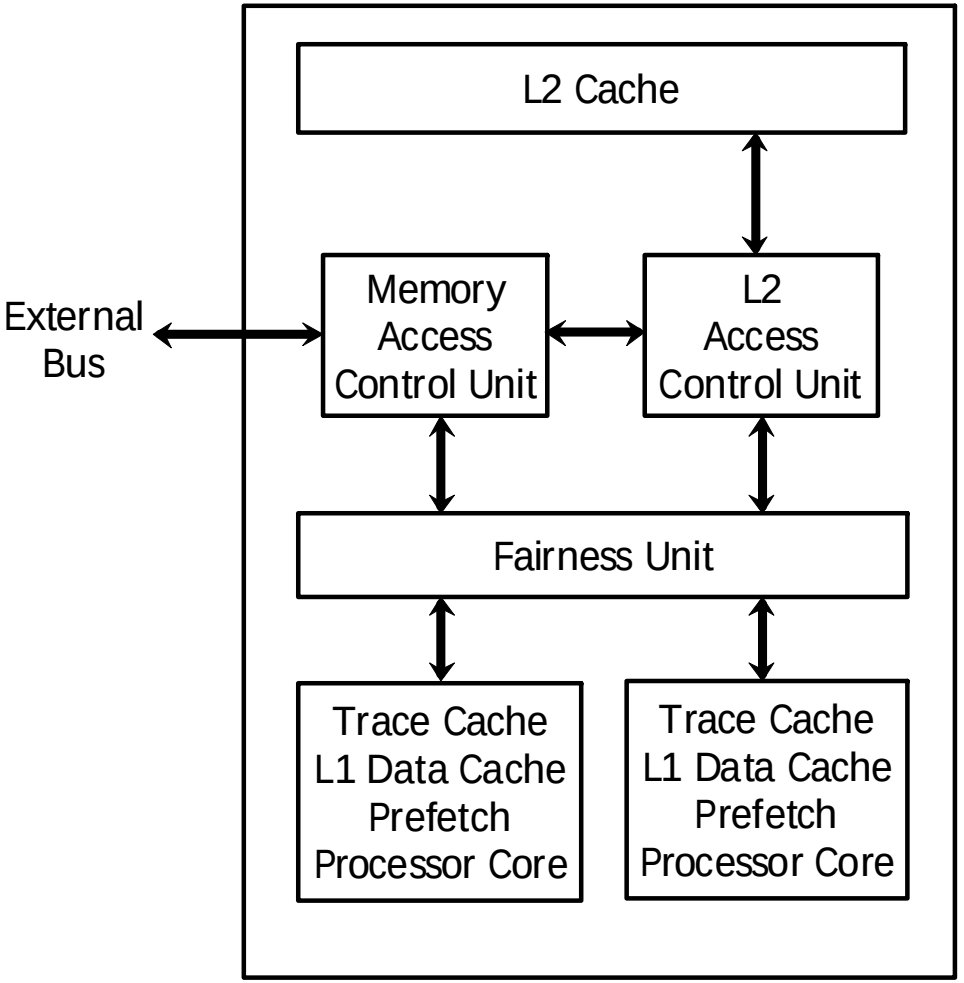
Employ trace caching instead



# Memory/L2 Access Control Unit



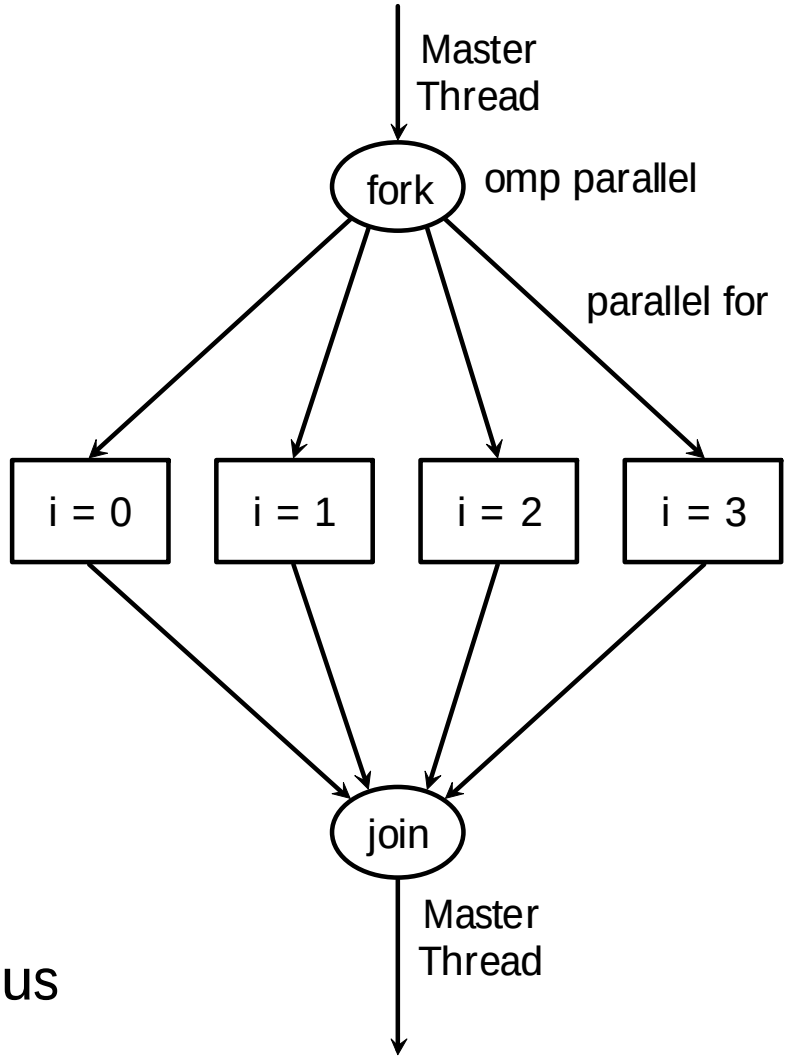

**Pentium-M**




**Core Duo**

# Communication Overhead in SMP

```
#pragma omp parallel
{
#pragma omp for
        for (i=0; i<4; i++)
                c[i] = a[i] + b[i];
}
```




## Sequential program requires

4 × load-add-store

## Parallel program requires

4 × update instruction cache over serial bus

1 × load-add-store

In most architectures cache updates more expensive than arithmetic

# Shared Memory Program — Vector Product

Compute $\sum_{i=0}^{3} \mathbf{a[i]} * \mathbf{b[i]}$ from data in shared memory

## Sequential Code

```
for (i=0; i<4; i++){
  load Ra, a[i]
  load Rb, b[i]
  Ra ← Ra * Rb
  add Racc, Ra
}
store p, Racc
```

CC ~ overhead + 4 × 4 + 1

Neglecting overhead

$$S \sim \frac{17}{12} = 1.42$$

## Parallel Code

```
fork 4 threads with private i=0,1,2,3
  load Ra, a[i]          …   load Ra, a[i]
  load Rb, b[i]              load Rb, b[i]
  Ra ← Ra * Rb               Ra ← Ra * Rb
  store p[i], Ra             store p[i], Ra
fork 2 threads with private i=0,2
  load Ra, p[i]          …   load Ra, p[i]
  load Rb, p[i+1]            load Rb, p[i+1]
  Ra ← Ra + Rb               Ra ← Ra + Rb
  store p[i], Ra             store p[i], Ra
load Ra, p[0]
load Rb, p[2]
Ra ← Ra + Rb
store p, Ra
```

CC ~ overhead + 1 × 4 + 1 × 4 + 4

# Shared Memory Multiprocessor Capacity

## Capacity limitation

CPUs operate independently on data cache

Access shared memory to exchange data when required

Data exchange required $\Rightarrow$ cache miss on at least one CPU

## Capacity definition

Interconnection network capacity $\geq$ volume of exchanged data

Exchange demand volume $= N \times D_x \times M$

$N =$ CPUs

$D_x =$ average data access rate (bytes per second)

$M =$ cache miss rate $=$ inter-CPU access rate

Interconnection capacity $= R_n \times W$

$R_n =$ transfer rate (transfer cycles per second)

$W =$ transfer width (bytes per transfer)

$$N \leq \frac{R_n W}{D_x M}$$

# Capacity Example

**Standard PCI express bus — 16 bytes per cycle at 1 GHz**

**Average data access rate depends on**

- Integer width
- Loads per instruction
- Instructions per second = 1/[(seconds per CC) × (CC per instruction)]

**Miss rate depends on number of data reads between cache updates**

- Compute dominated M ~ 0.01
- Communication dominated M ~ 0.1

$$N \le \frac{R_n W}{D_x M}$$

$$N \le \frac{10^9 \text{ transfers/sec} \times 16 \text{ bytes/transfer}}{10^9 \text{ bytes/sec} \times 0.1} = 160$$

$R_n = 1$ GHz

$W = 16$ bytes/transfer

$$D_x = (4 \text{ bytes/load}) \times (0.25 \text{ data loads/instruction}) \times (10^9 \text{ instruction/sec})$$

$$= 10^9 \text{ bytes/sec}$$

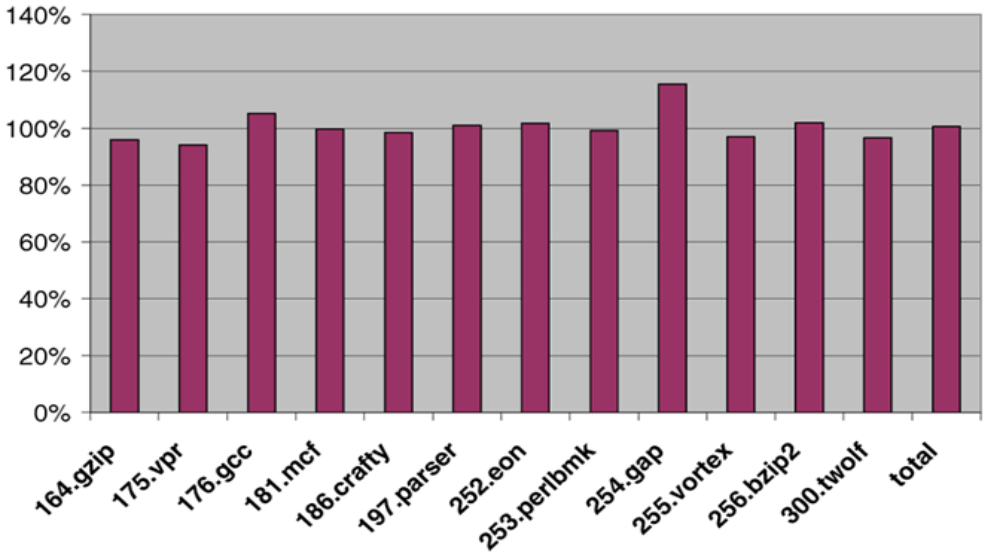
$M = 0.1$ (communication dominated miss rate)

# Single-Threaded Performance

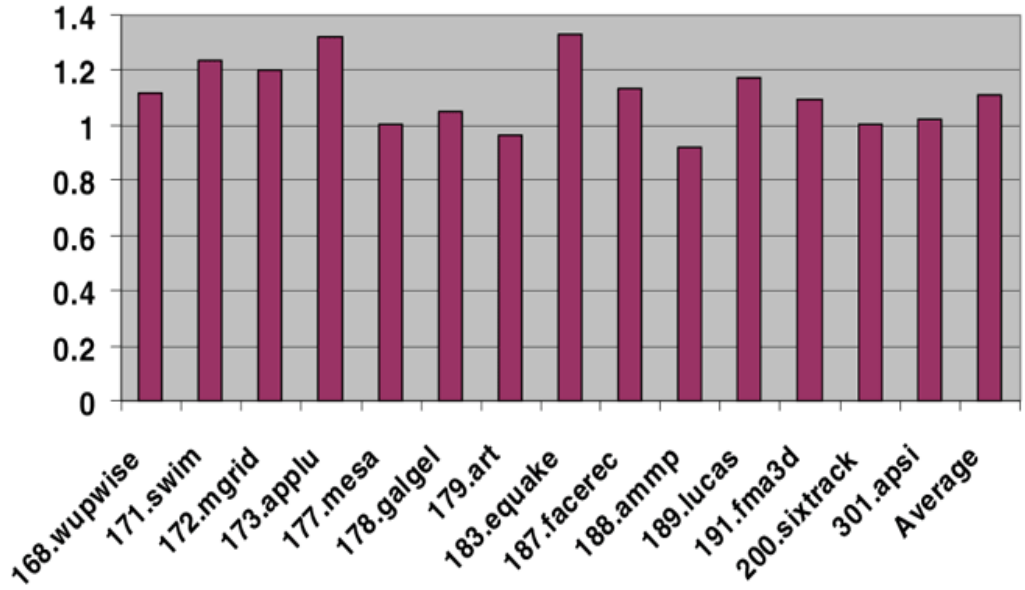
## Core Duo vs. Pentium M

Single-threaded SPEC code performance

### Cint



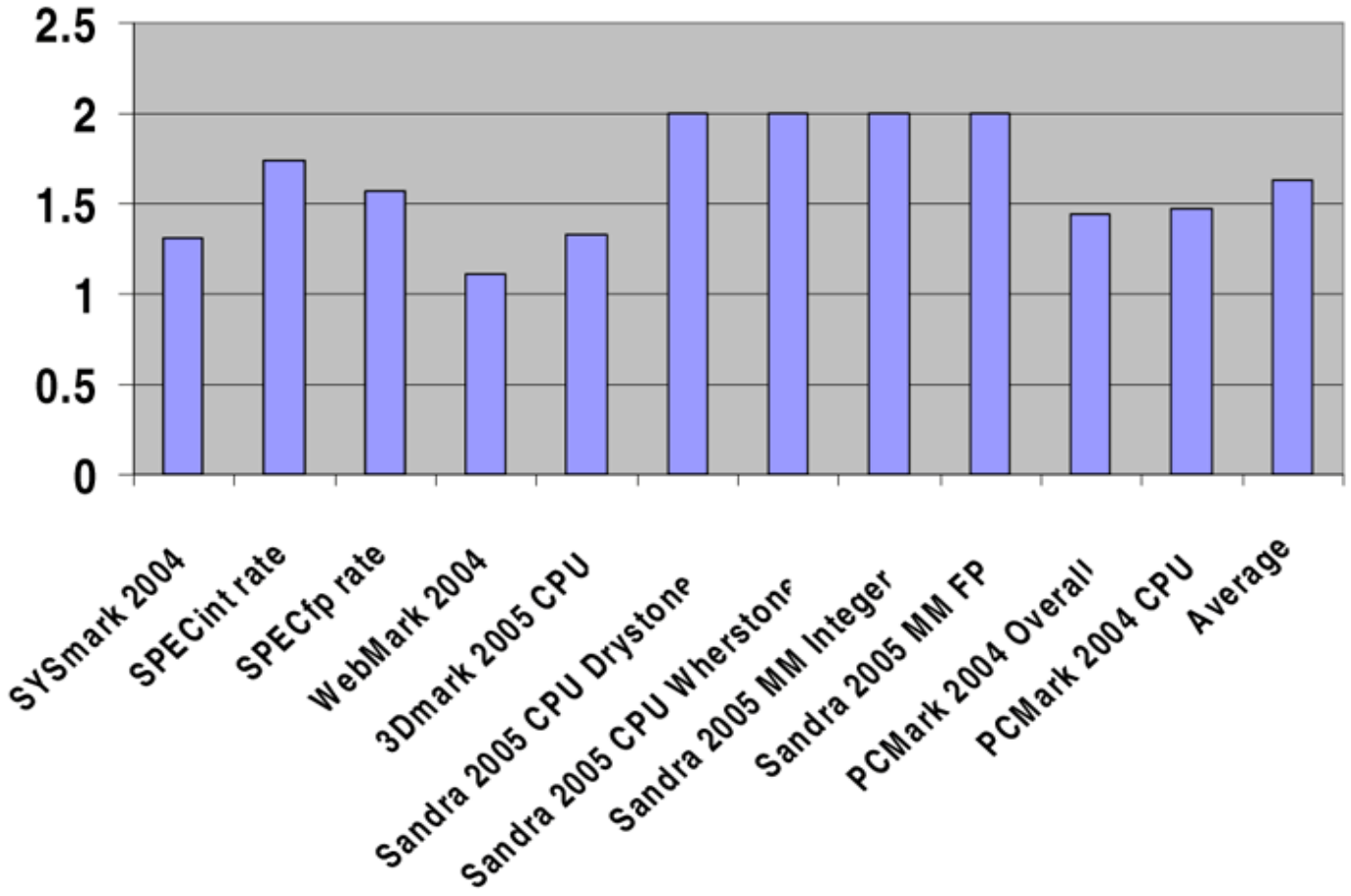

### Cfp



# Multi-Threaded Performance

## Core Duo vs. Pentium M

Multithreaded code performance



# Effect of Threading Model

## Data domain decomposition

Divide large data arrays among processors

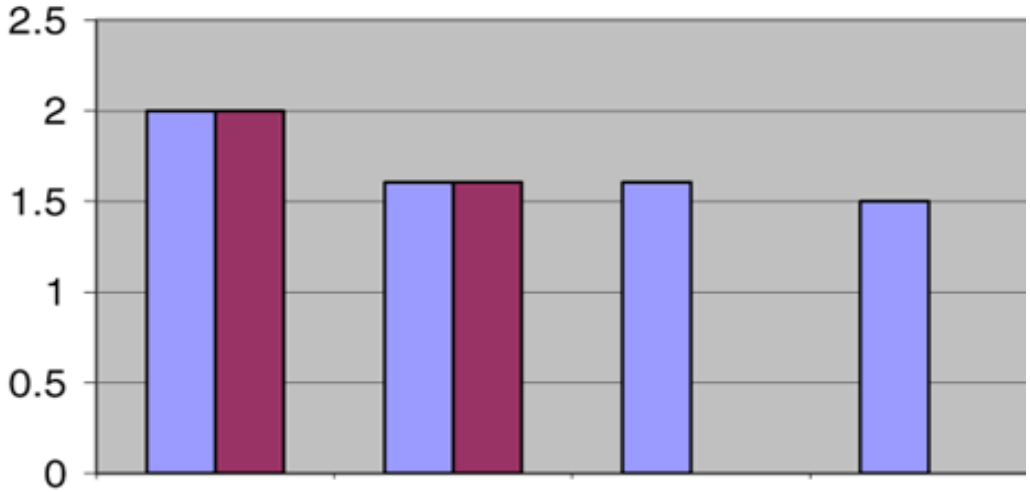
Works well with shared memory SMP

## Functional domain decomposition

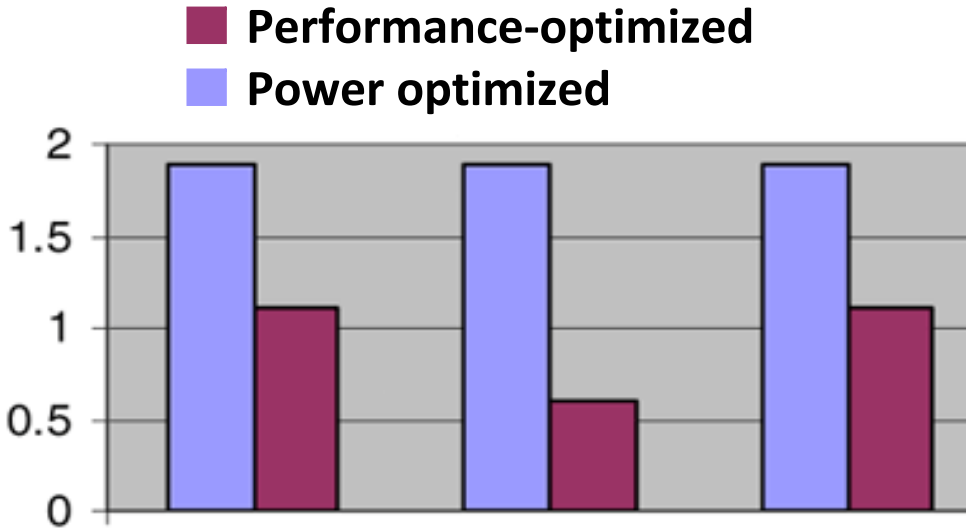
Divide tasks among processors

Tends toward load balancing problems on SMP

Works better with work queue systems on message passing systems




Data decomposition on Core Duo




Functional decomposition on Core Duo

# Data Hazards in Shared Memory SMP

## Three levels of data hazard in shared memory systems

- Program level
  - Concurrent programming of inherently sequential operations
  - Handled with programmed synchronization directives
    - Atomic read/write, critical sections, barrier
- Cache consistency
  - Multiple processors writing shared copies of data
  - Protocols for maintaining valid copies of data values
- Hardware level consistency
  - Instruction-level memory semantics are abstractions
  - Real hardware operates in more complex manner

## General approach to handle hazards

- Enforce operational definition for consistency
- Enables unambiguous program validation

Ref: Adve, Gharachorloo, "Shared Memory Consistency Models: A Tutorial", http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf

# Sequential Consistency

## Strict consistency

- Memory operations performed in order intended by programmer
- Possible to implement only on single processor systems

## Sequential consistency (Lamport 1979)

- Clear, consistent, repeatable definition of execution order
- Result of any execution identical to result of execution in which:
  - Operations on all processors executed in specified sequential order
  - Operations on each processor appear in order specified by program
- Specified sequential order
  - Any well-defined interleaving
  - Example — round robin

## Implications for programmer

- Unsynchronized threads assumed to execute interleaved
- Cache assumed to enforce write-order consistency
- Hardware assumed to enforce read/write-order consistency

# Implementing Critical Section with Semaphore

## Semaphore

Unsigned number with 2 operations

$$\mathbf{P(s)}: \begin{cases} \mathbf{s} \leftarrow \mathbf{s}-\mathbf{1}, & \mathbf{s} > \mathbf{0} \\ \text{wait}, & \mathbf{s} = \mathbf{0} \end{cases}$$

$$\mathbf{V(s)}: \quad \mathbf{s} \leftarrow \mathbf{s}+\mathbf{1}$$

## Binary semaphore

Mutual exclusion (mutex) or lock

**s** initialized to **1**

## Critical section

```
P(s)      /* section begins (s=1) or blocks (s=0) */
    x = x + 1;
V(s)      /* s ← 1 permits another thread to operate */
```

## Difficulty

Requires system-wide atomic semaphore operation

Impractical to disable all system interrupts and interleaving during P and V

# Shared Variable Lock

## Shared variables `Flag1` and `Flag2` initialized to zero

| Thread 1 | Thread 2 |
|---|---|
| `Flag1 = 1;`<br>`loop: if (Flag2 == 1) loop`<br>`<critical section>`<br>`Flag1 = 0;` | `Flag2 = 1;`<br>`loop: if (Flag1 == 1) loop`<br>`<critical section>`<br>`Flag2 = 0;` |

## Spin-loop

Loop instruction repeats until condition clears

## Interleaved execution

Actual execution order `t1, t2, t3, t4`

Creates deadlock

| Thread 1 | Thread 2 |
|---|---|
| `Flag1 = 1;` `t1`<br>`loop: if (Flag2 == 1) loop` `t3`<br>`<critical section>`<br>`Flag1 = 0;` | `Flag2 = 1;` `t2`<br>`loop: if (Flag1 == 1) loop` `t4`<br>`<critical section>`<br>`Flag2 = 0;` |

# Modified Shared Variable Lock

## Shared variables `Flag1` and `Flag2` initialized to zero

| Thread 1 | Thread 2 |
|---|---|
| `loop: Flag1 = 1;`<br>`if (Flag2 == 1)`<br>`{Flag1 = 0 ; loop}`<br>`<critical section>`<br>`Flag1 = 0;` | `loop: Flag2 = 1;`<br>`if (Flag1 == 1)`<br>`{Flag2 = 0 ; loop}`<br>`<critical section>`<br>`Flag2 = 0;` |

## Interleaved execution

Order `t1, t2, t3, t4`

No deadlock but possible livelock if hardware writes not atomic

| Thread 1 | Thread 2 |
|---|---|
| `loop: Flag1 = 1;` `t1`<br>`if (Flag2 == 1)`<br>`{Flag1 = 0 ; loop}` `t3`<br>`<critical section>`<br>`Flag1 = 0;` | `loop: Flag2 = 1;` `t2`<br>`if (Flag1 == 1)`<br>`{Flag2 = 0 ; loop}` `t4`<br>`<critical section>`<br>`Flag2 = 0;` |

# Dekker Algorithm

## Shared variables `Flag1, Flag2 = 0 ; turn = 1`

| Thread 1 | Thread 2 |
|---|---|
| `Flag1 = 1;`<br>`turn = 1;`<br>`loop: if (Flag2 == 1`<br>`&& turn == 1) loop`<br>`<critical section>`<br>`Flag1 = 0;` | `Flag2 = 1;`<br>`turn = 2;`<br>`loop: if (Flag1 == 1)`<br>`&& turn == 2) loop`<br>`<critical section>`<br>`Flag2 = 0;` |

| Thread 1 | Thread 2 |
|---|---|
| `Flag1 = 1; t1`<br>`turn = 1; t3`<br>`loop: if (Flag2 == 1`<br>`&& turn == 1) loop t5`<br>`<critical section>`<br>`Flag1 = 0;` | `Flag2 = 1; t2`<br>`turn = 2; t4`<br>`loop: if (Flag1 == 1)`<br>`&& turn == 2) loop t6`<br>`<critical section>`<br>`Flag2 = 0;` |

## No deadlock or livelock

Generalization to n > 2 threads — Lamport bakery algorithm

# Machine Language Support

## Atomic instruction primitives in processor ISA

Provide hardware-level semaphore **M** for well-defined atomic memory access

Enable implementation of atomic constructs at compiler level

| Instruction primitive | Application |
|---|---|
| **Test_and_Set M, R**<br>**Regs[R] ← Mem[M]**<br>**if Regs[R] == 0 Mem[M] ← 1** | **L1: Test_and_Set M, R1 ; spinlock**<br>**BNEZ R1, L1**<br>**<critical section>**<br>**Swap M, R1** |
| **Fetch_and_Add M, R1, R2**<br>**Regs[R1] ← Mem[M]**<br>**Mem[M] ← Regs[R1] + Regs[R2]** | **ADDI R2, R0, #1**<br>**L1: Fetch_and_Add M, R1, R2**<br>**BNEZ R1, L1**<br>**<critical section>**<br>**Swap M, R1** |
| **Swap M, R:**<br>**Regs[$R_{temp}$] ← Mem[M]**<br>**Mem[M] ← Regs[R]**<br>**Regs[R] ← Regs[$R_{temp}$]** | |

