Computing System Fundamentals/Trends + Review of Performance Evaluation and ISA Design

- **Computing Element Choices:**
  - Computing Element Programmability
  - Spatial vs. Temporal Computing
  - Main Processor Types/Applications
- **General Purpose Processor Generations**
- **The Von Neumann Computer Model**
- CPU Organization (Design)
- Recent Trends in Computer Design/performance
- **Hierarchy of Computer Architecture**
- Computer Architecture Vs. Computer Organization
- **Review of Performance Evaluation Review from 350:**
  - The CPU Performance Equation
  - Metrics of Computer Performance
  - MIPS Rating
  - MFLOPS Rating
  - **Amdahl's Law**
- **Instruction Set Architecture (ISA) Review from 350:**
  - Definition and purpose
  - ISA Types and characteristics
  - CISC vs. RISC
- A RISC Instruction Set Example: **MIPS64**
- The Role of Compilers in Performance Optimization

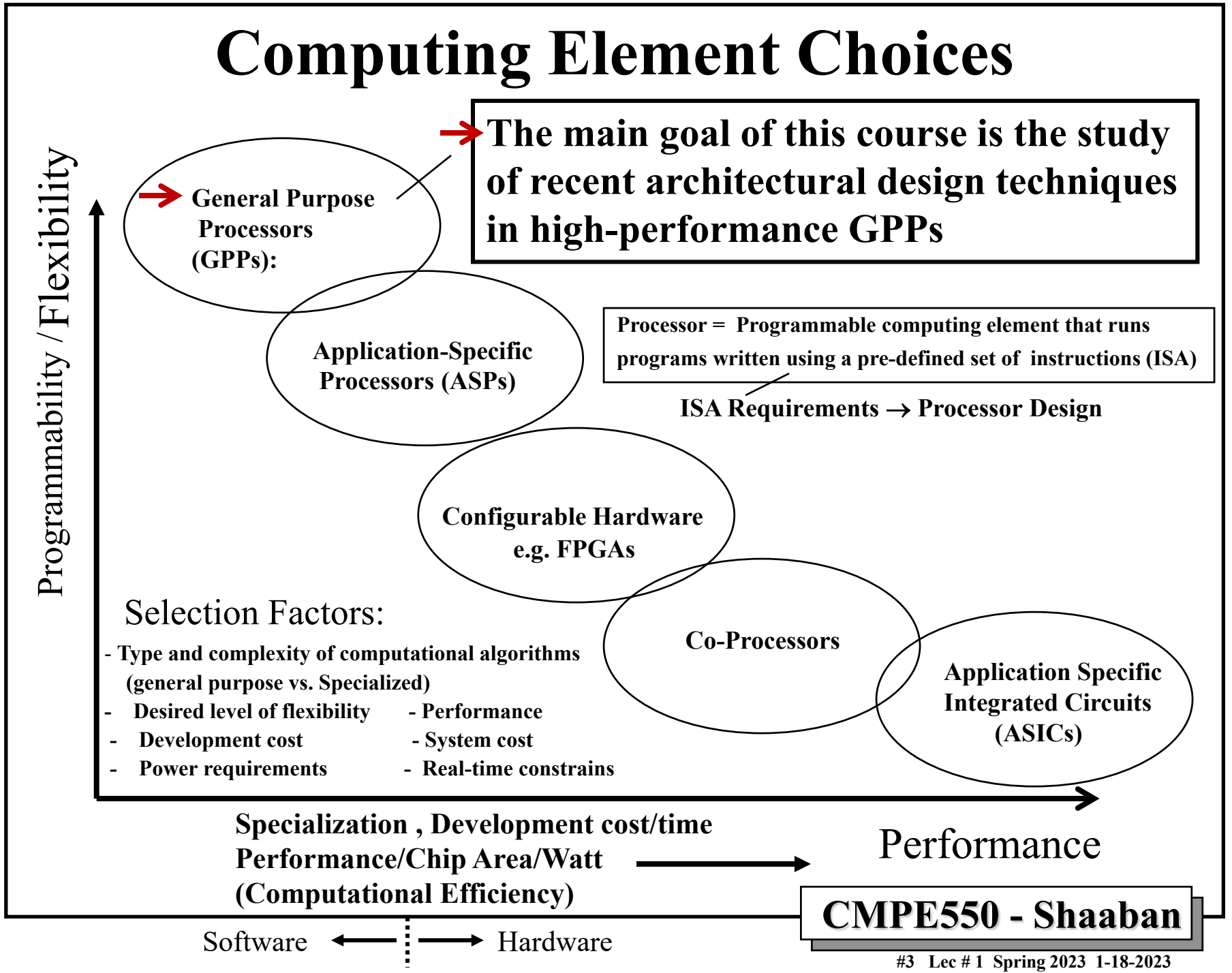
4th Edition: Chapter 1, Appendix B (ISA)
3rd Edition: Chapters 1 and 2

Computing Element Choices

- **General Purpose Processors (GPPs):** Intended for general purpose computing (desktops, servers, clusters..)
- **Application-Specific Processors (ASPs):** Processors with ISAs and architectural features tailored towards specific application domains
  - E.g Digital Signal Processors (DSPs), Network Processors (NPs), Media Processors, Graphics Processing Units (GPUs), Vector Processors??? ...
- **Co-Processors:** A hardware (hardwired) implementation of specific algorithms with limited programming interface (augment GPPs or ASPs)
- **Configurable Hardware:**
  - Field Programmable Gate Arrays (FPGAs)
  - Configurable array of simple processing elements
- **Application Specific Integrated Circuits (ASICs):** A custom VLSI hardware solution for a specific computational task
- The choice of one or more depends on a number of factors including:
  - Type and complexity of computational algorithm (general purpose vs. Specialized)
  - Desired level of flexibility/ programmability
  - Performance requirements
  - Development cost/time
  - System cost
  - Power requirements
  - Real-time constrains

The main goal of this course is to study recent architectural design techniques in high-performance GPPs

Computing Element Choices



Computing Element Choices:

Computing Element Programmability

(Hardware)

Fixed Function:

Or Algorithm

- **Computes one function (e.g. FP-multiply, divider, DCT)**
- **Function defined at fabrication time**
- **e.g hardware (ASICs)**

(Processor)

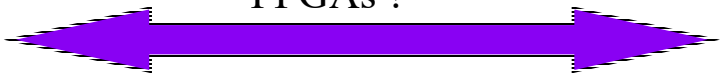
Programmable:

Software

Program

- **Computes "any" computable function (e.g. Processors)**
- **Function defined after fabrication**
- **Instruction Set (ISA)**

FPGAs ?



ISA Requirements → *CPU Design*

Parameterizable Hardware:
Performs limited "set" of functions

e.g. Co-Processors

Processor = Programmable computing element that runs programs written using pre-defined instructions (ISA)

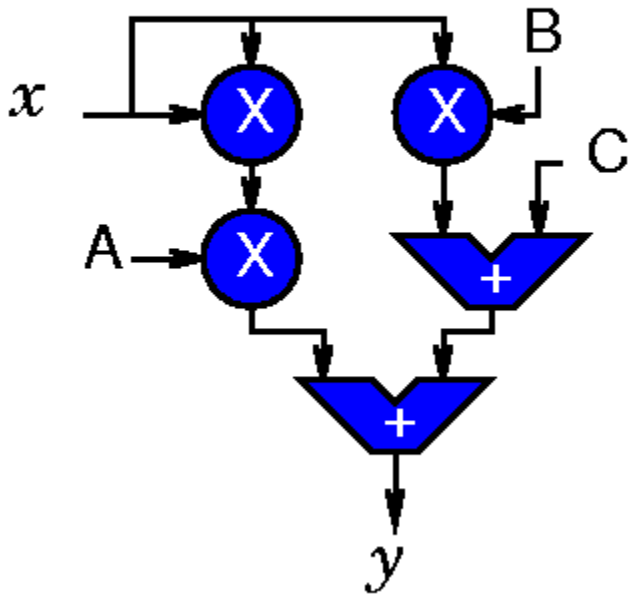
Computing Element Choices:

