# Review of Basic Computer Architecture

החומר בפרק הזה חזר 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



## Computer Architecture
### What is Computer 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.



## Computer Architecture
### What is Computer Architecture

**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.



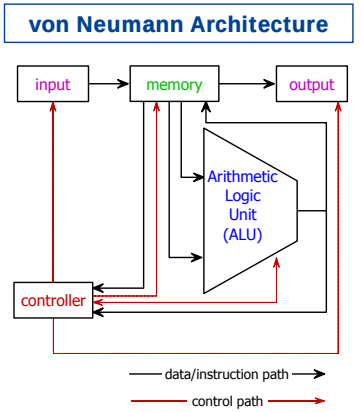
## Theory — Goals — Specification

**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

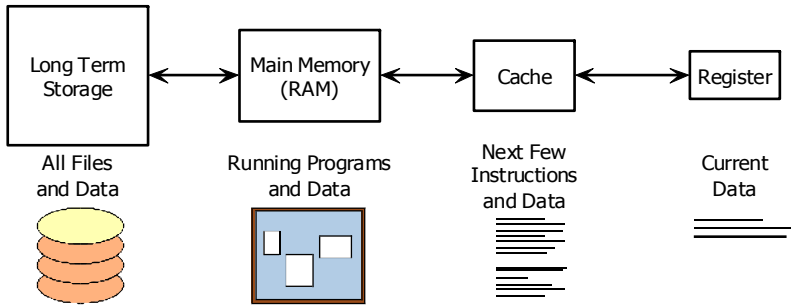
System calls, virtual memory management instructions

### 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 |

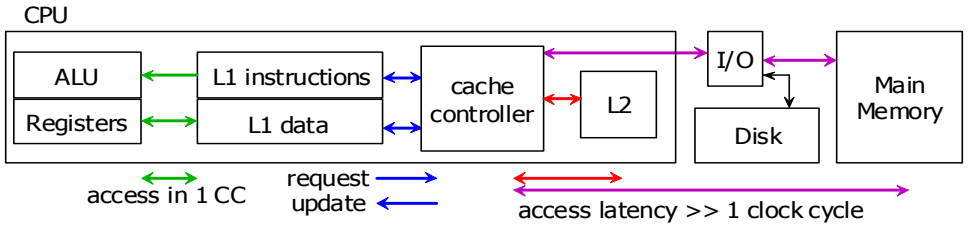





# CPU and Memory Hierarchy

**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`

R3
11223340

**Memory operands**

`address` = a memory storage location
`MEM[address]` = data stored in memory
`MEM[11223344]` = data stored at address `11223344` = `45`

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

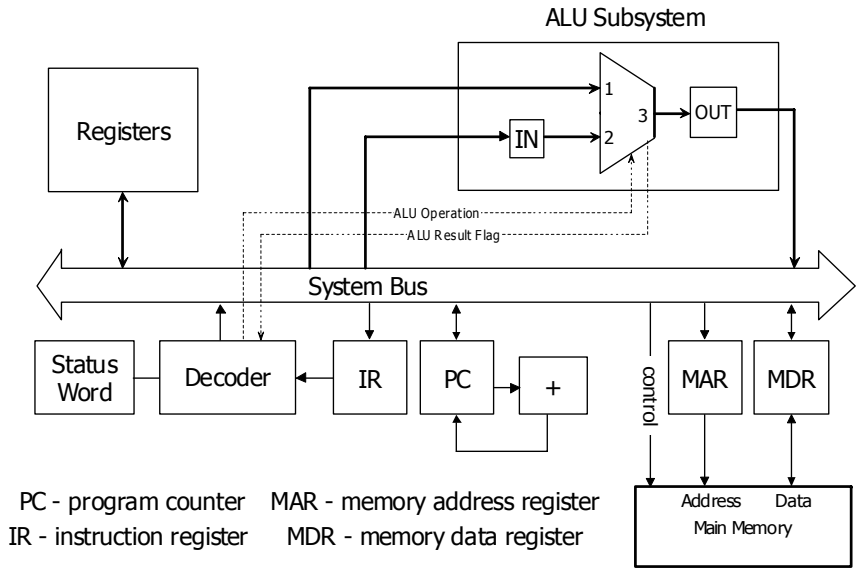
Design for user-friendly programming and small memory use