# Compare and Swap (CAS)

**Swaps M and R2 if M = R1**

```
Compare_and_Swap M, R1, R2
    if (Regs[R1] == Mem[M]){
        Mem[M] = Regs[R2]
        Regs[R2] = Regs[R1]
        Cflag ← 1
     }
    else Cflag ← 0
```

**No lock**

Non-blocking atomic operation more efficient (M. Herlihy — 1991)

| | |
|---|---|
| **Critical Section** | `#pragma omp critical`<br>`  x = x + 1;` |
| **Assembly Code** | `L1:  LW R1, x  ; load x`<br>`  ADDI R2, R1, #1  ; prepare new value for x`<br>`  CAS x, R1, R2  ; if no change in stored x, update x`<br>`  BEQZ Cflag, L1  ; else start again` |

# Load Reserve and Store Conditional

## Load-reserve

Returns current value of memory location

Associates reservation flag with address

Flag can be reset by subsequent load-reserve

## Store-conditional

Performs write if reservation flag still set

## Stronger than compare and swap

Prevents store in location that was written after read

Even if original value was restored

```
load-reserve R, M
    <flag, adr> ← <1, M>;
    Regs[R] ← Mem[M];
```

```
store-conditional M, R
    if <flag, adr> == <1, M>
      clear <flag, adr>;
      Mem[M] ← Regs[R];
      status ← 1;
    else  status ← 0;
```

```
L1: load-reserve R1, x
    ADDI R1, R1, #1
    store-conditional x, R1
    BEQZ status, L1
```

# Hardware Level Consistency without Cache

## Instruction-level memory semantics

ISA defines unambiguous sequential actions

**`LW R1, 32(R2)` ⇒ `Regs[R1]` ← `Mem[Regs[R2] + 32]`**

**`SW 32(R2), R1` ⇒ `Mem[Regs[R2] + 32]` ← `Regs[R1]`**

## Hardware-level implementation

Hardware optimizations alter presumed order of instruction actions

Instruction behavior meets definition on single processor machines

Can create hazards in multiprocessors

## Typical hazards

Single processor may reorder data reads and writes

Multiple processors may reorder reads and writes among themselves

Multiple processors write to multiple memory devices simultaneously

# Example — Write Buffer with Bypass

## Write buffer

Processor writes updates to memory through FIFO buffer

Prevents memory write latency from stalling pipeline

## Memory reordering

Write buffer permits reads to bypass previous writes

Prevents memory write queue from stalling subsequent reads

## Forwarding

Read hit to address in write buffer updates directly from FIFO

No data hazards for single processor system

## Violates sequential coherency in multiprocessor system

Example — Dekker algorithm with bypassing write buffer

Load **`other_flag`** bypasses Store **`my_flag`** in both processors

Load **`other_flag`** always returns **`0`**

# Example — Write Overlapping

## Multiprocessor system with general interconnect network

- Crossbar or multiple bus network
- Memory partitioned into multiple physical devices
- Permits multiple memory writes per transfer cycle

## Violates sequential coherency in multiprocessor system

- Example

```
#pragma omp critical
  x = x + 1;
```

- Critical section implemented with Dekker algorithm (not CAS)
- Thread 1
  - Issues write instructions to **`x`** and **`Flag1`** releasing lock
  - Write to **`x`** delayed in network
  - Write to **`Flag1`** completes
- Thread 2 completes
- Thread 1 write to **`x`** completes after Thread 2 write to **`x`**

# Example — Read/Write Reordering

## Multiprocessor system with general interconnect network

Permits multiple memory writes per transfer cycle

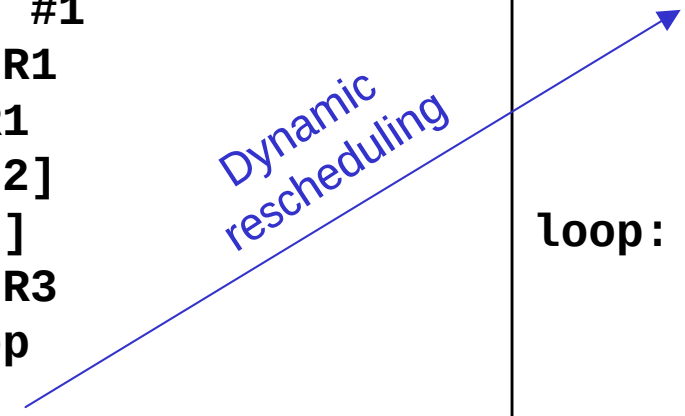
## Violates sequential coherency in multiprocessor system

Example
```
#pragma omp critical
    x = x + 1;
```

Critical section implemented with Dekker algorithm (not CAS)

Thread 1 runs out-of-order but Thread 2 runs in-order

| Thread 1 Listing | Thread 1 Out-of-Order Execution |
|---|---|
| `ADDI R1, R0, #1`<br>`SW [Flag1], R1`<br>`SW [turn], R1`<br>`loop: LW R2, [Flag2]`<br>`LW R3, [turn]`<br>`AND R4, R2, R3`<br>`BNEZ R4, loop`<br>`LW R5, [x]`<br>`ADDI R5, R5, #1`<br>`SW [x], R5`<br>`SW [Flag1], R0` | `LW R5, [x]`<br>`ADDI R1, R0, #1`<br>`SW [Flag1], R1`<br>`SW [turn], R1`<br>`loop: LW R2, [Flag2]`<br>`LW R3, [turn]`<br>`AND R4, R2, R3`<br>`BNEZ R4, loop`<br>`ADDI R5, R5, #1`<br>`SW [x], R5`<br>`SW [Flag1], R0` |



# Memory Fences

## Memory barrier (membar)

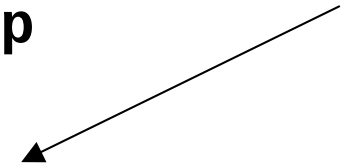
Machine-level instruction inserted by programmer or compiler

Enforces sequential consistency in rescheduling

Instructions are not moved past a membar

## Example

```
        ...
loop:   BNEZ R4, loop
        MEMBAR
        LW R5, [x]
        ADDI R5, R5, #1
        ...
```



Processor will not execute load before memory barrier

# Cache Policies in Single CPU System

## Read miss

- Update cache (copy block from memory)

## Write miss

- Write allocate policy
  - Update cache
- No write allocate
  - No cache update
  - Write data directly to memory

## Write to cache

- Write through
  - Update memory (copy block from cache) on every cache write
  - Maintains cache-memory consistency
- Write back
  - Update memory when block swapped out
  - Requires fewer memory bus operations

# Cache Consistency and Coherency

## Multiple processors access shared data in main memory

- Each processor copies data blocks to private cache
- Caches and main memory become inconsistent

## Example

- CPU-0 reads **X** to cache from main memory
- CPU-1 reads **X** to cache from main memory
- CPU-0 writes to **X** in cache
  - Copies of **X** in CPU-1 cache and main memory not valid

## Strict consistency

- All copies of any data location are identical
- Almost impossible to implement

## Sequential consistency

- Operations on all processors executed in specified sequential order
- Operations on each processor appear in order specified by program

## Cache coherency

- Implementation of sequential consistency for cache and main memory
- Requires processor hardware support for cache coherency management

# Cache Inconsistency in Write Back Policy

## Concurrent threads

```
#pragma omp critical
 x = x + 1;
```

Each thread enforces sequential consistency with its private cache

Criticality not enforced on all copies of **x**

## Example

| CPU-0 | Cache 0 | Memory | Cache 1 | CPU-1 |
|---|---|---|---|---|
| LW R5, [x] | x = 2 | x = 2<br>y = 10 | | |
| ADDI R5, R5, #1 | x = 2 | x = 2<br>y = 10 | x = 2 | LW R5, [x] |
| SW [x], R5 | x = 3 | x = 2<br>y = 10 | x = 2 | ADDI R5, R5, #1 |
| LW R6, [y]<br>; write back x | y = 10 | x = 3<br>y = 10 | x = 3 | SW [x], R5 |
| | | x = 3<br>y = 10 | y = 10 | LW R6, [y]<br>; write back x |

# Cache Inconsistency in Write Through Policy

## Concurrent threads

```
#pragma omp critical
 x = x + 1;
```

Each thread enforces sequential consistency with its private cache

Criticality not enforced on all copies of **x**

## Example

| CPU-0 | Cache 0 | Memory | Cache 1 | CPU-1 |
|---|---|---|---|---|
| LW R5, [x] | x = 2 | x = 2 | | |
| ADDI R5, R5, #1 | x = 2 | x = 2 | x = 2 | LW R5, [x] |
| SW [x], R5 | x = 3 | x = 3 | x = 2 | ADDI R5, R5, #1 |
| | | x = 3 | x = 3 | SW [x], R5 |

# Cache Coherence Protocols

## Cache coherence

Enforce sequential consistency for all caches and main memory

Enables system-wide atomic and critical constructs

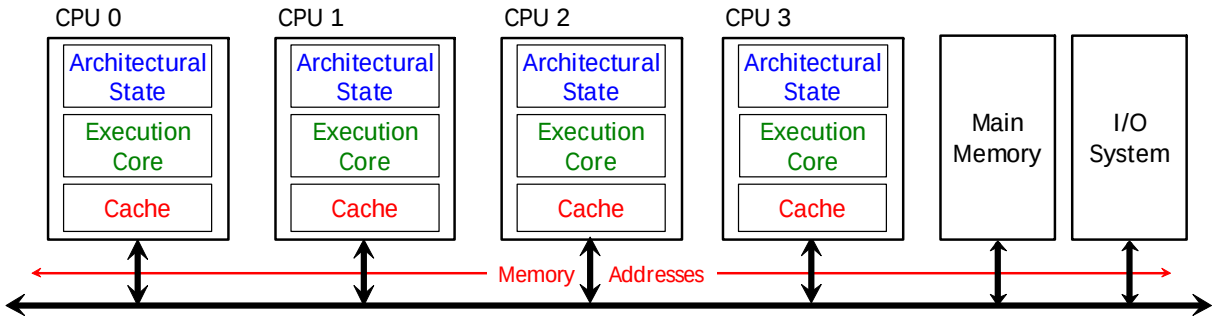
## Snoopy cache

Cache blocks tags with status bits depending on coherence policy

- Status depends on access history to data in block
- Change in status can initiate write back before usual block eviction

Cache manager

- Monitors all addresses written on system bus by all processors
- Compares addresses with blocks in cache
- Updates state of cached block on address hit



# Cache Coherence Policy Types

## Write update

- Processor W places write address on bus
- Snooping processors $S_i$ mark copy of block invalid on address hit
- W writes to cache and to bus
- $S_i$ update local copies of block and mark updated blocks valid
- Creates large bus overhead

## Write invalidate with inquire

- Processor W places write address on bus
- Snooping processors $S_i$ mark copy of block invalid on address hit
- W writes to local cache — creates unique valid copy of data block
- Inquire — $S_i$ place address on bus to access copy of block in W
- Lower bus overhead

## Typical implementation in shared memory system

- Write allocate — W loads to cache on write miss
- Write invalidate — $S_i$ invalidate cache block on write by W

# State Definitions

**Possible states of data block**

Indicated by status bits in block tag

**Modified**

Unique valid copy of block

Block has been modified since loading

**Owned**

Device is cache is owner of block

Device services requests by other processors for block

**Exclusive**

Unique valid copy of block

Block has not been modified since loading

**Shared**

Block held by multiple caches in system

**Invalid**

Block must be reloaded before next access

# Processor Behavior — Modified

**Processor W cache holds unique valid copy of data block**

- Block is dirty
  - Memory copy is different from W copy
- Cache copies in other processors $S_i$ marked Invalid
  - $S_i$ inquire (request update from this copy) before accessing block

**W can continue to update this copy**

- No memory update on cache writes
- Memory update on cache swap or inquire

**W responds to inquire**

- Snoops memory address from $S_i$ on bus
- W updates memory
- Marks block invalid on write inquire

# Processor Behavior — Shared

## Processor W cache holds one of many valid copies of data line

- Block is clean
  - Memory copy is same as W copy
- Other processors $S_i$ may have copies of block

## W can update this copy

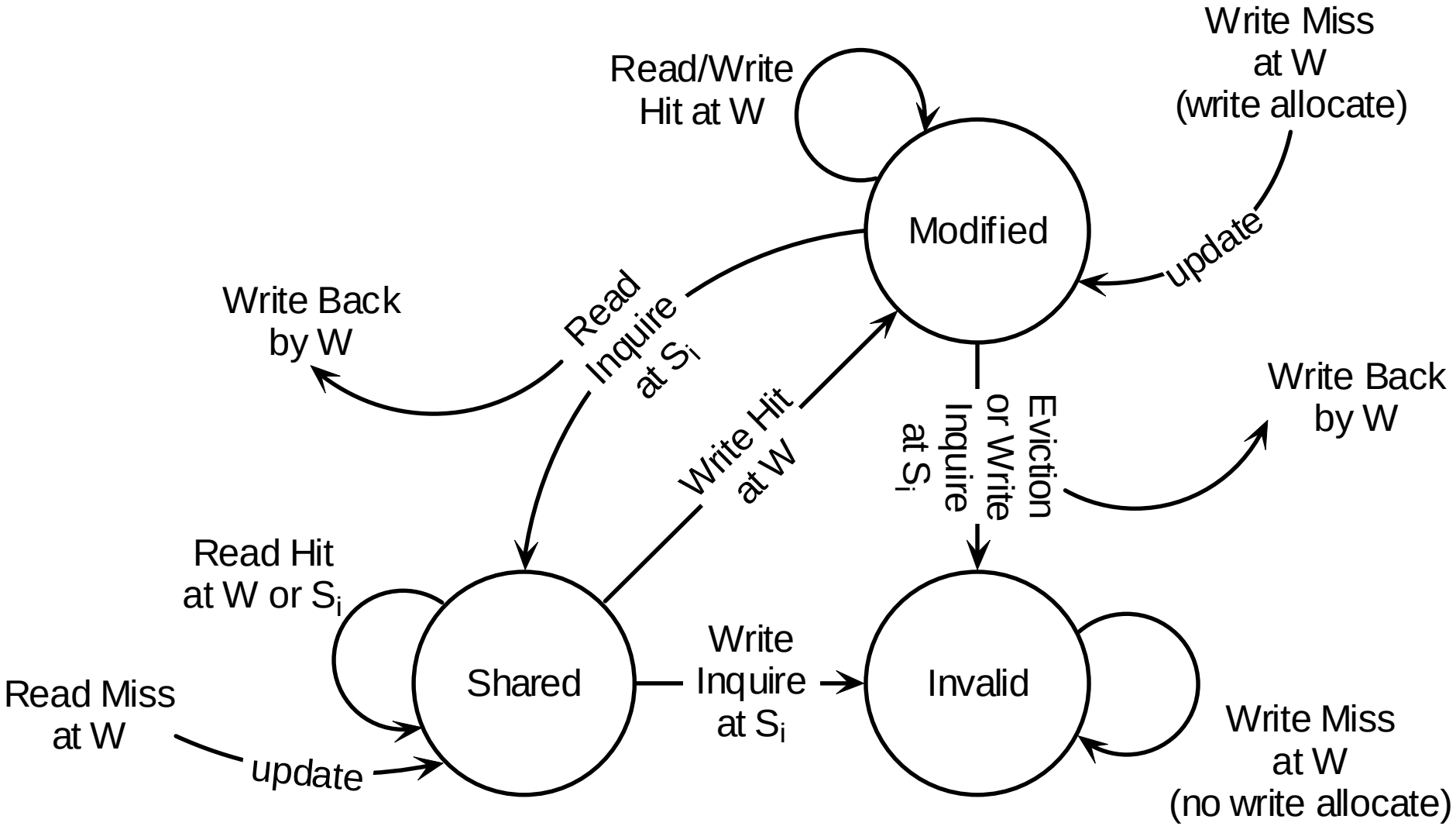
- Writes update address to memory bus
  - Processors $S_i$ mark block invalid
- Updates cache
  - W marks block Modified

# Processor Behavior — Invalid

## Invalid

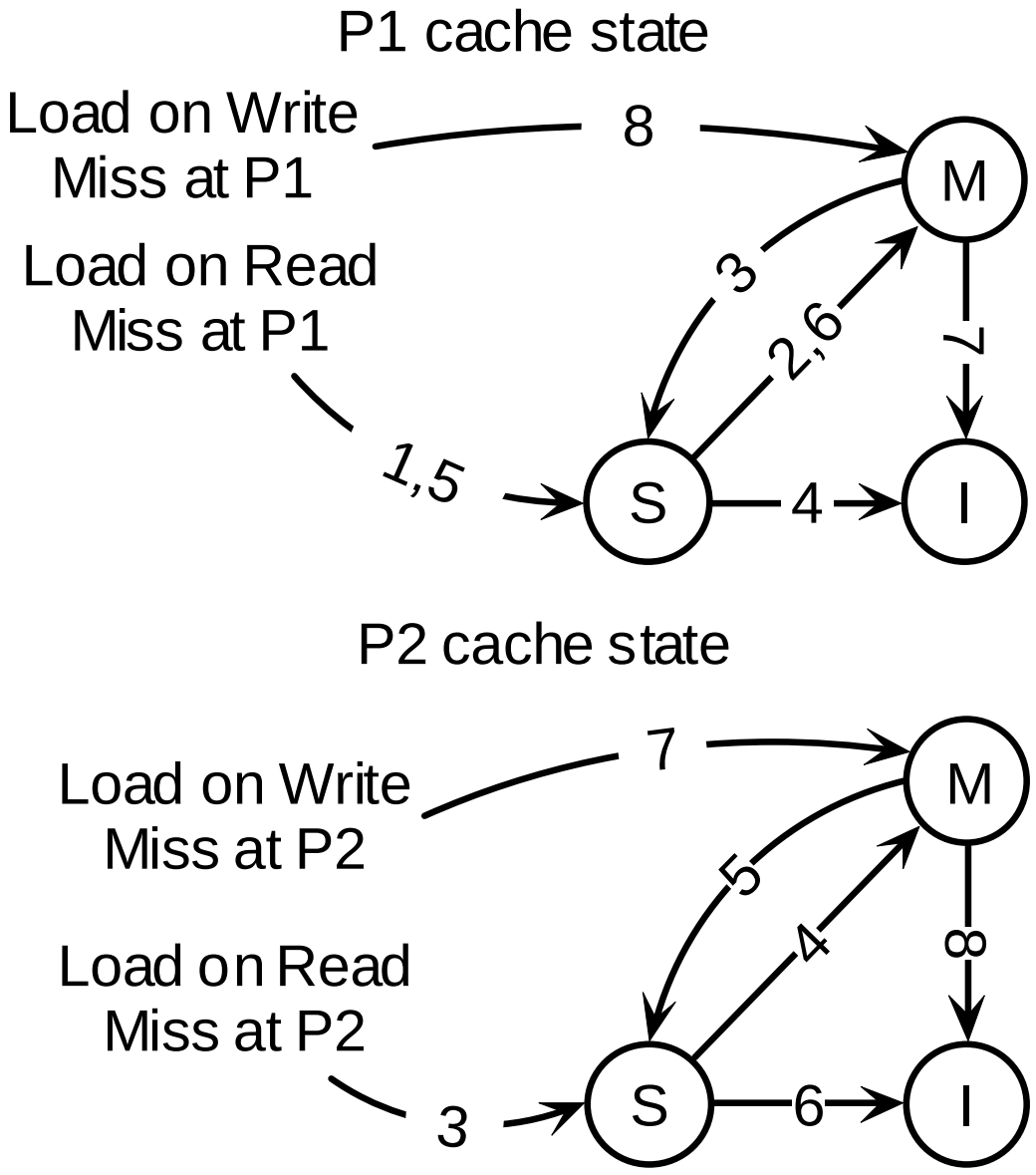
- Block must be reloaded before next read
  - State includes blocks tagged invalid and blocks not in cache
- Read and write accesses are cache misses
- Write allocate
  - Write miss at W initiates cache update
- No write allocate
  - Write miss at W does not initiate cache update
  - Write directly to memory

# MSI Protocol State Diagram



# MSI Example

| | P1 | P2 |
|---|---|---|
| 1 | Read | |
| 2 | Write | |
| 3 | | Read |
| 4 | | Write |
| 5 | Read | |
| 6 | Write | |
| 7 | | Write |
| 8 | Write | |



# Processor Behavior — Exclusive

## Processor W cache holds only valid copy of data line

- Line is clean
  - Memory copy is same as W copy
- Cache copies in other processors $S_i$ marked invalid
  - $S_i$ inquire (request update from this copy) before accessing block

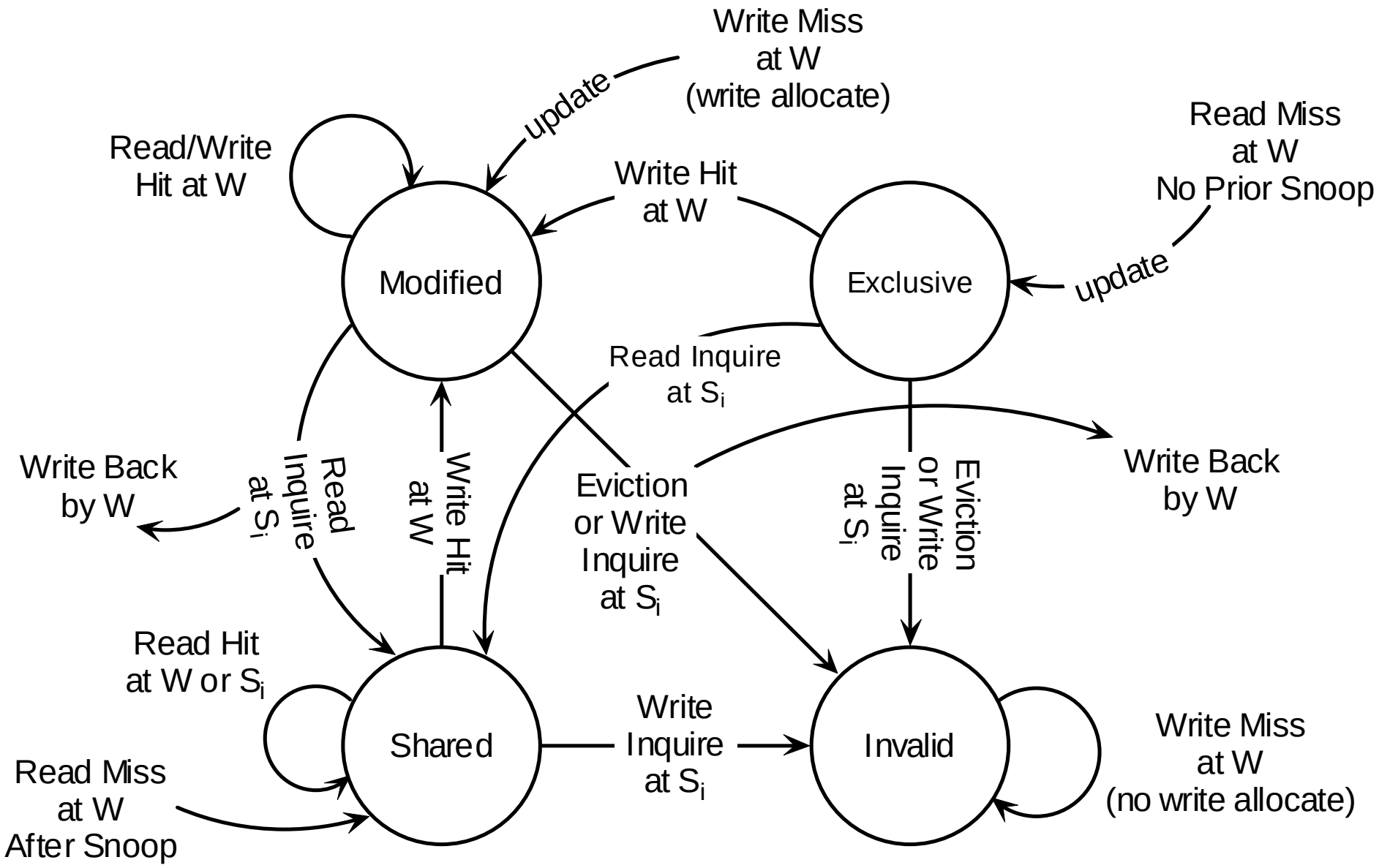
## W responds to inquire

- Snoops memory address from $S_i$ on bus
- W updates memory
- Marks block invalid on write inquire

## Improved performance on private data

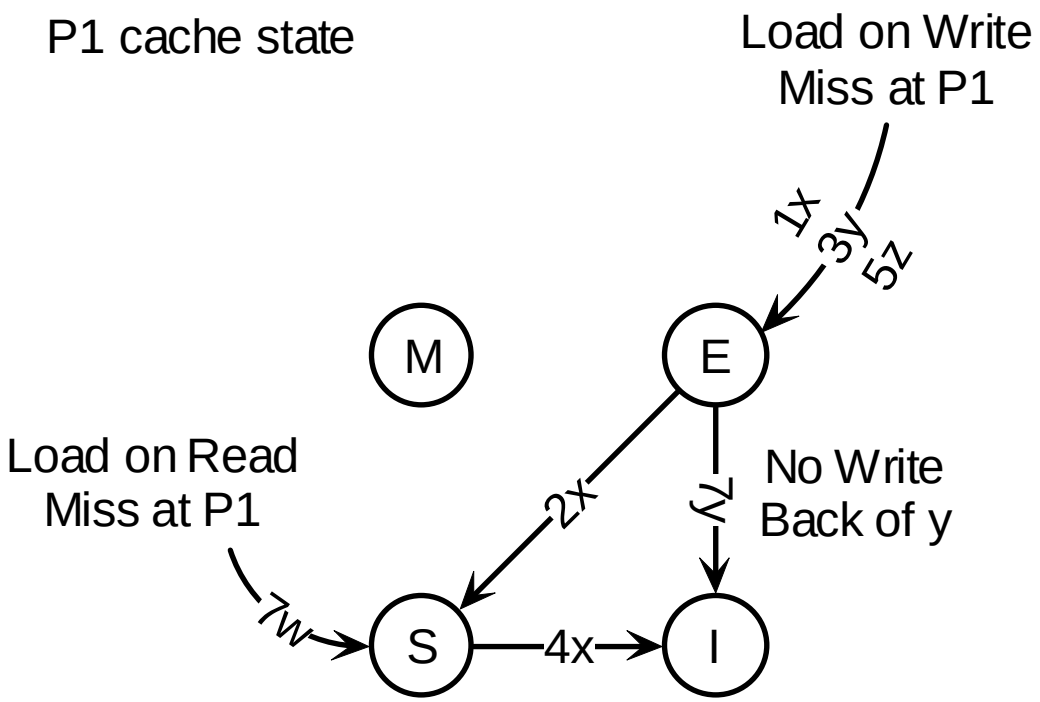
- No write back to memory on eviction of exclusive block