Space vs. Time Tradeoff

Spatial vs. Temporal Computing

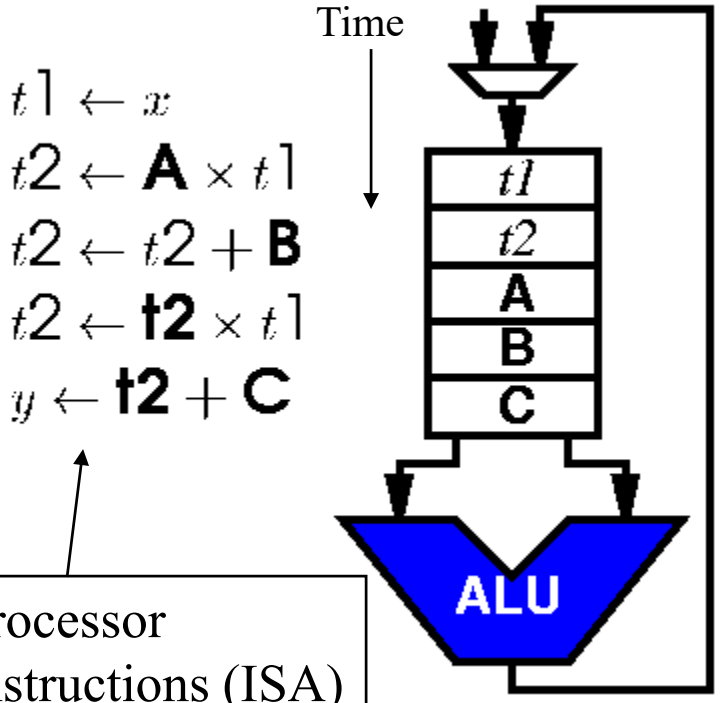
Spatial

(using hardware)



Temporal

(using software/program running on a processor)



Processor Instructions (ISA)

ISA Requirements → Processor Design

Processor = Programmable computing element that runs programs written using a pre-defined set of instructions (ISA)

Main Processor Types/Applications

- **General Purpose Processors (GPPs) - high performance.**
  - RISC or CISC: Intel P4, IBM Power4, SPARC, PowerPC, MIPS ...
  - Used for general purpose software
  - Heavy weight OS - Windows, UNIX
  - Workstations, Desktops (PC's), Clusters

  64 bit

- **Embedded processors and processor cores**
  - e.g: Intel XScale, ARM, 486SX, Hitachi SH7000, NEC V800...
  - Often require Digital signal processing (DSP) support or other application-specific support (e.g network, media processing)
  - Single program
  - Lightweight, often realtime OS or no OS

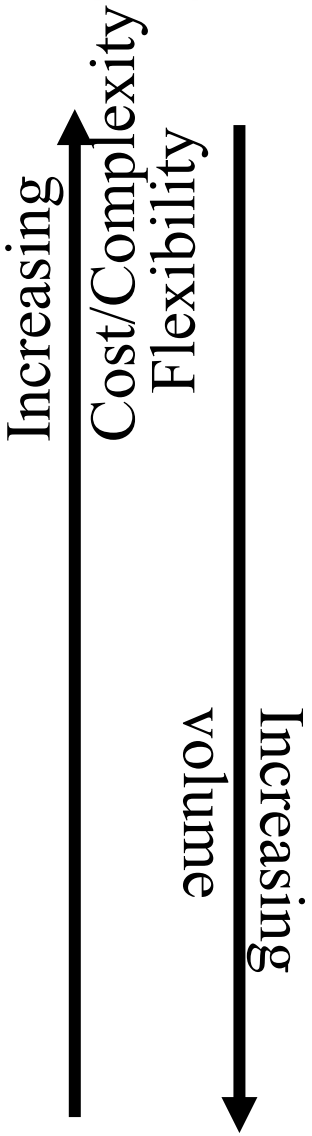
  16-32 bit

  - Examples: Cellular phones, consumer electronics .. (e.g. CD players)

- **Microcontrollers**
  - Extremely cost/power sensitive
  - Single program
  - Small word size - 8 bit common
  - Highest volume processors by far

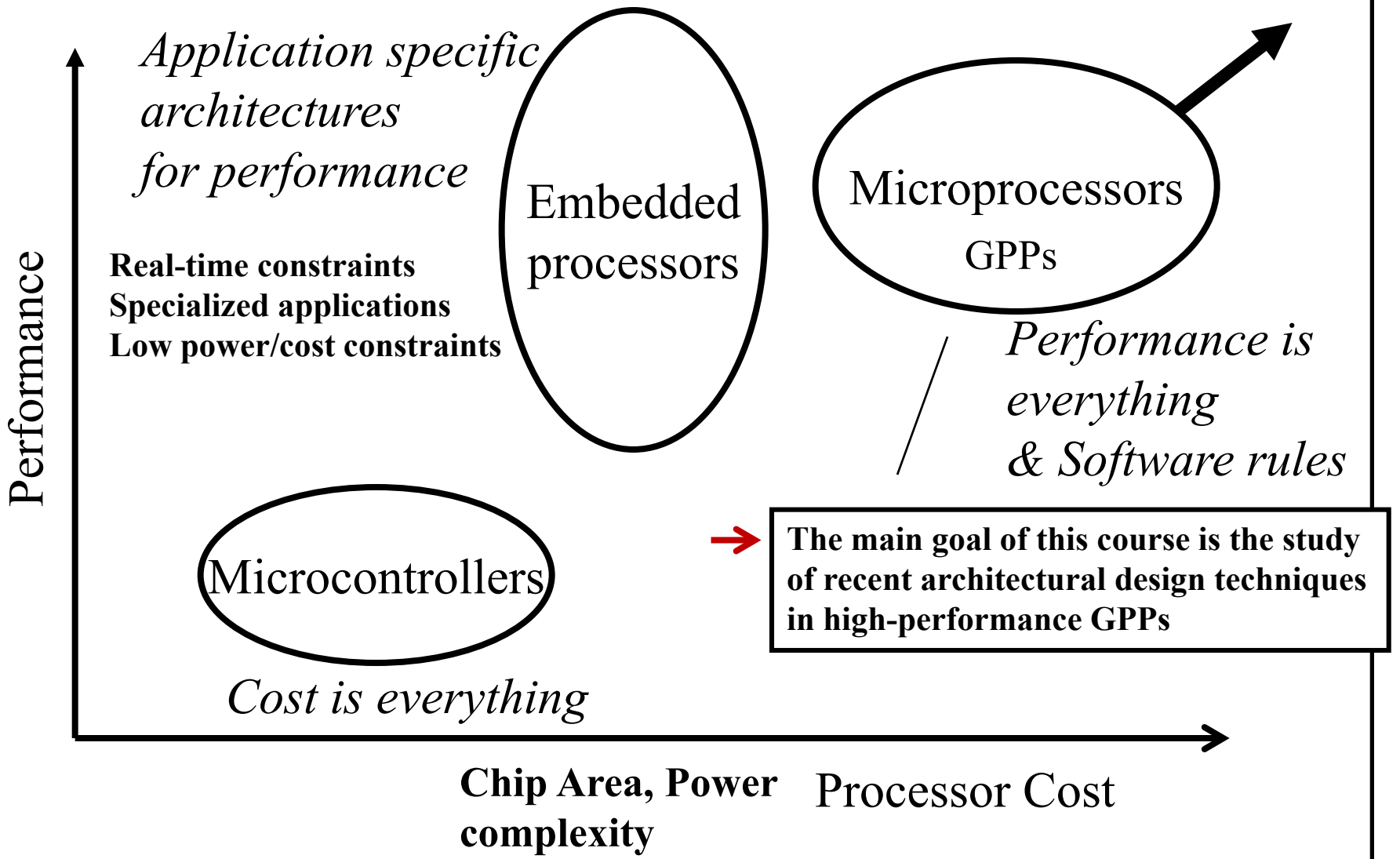
  8-16 bit ?

  - Examples: Control systems, Automobiles, toasters, thermostats, ...



Examples of Application-Specific Processors (ASPs)

The Processor Design Space



The main goal of this course is the study of recent architectural design techniques in high-performance GPPs

Processor = Programmable computing element that runs programs written using a pre-defined set of instructions (ISA)

General Purpose Processor (GPP) Generations

Classified according to implementation technology of logic devices:

- **The First Generation, 1946-59: Vacuum Tubes, Relays, Mercury Delay Lines:**
  - **ENIAC (Electronic Numerical Integrator and Computer): First electronic computer, 18000 vacuum tubes, 1500 relays, 5000 additions/sec (1944).**
  - **First stored program computer: EDSAC (Electronic Delay Storage Automatic Calculator), 1949.**
- **The Second Generation, 1959-64: Discrete Transistors.**
  - **e.g. IBM Main frames**
- **The Third Generation, 1964-75: Small and Medium-Scale Integrated (MSI) Circuits.**
  - **e.g Main frames (IBM 360) , mini computers (DEC PDP-8, PDP-11).**
- **The Fourth Generation, 1975-Present: The Microcomputer. VLSI-based Microprocessors.** (Microprocessor = VLSI-based Single-chip processor)
  - **First microprocessor: Intel's 4-bit 4004 (2300 transistors), 1970.**
  - **Personal Computer (PCs), laptops, PDAs, servers, clusters …**
  - **Reduced Instruction Set Computer (RISC) 1984**