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 MAR - memory address register

IR - instruction register MDR - memory data register

## Decoding Machine Instructions

**Machine Language Instruction**

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

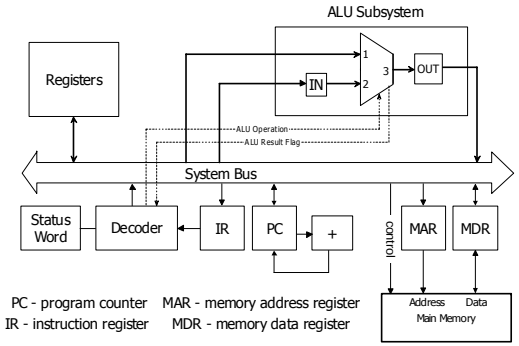
**Microcode Instruction Sequence (Microprogram)**

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

**Microcode instruction**

Hardware level atomic operation

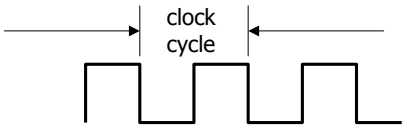
9 lines = 9 clock cycles






## 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)

**Run time** = clock cycles to run program × seconds per clock cycles

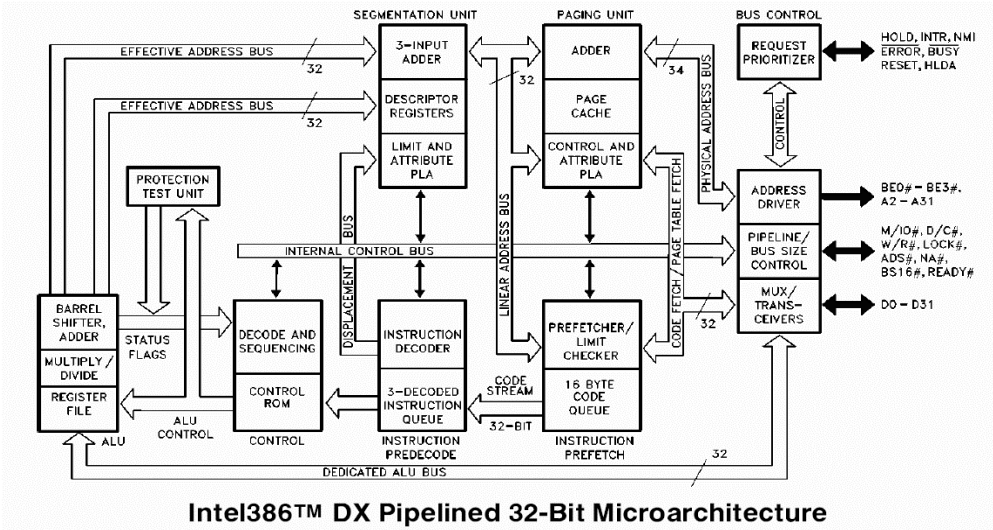
$$= \frac{\text{clock cycles to run program}}{\text{clock cycles per second}}$$

Higher clock rate ⇒ shorter run time

More clock cycles (at constant clock rate) ⇒ 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 ⇒ 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

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

$S_i = \frac{t_i}{t_i'}$ = **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

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

**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**

| 31 26 | 25 21 | 20 16 | 15 5 | 4 0 | |
|---|---|---|---|---|---|
| **Opcode** | **Number** | | | | PALcode type |
| **Opcode** | **Ra** | **Disp** | | | Branch type |
| **Opcode** | **Ra** | **Rb** | **Disp** | | Memory type |
| **Opcode** | **Ra** | **Rb** | **Function** | **Rc** | Operate type |