# MESI Protocol State Diagram

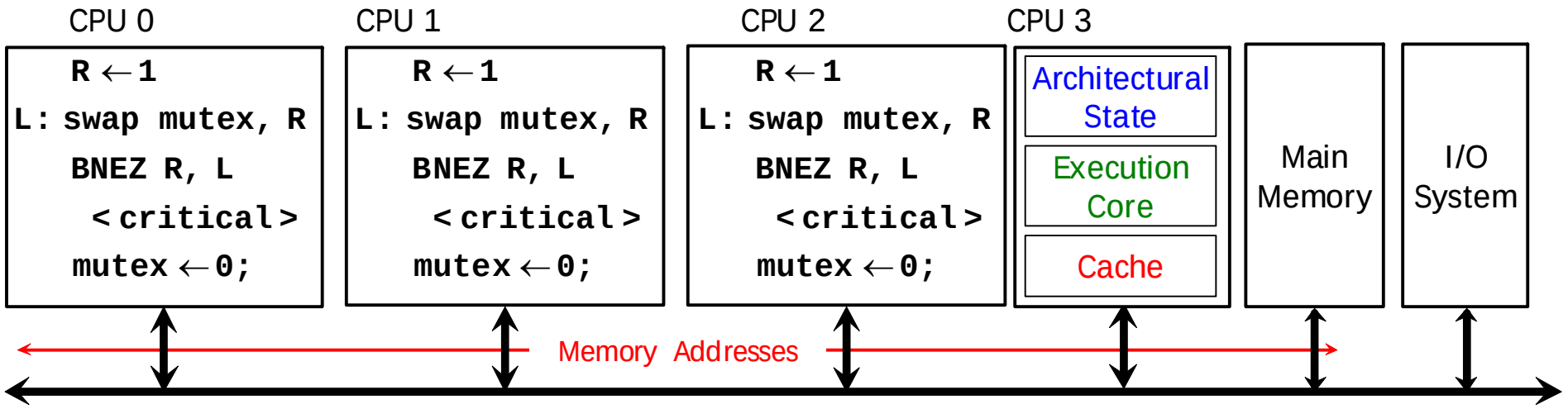


# MESI Example

| | P1 | P2 |
|---|---|---|
| 1 | Read x | |
| 2 | | Read x |
| 3 | Read y | |
| 4 | | Write x |
| 5 | Read z<br>Replaces x in cache | |
| 6 | | Read w |
| 7 | Evict y without write back<br>Read w | |
| 8 | | Evict x with write back<br>Read y |



# Memory Thrashing with MSI




## Value of shared variable mutex

- Required to enforce criticality
- Each swap causes
  - Read miss and cache update
  - Write to one local cache (write allocate)
  - Invalidation all other caches (write invalidate)

## Inefficient movement of data across bus

- Multiple mutex cache misses/loads creates large overhead
- Overhead much larger than read/writes for critical section

# Load-Reserve and Store-Conditional with MSI

## Implement critical section

Use load-reserve/store-conditional

No shared mutex variable

```
   R ← 1
L: swap mutex, R;
   BNEZ R, L;
     <critical(x)>
   mutex ← 0;
```

```
L: load-reserve R, x
   <critical>
   store-conditional x, R
   BEQZ status, L
```

## Each processor has private reservation flag

Flag set on load-reserve

Flag checked on store-conditional attempt

Snooping cache manager clears flag on write to memory location

Requires restart of load and critical section

## Improved overall efficiency

Overhead created by mutex → multiple reads of critical variable **x**

Load-reserve/store-conditional eliminates stores to mutex variables

# False Sharing

## False sharing

Multiple threads access distinct address ranges in same cache block

Write to block causes invalidation in other processors

Creates memory thrashing

## Example

```
int counter[THREAD_NUM];
int inc_counter ()
{
counter[my_tid]++;
return counter[my_tid];
}
```

## Solution

Avoid small array accessed by multiple threads

Avoid global data or static data variables in same cache block

# Cache Coherence and Hardware Optimizations

**Memory hardware optimizations**

- Equivalent to memory semantics within processor

**Cache coherence**

- Enforces sequential consistency between processors

**Write buffer with read-bypass**

- Cache/memory updates follow cache invalidation
  - Load cannot bypass store on two processors

**Write overlapping**

- Writes invalidate other cache copies
- Physical writes occur in correct order

**Read / write reordering**

- Writes invalidate other cache copies
- Rescheduled read invalidated when other CPU writes to variable

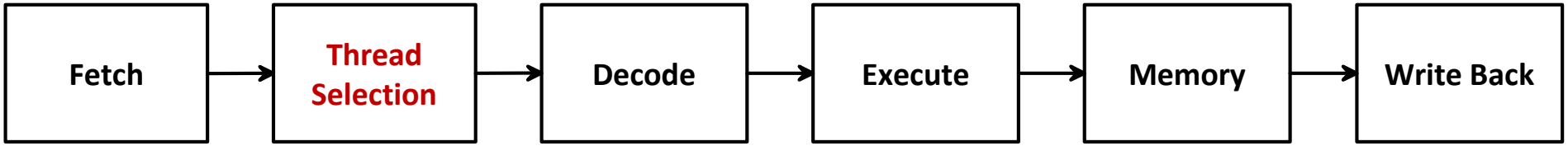
# Sun T1 Multiprocessor

## Server processor

- Introduced by Sun Microsystems in 2005
- Based on UltraSPARC architecture
- Optimized for TLP for online transaction processing (OLTP)

## Core

- 8 processor cores
  - Single-issue — not superscalar
  - In-order execution — no dynamic scheduling
  - 6 stage pipeline — 5 standard stages plus thread selection




## Threads

- 4 threads per core
  - Fine grained multithreading
  - Switches thread on each clock cycle
  - Skips over stalled threads

# Sun T1 Organization

## 8 processor cores

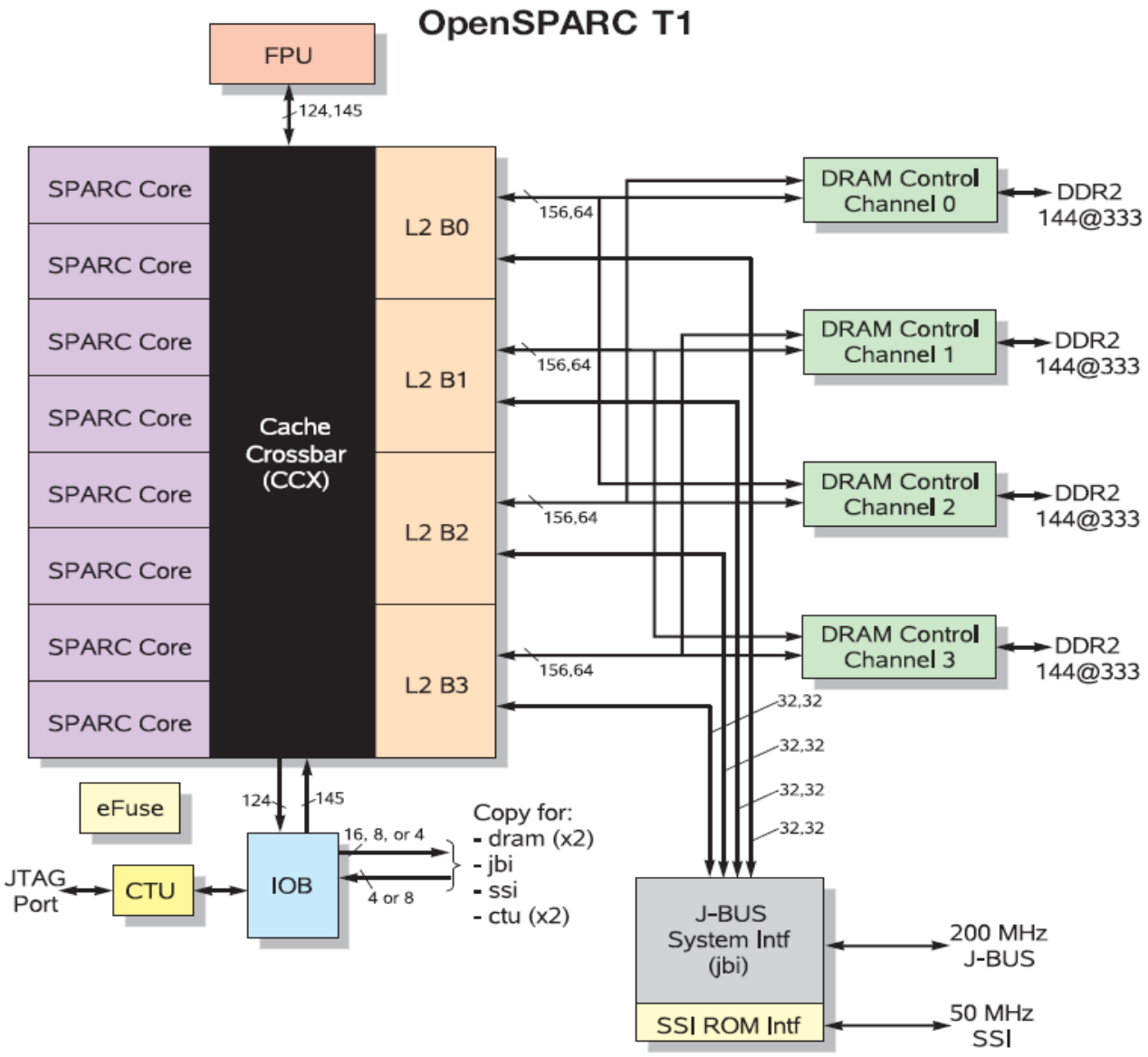
- 1 shared FPU
- Fine grain multithreading
  - 4 threads per core
- 1.4 GHz clock

## Private L1 cache

- 16 KB instruction
- 8 KB data
- 64-byte block
- Every L1 block in L2
- L1 miss penalty = 23 CC

## Shared L2 cache

- 3 MB in 4 banks
- L2 miss penalty = 110 CC
- Distributed directory in L2
  - Maintains coherency among L1 caches



# Oracle (Sun) T4 Multiprocessor

## T1 (2005) → T2 (2007) → T3 (2010) → T4 (2011)

| | T1 | T4 |
|---|---|---|
| **Threads / core** | 4 | 8 |
| **Threads / chip** | 32 | 64 |
| **Pipeline model** | In-order | Dynamic scheduling |
| **Pipeline stages** | 6 | 16 |
| **Shared EUs** | 1 FPU | 1 FPU + 1 Cryptographic |
| **Clock rate** | 1.4 GHz | 3.0 GHz |
| **L1 cache** | 8 × 16 KB I + 8 × 8 KB D | 8 × 16 KB I + 8 × 16 KB D |
| **L2 cache** | 3 MB | 8 × 128 KB |
| **L3 cache** | — | 4 MB |

## Oracle T4 server

4 SPARC T4 CPUs (256 threads) + 1 TB RAM

25% faster than HP 4 Xeon server on OLTP benchmark TPC-H

http://www.oracle.com/us/solutions/performance-scalability/sparc-t4-4-tpc-113011-bmark-1385055.html

# Message Passing Architecture

# Message Passing Multiprocessors

## Collection of N nodes

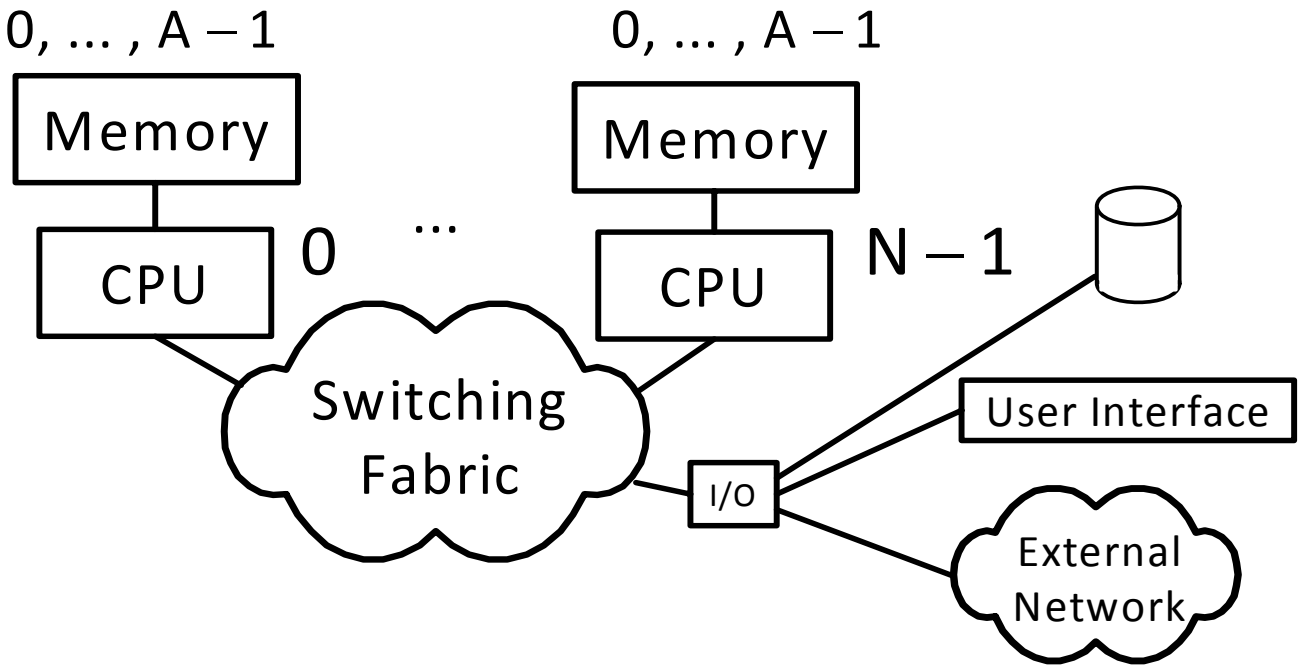
Node = CPU with cache and private memory space

Node i has address space 0, ... , $A_i - 1$

No shared memory locations

## Processes communicate by exchange of structured messages

Switching fabric — network



# Message Passing Systems

## Interprocessor communication

- Send / receive transactions
- Message Passing Interface (MPI)
  - API for message passing parallel programming
  - Transaction primitives between processor and interprocessor network
  - Implementation depends on OS and hardware

## Interprocessor network performance

- General computer network
- Communication protocol and message types
- Routing, network access, segmentation/reassembly (SAR)
- Data requests and transfers

## Scalable to large systems

- Depends on network capacity
- No geographical restriction
- Grows naturally into cluster computing

# Message Passing Issues

## Basic message types

- Send data
- Send request (for data)
- Respond to request (data or status)
- Send system command

## Structured messages

- Header
  - Source ID + destination ID + timestamp
- Content
  - Data / request / response
- Message management ⇒ overhead

## Sequential consistency

- No shared memory ⇒ no snooping ⇒ no snooping overhead
- Code expects specific content from specific source
- Message header + content ⇒ consistent write order

# Message Passing Example — Scalar Product

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

| 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 P1 data from P2 data by source ID
- No data hazard

# Blocking versus Non-Blocking Models

## Program must pair

Send instruction in one thread

Receive instruction in another thread

RTS = Request to Send
CTS = Clear to Send

## Synchronous (blocking) model

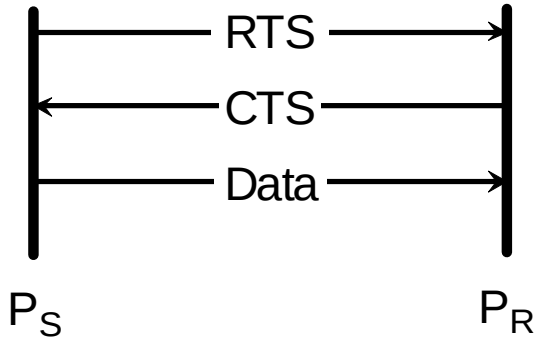
Threads synchronize on send/receive pair

Send and receive instructions block in code

Receive waits for RTS

Send waits for CTS

Creates synchronization barrier at transfer




## Asynchronous (non-blocking) model

Sender execute send instruction and continues

- Message buffered at sender until sent through network

Message buffered at receiver until needed

- Receiver accepts message when executing receive instruction

# Some MPI Environment Messages

| | |
|---|---|
| **`MPI_Init (&argc,&argv)`** | Initialize MPI execution environment<br>Broadcast command line arguments to all processes |
| **`MPI_COMM_WORLD`** | Environment variable<br>Lists MPI-aware processes |
| **`MPI_Comm_size (comm,&size)`** | Returns number of processes in group |
| **`MPI_Comm_rank (comm,&rank)`** | Returns process number of calling process |
| **`MPI_Finalize ()`** | Terminates MPI execution environment<br>Last MPI routine in every MPI program |

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

# Some MPI Point-to-Point Messages

| |
|---|
| **`MPI_Send(buffer,count,data_type,destination_task,id_tag,comm)`** |
| Blocking send<br>id_tag — defined by user to distinguish specific message<br>comm — identifies group of related tasks (usually **`MPI_COMM_WORLD`**) |
| **`MPI_Recv(buffer,count,data_type,source,id_tag,comm,status)`** |
| Blocking receive<br>status — collection of error flags |
| **`MPI_Isend(buffer,count,data_type,destination_task,id_tag,comm,`**<br>**`&request)`** |
| Non-blocking send<br>request — system returns request number for subsequent synchronization |
| **`MPI_Irecv(buffer,count,data_type,source_task,id_tag,comm,&request)`** |
| Non-blocking receive |
| **`MPI_Wait (&request,&status)`** |
| Blocks until a specified non-blocking send or receive operation has completed |

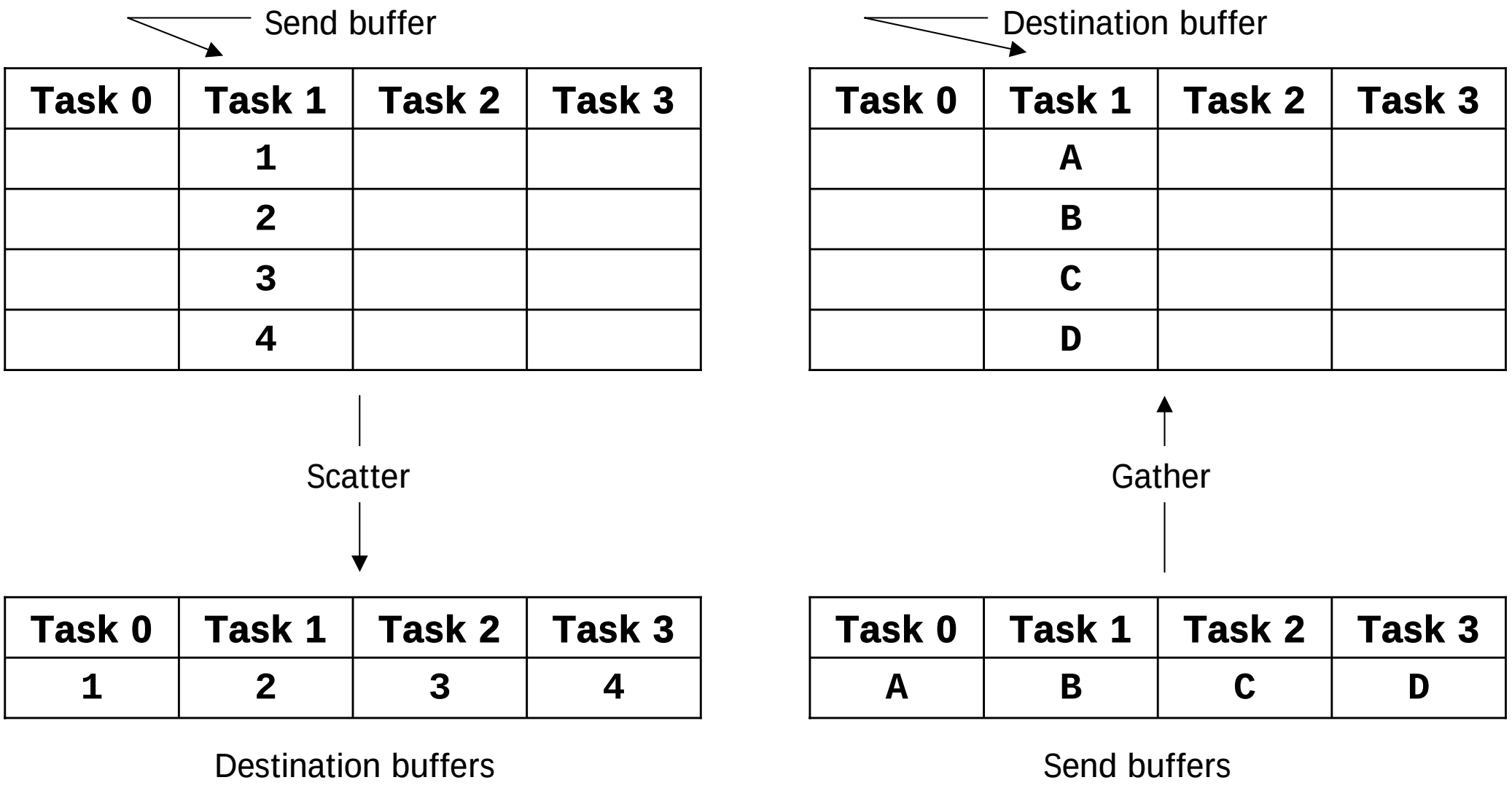
Ref: https://computing.llnl.gov/tutorials/mpi/

# Some Collective Communication Messages

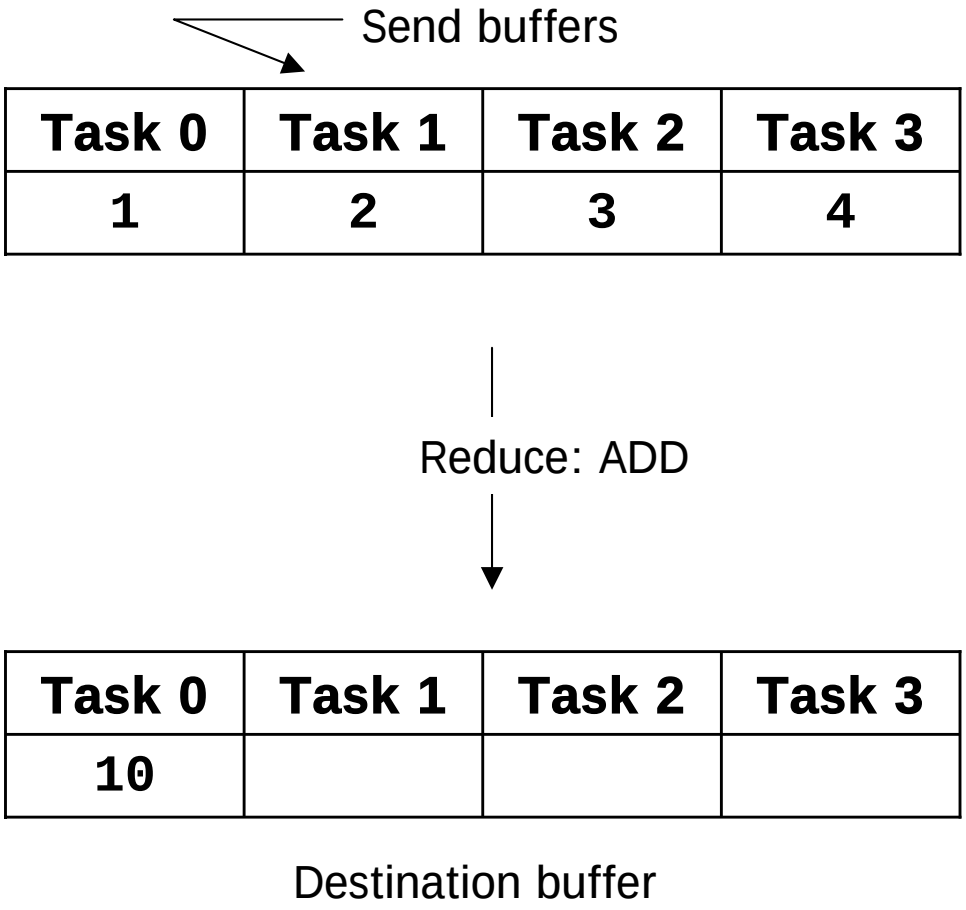
| |
|---|
| **`MPI_Barrier(comm)`** |
| Each task reaching MPI_Barrier call blocks until all tasks in group reach same MPI_Barrier |
| **`MPI_Bcast(buffer,count,data_type,root,comm)`** |
| Sends broadcast message from process root to all other processes in group |
| **`MPI_Scatter(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,recvtype, root,comm)`** |
| Distributes distinct messages from a single root task to each task in group |
| **`MPI_Gather(sendbuf,sendcnt,sendtype,recvbuf,recvcount,recvtype, root,comm)`** |
| Gathers distinct messages from each task in group to a single destination task |
| **`MPI_Reduce(sendbuf,recvbuf,count,data_type,operation,root,comm)`** |
| Applies a reduction operation on all tasks in group and places result in one root task |

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

# 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"

# Send/Receive Instructions

## Send instruction

| | |
|---|---|
| `MPI_Send` | Instruction name |
| `message` | Pointer to message buffer |
| `strlen(message)+1` | Length of message |
| `MPI_CHAR` | Data type in message |
| `1` | Destination process |
| `99` | Message identifier (tags specific message from source) |
| `MPI_COMM_WORLD` | List of running MPI-aware processes |

## Receive instruction

| | |
|---|---|
| `MPI_Recv` | Instruction name |
| `message` | Pointer to message buffer |
| `20` | Maximum length of message |
| `MPI_CHAR` | Data type in message |
| `0` | Source process |
| `99` | Message identifier (filters on tag) |
| `MPI_COMM_WORLD` | List of running MPI-aware processes |
| `&status` | Error flags |

# Scalar Product with MPI Constructs

Compute $\sum_{i=0}^{3} \mathbf{a}[\mathbf{i}] * \mathbf{b}[\mathbf{i}]$ on 4 nodes

```
/* scatter data from root node 0 */
/* each node receives 1 component of a and one of b */
MPI_Scatter(a,1,MPI_INT,a,1,MPI_INT,0,MPI_COMM_WORLD)
MPI_Scatter(b,1,MPI_INT,b,1,MPI_INT,0,MPI_COMM_WORLD)

/* each node calculates */
p = a * b

/* OS for root node adds 1 integer from each node and places sum
     in root 0 */
MPI_REDUCE(p,p,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
```