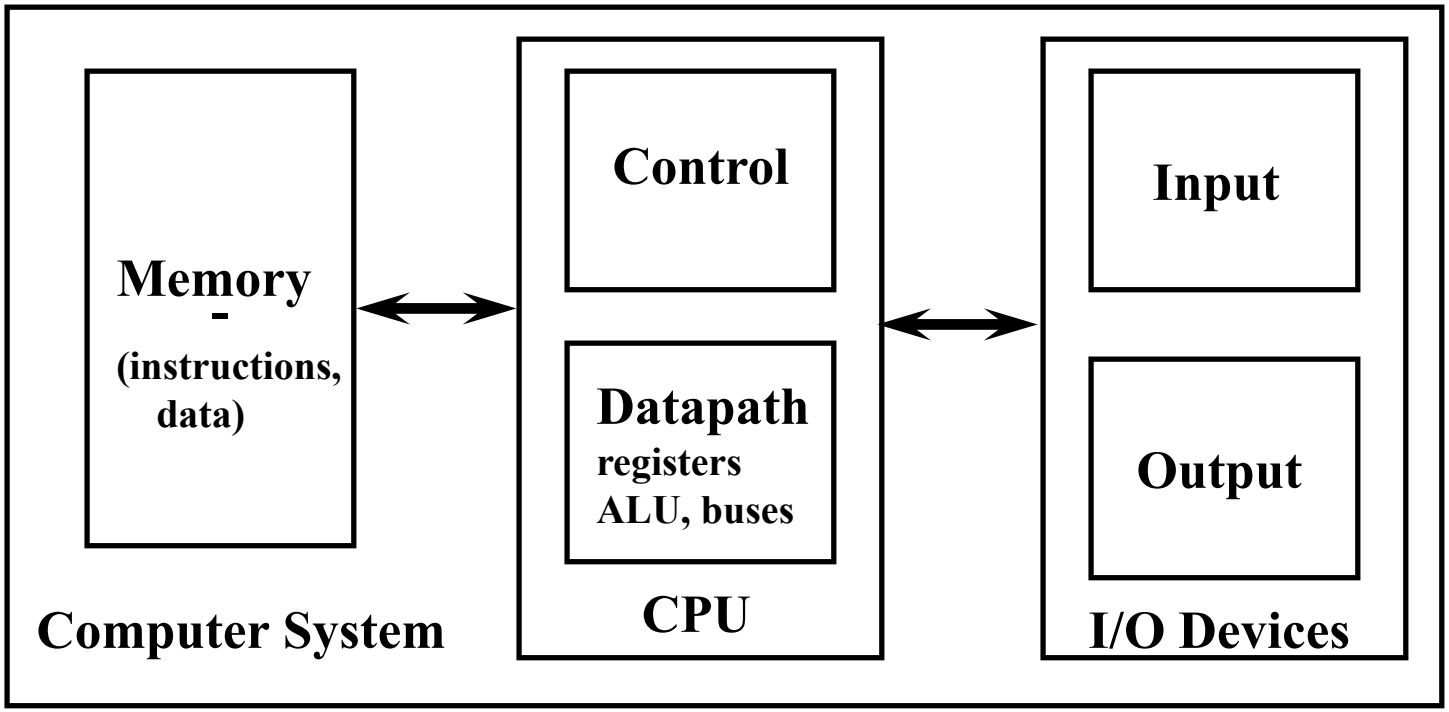
Common factor among all generations:
All target The Von Neumann Computer Model or paradigm

The Von Neumann Computer Model

- **Partitioning of the programmable computing engine into components:**
  - **Central Processing Unit (CPU):** Control Unit (instruction decode , sequencing of operations), Datapath (registers, arithmetic and logic unit, buses).
  - **Memory:** Instruction and operand storage.
  - **Input/Output (I/O) sub-system:** I/O bus, interfaces, devices.
  - **The stored program concept:** Instructions from an instruction set are fetched from a common memory and executed one at a time

AKA Program Counter
PC-Based Architecture

The Program Counter (PC) points to next instruction to be processed

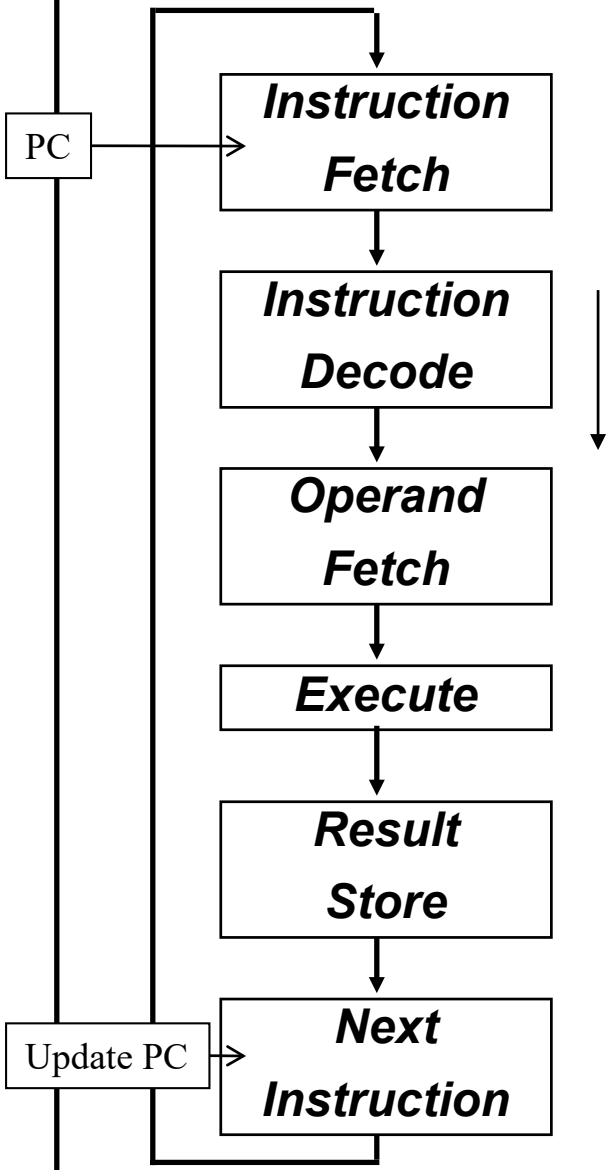


Major CPU Performance Limitation: The Von Neumann computing model implies sequential execution one instruction at a time

Another Performance Limitation: Separation of CPU and memory (The Von Neumann memory bottleneck)

Generic CPU Machine Instruction Processing Steps

(Implied by The Von Neumann Computer Model)



Obtain instruction from program storage

The Program Counter (PC) points to next instruction to be processed

Determine required actions and instruction size

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor or next instruction

(i.e Update PC)

Major CPU Performance Limitation: The Von Neumann computing model implies sequential execution one instruction at a time

CPU Organization (Design)

- **Datapath Design:**

  Components & their connections needed by ISA instructions

  - Capabilities & performance characteristics of principal Functional Units (FUs):
    - (e.g., Registers, ALU, Shifters, Logic Units, ...)
  - Ways in which these components are interconnected (buses connections, multiplexors, etc.).
  - How information flows between components.

- **Control Unit Design:**

  Control/sequencing of operations of datapath components to realize ISA instructions

  - Logic and means by which such information flow is controlled.
  - Control and coordination of FUs operation to realize the targeted Instruction Set Architecture to be implemented (can either be implemented using a finite state machine or a microprogram).

- Description of hardware operations with a suitable language, possibly using Register Transfer Notation (RTN).