**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`**

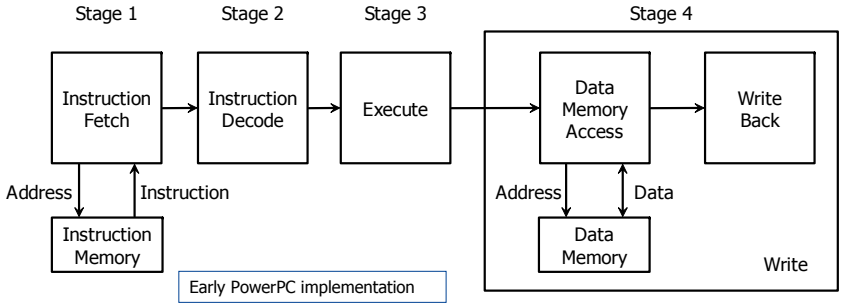
**Memory** — move between **`Ra`** and **`Mem[Regs[Rb] + Disp]`**

**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**

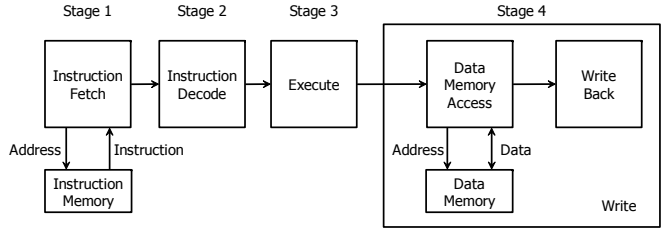


## Pipelining — The RISC Advantage

**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 |

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

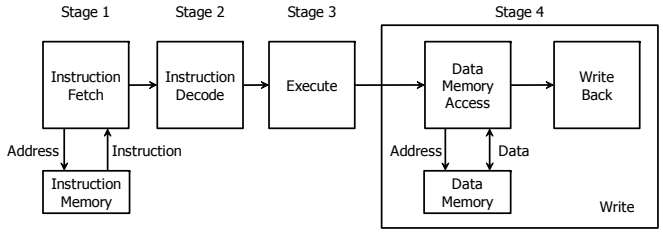
$$CPI^{\text{ideal}} = \frac{N^{ideal}}{IC}$$

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

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



## 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**

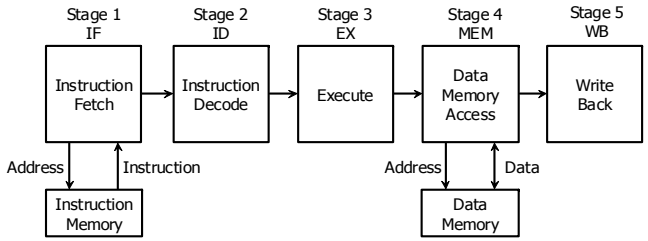
Only LOAD and STORE access data memory

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

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

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

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

**Eliminate cause of stall**

Improve implementation based on analysis of stalls

Main activity of hardware architects



## Structural Hazards

**Conflict over access to resource**

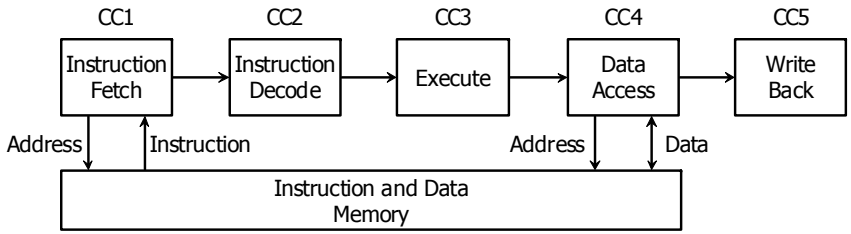
**Typical structural hazard — unified cache hazard**

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%

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

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

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

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

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

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

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

## 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

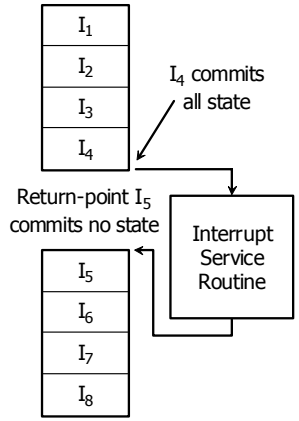
No following operations commit any results to state

**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/