# Implementation Issues for Processor

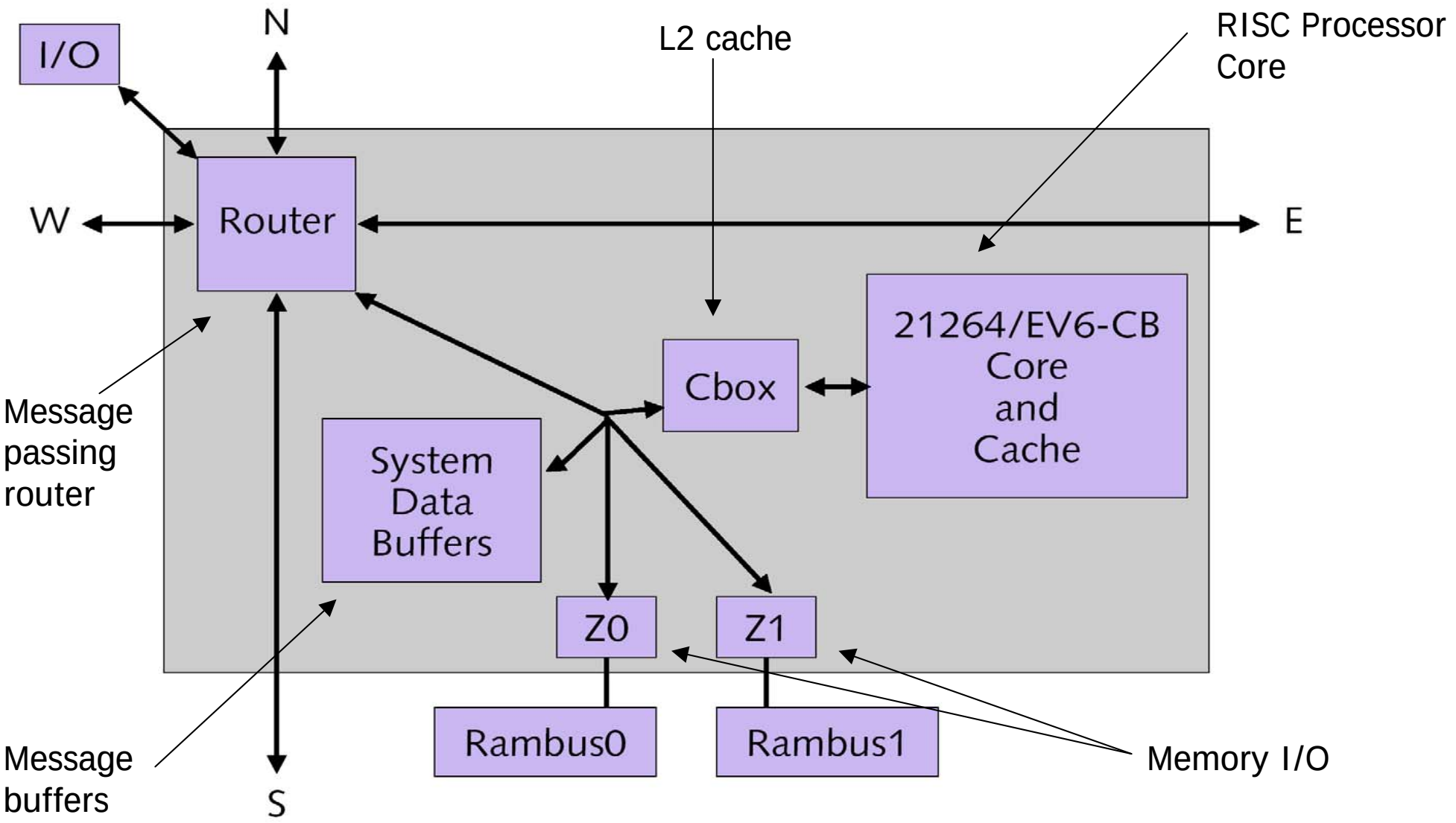
## Processor directly supporting message passing

- Synchronous model
  - Message send/receive port
  - Hardware blocking on port access
- Asynchronous model
  - Message send/receive port or equivalent
  - Internal send/receive buffer for messages

## Interconnect support for message passing

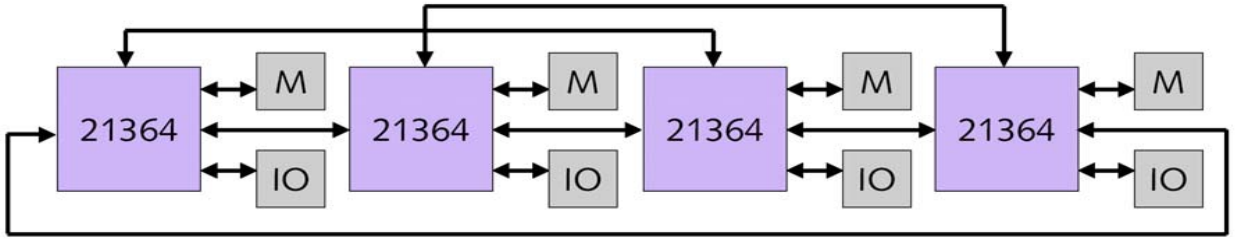
- No direct support in processor
- Interconnect buffers messages for processor
- Synchronous model
  - Processor blocks on send receive instruction
  - Interconnect sends/receives messages
- Asynchronous model
  - Interconnect buffers and sends messages for processor
  - Interconnect buffers received messages and interrupts processor

# Message Passing Support in Alpha 21364

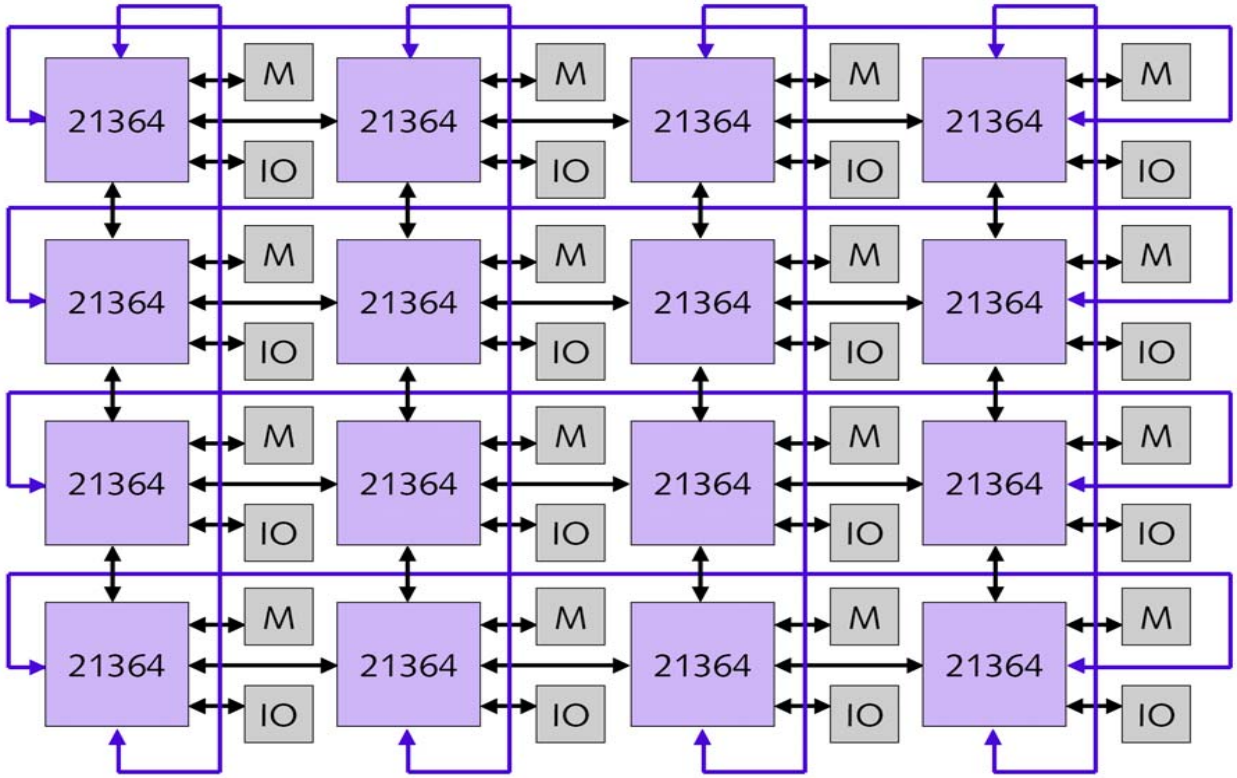



Ref: Kevin Krewell, "Alpha EV7 Processor", Microprocessor Report, 2002

# Message Passing Multiprocessor Configurations




Four-Processor Configuration




Sixteen-Processor 2D Torus Configuration

# Cluster Computing

## Large message passing distributed memory system

Exploits MPI scalability

Up to millions of nodes

## Typical node

Standard workstation

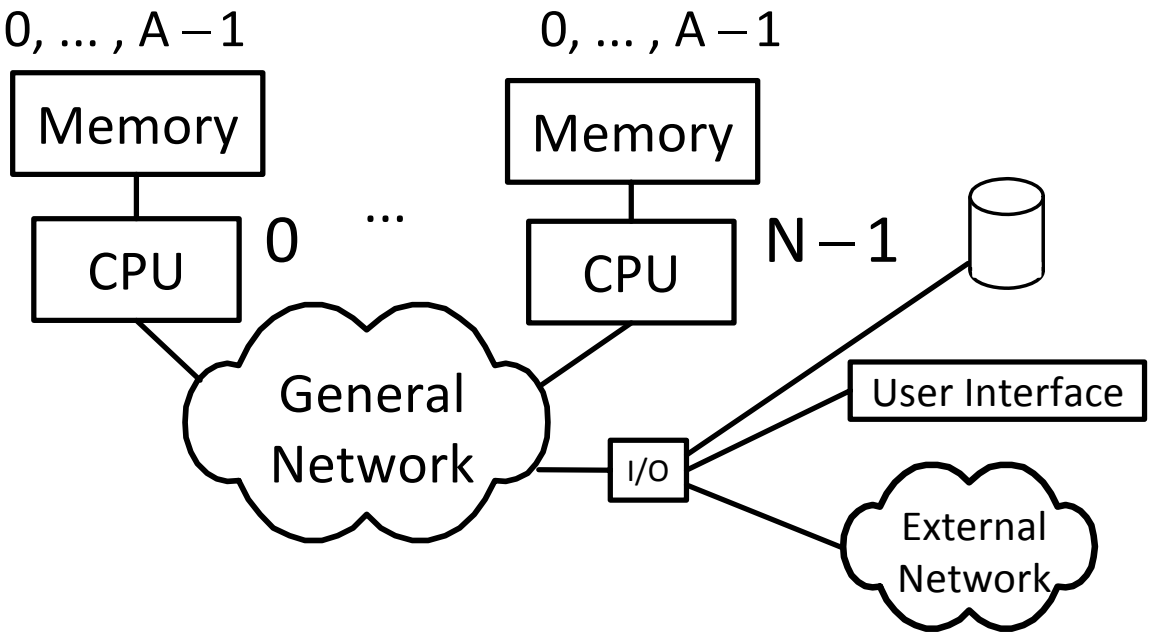
## Node-to-node scale

Physical bus

Crossbar switch

LAN

WAN




## Interconnection network

LAN / WAN

**`MPI_COMM_WORLD`** includes network addresses

# Benefits of Clusters

## Scalability in parallel computing

- Message passing parallel algorithms scale to large number of nodes
  - Typical communication / computation ratio low to medium
  - Less sensitive to latency than shared memory
- Very high scalability for thread-replicating servers
  - Web servers
  - Database access servers
  - On-line transaction processing (OLTP)

## Availability

- Nodes provide backup to each other
- On device failure load transferred to another node within cluster
- Transparent to user

## Manageability

- Single system image $\Rightarrow$ simple system restoration
- Automatic load balancing

# MPI and Socket API

## Socket API maps endpoints to socket numbers

- Socket number identifies connection to remote endpoint
  - Endpoint = IP address:port
- Client — socket identifies connection to server endpoint
- Server — socket identifies connection from client endpoint

## Systems with MPI and sockets

- MPI cluster implemented as client/server system
  - Primary node (client) opens connection to each remote server
  - Secondary nodes run MPI server listening on service port number
- MPI maps socket numbers to **MPI rank construct**
  - Socket number provides ID number for node in MPI_COMM_WORLD

## Node performs MPI send to remote node

- Writes to socket for remote node

## Node performs MPI receive

- Reads from socket for remote node

# Network Latency in Cluster Computing

## Amdahl's law with communication overhead

$$S = \frac{1}{(1 - F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)} \quad , \quad F_P = \frac{\text{Parallel instructions}}{\text{All instructions}}$$

$$F_{overhead} = \frac{CPI^{comm}}{CPI} = \frac{\text{Communication activity CC per instruction}}{\text{Computation activity CC per instruction}}$$

Cluster computing works well for $F_P$ and $N$ large, and $F_{overhead}$ small

### Communications overhead

Networking software

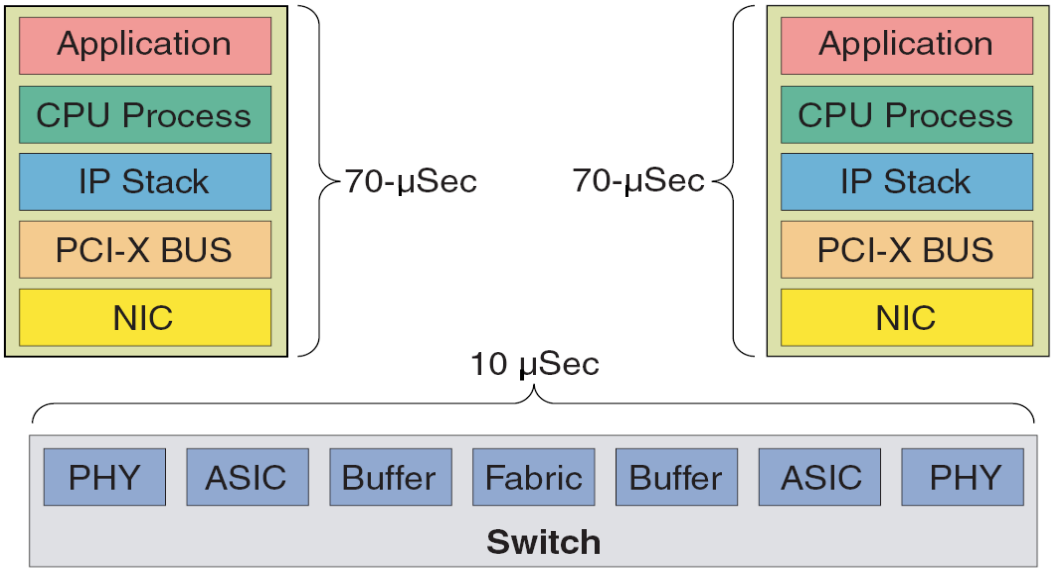
Network hardware

CPU CC ≈ $10^{-9}$ sec

Message latency ≈ $150 \times 10^{-6}$ sec

≈ $1.5 \times 10^{5}$ CC

Comparable to page fault




Ref: Cisco Systems, "Cluster Computing"

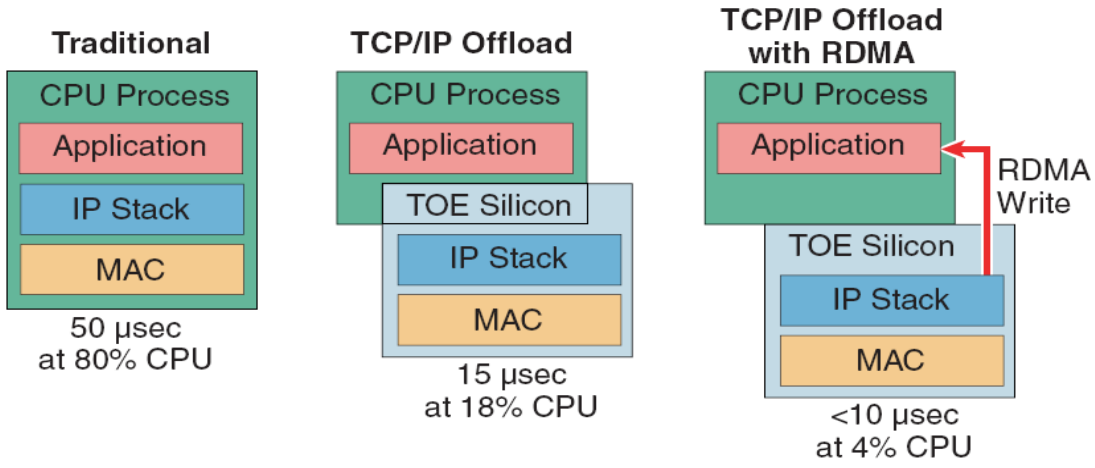
# Hardware Support for Cluster Processing

## TCP offload engine (TOE)

- Dedicated communication processor
- Processes TCP software for CPU(s) in node
  - Segmentation/reassembly (SAR)
  - Error checking
- Feature of mainframes since 1960s
- Returning as asymmetric multicore processors

## Remote Direct Memory Access (RDMA)

- Performs DMA functions between nodes




Ref: Cisco Systems, "Cluster Computing"

# Typical Cluster Configuration

## Node

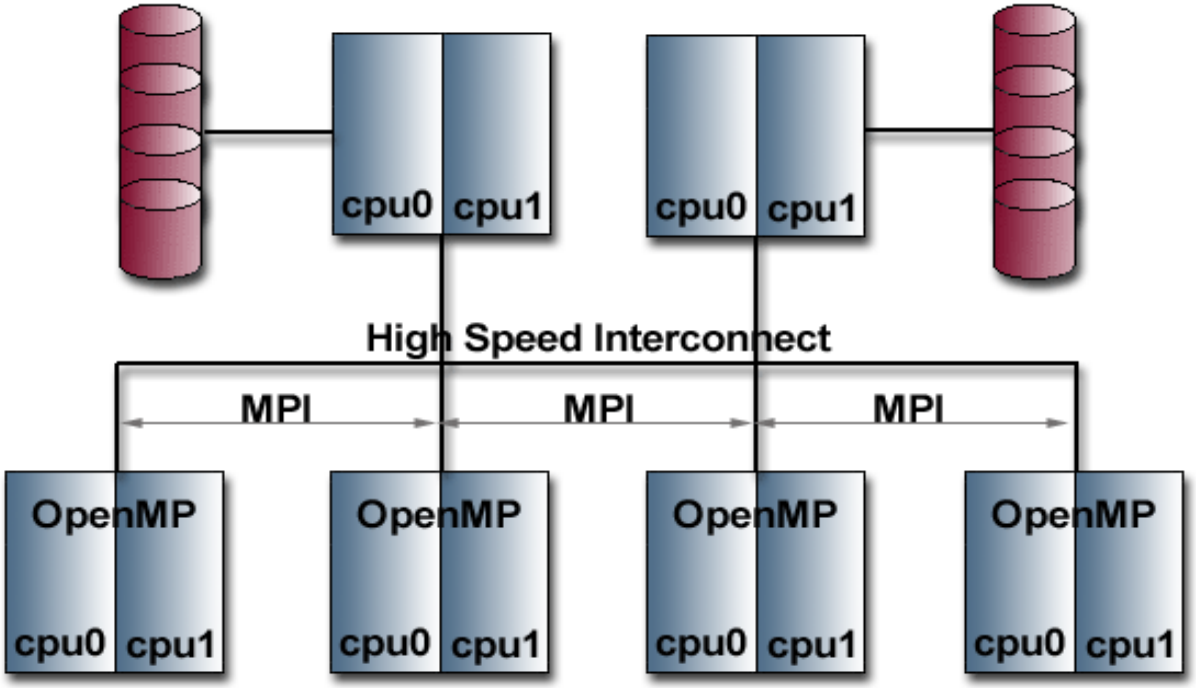
Shared memory SMP running tightly coupled code over OpenMP

## High speed interconnect

TCP/IP over Gigabit Ethernet (1 Gbps serial transmission)

## Cluster of SMPs (CluMP)

Nodes exchange data for loosely coupled code over MPI




https://computing.llnl.gov/tutorials/linux_clusters/

# Thunder Cluster

## CluMP type cluster computer

1024 Linux nodes

Each node is 4-CPU Itanium 2 "Madison Tiger4" SMP

## Built at Lawrence Livermore National Laboratory (LLNL)

US Department of Energy National Nuclear Security Administration

Second fastest supercomputer in June 2004 (http://www.top500.org)

Peak performance: 23 TFLOPS



https://computing.llnl.gov/tutorials/linux_clusters/thunder.html

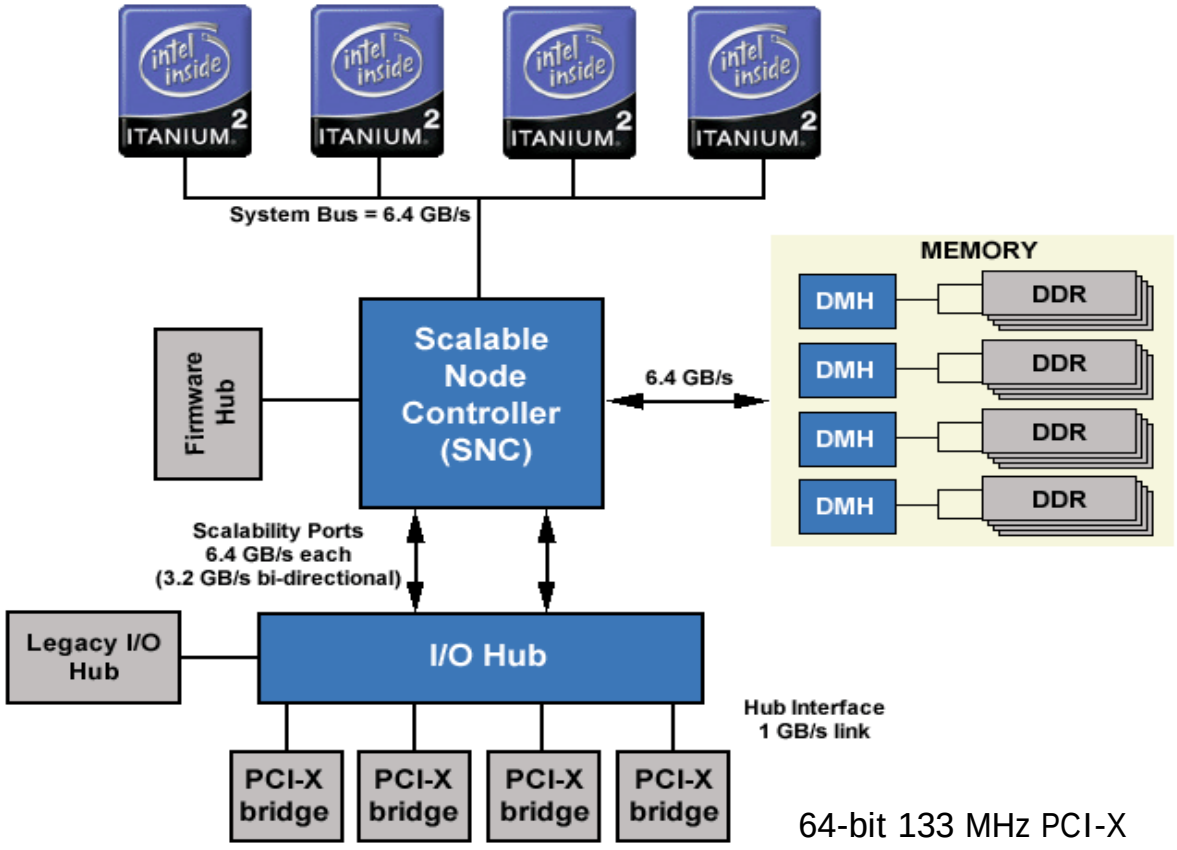
# Node Technical Details

## 1024 nodes

- 4-CPU 1.4 GHz Itanium 2 "Madison Tiger4"
  - VLIW explicitly parallel RISC language machine
- 8 GB DDR per node (8 TB total)
- 16KB L1 Data Cache
- 16KB L1 Instruction Cache
- 256 KB L2 Cache
- 4 MB L3 Cache

## Intel E8870 chipset

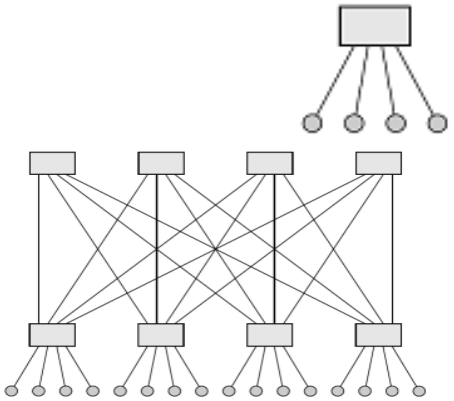
- 6.4 GB/sec switch



# Interconnect Details

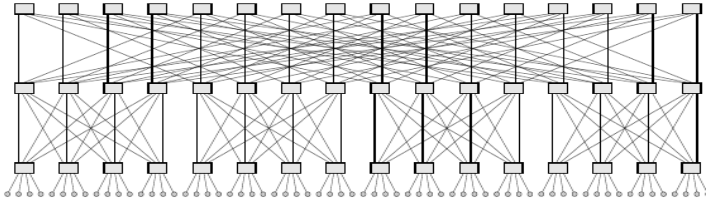
## Quadrics QsNetII

- PCI-to-PCI switch
- 900 MB/s of user space to user space bandwidth
- Optical interconnect extends maximum link length to 100m
- Fat tree topology
  - Up to 4096 nodes in 5 switching stages
- Switch built from Elan4 ASIC
  - 8 port custom switch
  - Integrated DMA controller
- Supports collective communication
  - Barrier synch
  - Broadcast
  - Multicast



1 stage
4 node
system

2 stage
16 node
system



3 stage
128 node
system

http://www.quadrics.com

# Blue Gene/L

## Massively parallel supercomputer

65,536 dual-processor nodes

32 TB (32,768 GB) main memory

Based on IBM system-on-a-chip (SOC) technology

Peak performance of 596 teraflops

Gara, et. al., "Overview of the Blue Gene/L system architecture", IBM Technical Journal

## Built at Lawrence Livermore National Laboratory (LLNL)

US Department of Energy National Nuclear Security Administration

2nd fastest supercomputer in June 2008 (1st in 2007)