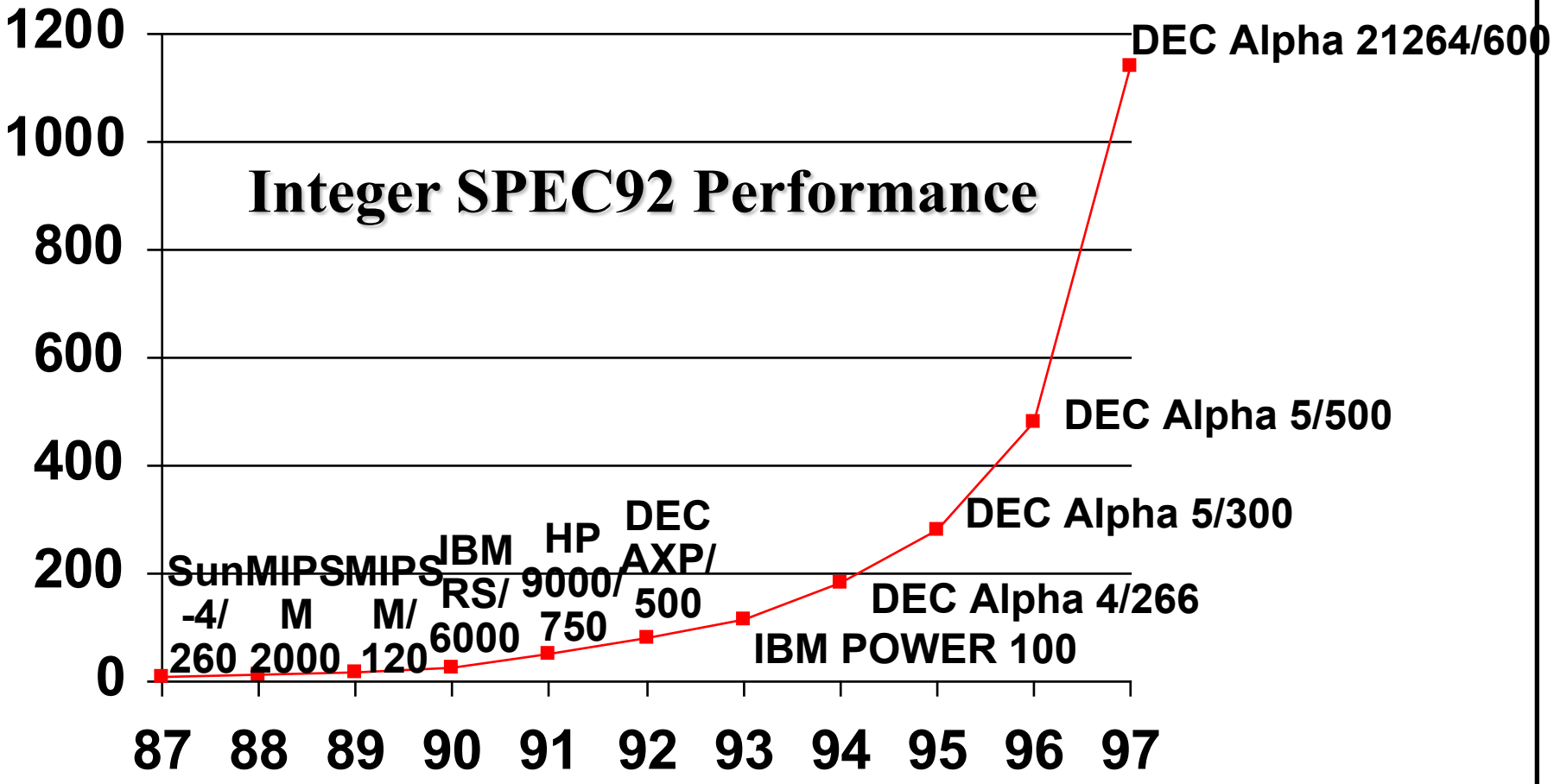
(From 350)

Recent Trends in Computer Design

- The cost/performance ratio of computing systems have seen a steady decline due to advances in:
  - Integrated circuit technology: *decreasing feature size,* λ
    - Clock rate improves roughly proportional to improvement in λ
    - Number of transistors improves proportional to λ^2 (or faster).
  - Architectural improvements in CPU design.
- Microprocessor systems directly reflect IC and architectural improvement in terms of a yearly 35 to 55% improvement in performance.
- Assembly language has been mostly eliminated and replaced by other alternatives such as C or C++
- Standard operating Systems (UNIX, Windows) lowered the cost of introducing new architectures.
- Emergence of RISC architectures and RISC-core (x86) architectures.
- Adoption of quantitative approaches to computer design based on empirical performance observations.
- Increased importance of exploiting thread-level parallelism (TLP) in main-stream computing systems.

e.g Multiple (2 to 8) processor cores on a single chip (multi-core)

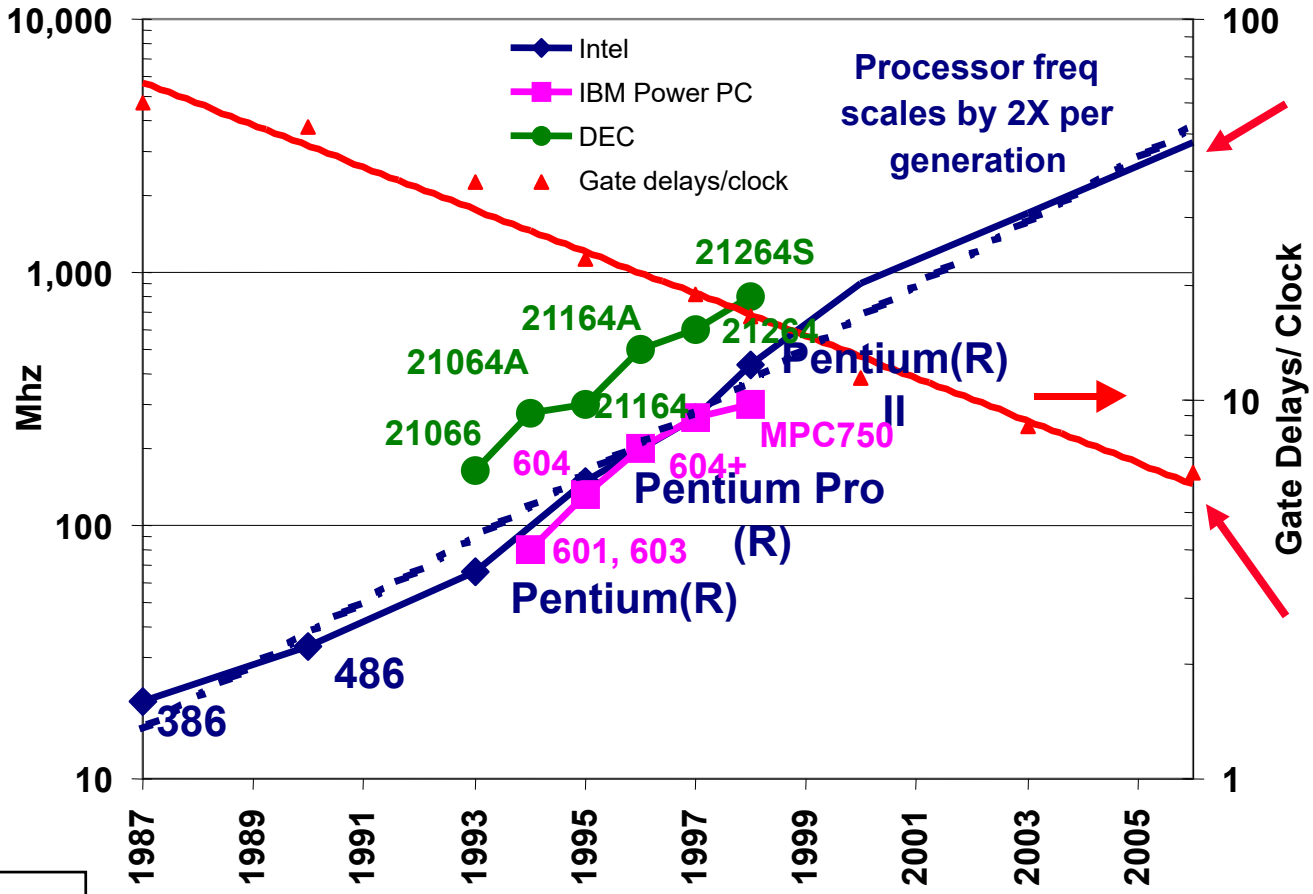
Microprocessor Performance 1987-97



> 100x performance increase in the last decade

T = I x CPI x C

Microprocessor Frequency Trend



Realty Check:
Clock frequency scaling is slowing down!
(Did silicone finally hit the wall?)

Why?
1- Static power leakage
2- Clock distribution delays

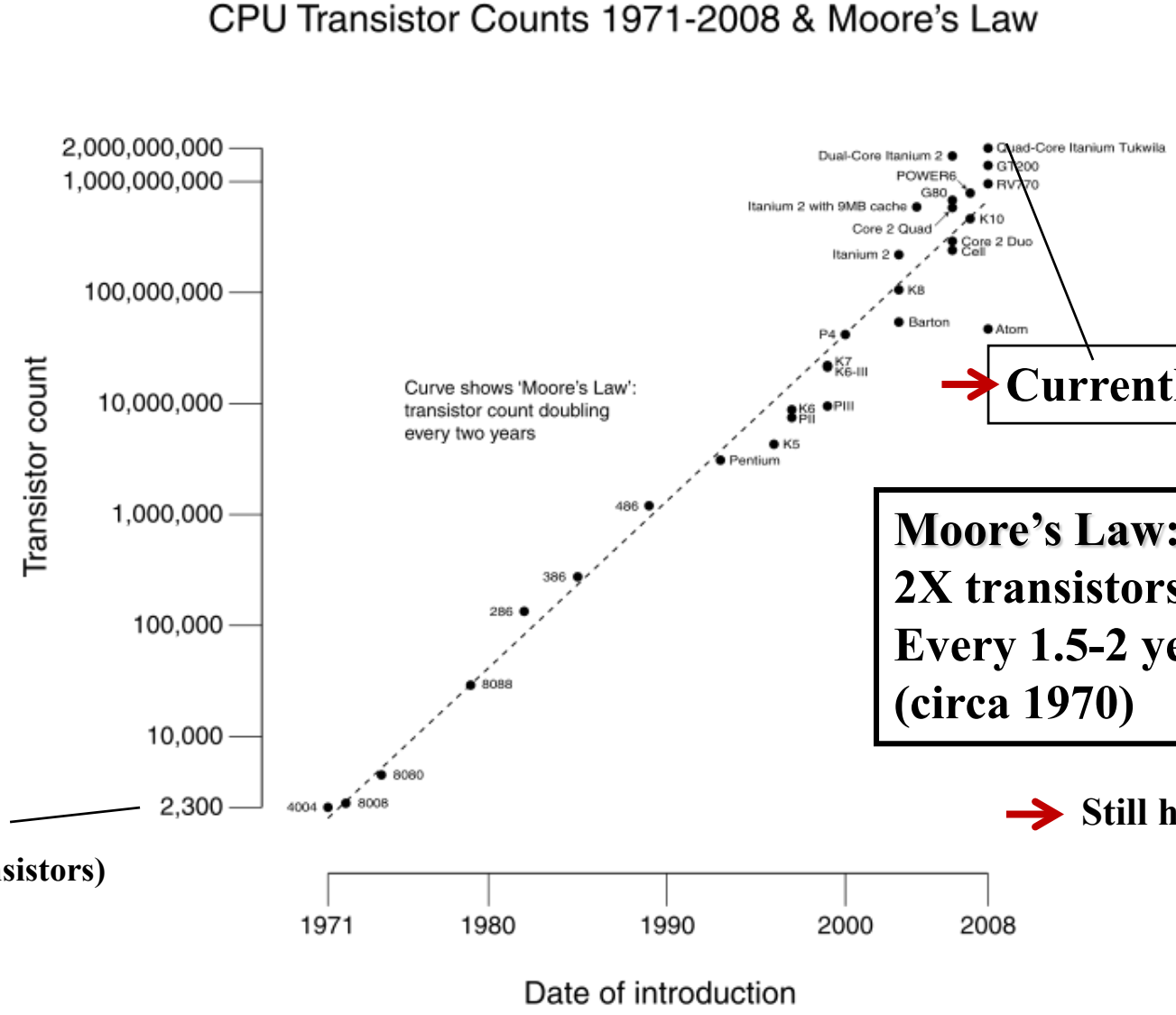
Result:
Deeper Pipelines
Longer stalls
Higher CPI
(lowers effective performance per cycle)

No longer the case

1. Frequency doubles each generation ?
2. Number of gates/clock reduce by 25%
3. Leads to deeper pipelines with more stages
   (e.g Intel Pentium 4E has 30+ pipeline stages)

$$T = I \times CPI \times C$$

Microprocessor Transistor Count Growth Rate



Currently ~ 9 Billion

Moore's Law:
2X transistors/Chip
Every 1.5-2 years
(circa 1970)

Still holds today

4-Bit
Intel 4004
(2300 transistors)

~ 4,000,000x transistor density increase in the last 50 years

Computer Technology Trends: *Evolutionary but Rapid Change*

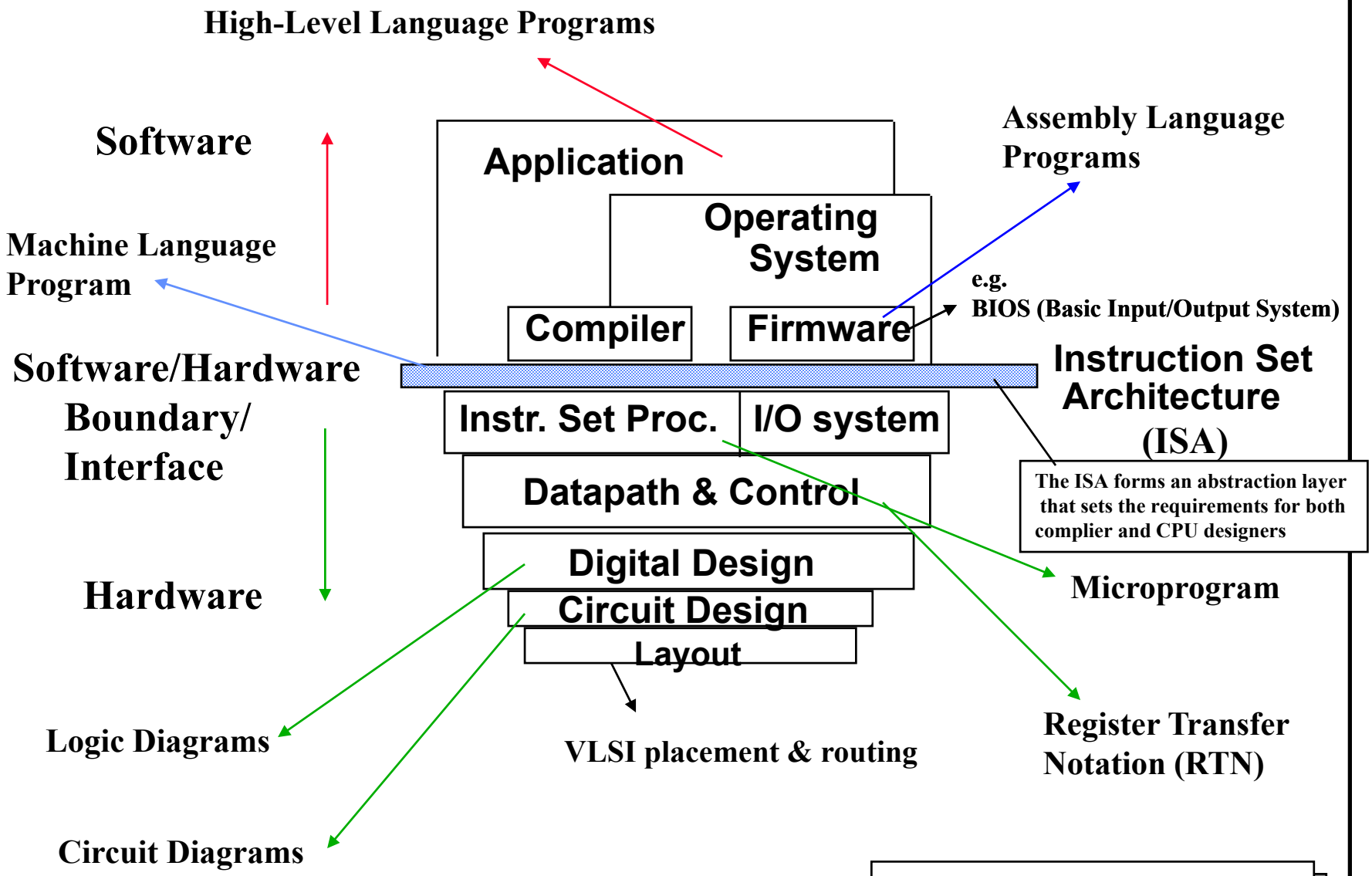
- **Processor:**
  - 1.5-1.6 performance improvement every year; Over 100X performance in last decade.
- **Memory:** Main or System Memory
  - DRAM capacity: > 2x every 1.5 years; 1000X size in last decade.
  - Cost per bit: Improves about 25% or more per year.
  - Only 15-25% performance improvement per year.
- **Disk:** i.e. I/O Subsystem
  - Capacity: > 2X in size every 1.5 years.
  - Cost per bit: Improves about 60% per year.
  - 200X size in last decade.
  - Only 10% performance improvement per year, due to mechanical limitations.

Performance gap compared to CPU performance causes system performance bottlenecks

- **State-of-the-art PC First Quarter 2023 :**
  - Processor clock speed: ~ 4000 MegaHertz (4 Giga Hertz)
  - Memory capacity: ~ 16000 MegaByte (16 Giga Bytes)
  - Disk capacity: ~ 8000 GigaBytes (8 Tera Bytes)

With 2-32 processor cores on a single chip

Hierarchy of Computer Architecture



ISA Requirements → Processor Design

Computer Architecture Vs. Computer Organization

- The term Computer architecture is sometimes erroneously restricted to computer instruction set (ISA) design, with other aspects of computer design called implementation
- More accurate definitions:
  - Instruction set architecture (ISA): The actual programmer-visible instruction set and serves as the boundary between the software and hardware.
  - Implementation of a machine has two components:
    - Organization: includes the high-level aspects of a computer's design such as: The memory system, the bus structure, the internal CPU unit which includes implementations of arithmetic, logic, branching, and data transfer operations.
    - Hardware: Refers to the specifics of the machine such as detailed logic design and packaging technology.
- In general, Computer Architecture refers to the above three aspects:

  Instruction set architecture, organization, and hardware.