## Target applications

Large compute-intensive problems

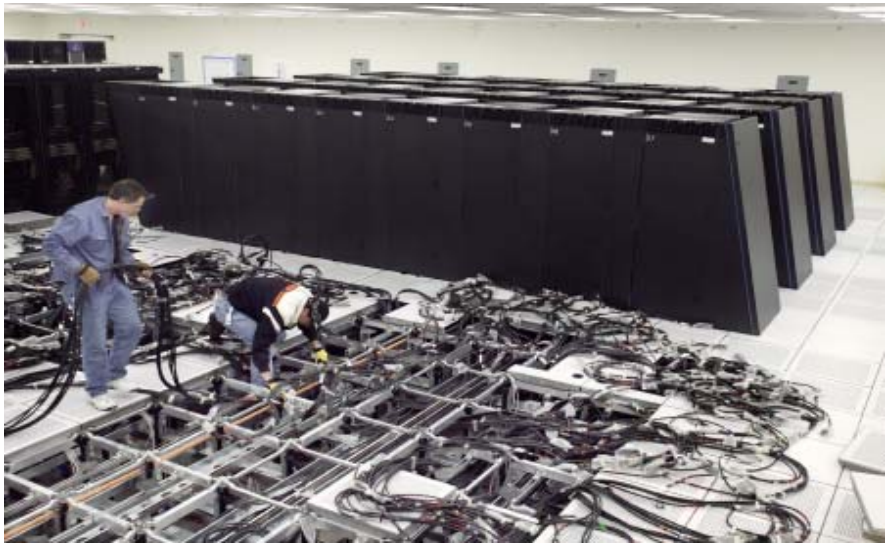
Simulation of physical phenomena

Offline data analysis

## Architectural goals

High performance on target applications

Cost/performance of typical server



# Target Applications

## Simulations of physical phenomena

- Represent physical state variables on lattice
- Time evolution through sequence of iterations
  - Exchange state variables among lattice points
  - Update local state variables according to physical laws
- Short-range physical interactions
  - Exchange state variables within small volume of lattice points
- Spectral methods
  - FFT and others
  - Require long-range communication between lattice point
  - Preferable for good convergence behavior

## Mapping lattice sites onto multiple processors straightforward

- Each node allocated subset of problem
- Spatial decomposition $\Rightarrow$ nodes communicate with small group of nodes

## Global reduction

- Normalization over full data set
- Decision step in algorithm

# Scaling Issues in Very Large Systems

## Cost / performance ratio scales badly at massive scale

- Equipment overhead
  - Equipment racks
  - Floor space
  - Electrical power
  - Cooling
- Communication complexity
  - Fully connected network grows as (number of processors)
  - Raises communication overhead FP

## Computing power versus electrical power

- Performance / rack = (Performance / Watt) × (Watts / rack)
- Watts / rack ~ constant ~ 20 kW / rack
- Rack performance limited by Performance / Watt
- Performance / Watt scales badly with performance

# Architectural Decisions

## High integration and hardware density

- Fewer racks $\Rightarrow$ lower rack overhead
- Higher performance per rack
- Shorter and simpler communication paths

## Requirements

- High performance / Watt
  - Medium performance processors
  - High performance processor $\Rightarrow$ high ILP at high power
  - TLP more significant than ILP in target applications
- Highly integrated system on a chip (SOC)
  - Multiple cores
  - Processor + memory + I/O on single integrated circuit
  - High performance interconnect
  - Application specific integrated circuit (ASIC)

# Implementation

## 65,536 nodes

- 130-nm copper IBM CMOS 8SFG technology
- Single ASIC
  - two processors
  - nine DDR SDRAM chips
- Nodes physically small
  - 2 to 10 times higher density greater than high-frequency uniprocessors

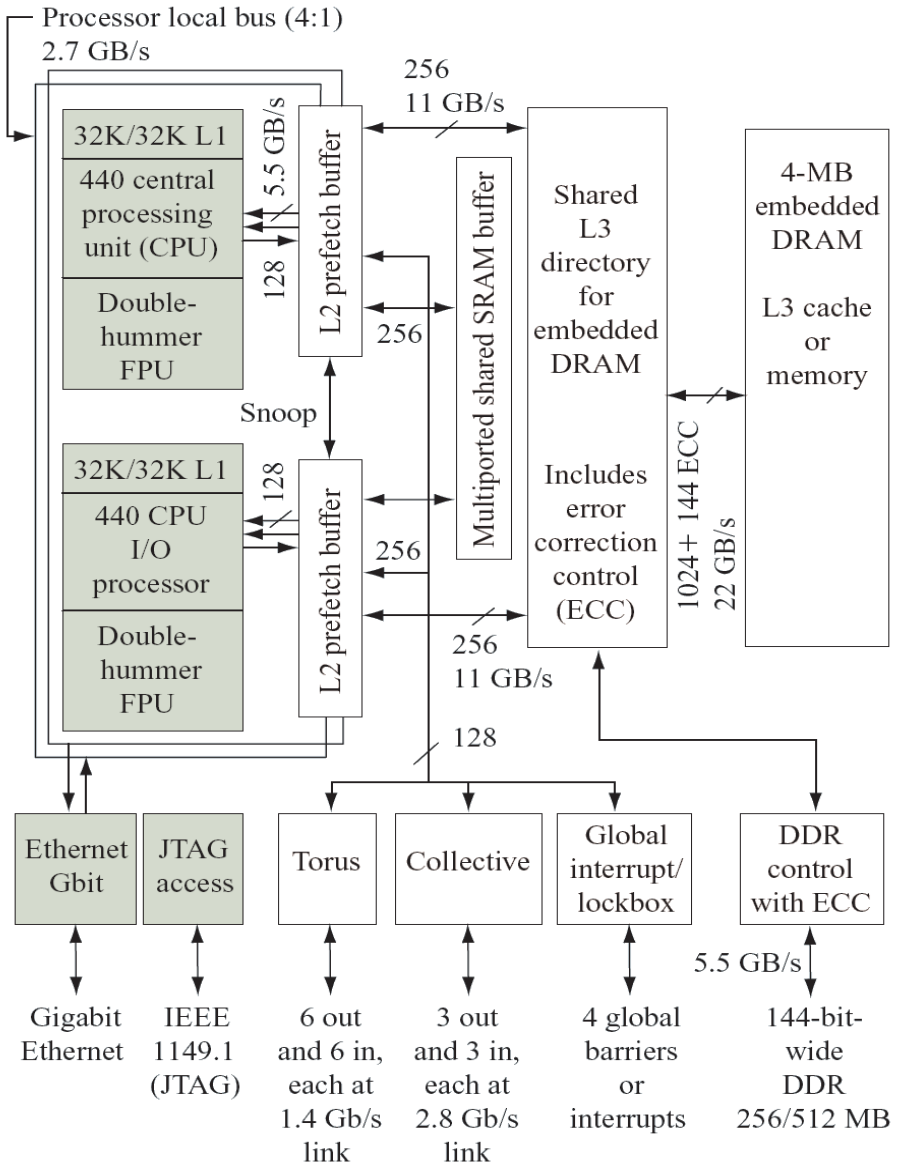
## 1,024 dual-processor nodes per rack

- 27.5 kW of total power
- 85% of inter-node connectivity contained within racks

## Five interconnection networks

- High connectivity static switching fabric for bulk communications
- Collective network supporting scatter/gather/broadcast/reduction
- Barrier network supporting barrier synchronization
- Control system network
- Gigabit Ethernet network supporting I/O

# Node ASIC


Processor local bus (4:1)
2.7 GB/s
256
11 GB/s
32K/32K L1
440 central processing unit (CPU)
Double-hummer FPU
5.5 GB/s
128
L2 prefetch buffer
256
Multiported shared SRAM buffer
Shared L3 directory for embedded DRAM
Includes error correction control (ECC)
4-MB embedded DRAM
L3 cache or memory
1024+ 144 ECC
22 GB/s
Snoop
32K/32K L1
440 CPU I/O processor
Double-hummer FPU
128
L2 prefetch buffer
256
256
11 GB/s
128
Ethernet Gbit
JTAG access
Torus
Collective
Global interrupt/ lockbox
DDR control with ECC
5.5 GB/s
Gigabit Ethernet
IEEE 1149.1 (JTAG)
6 out and 6 in, each at 1.4 Gb/s link
3 out and 3 in, each at 2.8 Gb/s link
4 global barriers or interrupts
144-bit-wide DDR 256/512 MB

# Interconnect Details

## Static interconnection network

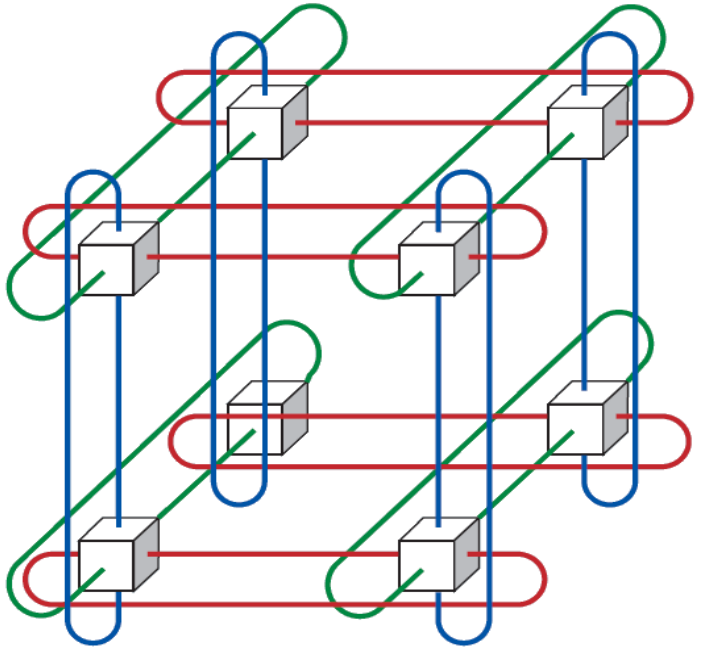
- 64 × 32 × 32 three-dimensional torus for bulk communication
- Nodes communicate with neighboring nodes
- Same bandwidth and nearly same latency to all nodes
  - No edges in torus configuration
- Simplifies programming model
- Nearest-neighbor link rate is 1.4 Gb/s

## Collective network

- Global broadcast/scatter/gather of data
- Improvement over 3D torus network
- 1% to 10% of typical supercomputer latency on collective operations
- Built-in ALU hardware on network
  - Supports reduction operations
  - Min, max, sum, OR, AND, XOR

## Barrier network

- Propagates OR of nodes to trigger interrupts
- Propagates AND of nodes for synchronization



2 × 2 × 2 torus

# IBM Sequoia — Blue Gene/Q

## Massively parallel supercomputer

- 96 K (98,304) 16-core nodes
- 1.6 PB (1.6 × 1024 × 1024 GB) main memory
- Based on IBM POWER system-on-a-chip (SOC)
- Peak performance of 16 petaflops

## Built at Lawrence Livermore National Laboratory (LLNL)

- US Department of Energy National Nuclear Security Administration
- Fastest supercomputer in June 2012

## Operating systems

- Red Hat Enterprise Linux on I/O nodes
  - Connect to file system
- Compute Node Linux (CNL) on application processors
  - Runtime environment based on Linux kernel

## Target applications

- Advanced Simulation and Computing Program
- Simulated testing of US nuclear arsenal
- Nuclear detonations banned since 1992

# Example of Cluster Application

## Parallel multiplication of N × N matrix by vector

Medium coupled computation

$$c[i] = \sum_{j=0}^{N-1} A[i][j] * b[j] = A[i] \text{ dot } b$$

Works well on shared memory SMP

Dominated by communication overhead on cluster

### Manager code

```
setup
initialize matrix a[i][j]
for (i=0 ; i<N ; i++)
 MPI_Send row A[i] to rank_i
for (i=0 ; i<N ; i++)
 MPI_Recv c[i] as row * vector
```

### Worker code

```
setup
initialize vector b[i]
for (i=0 ; i<N ; i++)
 MPI_Recv row A[i]
dotp = 0.0
for (i=0 ; i<N ; i++)
 dotp += c[i] * b[i]
MPI_Send dotp
```

# Speedup?

## Computations

Each row of vector result

- N multiplications
- N – 1 additions
- 2N – 1 flops

$$S = \frac{1}{(1 - F_P) + F_P\left(\frac{1}{N} + F_{overhead}\right)}$$

## Communications

Send N × N matrix ⇒ $N^2$ floats

Return N vector components ⇒ N floats

Total = $N^2$ + N floats sent

Send ($N^2$ + N) / (2N – 1) floats per flop

## Overhead Factor

$$F_{overhead} = \frac{CPI^{comm}}{CPI} = \frac{N^2 + N}{2N - 1} \times \frac{2CC^{communication}}{CC^{processor}} \sim N \text{ on fast interconnect}$$

$$S = \frac{1}{(1 - F_P) + F_P\left(\frac{1}{N} + N\right)} \sim \frac{1}{N} \xrightarrow[F_P \approx 1 \quad N \text{ large}]{} 0$$

# Example of Reasonable Cluster Application

Calculate $\pi = \int_0^1 \frac{4}{1+x^2} dx$ numerically

## Sequential Version

```
N steps
step = 1 / N
for (i=0; i<N; i++)
 {
  x = (i+0.5)*step;
  sum = sum + 4.0/(1.0 +
    x*x);
 }
pi = step * sum;
```

## Loop Computations

```
N * (sum, product, product,
     sum, division, sum) +
     product
~ 6N + 1 flops
```

## Cluster Version

```
N processors
processor i computes
x = (i+0.5)*step;
sum = 4.0/(1.0 + x*x);
MPI_REDUCE (p, N, 1, MPI_INT,
    MPI_SUM,0,MPI_COMM_WORLD)
pi = step * p;
```

## Computations

```
sum, product, product, sum,
    division + reduce_add
~ 6 flops
```

## Communications

```
Send / receive 1 float per 6 flops
```

$$F_{overhead} \sim \frac{1}{6} \times 2 \Rightarrow S \sim 3$$

# Examples of Good Cluster Applications

## Thread-replicating servers

- Web servers
- Database access servers
- On-line transaction processing (OLTP)

## Number crunching on separable data

- Numerical integration
- Ray tracing
- VLSI layout planning

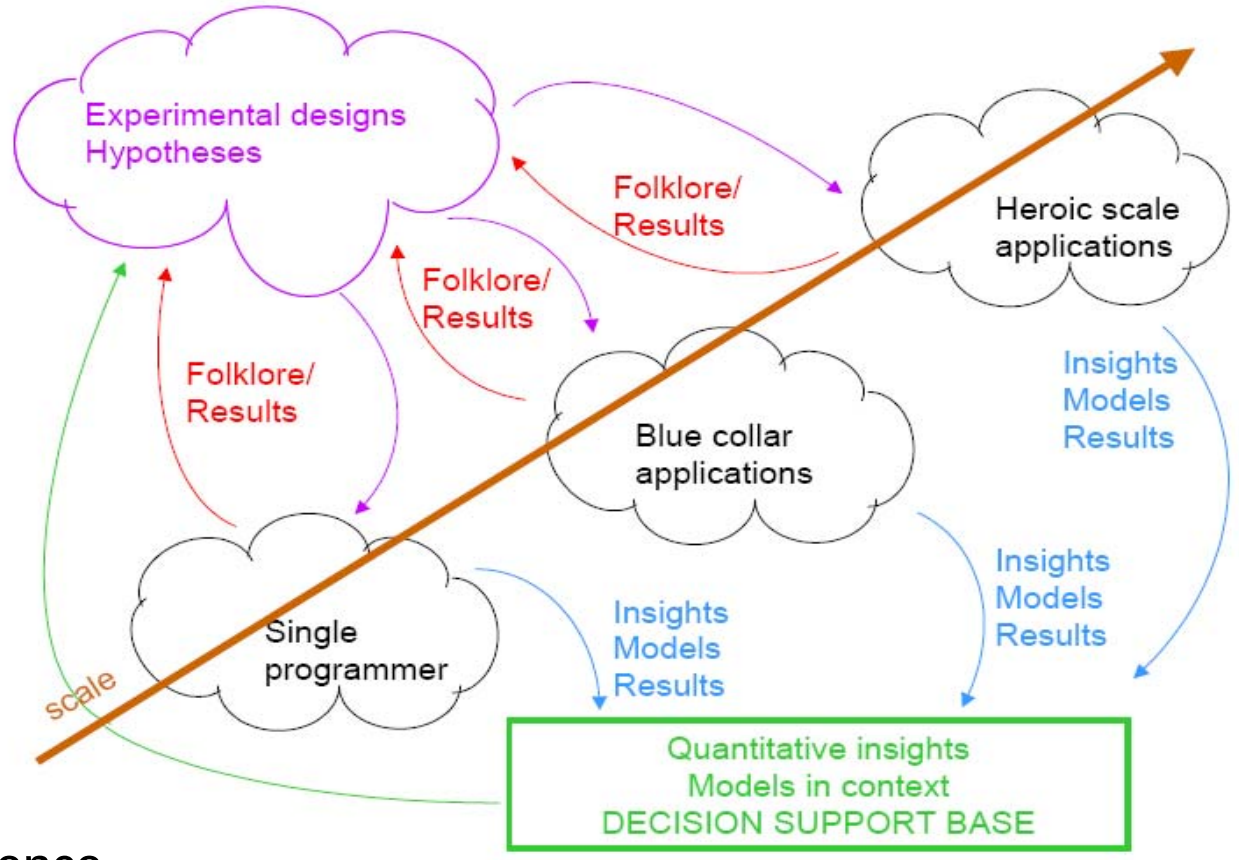
## Numerical solutions for 2nd order partial differential equations

- Electrodynamics and optics
- Quantum mechanics and quantum chemistry
- Thermodynamics, hydrodynamics, aerodynamics
- Plasma magnetohydrodynamics

# Learning Curve in Parallel Programming

## Small scale projects

- Single programmer
- "Embarrassingly" parallel
  - Obvious task/data decomposition
- Follow design rules
  - Vector constructs
  - OpenMP constructs
  - Structure optimization
  - Sequential consistency




## Medium scale projects

- Coordination — folklore or science
- Debugging consistency across team of programmers

## Large scale projects

- High performance computing (HPC) for difficult mathematical problems
- Requires expertise in programming and heuristics of mathematical problem
- Programmers become specialists in specific HPC field

# Empirical Results

## Study of student programmers

Graduate-level courses in parallel programming

Students implement "game of life" program in C on PC cluster

Compare serial with parallel solutions (OpenMP and MPI)

| Programming model | Speedup on 8 processors<br>Parallel version compared to sequential | |
|---|---|---|
| | **Mean** | **Standard Deviation** |
| **MPI** | 4.7 | 1.97 |
| | 2.8 | 1.9 |
| **OpenMP** | 6.7 | 9.1 |
| | Speedup on parallel version with N = 8 compared to parallel version with N = 1 | |
| **MPI** | 5.0 | 2.1 |
| | 4.8 | 2.0 |
| | 5.6 | 2.5 |
| **OpenMP** | 5.7 | 3.0 |

# Virtual Machines

# Virtual Machine (VM)

## Layered model of computation

Software and hardware divided into logical layers

Layer n

- Receives services from server layer n – 1
- Provides services to client layer n + 1

Layers interact through well-defined programming interface

## Virtual layer

Software emulation of hardware or software layer n

Transparent to layer n + 1

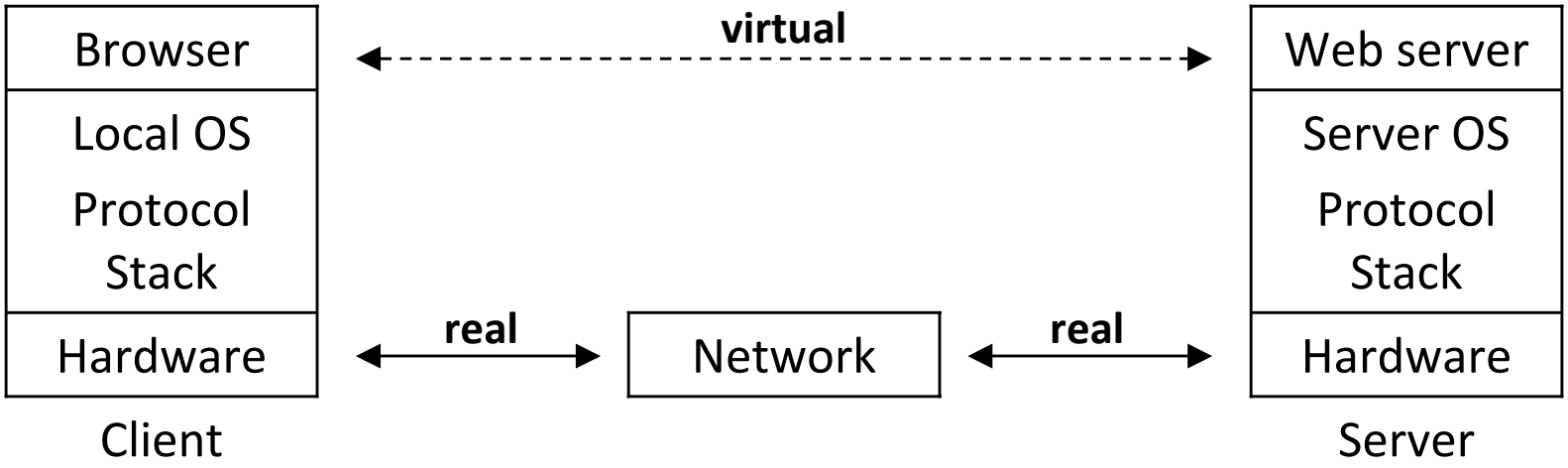
- Provides service to layer n + 1 as expected from real layer n

Virtual layer n can run at some layer m ≠ n in real system

| Virtual System | Real System |
|---|---|
| n + 1 | n + 1 |
| n | virtual n = m |
| n – 1 | m – 1 |

# Examples of Virtual Systems

## Web browser exchanges data with server




## Cloud computing

- Virtual
  - Service level agreement (SLA) specifies infrastructure requirements
  - User sees hardware / software configuration / performance
- Real
  - Provider assembles virtual configuration
  - Meets SLA requirements
  - May be implemented in any way

# Types of Virtual Machine

## Process Virtual Machine

VM provides application interpretation above OS

## Hosted Virtual Machine

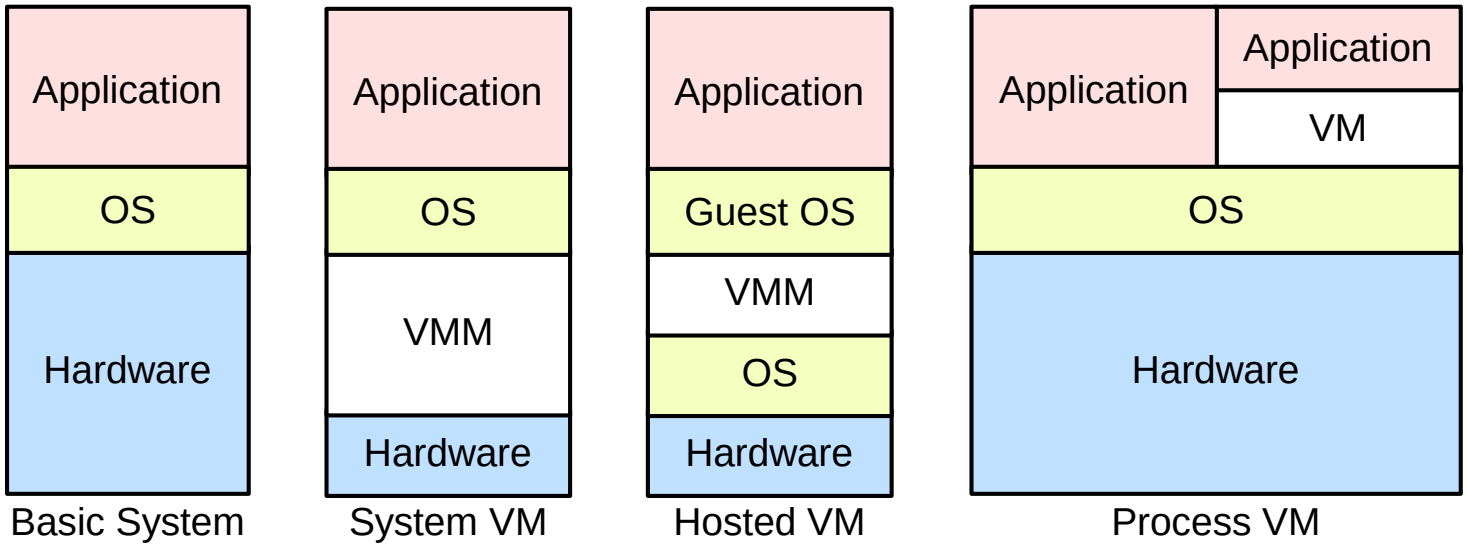
Virtual machine monitor (VMM)

- Runs above primary OS / below guest OS
- Provides guest OS with software emulation of real hardware system

## System Virtual Machine

Emulation of system-level hardware environment

- Runs above physical hardware and below one or more OSs



# Process VM Example — Java

## Designed for program portability between platforms

Provides standard interface to software

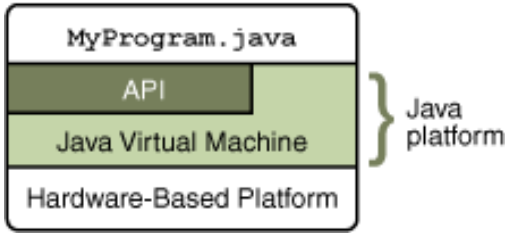
## Java VM located above a standard OS

Interface to hardware implementation dependent

I/O operations performed by calls to OS

Java compiled to bytecode

Bytecode usually run (interpreted) in Java VM







## Java without VM

Java bytecode processor in IBM mainframes

Native machine language (ISA) is Java bytecode

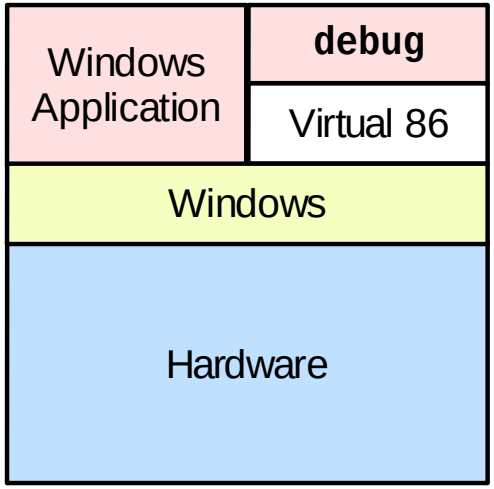
Execute Java bytecode without interpretation

http://java.sun.com/docs/books/tutorial/getStarted/intro/definition.html

# Hosted VM Example — Guest OS Over OS

## DOS command line interface over Windows

- Windows allocates 1 MB virtual memory space
- Copies DOS kernel into low memory
  - System calls handled by guest DOS kernel
- DOS accesses to hardware
  - Trapped and served by Windows host OS
  - Responses returned to DOS
- Concurrent DOS windows
  - Multiple allocations of 1 MB virtual memory spaces
- Debug
  - Application running in virtual DOS machine
  - Sees 1 MB memory space allocated by Windows
  - Register values
    - Windows emulates real values to DOS
    - Debug emulates DOS values to user




## Parallels, VirtualBox, VMware, DOSBox, ...

- Host Windows, Linux, DOS, ... as guest OSs over host OS

# Virtual Machine in IBM z/990 Mainframe

**Hardware**

CPUs, I/O system, internal communication network

**VMM (hypervisor)**

Operator console for partitioning/configuring CPUs and I/O

Provides hardware emulation as abstraction to OS layer

**OS**

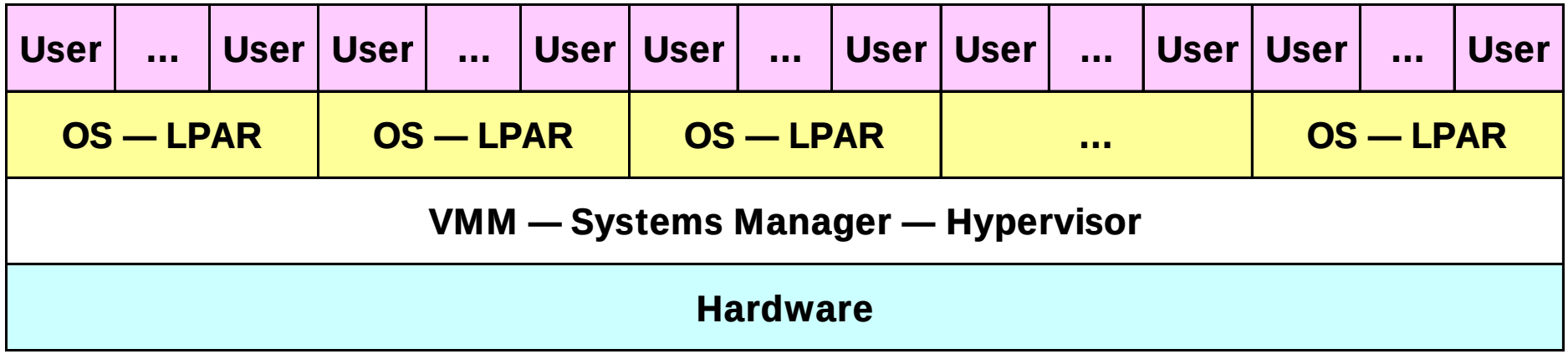
Logical partition (LPAR) runs separate instance of operating system

Run z/OS, MVS, VM, Unix, Linux, Windows, ... instances in parallel

Non-Windows OS versions expect to see hypervisor (not hardware)

**User**

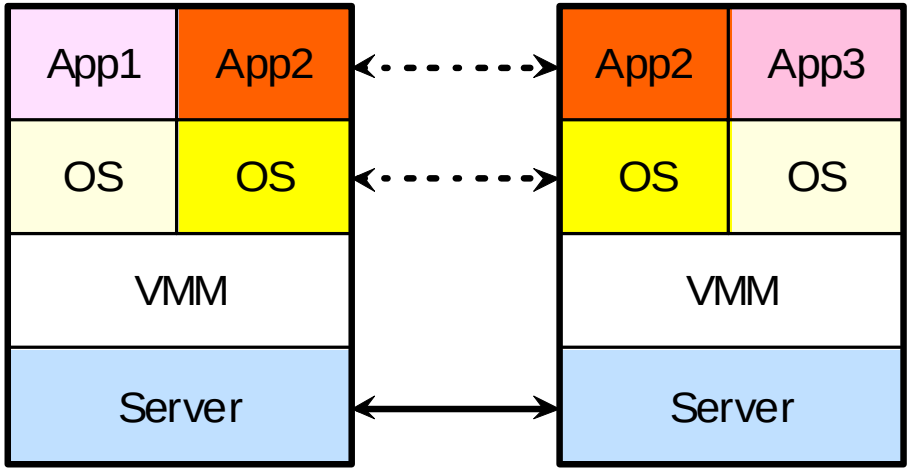
User sees single-user interface provided by one OS



# VM as System Management Tool

## Isolate user environments on single hardware platform

- Multiple copies of single operating system running independently
- Multiple operating systems running concurrently
- Maintain higher security




## Resource management

- Hardware redundancy
  - High availability
  - Recovery management
- Hardware pooling
  - Assemble hardware cluster
  - Map applications to hardware efficiently
- Load balancing
  - Remap applications to hardware

# Virtualization for Server Systems

## Old file server model

- Run one application per physical server
- Server specified for worst case load
- Large number of typically underutilized servers
- Huge aggregate space capacity

## Competition from mainframes

- VMM provides dynamic load balancing
- Hardware provides centralized power, cooling, monitoring, backup
- High SAR — scalability, availability, reliability
- Lower cost per served client than server farm

## Virtualization in server

- Partition hardware resources to run independent applications
- Intel virtualization
  - IA-32 and IA-64 ISA support
  - I/O chipset support

# HP Virtual Partitions (vPars)

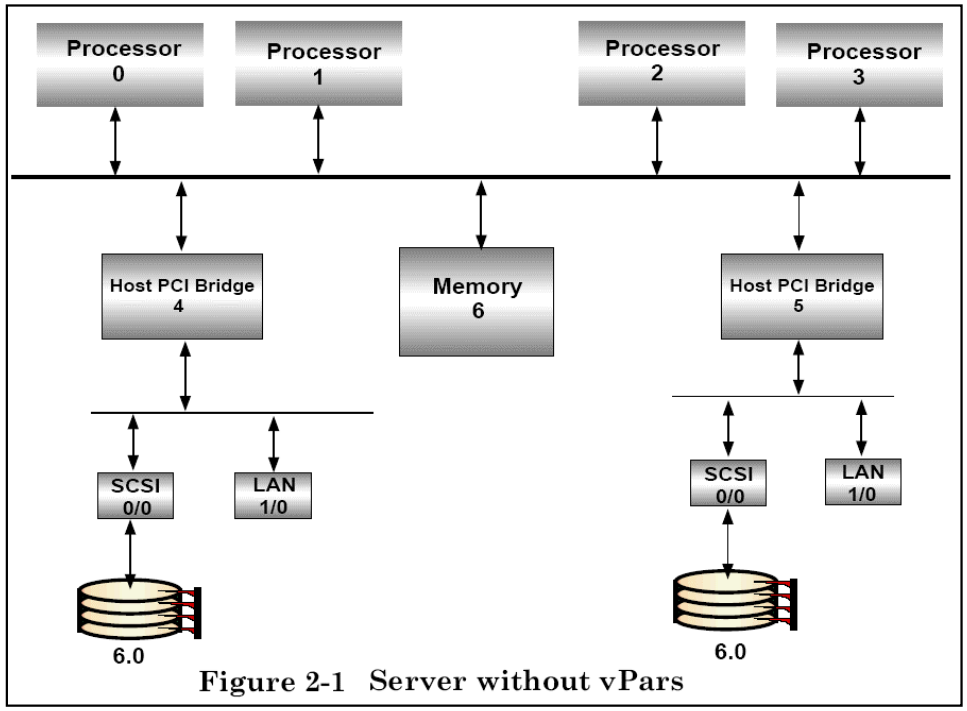



Figure 2-1 Server without vPars

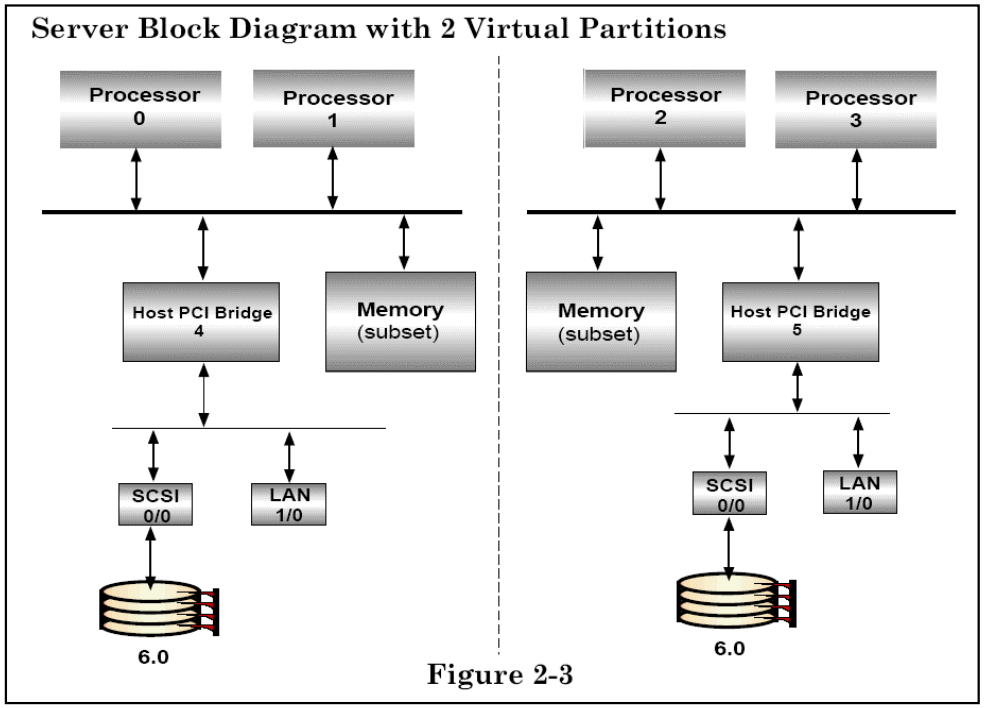
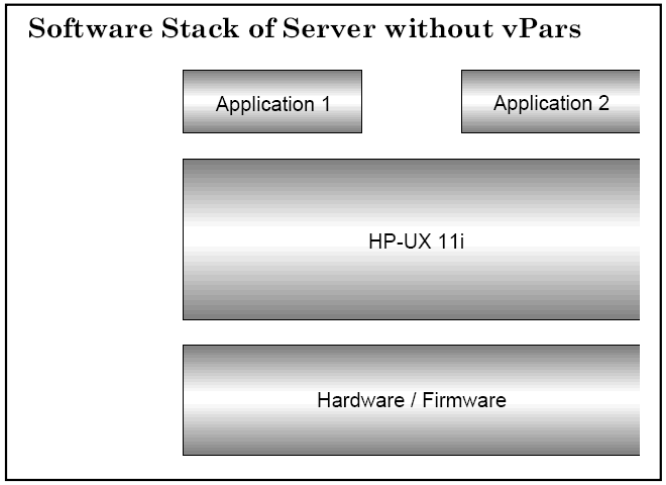
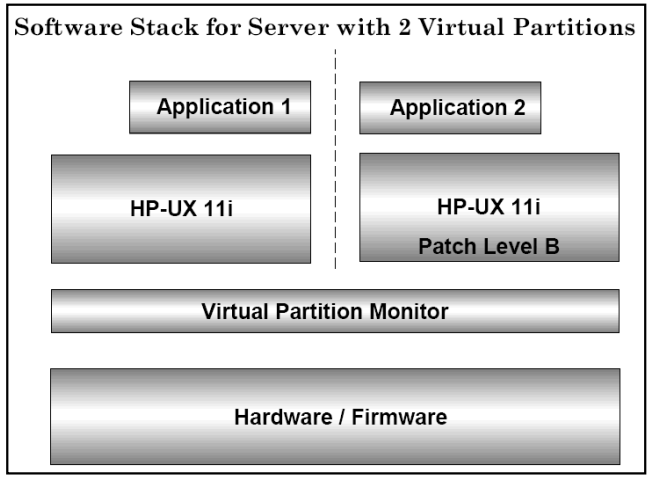



Figure 2-3




Boot Order




Hewlett-Packard, "Installing and Managing HP-UX Virtual Partitions (vPars)"

# System VM Organization

## Hypervisor

- Virtual machine monitor (VMM)
- Lowest layer above physical hardware (host)
  - Uniprocessor or multiprocessor system
- Creates virtual machine (VM) environments for OSs
  - Allocates physical host resources to virtual resources

## VM overhead

- Processor intensive applications — low overhead
  - Infrequent use of OS calls
  - Most instructions run directly on hardware
- I/O intensive applications — high overhead
  - Frequent use of OS calls
  - OS calls for I/O services run in emulation
- I/O-limited applications
  - Program throughput limited by I/O latency
  - Emulation adds relatively small overhead

# VMM Requirements

## Hardware abstraction

Guest environment must replicate hardware

VMM must present well-defined emulated interface to OS

## Protection

Isolate guests from one another

Protect VMM from guest OS and application software

Guest software cannot change allocation of physical resources

## Privilege

VMM runs in kernel mode

Guest OSs and applications run in user mode

## Hardware support for VMM

Virtualization primitives built into mainframe ISA

Any OS or application access to hardware causes trap to VMM

VMM catches every access to hardware abstraction layer (HAL)

# Virtualization Awareness

## Virtualization-unaware guest OS

- OS written to run above physical hardware
- Expects full and direct control of real hardware
- Requires extensive intervention and remapping by VMM

## Virtualization-aware guest OS

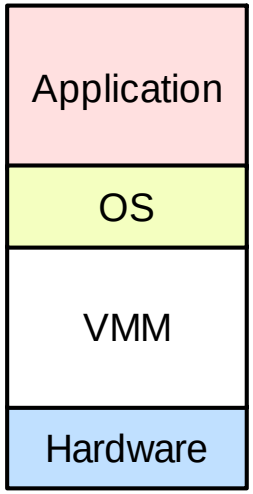
- OS written to run above VMM/hypervisor
- Expects to interact with virtual host
- Does not expect full or direct control of physical hardware
  - OS code interfaces with VMM code
  - No need to remap (bluff) pointers intended for real hardware
- May be presented with view of real system for limited operations
- Example — mainframe OS
  - Writes I/O outputs to hypervisor interface
  - Does not attempt to configure I/O hardware devices
  - Particular OS may be given direct control of particular I/O device

# Hardware Emulation Activities

## OS sees hardware through operations

OS instructions cause to CPU initiate memory and I/O operations

I/O devices initiate DMA operations and interrupts




| Operation | Real Hardware | VMM Emulation |
|---|---|---|
| CPU Memory Access | Read data or instruction<br>Write data | Translate data/instruction from guest to host format<br>Remap address space<br>Read/Write to real host memory |
| CPU I/O Device Access | Read data or instruction<br>Write data | Translate data/instruction from guest to host format<br>Remap I/O port space<br>Read/Write to real host I/O device |
| I/O device actions | DMA or IRQ | VMM manages I/O device DMA<br>Translate OS interrupt handlers from guest format to host format |

# Full/Partial System Emulation

## Full system emulation

VMM intervenes in every OS access to hardware

| | |
|---|---|
| **CPU** | **Translates guest ISA to host ISA** |
| **Memory** | **Translates memory size and organization** |
| **Chipset** | **Translates guest configuration instructions to host** |
| **I/O devices** | **Translates guest driver to host driver** |

## CPU emulation example

Run Nintendo game on PC

Translate each Nintendo instruction to IA-32 instruction set

## Partial system emulation

Part of host hardware presented to OS unchanged

VMM passes guest operations to host with minimal intervention

Most system VMs emulate subset/superset of real host hardware

CPU emulation only in special cases

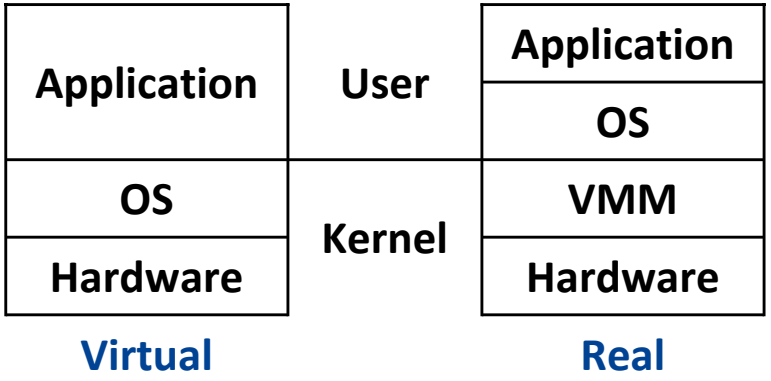
# Virtualization Difficulties for IA-32

## IA-32 designed to provide hardware support to OS

- Memory segmentation
- Virtual memory and paging
- Task management
- Interrupt management
- Protection and privilege for segmentation, paging, interrupts

## Workaround virtualization

- Treat OS like user application
- Can create a kludge on IA-32 systems
- IA-32 operating systems
  - Expect to have highest privilege
    - Can easily discover their lower privilege
  - Expect to see full virtual memory space
    - Suffer memory and table compression
    - VMM must reserve address and table space for itself



# Intel Virtualization Technology (VT)

## Virtual machine monitor

- Hardware boots (3rd party) VMM software instead of OS
- VMM configures hardware resources among guest systems
  - Remaps hardware locations to virtual pointers for guests
  - OSs boot within guest partitions

## Hardware support for virtualization

- VT-enabled processors alternate between operating modes
  - Root mode grants full hardware control to VMM
  - Non-root mode presents virtual pointers to guest OS
- VT-enabled chipset
  - Grants control of I/O to root mode
  - Remaps I/O channels for non-root mode

| | | |
|---|---|---|
| VMX non-root | User | Ring 3 User Privilege |
| | OS | Ring 0 Virtual Full Privilege |
| VMX root | VMM | Real Full Privilege |

## Operating system

- Sees virtual machine as real system
- Operates in ring 0 for maximum privilege
- Sends instructions to hardware pointers in usual way

http://www.intel.com/technology/itj/2006/v10i3/index.htm

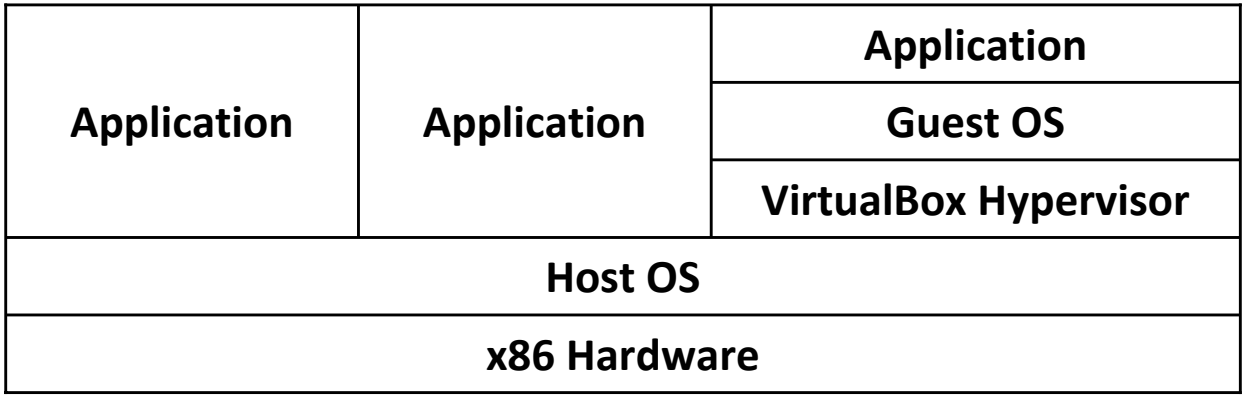
# VirtualBox

## Open source hosted VMM by Oracle (Sun Microsystems)

- Runs on Intel and AMD x86 hardware
- Runs above Windows, Linux, Mac OS X (Intel), Solaris
- Provides VM with guest OS
  - Standard DOS, Windows, Linux, OS/2, FreeBSD, Solaris
- Uses hardware virtualization support if available (not required)

## Scheduling

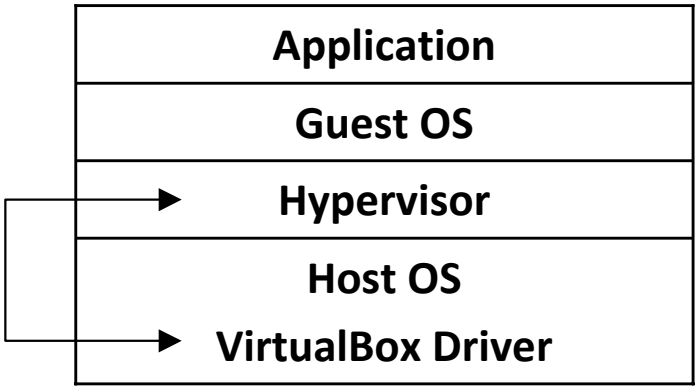
- Host OS grants timeslice to VM
- VM sub-processes scheduled by guest OS



# VirtualBox Architecture

## Front-end (client)

- VirtualBox hypervisor
- Runs above host OS
- Without Intel VT performs workaround virtualization
  - Hypervisor runs in ring 0 of guest context
  - Guest OS runs as user program in ring 1 of guest context
- Limited use of Intel VT if available




## Back-end (server)

- Ring 0 driver in host OS
- Copes with "gory details of x86 architecture"
- Allocates physical memory for VM (guest OS)
- Saves/restores guest CPU context during host interrupt
  - Registers and descriptor tables
- No intervention in guest OS process management

# CPU Operating Modes

## Runs native (not emulated) on CPU

- Host applications at ring 3
- Host OS code at ring 0
- Guest "safe" application code at ring 3
  - Non-system activities
  - Makes system calls to guest OS

## Runs emulated on CPU at ring 3

- Guest application code that causes guest OS interventions
  - Disable interrupts
  - Trap of prohibited accesses
  - Executes real mode code
- Each instruction interpreted by VirtualBox driver
  - CPU runs interpreted code instead of native code

## Runs native on CPU at ring 1

- Guest OS ring 0 code
- VirtualBox driver handles "gory details" of workaround

# Xen

## Open source system VMM

- Runs on Intel and AMD x86 hardware
- Runs directly above hardware
  - Linux required to build and install Xen
- Provides VMs with guest OSs
  - Linux, Solaris, Windows XP, 2003 Server
- Hardware virtualization support required for Windows guest OS

## Paravirtualization for Linux / Unix guest OS

- OS kernel modified to work with Xen explicitly
- Operating systems ported to run on Xen
  - Similar effort to porting OS to new hardware platform
  - Paravirtual machine architecture very similar to native hardware
- User space applications and libraries not modified

Xen Architecture Overview, http://wiki.xensource.com/xenwiki

# Xen Architecture

## Xen hypervisor

Directly above hardware

Boots system on start-up

## Domain 0

Initialized by hypervisor on boot

Runs XenLinux — modified Linux kernel

Provides Domain Management and Control (DMC)

## Domain U

VM running guest OS

<table>
<tr><td>DMC</td><td>Application</td><td>Application</td><td>Application</td></tr>
<tr><td>XenLinux OS<br>Domain 0</td><td>Guest OS<br>Domain U</td><td>Guest OS<br>Domain U</td><td>Guest OS<br>Domain U</td></tr>
<tr><td colspan="4">Xen Hypervisor</td></tr>
<tr><td colspan="4">x86 Hardware</td></tr>
</table>

# Hypervisor

## Full privilege

Operates directly on hardware in ring 0

## Functions

CPU scheduling for virtual machines

Memory partitioning

Provides hardware abstraction to virtual machines

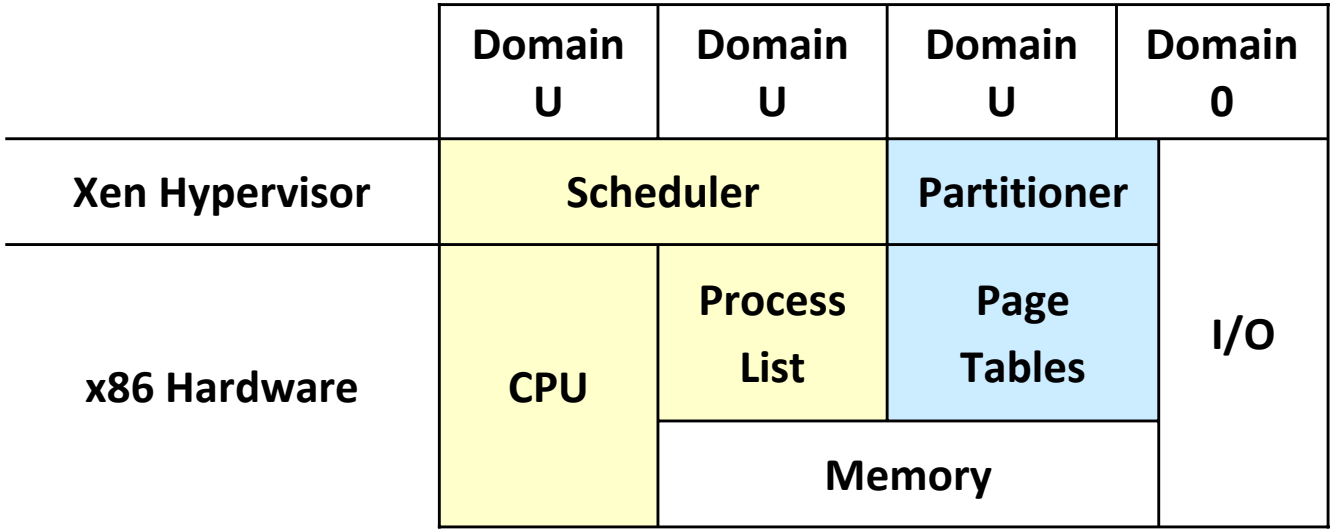
## No awareness of

Networking

External storage devices

Video

Common I/O



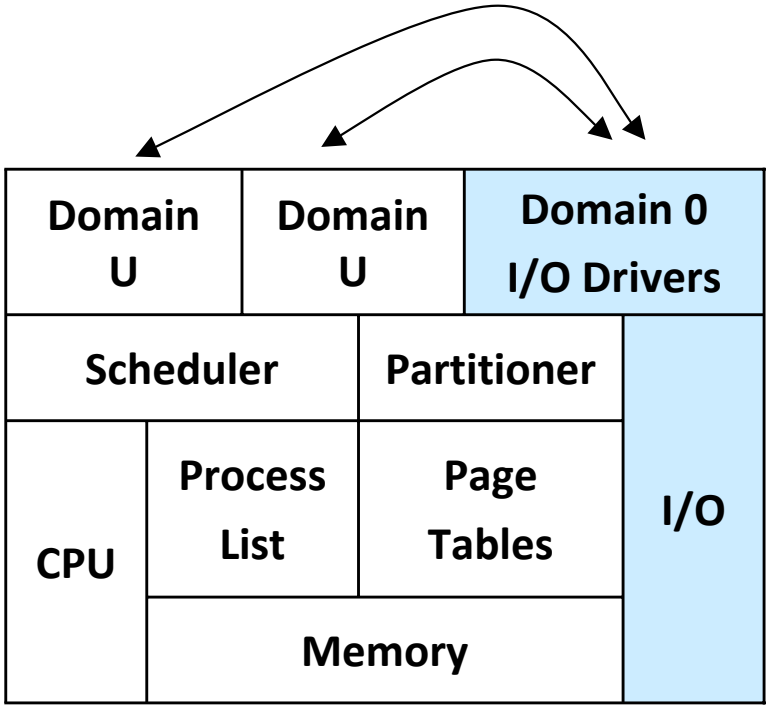
# Domain 0

## XenLinux

- Modified Linux kernel running in unique VM over hypervisor
- Direct privileged access rights to physical I/O resources
- Provides I/O virtualization to Domain U guest VMs