The ISA forms an abstraction layer that sets the requirements for both complier and CPU designers

CPU Micro-architecture (CPU design)

Hardware design and implementation

The Task of A Computer Designer

- Determine what attributes that are important to the design of the new machine (CPU).
- Design a machine to maximize performance while staying within cost and other constraints and metrics.
- It involves more than instruction set design.
  1. – Instruction set architecture. (ISA)
  2. – CPU Micro-architecture (CPU design).
  3. – Implementation.
- Implementation of a machine has two components:
  - Organization.
  - Hardware.

e.g
Power consumption
Heat dissipation
Real-time constraints

Recent Architectural Improvements

- Long memory latency-hiding techniques, including:
  - Increased optimization and utilization of multi-level cache systems.
- Improved handling of pipeline hazards.
- Improved hardware branch prediction techniques.
- Optimization of pipelined instruction execution:
  - Dynamic hardware-based pipeline scheduling.
  - Dynamic speculative execution.

AKA Out-of-Order Execution

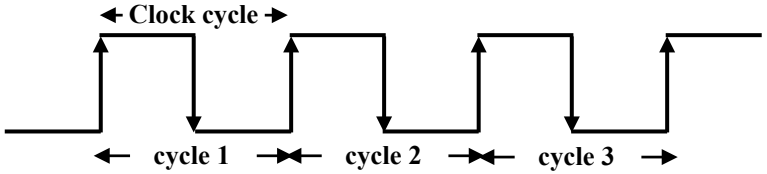
- Exploiting Instruction-Level Parallelism (ILP) in terms of multiple-instruction issue and multiple hardware functional units.
- Inclusion of special instructions to handle multimedia applications (limited vector processing).
- High-speed bus/interconnects designs to improve data transfer rates.

  - Also, increased utilization of point-to-point interconnects instead of one system bus (e.g. HyperTransport)

CPU Performance Evaluation: Cycles Per Instruction (CPI)

- **Most computers run synchronously utilizing a CPU clock running at a constant clock rate:**

where: f **Clock rate = 1 / clock cycle** C



- **The CPU clock rate depends on the specific CPU organization (design) and hardware implementation technology (VLSI) used**
- **A computer machine (ISA) instruction is comprised of a number of elementary or micro operations which vary in number and complexity depending on the instruction and the exact CPU organization (Design)**
  - **A micro operation is an elementary hardware operation that can be performed during one CPU clock cycle.**
  - **This corresponds to one micro-instruction in microprogrammed CPUs.**
  - **Examples: register operations: shift, load, clear, increment, ALU operations: add , subtract, etc.**

CPI

- **Thus a single machine/ISA instruction may take one or more CPU cycles to complete termed as the Cycles Per Instruction (CPI).**

CPI = 1/IPC

- **Average CPI of a program: The average CPI of all instructions executed in the program on a given CPU design.**

(From 350)

Instructions Per Cycle = IPC = 1/CPI

Computer Performance Measures: Program Execution Time

- For a specific program compiled to run on a specific machine (CPU) "A", the following parameters are provided:
  - The total instruction count of the program. (executed) **I**
  - The average number of cycles per instruction (average CPI). **CPI**
  - Clock cycle of machine "A" **C**

I =Dynamic instruction count executed

- How can one measure the performance of this machine running this program?
  - Intuitively the machine or CPU is said to be faster or has better performance running this program if the total execution time is shorter.
  - Thus the inverse of the total measured program execution time is a possible performance measure or metric:

$$\text{Performance}_A = 1 \;/\; \text{Execution Time}_A$$

How to compare performance of different machines or CPUs?

What factors affect performance? How to improve performance?

(From 350)

Comparing Computer Performance Using Execution Time

- To compare the performance of two machines (or CPUs) "A", "B" running a given specific program:

$$\text{Performance}_A = 1 / \text{Execution Time}_A$$
$$\text{Performance}_B = 1 / \text{Execution Time}_B$$

- Machine A is n times faster than machine B means (or slower? if n < 1) :

$$\text{Speedup} = n = \frac{\text{Performance}_A}{\text{Performance}_B} = \frac{\text{Execution Time}_B}{\text{Execution Time}_A}$$

- Example: (i.e Speedup is ratio of performance, no units)

For a given program:

Execution time on machine A: $\text{Execution}_A = 1$ second

Execution time on machine B: $\text{Execution}_B = 10$ seconds

Speedup= $\text{Performance}_A / = \text{Execution Time}_B / \text{Execution Time}_A$

$\text{Performance}_B = 10 / 1 = 10$

The performance of machine A is 10 times the performance of machine B when running this program, or: Machine A is said to be 10 times faster than machine B when running this program.

(From 350) The two CPUs may target different ISAs provided the program is written in a high level language (HLL)

CPU Execution Time: The CPU Equation

- A program is comprised of a number of instructions executed , I
  - Measured in: instructions/program

I =Dynamic instruction count executed

- The average instruction executed takes a number of *cycles per instruction (CPI)* to be completed.
  - Measured in: cycles/instruction, CPI

Or Instructions Per Cycle (IPC): IPC= 1/CPI

- CPU has a fixed clock cycle time C = 1/clock rate
  - Measured in: seconds/cycle

C = 1/f

- CPU execution time is the product of the above three parameters as follows:

Executed

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

$$\rightarrow T = I \times CPI \times C$$

- T: execution Time per program in seconds
- I: Number of instructions executed
- CPI: Average CPI for program
- C: CPU Clock Cycle

(This equation is commonly known as the CPU performance equation)

(From 350)

CPU Execution Time: Example

- **A Program is running on a specific machine (CPU) with the following parameters:**
  - **Total executed instruction count: 10,000,000 instructions**
    Average CPI for the program: 2.5 cycles/instruction.
  - **CPU clock rate: 200 MHz.** (clock cycle = 5×10^{-9} seconds)
- **What is the execution time for this program:**

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

CPU time = Instruction count x CPI x Clock cycle

= 10,000,000 x 2.5 x 1 / clock rate

= 10,000,000 x 2.5 x 5×10^{-9}

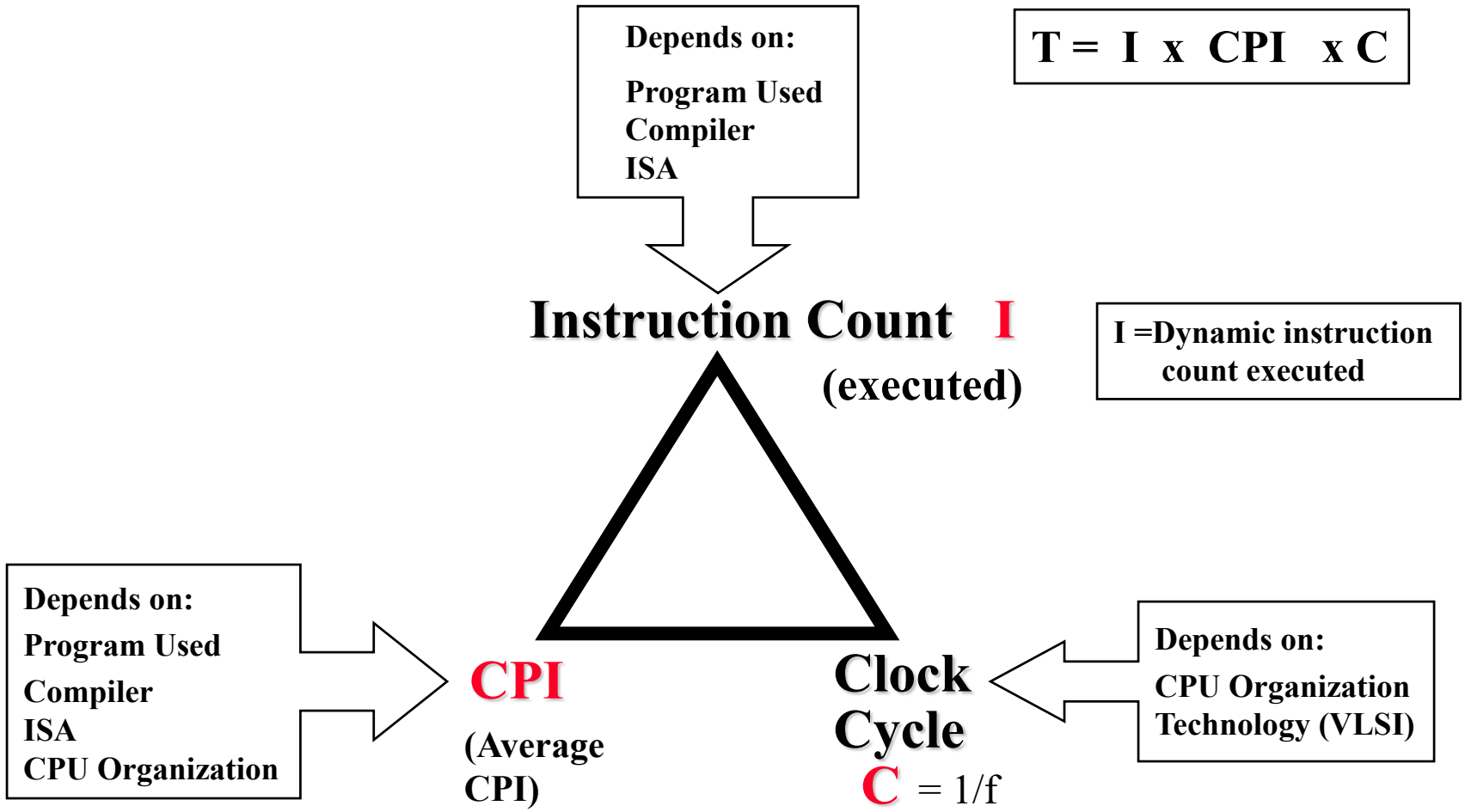
= .125 seconds

(From 350)