## Generic I/O drivers

- Network Backend Driver
  - Manages local networking hardware
  - Processes all VM networking requests from Domain U guests
- Block Backend Driver
  - Manages local storage disk
  - Processes all read/write data requests from Domain U guests



# Domain U Paravirtualization (PV)

## Domain U PV guests

Paravirtualized VM running modified Linux/UNIX kernels

## OS expectations

No direct access to host hardware

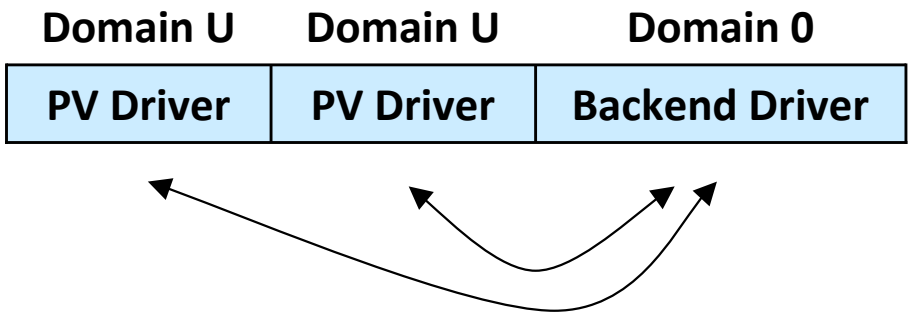
Shares host hardware with other VMs

## Guest drivers provide I/O access

Access backend drivers in Domain 0

PV Network Driver

PV Block Driver



# Domain U HVM

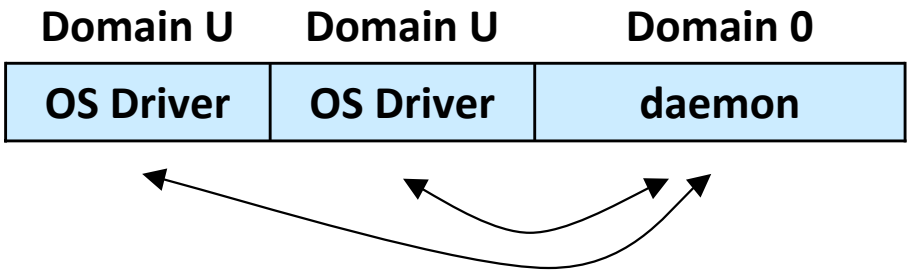
## Domain U HVM Guests

Fully virtualized machines

Run standard Windows or other unmodified OS

OS runs as VMX non-root operation with VT-x

## OS expectations




No hardware virtualization

Not sharing with other VMs

Normal hardware access for boot

- Xen virtual firmware runs as VMX root operation with VT-x
- Simulates BIOS expected by OS on initial startup

## I/O support

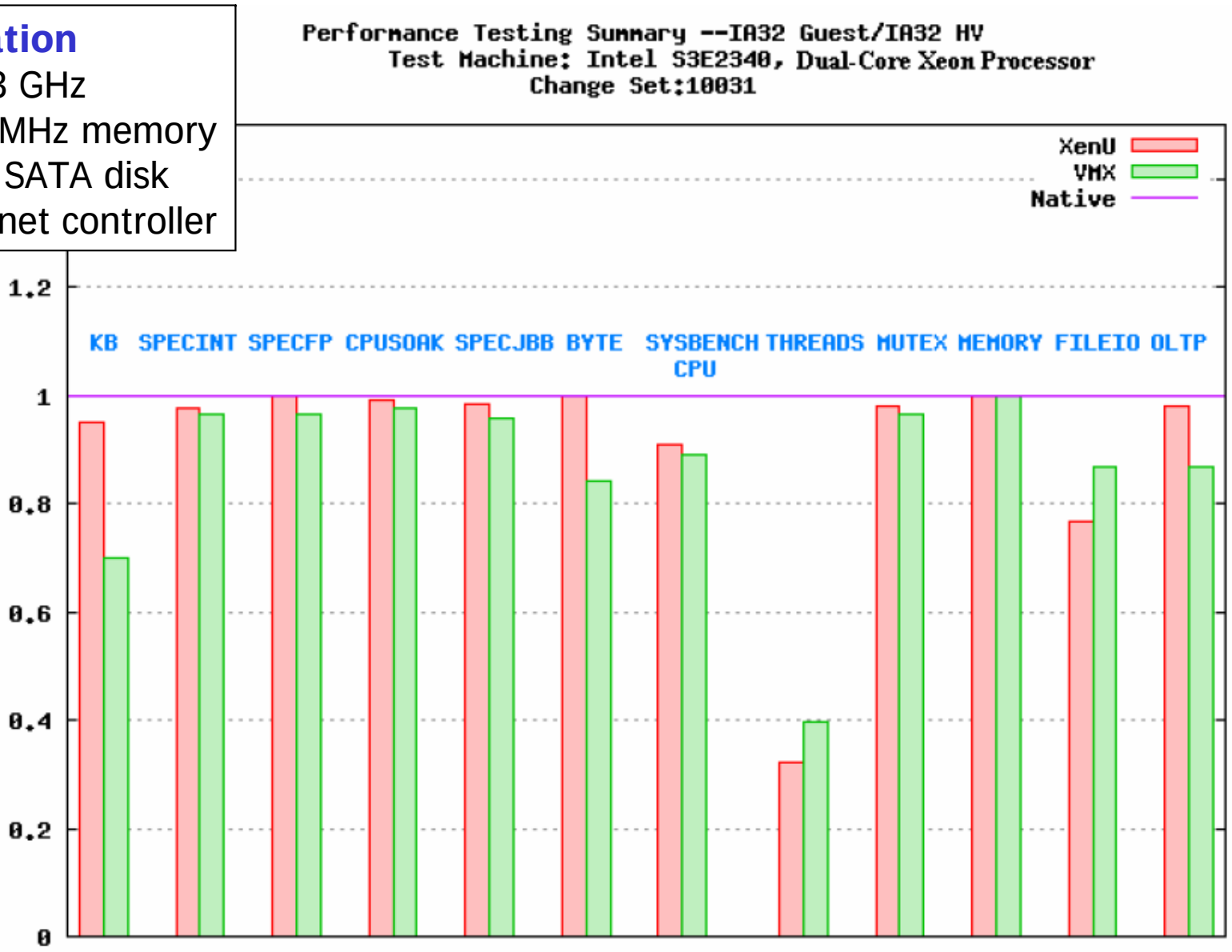
No special drivers

Domain 0 runs Qemu-dm daemon for each HVM Guest

- Supports Domain U HVM Guest for networking and disk access requests

# Xen PV and HVM Performance

**Test Configuration**
Intel Xeon @ 2.3 GHz
4 GB DDR2 533 MHz memory
160 GB Seagate SATA disk
Intel E100 Ethernet controller




Dong, et. al., "Extending Xen with Intel Virtualization Technology", Intel Technical Journal

# I/O Bottleneck

## Bottleneck — Single Ethernet controller

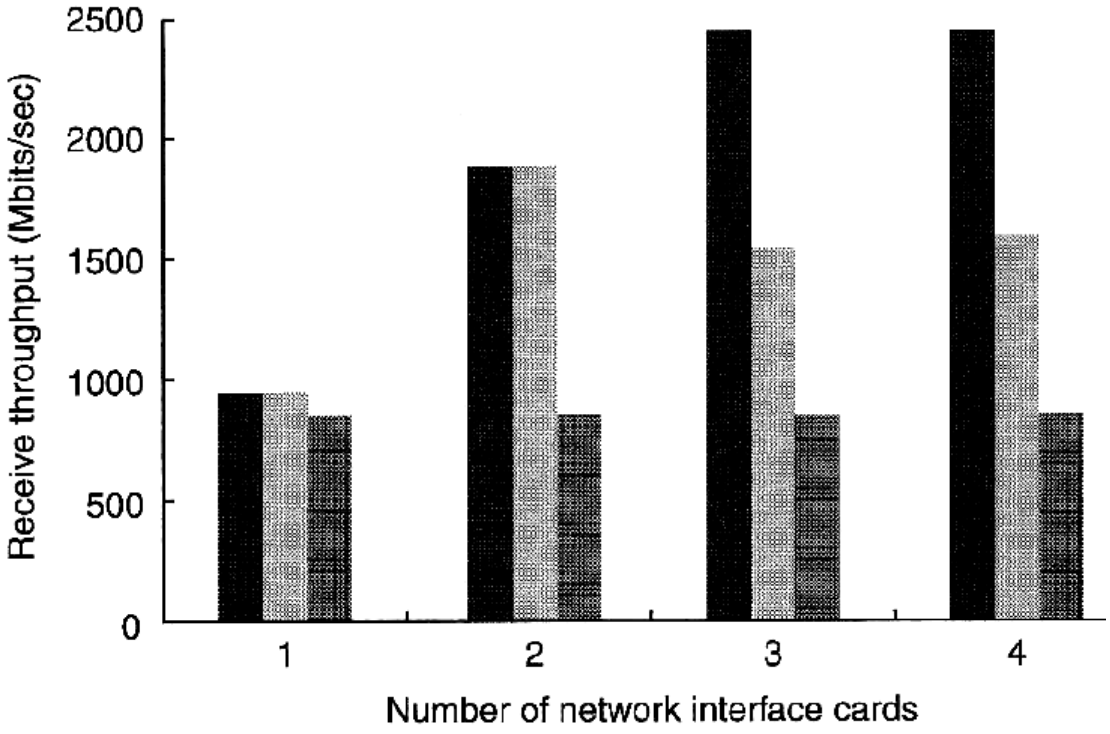
Guest OS tasks waiting for I/O access

hides performance degradation caused by virtualization

Web server running over native Linux without Xen
Threads compete above 2.5 Gbps

Web server running over XenLinux in Domain 0
Threads compete above 1.9 Gbps

Web server running over XenLinux in Domain U PV
Threads compete above 0.9 Gbps



# What is Cloud Computing?

## Outsourcing service model

- Customer obtains computing services from service provider
  - Service Level Agreement (SLA) guarantees service to customer
- Provider handles operations+administration+maintenance (OAM)
- Business advantages to customer
  - Economies of scale to large provider
  - Cuts labor/capital costs from balance sheet → happy investors

## Based on standard technologies

- Cloud service organized from conventional resources
  - Hardware + software + network
  - Provider offers menu of services
- Not a fundamentally different computing technology

## Unique technological issues

- Service reliability — provider committed to SLA
- Optimization of provider-side resource configuration
- Optimization of user-side resource configuration

# Service Hierarchy in Cloud Computing

## Infrastructure as a service (IaaS)

User sees virtual hardware environment

Real hardware or hypervisor / system virtual machine

User installs OS → installs software → runs jobs

## Platform as a service (PaaS)

User sees virtual OS environment

OS on single hardware platform or virtual OS

User installs software → runs jobs

## Software as a service (SaaS)

User sees virtual application software environment

Applications running on private OS or "sandboxed" on shared OS

Sandbox — private execution environment per application instance

User runs jobs

**Storage as a service (STaaS)**

User sees virtual mounted storage device

# Centralize → Decentralize → Centralize → ?

**1950s — 60s**

Centralized mainframe computer + multiple OS instances over hypervisor

Timesharing OS serves multiple users

User sees OS environment via dumb terminal (thin client)

**1970s**

User applications offloaded to minicomputers + timesharing services

User sees timeshared OS environment via dumb terminal

**1980s**

User applications offloaded to personal workstations (PC)

User sees single-user OS environment running locally

**1990s**

Network single user workstations

User sees single-user OS environment running locally

**2000s**

Centralized control of local OS environment by IT departments

**2010s**

Cloud + netbook / tablet / smart phone — dumb terminal with high-res GUI

# Issues in Cloud Computing

## Cost

- Provider issues
  - Economies of scale $\Rightarrow$ lower cost per compute job
- User issues
  - Capital + OAM costs $\rightarrow$ operating costs
  - Lower start-up costs $\Rightarrow$ operating debt

## Reliability

- Provider issues
  - Redundant infrastructure $\rightarrow$ continuity + disaster recovery
  - Centralized management of OAM, security, performance
  - Virtualization $\rightarrow$ serve multiple users on physical server
  - Multitenancy $\rightarrow$ provide multiple sandboxed application instances on OS
- User sees guaranteed service

## Agility

- User / provider reconfigure service / infrastructure as needed
- Growth, load balancing, time-zone serving

# Cloud Ownership

## Public cloud

- Service provider as public utility — sells / rents computing service
  - Initial providers leverage large existing infrastructure
  - Amazon, Microsoft, Google, IBM
- Menu of services at fixed prices

## Private cloud

- Cloud infrastructure for private organization
- Managed internally or outsourced
- Isolates service developers from implementation issues
  - Standard development platform
- Requirements for economic justification
  - Large organization
  - Technology-based services
  - Frequent new service
  - Example — internet content provider

# Energy Efficient Architecture

# Energy Efficiency

## Initial motivation — mobile computing

- Large market potential identified in 1990s
- Market synergy with WiFi
  - Wireless networking makes laptop more attractive
  - Laptop makes WiFi hardware more attractive
- Limited by battery life-time
- Wireless transceiver in laptop limits battery life-time far more

## Subsequent insight

- Electrical power limits computing power
  - Performance / rack = (performance / kW) × (kW / rack)
  - Improving (performance / kW) improves large system performance
- Electric bills for large server farms (ignoring cooling)
  - (400 W / PC server) × 1000 servers = 400 kW
  - 400 kW ⇒ 292,000 kW-Hr / month ~ ח"ש 146,000

## Competition

- Mainframe centralization ⇒ greater energy efficiency

# Power Consumption

## Consumed power in digital circuits

Most current flows during 0 $\rightarrow$ 1 and 1 $\rightarrow$ 0 state transitions

$$\text{Power} = P = \frac{\text{energy}}{\text{transition}} \times \frac{\text{transitions}}{\text{second}} = \frac{\text{energy}}{\text{clock cycle}} \times F, \; F = \text{clock frequency}$$

$$= \frac{\text{energy}}{\text{task}} \times \frac{\text{tasks}}{\text{clock cycle}} \times \frac{1}{\tau} = \frac{\text{energy}}{\text{task}} \times \frac{1}{\text{clock cycle per task}} \times \frac{1}{\tau}$$

$$= \frac{\text{energy}}{\text{task}} \times \frac{1}{CPI \times IC} \times \frac{1}{\tau}$$

$$= \frac{\text{energy}}{\text{task}} \times \frac{1}{CPI \times IC \times \tau} = \frac{\text{energy}}{\text{task}} \times \frac{1}{T}, \qquad T = \text{run-time}$$

## Reduce power consumption

Lower clock frequency $F$

Lower energy per transition

- Raise energy efficiency of transistor circuits
- Maintain performance with fewer circuit transitions per operation

# Power and Performance

## Energy consumption

$$\text{Power} = P = \frac{\text{energy}}{\text{task}} \times \frac{1}{T} \quad \Rightarrow \quad \frac{\text{energy}}{\text{task}} = P \times T$$

$T = \text{run-time}$

## Performance

$$\text{Performance} = \frac{\text{tasks}}{\text{run-time}} = \frac{IC}{T} = \frac{IC}{CPI \times IC \times \tau} = \frac{F}{CPI} = IPC \times F$$

$IPC = \text{instructions per clock cycle}$

## Energy and performance

$$\text{Performance} = \frac{IC}{T} \quad \Rightarrow \quad T = \frac{IC}{\text{Performance}}$$

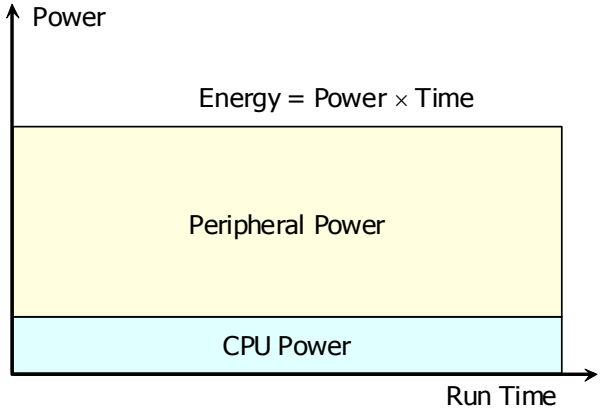
$$\frac{\text{energy}}{\text{task}} = P \times T = \frac{\text{power}}{\text{Performance}} \times IC$$

# System Issues

## CPU

Consumes 10% of power

Higher performance ⇒ higher CPU power




## Peripherals

Consume 90% of power

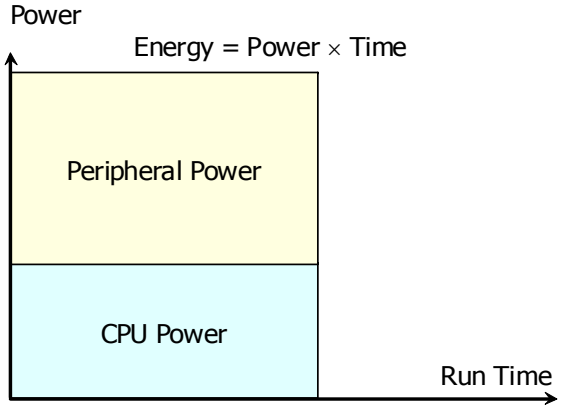
Power independent of CPU performance

LCD, hard disk, memory system, networking components, I/O system

## Idle CPU ⇒ system decreases power to CPU and peripherals

Increase performance allowing energy per task to increase

- Increases power / performance ratio
- Increases total CPU energy consumption
- Decreases run-time
- Decreases total peripheral energy consumption




## Overall strategy

Fast CPU clock ⇒ maximize CPU idle time

Minimize total consumed energy

# Classes of Architectural Changes

## Changes to processor architecture affect

Power consumption

Performance = IC / run-time

Energy = Power × run-time

| | | Run-Time | |
|---|---|---|---|
| | | **Decrease ↓** | **Increase ↑** |
| **Consumed Power** | **Decrease ↓** | Energy per Task ↓<br>Total Consumed Energy ↓<br>Improvement in all categories | Total Consumed Energy ↓ or ↑<br>Worthwhile for small increase in run-time |
| | **Increase ↑** | Total Consumed Energy ↓ or ↑<br>Possibly worthwhile | Total Consumed Energy ↑<br>Degradation in all categories |

# Choosing Architectural Changes

## Default processor operating point (zero point)

Consumes power $P_0$

Achieves performance $S_0$

Defines default state vector $(P_0, S_0)$

## Change in processor operating point

$$(P_0, S_0) \rightarrow (P_0 + \Delta P, S_0 + \Delta S) = (P_0, S_0) + (\Delta P, \Delta S)$$

## Energy efficiency criteria

$\Delta P < 0$ and $\Delta S > 0 \Rightarrow$ accept as improvement

$\Delta P < 0$ and $\Delta S < 0 \Rightarrow$ accept as improvement if same $\Delta P$ cannot be achieved with a smaller $|\Delta S|$

$\Delta P > 0$ and $\Delta S > 0 \Rightarrow$ accept as improvement for small $\Delta P$ and large $\Delta S$

$\Delta P > 0$ and $\Delta S < 0 \Rightarrow$ reject

# Physical Parameters in Circuit Performance

1. **Consumed power at zero point (default working Voltage)**

$$P_0 = \alpha \times C_0 \times V_0^2 \times F_0$$

$\alpha$ = constant

$C_0$ = effective capacitance

$V_0$ = default CPU voltage

$F_0$ = default clock frequency

2. **Effective contribution of clock frequency**

$$F_0 = K_f \times V_0$$

$K_f$ = constant

3. **Performance for given clock frequency**

$$S_0 = IPC_0 \times F_0$$

$$IPC_0 = \frac{1}{CPI_0}$$

4. **Total energy consumed by task**

$$E_0 = P_0 \times \text{run-time}$$

# Voltage Scaling Relations

**Small change in Voltage** $V_0 \rightarrow V_0 + \Delta V \Rightarrow$ changes $(\Delta P, \Delta S)$

**Combine (1) and (2)** $\Rightarrow P_0 = \left(\alpha C_0 V_0^{\,2}\right)\left(K_f V_0\right) = \alpha C_0 K_f V_0^{\,3}$

**For small Voltage change** $\Delta V$

$$\Delta P \approx \frac{dP_0}{dV_0} \times \Delta V = \alpha C_0 K_f \times 3V_0^{\,2} \times \Delta V$$

**Relative change in power**

$$\frac{\Delta P}{P_0} = \frac{3\alpha C_0 K_f V_0^{\,2} \Delta V}{\alpha C_0 K_f V_0^{\,3}} = \frac{3\Delta V}{V_0}$$

**Combine (2) and (3)** $\Rightarrow \quad S_0 = \left(IPC_0\right)\left(K_f V_0\right) = IPC_0 K_f V_0$

**Relative change in performance**

$$\frac{\Delta S}{S_0} = \frac{IPC_0 K_f \Delta V}{IPC_0 K_f V} = \frac{\Delta V}{V_0}$$

# Energy Scaling Relations

## Relative change in energy

$$E_0 = P_0 \times \text{run-time} = \frac{P_0}{S_0} = \frac{\alpha C_0 V_0^{\,2}}{IPC_0} \Rightarrow \frac{\Delta E}{E_0} = \frac{2\Delta V}{V_0}$$

## General change in operating point

$$\frac{\Delta E}{E_0} = \frac{1}{E_0}\left[\frac{\partial E_0}{\partial P_0}\Delta P + \frac{\partial E_0}{\partial S_0}\Delta S\right] = \frac{S_0}{P_0}\left[\frac{1}{S_0}\Delta P + \frac{-P_0}{S_0^{\,2}}\Delta S\right] = \frac{\Delta P}{P_0} - \frac{\Delta S}{S_0}$$

## Energy break - even point

$$\frac{\Delta E}{E_0} = 0 \Rightarrow \frac{\Delta P}{P_0} = \frac{\Delta S}{S_0}$$

## Voltage scaling stronger than break - even

$$\frac{\Delta E}{E_0} = \left[\frac{\Delta P}{P_0} - \frac{\Delta S}{S_0}\right]_{\substack{\text{Voltage}\\ \text{scaling}}} = 3\frac{\Delta V}{V_0} - \frac{\Delta V}{V_0} = 2\frac{\Delta V}{V_0}$$

# Voltage Scaling Criterion

## Voltage Scaling Relations

$$\frac{\Delta S}{S_0} = \frac{\Delta V}{V_0} \qquad \frac{\Delta P}{P_0} = 3\frac{\Delta V}{V_0} \qquad \frac{\Delta E}{E_0} = 2\frac{\Delta V}{V_0}$$

**1% drop in CPU voltage** $\Rightarrow \frac{\Delta V}{V_0} = -0.01 \Rightarrow$

3% drop in power consumed
1% drop in performance
2% drop in total energy consumed

## Criterion for architectural enhancement

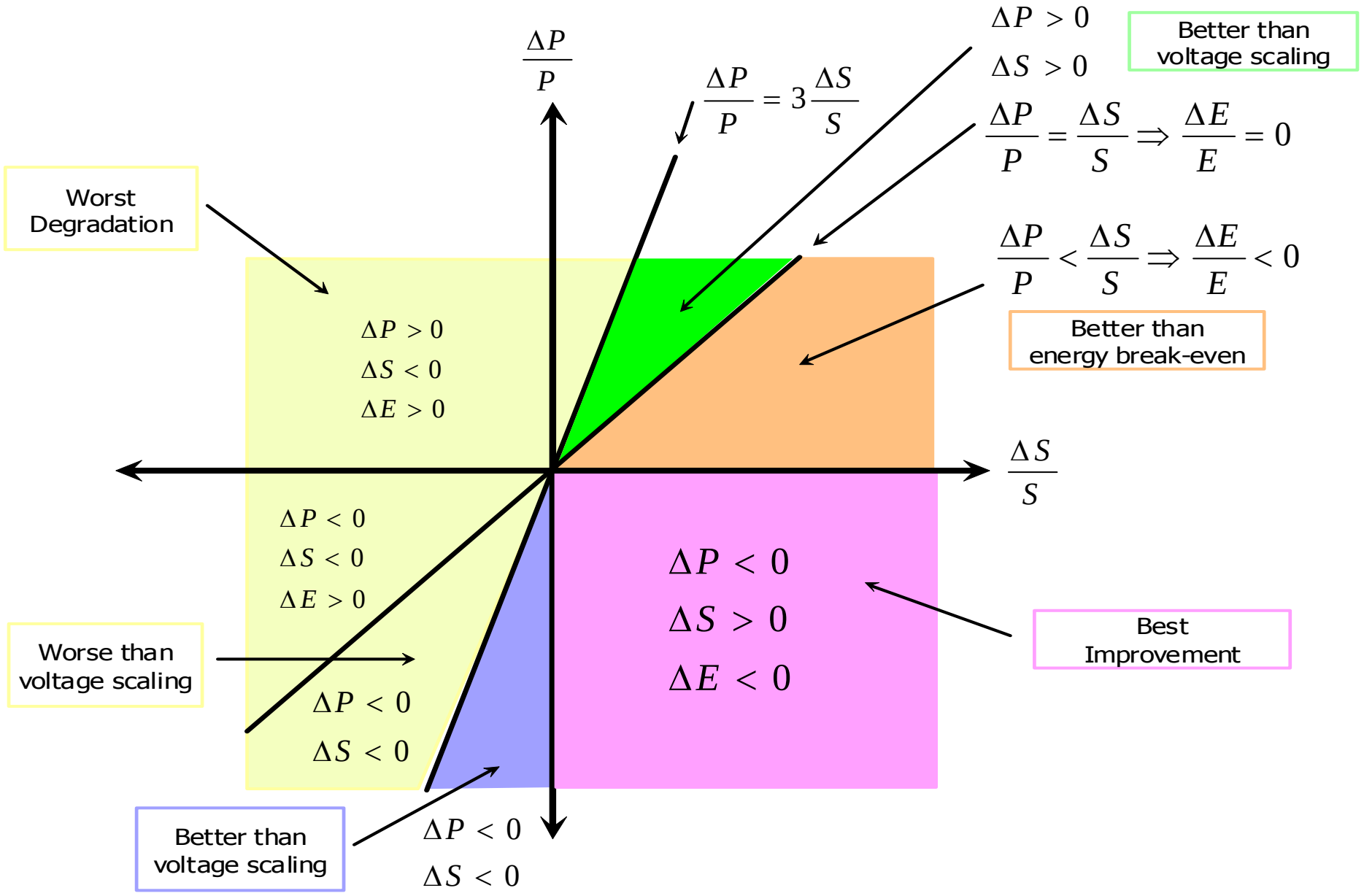
Enhancement worse than voltage scaling

- Lowers power by 3% ( $\Delta P / P_0 = -0.03$ )
- Lowers performance by more than 1% ( $|\Delta S / S_0| > 0.01$ )
- Preferable to lower CPU voltage

Enhancement better than voltage scaling

- Lowers power by 3% ( $\Delta P / P_0 = -0.03$ )
- Lowers performance by less than 1% ( $|\Delta S / S_0| < 0.01$ )
- More cost-effective than lowering CPU voltage

# Architectural Enhancement Criteria



# Intel Pentium M

## Intended for laptop computers

- Processor component in Centrino system
- M = mobile

## Design requirements set by Intel

- Satisfy IA-32 Instruction Set Architecture (ISA)
  - IA-32 instruction set for integer and floating point numbers
  - MMX and SSE/SSE2 vector extensions
- Implement Net-burst micro-architecture
  - Enhancements of P6 architecture used on Pentium II and III
  - Perform internal operations as on Pentium 4
- Save power
  - Maximize battery life in laptop
- Provide "close-to-desktop" performance
  - Performance comparable to Pentium 4 of same generation

## Strategy

- Introduce architectural changes that meet Voltage scaling criterion

# Enhancements to Lower Power

## Improve branch prediction

- Decrease number of instructions per task
  - Fewer mispredicted speculative loads
- Decrease run time
  - Higher performance and lower power

## Decrease number of micro-ops per instruction

- Micro-op fusion — combine micro-ops
  - Less micro-op independence
  - Lower performance at lower total energy

## Decrease number of transistor switches per micro-op

- Lower performance at lower total energy

## Decrease energy per transistor switch

- Higher quality transistors
- Higher performance and lower power

# Branch Prediction Enhancements

## Loop detector

- Branch instruction executes N+1 times
- Either
  - Taken N times and not-taken once
- Or
  - Not-taken N times and taken once
- Detector allocates counter to accurately predict subsequent loops

## Indirect branch predictor

- Branch target is data-dependent
- Records global target history
- Lowers misprediction of branch target

## Jump detector

- Branch instruction misses in branch predictor (first encounter)
- Predicted not taken if jumps forward
- Predicted taken if jumps backward

## Pentium-M misprediction rate

- 20% lower than Pentium 4

# Micro-Op Fusion

## P6 / Netburst architecture

- IA-32 instructions decoded to micro-ops
- Out-of-order micro-op execution produces high throughput

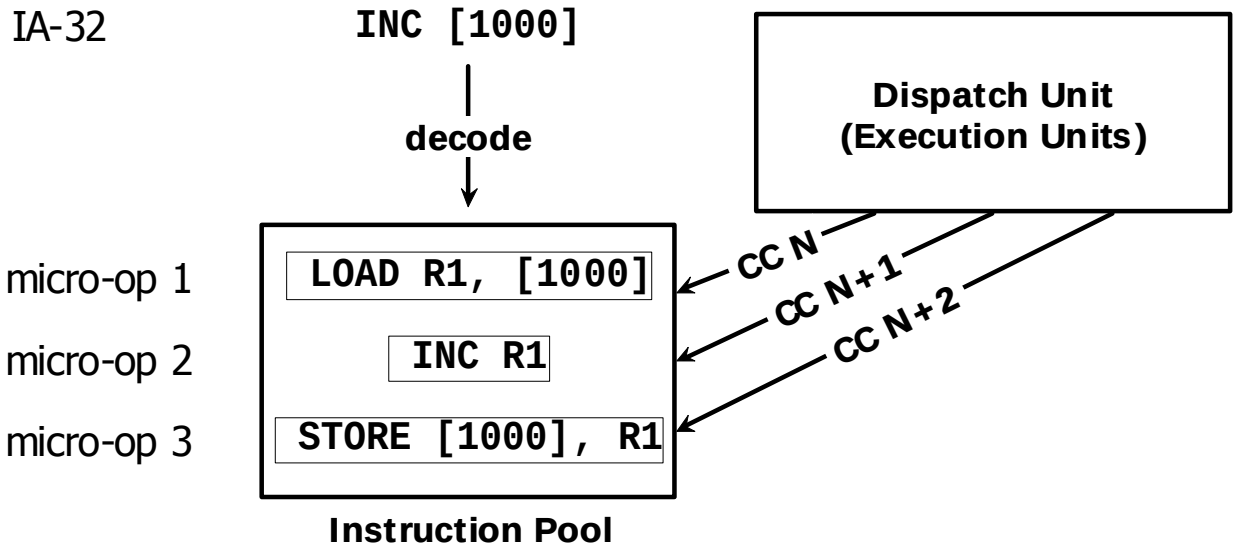
## Cost of execution efficiency

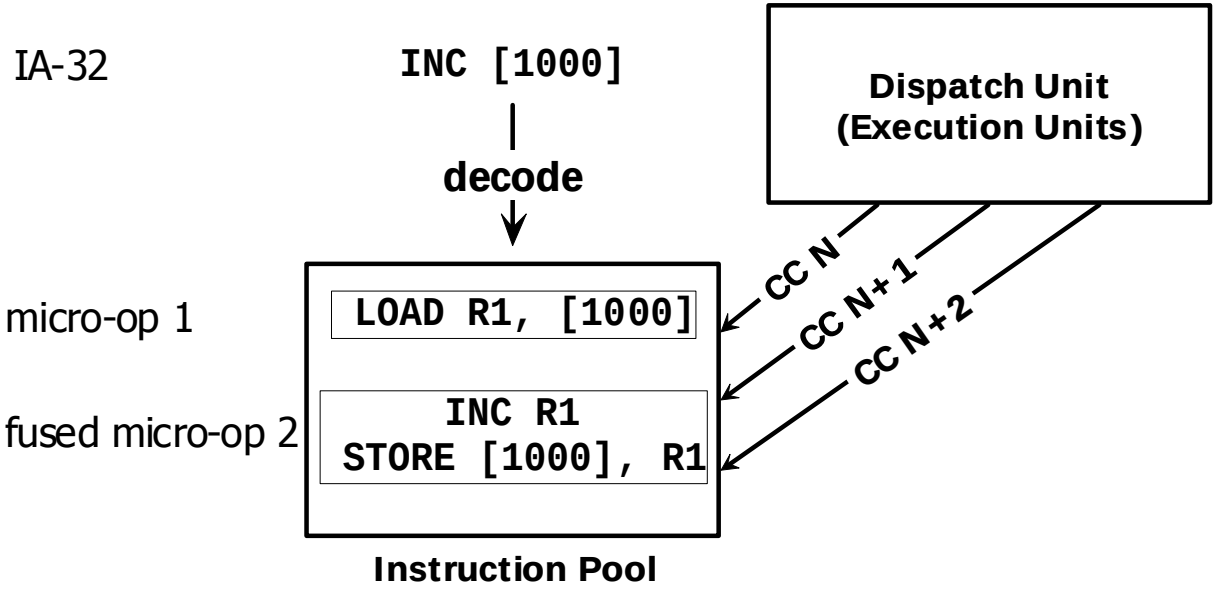
- Resource requirements depend on number of micro-ops
- Decoders, instruction pool registers, registration station
- Independent processing of micro-ops consumes more power

## Pentium-M

- Decodes certain IA-32 instructions to fewer micro-ops
- Fused micro-ops
  - Contain multiple inherently sequential P6 micro-ops
  - Stored as single fused micro-op with independent sections
  - Dispatch unit acts on one independent section of micro-op
  - Executed in dispatch unit once for each section
- Reduces number of micro-ops by about 10%

# Example of Fused Micro-Op



# Dedicated Stack Engine

## IA-32 stack operations operate on hardware stack pointer

| PUSH source | POP destination |
|---|---|
| ESP ← ESP – 4 | destination ← SS:ESP |
| SS:ESP ← source | ESP ← ESP + 4 |

## P6/Netburst implements ESP updates as micro-ops

**POP EAX** decoded as

**EAX ← SS:ESP**

**ESP ← ESP + 4**

**ESP** update micro-op executed in execution unit

## Pentium-M updates ESP in dedicated hardware

Each **PUSH/POP** decoded as transfer and update

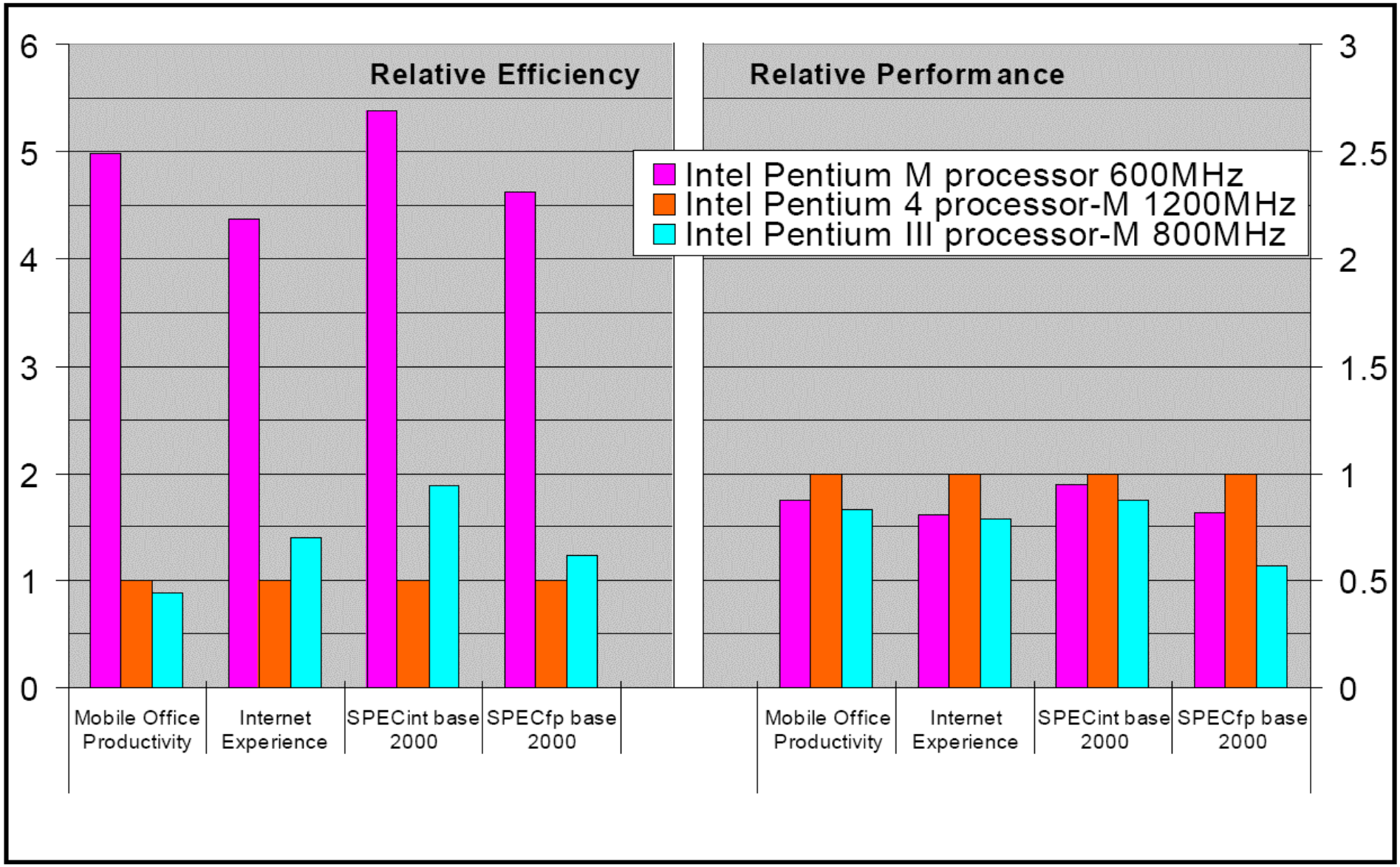
$$ESP_{IA\text{-}32} = ESP_{decode\text{-}time} + ESP_{delta}$$

**ESP** updates performed on $ESP_{delta}$ until retirement of $ESP_{IA\text{-}32}$

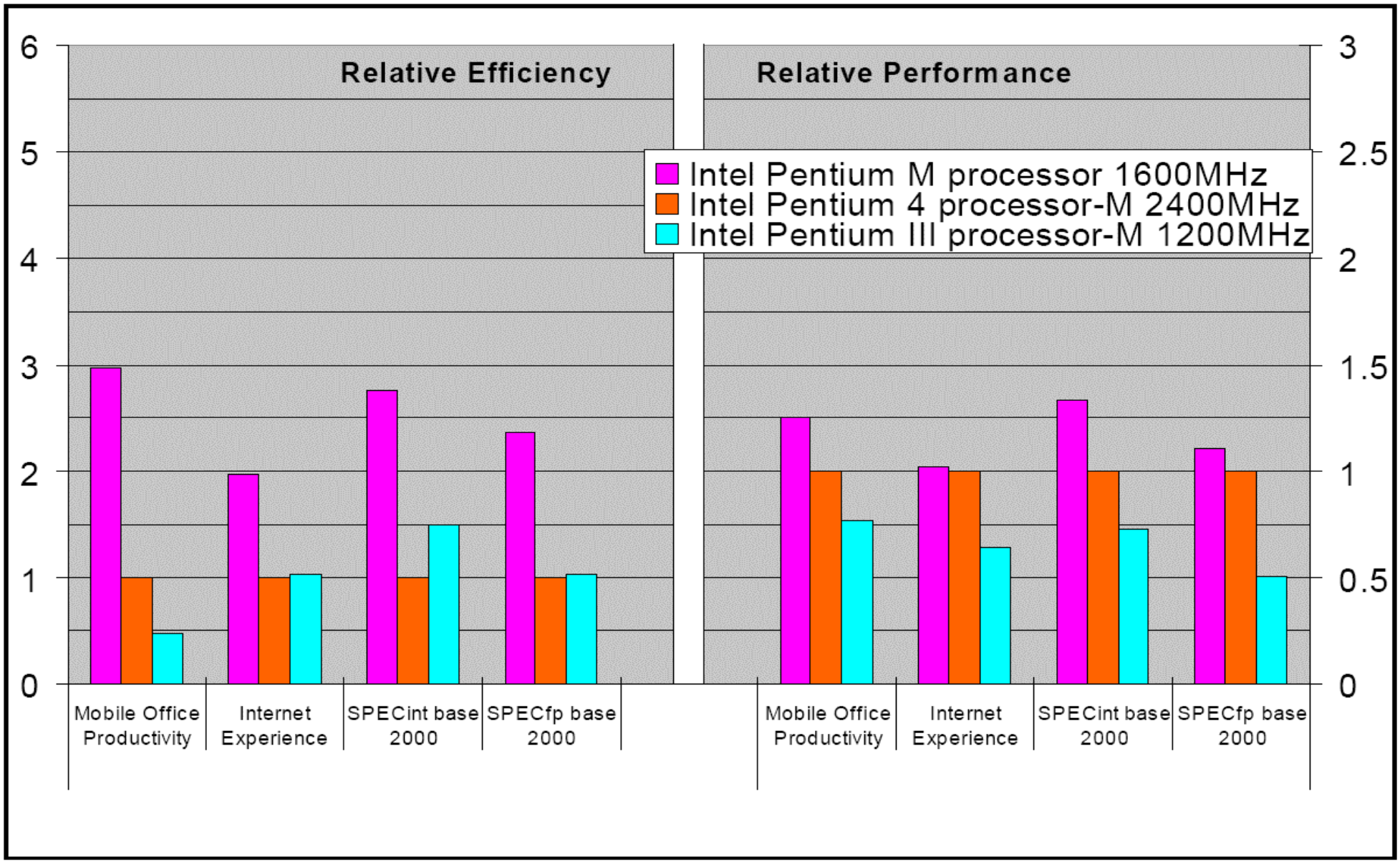
Dedicated counter keeps track of each $ESP_{delta}$ and status

$ESP_{IA\text{-}32}$ retirement updates $ESP_{decode\text{-}time}$ and zeros $ESP_{delta}$

# Maximum Battery Save Mode



# Always-On Full Speed Mode



# Intel Core Duo

## Follow-up to Pentium-M

Contains two enhanced Pentium-M cores

Extends Intel Speedstep Technology

## Design requirements set by Intel

Similar or better performance to Pentium M on single thread code

Improved performance for multithreaded and multi-process code

Average power consumption of dual core same as Pentium M

Meets thermal requirements of target systems

## Independent thermal control per core

Overall package-level power state control

Per-core power state control in package

Gochman, S. et. al., "Introduction to Intel Core Duo Processor Architecture." Intel Technology Journal.
Mendelson, A. et. al., "CMP Implementation in Systems Based on the Intel Core Duo Processor" Intel Technology Journal
Naveh, A. et. al., "Power and Thermal Management in the Intel Core Duo Processor" Intel Technology Journal

# ACPI

## Advanced Configuration and Power Interface

- Industry standard for OS interface to processor configuration
- Developed by HP, Intel, Microsoft, Phoenix, Toshiba

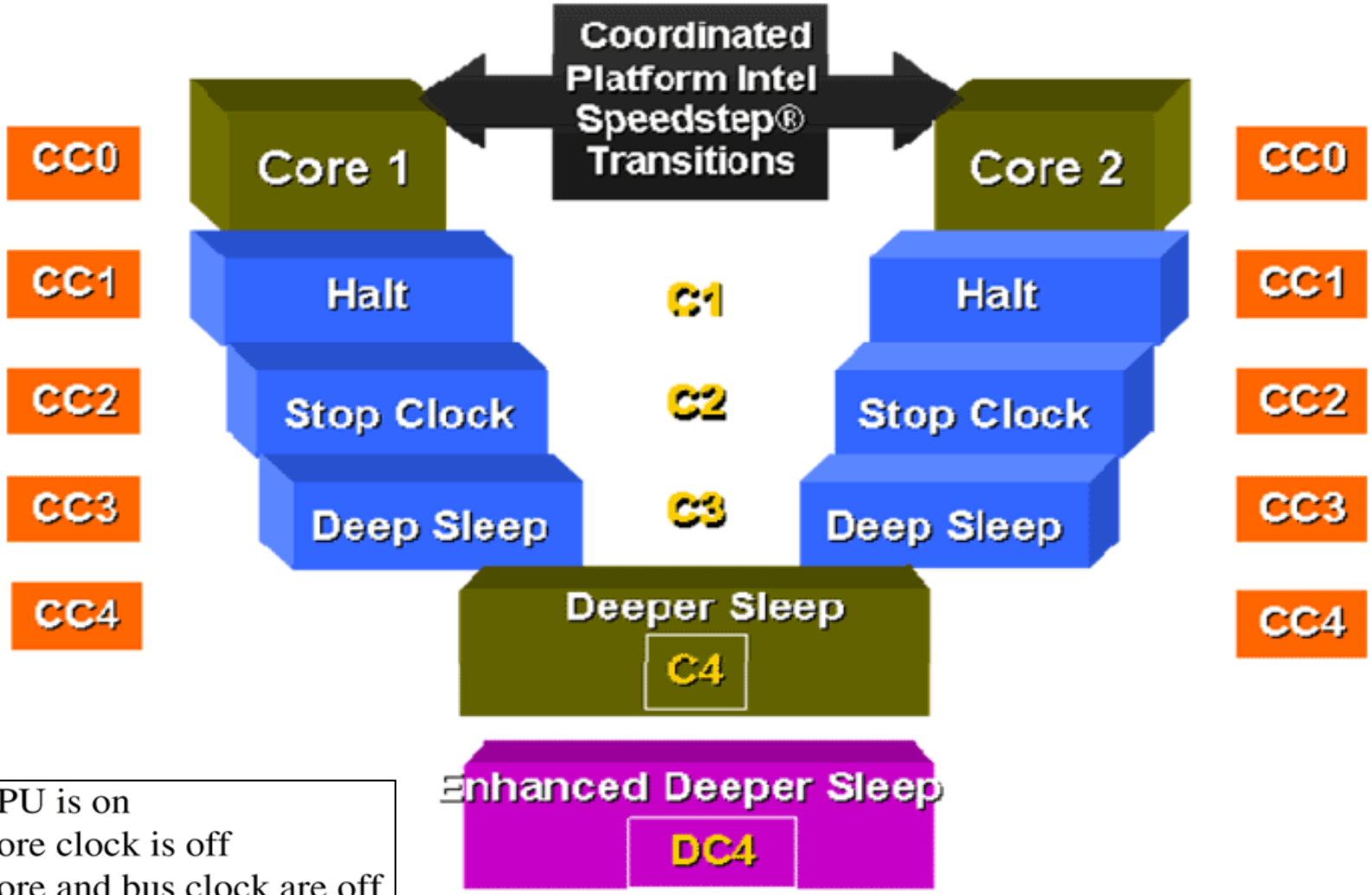
## Defines operating states

- Global (system) states
  - G0 (working), G1 (sleeping), G2 (software off), G3 (mechanical off)
- Device states
  - D0 (on), D1/D2 (device dependent) , D3 (off)
- CPU power states
  - C0 (active), C1 (halt), C2 (stop clock), C3(sleep), C4 (deep sleep), ...

## OS controls CPU power state

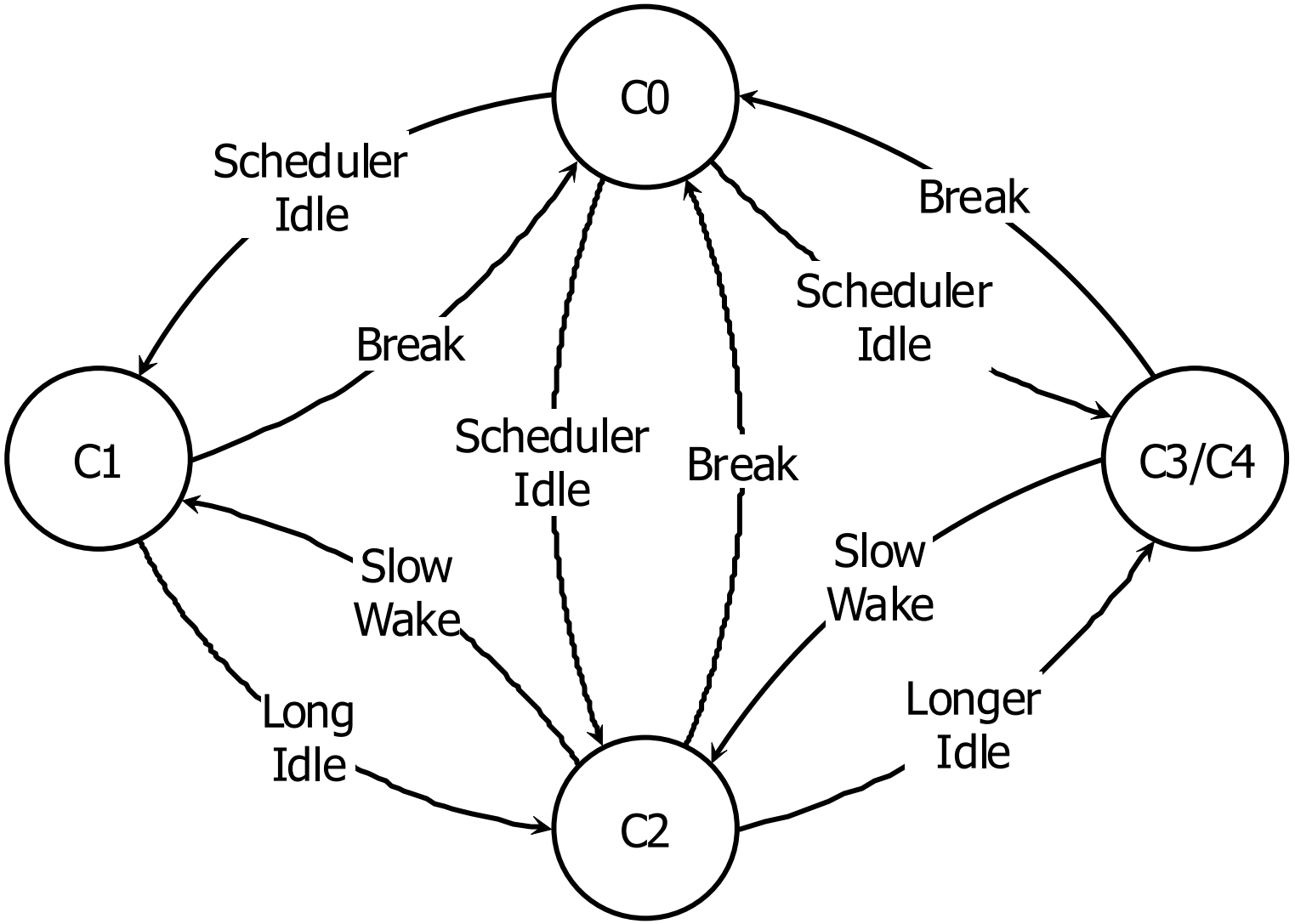
- Scheduler detects empty task queue
- Transitions processor into idle state with BIOS instruction sequence
- Processor remains idle until break event
  - Interrupt
  - Indicator of new task to be executed
- OS returns processor to C0 state

# Core Duo Sleep States




| | |
|---|---|
| **C0 – Active** | CPU is on |
| **C1 – Auto Halt** | Core clock is off |
| **C2 – Stop clock** | Core and bus clock are off |
| **C3 – Deep sleep** | Clock generator is OFF |
| **C4 – Deeper sleep** | Reduced VCC |
| **DC4 –Deeper C4** | Further reduced VCC |

# Possible ACPI Idle Management



# Anticipated Sleep State Transitions

## OS given range of sleep options

Can move idle processor from C0 directly into any sleep stage

C1, C2, ...

Can move idle processor from C0 to sleep states in stages

C1 → C2 → C3 → C4 → ...

## Core Duo sleep states

| | |
|---|---|
| **C1** | **Instruction execution and core clocks halted** |
| **C2** | **Processor cannot access external bus**<br>**Front side bus and chipset placed in lower power state** |
| **C3** | **Processor disables internal phase locked loops**<br>**L1 cache flushed and powered off** |
| **C4** | **Processor decreases internal voltage**<br>**No processing circuits can operate** |
| **DC4** | **L2 cache flushed and powered off** |