T = I x CPI x C

Aspects of CPU Execution Time

CPU Time = Instruction count x CPI x Clock cycle



(From 350)

Factors Affecting CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Instruction Count I	Average CPI	Clock Cycle C
Program	X	X	
Compiler	X	X	
Instruction Set Architecture (ISA)	X	X	
Organization (CPU Design)		X	X
Technology (VLSI)			X

(From 350)

$$T = I \times CPI \times C$$

Performance Comparison: Example

- From the previous example: A Program is running on a specific machine with the following parameters:
  - Total executed instruction count, I: 10,000,000 instructions
  - Average CPI for the program: 2.5 cycles/instruction.
  - CPU clock rate: 200 MHz.
- Using the same program with these changes:
  - A new compiler used: New instruction count executed I: 9,500,000
    New CPI: 3.0
  - Faster CPU implementation: New clock rate = 300 MHZ
- What is the speedup with the changes?

$$\text{Speedup} = \frac{\text{Old Execution Time}}{\text{New Execution Time}} = \frac{I_{old} \times CPI_{old} \times \text{Clock cycle}_{old}}{I_{new} \times CPI_{new} \times \text{Clock Cycle}_{new}}$$

$$\text{Speedup} = (10{,}000{,}000 \times 2.5 \times 5\times10^{-9}) / (9{,}500{,}000 \times 3 \times 3.33\times10^{-9})$$

$$= .125 / .095 = 1.32$$

or 32 % faster after changes.

(From 350)

Clock Cycle = 1/ Clock Rate

Instruction Types & Average CPI

- **Given a program with *n* types or classes of instructions executed on a given CPU design with the following characteristics:**

C_i = Count of instructions of type$_i$ **Executed**

CPI_i = Cycles per instruction for type$_i$

i = 1, 2, …. n

Then:

CPI = CPU Clock Cycles / Instruction Count I

Executed

Where:

$$CPU\ clock\ cycles = \sum_{i=1}^{n} \left(CPI_i \times C_i \right)$$

Executed

Instruction Count I = $\Sigma\ C_i$

i.e. Average or effective CPI

T = I x CPI x C

(From 350)

Instruction Types & CPI: An Example

- An instruction set has three instruction classes:

Instruction class	CPI
A	1
B	2
C	3

For a specific CPU design

- Two code sequences have the following instruction counts:

	Instruction counts for instruction class		
Code Sequence	A	B	C
1	2	1	2
2	4	1	1

- CPU cycles for sequence 1 = 2 x 1 + 1 x 2 + 2 x 3 = 10 cycles

  CPI for sequence 1 = clock cycles / instruction count

  = 10 /5 = 2

- CPU cycles for sequence 2 = 4 x 1 + 1 x 2 + 1 x 3 = 9 cycles

  CPI for sequence 2 = 9 / 6 = 1.5

$$CPU\ clock\ cycles = \sum_{i=1}^{n} (CPI_i \times C_i)$$

CPI = CPU Cycles / I

(From 350)

Instruction Frequency & Average CPI

Or Fractions

- **Given a program with *n* types or classes of instructions with the following characteristics:**

C_i = Count of instructions of type$_i$

CPI_i = Average cycles per instruction of type$_i$

F_i = Frequency or fraction of instruction type$_i$ executed

= C_i/ total executed instruction count = C_i/ I

i = 1, 2, …. n

Then:

Where: Executed Instruction Count I = ΣC_i

$$CPI = \sum_{i=1}^{n} \left(CPI_i \times F_i \right)$$

i.e average or effective CPI

Fraction of total execution time for instructions of type i = $\frac{CPI_i \times F_i}{CPI}$

Instruction Type Frequency & CPI: A RISC Example

Program Profile or Executed Instructions Mix

$\frac{CPI_i \times F_i}{CPI}$

Base Machine (Reg / Reg)

Given

Op	Freq, F_i	CPI_i	$CPI_i \times F_i$	% Time
ALU	50%	1	.5	23% = .5/2.2
Load	20%	5	1.0	45% = 1/2.2
Store	10%	3	.3	14% = .3/2.2
Branch	20%	2	.4	18% = .4/2.2
			CPI = Sum = 2.2	

Typical Mix

Average

$$CPI = \sum_{i=1}^{n} (CPI_i \times F_i)$$

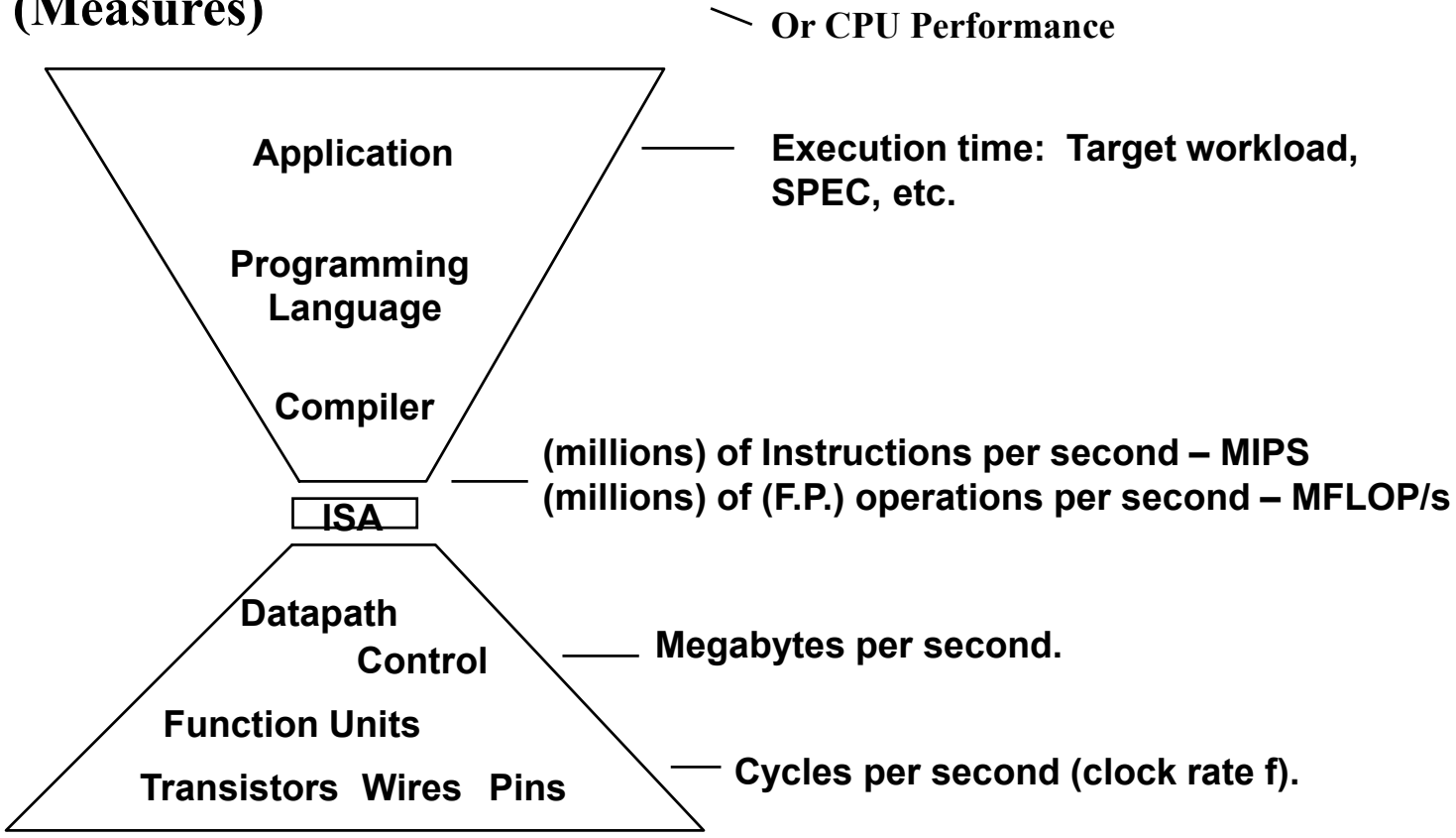
$$CPI = .5 \times 1 + .2 \times 5 + .1 \times 3 + .2 \times 2 = 2.2$$

$$= .5 + 1 + .3 + .4$$

(From 350)

Metrics of Computer Performance

(Measures)



Each metric has a purpose, and each can be misused.

Choosing Programs To Evaluate Performance

Levels of programs or benchmarks that could be used to evaluate performance:

- **Actual Target Workload:** Full applications that run on the target machine.

- **Real Full Program-based Benchmarks:**
  - Select a specific mix or suite of programs that are "typical" of targeted applications or workload (e.g SPEC95, SPEC CPU2000).

- **Small "Kernel" Benchmarks:** Also called synthetic benchmarks
  - Key computationally-intensive pieces extracted from real programs.
    - Examples: Matrix factorization, FFT, tree search, etc.
  - Best used to test specific aspects of the machine.

- **Microbenchmarks:**
  - Small, specially written programs to isolate a specific aspect of performance characteristics: Processing: integer, floating point, local memory, input/output, etc.

SPEC: System Performance Evaluation Corporation

The most popular and industry-standard set of CPU benchmarks.

Target Programs application domain: Engineering and scientific computation

- **SPECmarks, 1989:**
  - 10 programs yielding a single number ("SPECmarks").
- **SPEC92, 1992:**
  - SPECInt92 (6 integer programs) and SPECfp92 (14 floating point programs).
- **SPEC95, 1995:**
  - SPECint95 (8 integer programs):
    - go, m88ksim, gcc, compress, li, ijpeg, perl, vortex
  - SPECfp95 (10 floating-point intensive programs):
    - tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fppp, wave5
  - Performance relative to a Sun SuperSpark I (50 MHz) which is given a score of SPECint95 = SPECfp95 = 1
- **SPEC CPU2000, 1999:**
  - CINT2000 (11 integer programs). CFP2000 (14 floating-point intensive programs)
  - Performance relative to a Sun Ultra5_10 (300 MHz) which is given a score of SPECint2000 = SPECfp2000 = 100
- **SPEC CPU2006, 2006:**
  - CINT2006 (12 integer programs). CFP2006 (17 floating-point intensive programs)
  - Performance relative to a Sun Ultra Enterprise 2 workstation with a 296-MHz UltraSPARC II processor which is given a score of SPECint2006 = SPECfp2006 = 1

All based on execution time and give speedup over a reference CPU

SPEC CPU2000 Programs

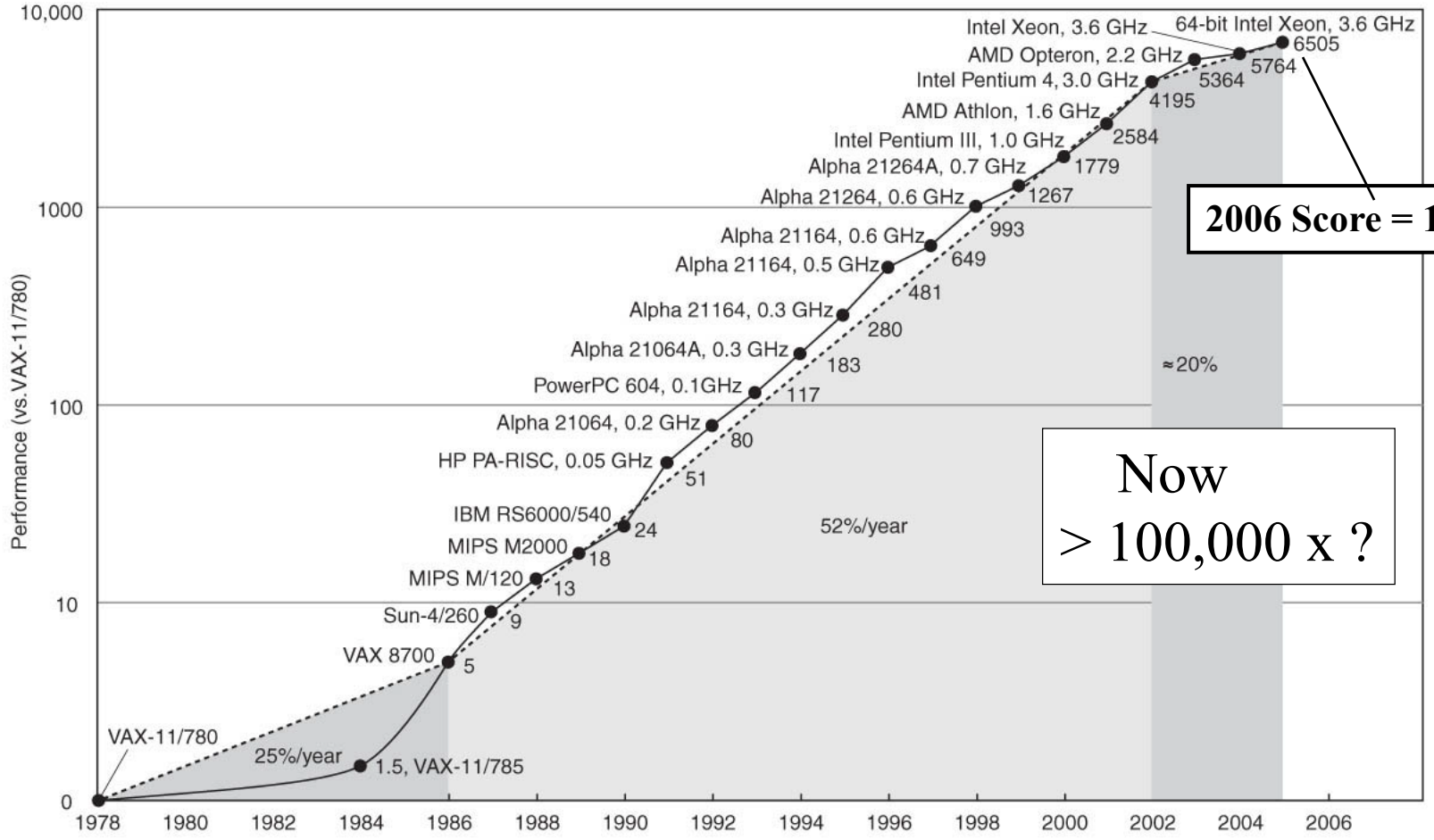
	Benchmark	Language	Descriptions
CINT2000 (Integer)	**164.gzip**	**C**	**Compression**
	175.vpr	**C**	**FPGA Circuit Placement and Routing**
	176.gcc	**C**	**C Programming Language Compiler**
	181.mcf	**C**	**Combinatorial Optimization**
	186.crafty	**C**	**Game Playing: Chess**
	197.parser	**C**	**Word Processing**
	252.eon	**C++**	**Computer Visualization**
	253.perlbmk	**C**	**PERL Programming Language**
	254.gap	**C**	**Group Theory, Interpreter**
	255.vortex	**C**	**Object-oriented Database**
	256.bzip2	**C**	**Compression**
	300.twolf	**C**	**Place and Route Simulator**
CFP2000 (Floating Point)	**168.wupwise**	**Fortran 77**	**Physics / Quantum Chromodynamics**
	171.swim	**Fortran 77**	**Shallow Water Modeling**
	172.mgrid	**Fortran 77**	**Multi-grid Solver: 3D Potential Field**
	173.applu	**Fortran 77**	**Parabolic / Elliptic Partial Differential Equations**
	177.mesa	**C**	**3-D Graphics Library**
	178.galgel	**Fortran 90**	**Computational Fluid Dynamics**
	179.art	**C**	**Image Recognition / Neural Networks**
	183.equake	**C**	**Seismic Wave Propagation Simulation**
	187.facerec	**Fortran 90**	**Image Processing: Face Recognition**
	188.ammp	**C**	**Computational Chemistry**
	189.lucas	**Fortran 90**	**Number Theory / Primality Testing**
	191.fma3d	**Fortran 90**	**Finite-element Crash Simulation**
	200.sixtrack	**Fortran 77**	**High Energy Nuclear Physics Accelerator Design**
	301.apsi	**Fortran 77**	**Meteorology: Pollutant Distribution**

Programs application domain: Engineering and scientific computation

Source: http://www.spec.org/osg/cpu2000/

Integer SPEC CPU2000 Microprocessor Performance 1978-2006

Performance relative to VAX 11/780 (given a score = 1)



1978 Score = 1

T = I x CPI x C

Top 20 SPEC CPU2000 Results (As of October 2006)

Top 20 SPECint2000

#	MHz	Processor	int peak	int base
1	2933	Core 2 Duo EE X86	3119	3108
2	3000	Xeon 51xx	3102	3089
3	2666	Core 2 Duo	2848	2844
4	2660	Xeon 30xx	2835	2826
5	3000	Opteron	2119	1942
6	2800	Athlon 64 FX	2061	1923
7	2800	Opteron AM2	1960	1749
8	2300	POWER5+	1900	1820
9	3733	Pentium 4 E	1872	1870
10	3800	Pentium 4 Xeon X86	1856	1854
11	2260	Pentium M	1839	1812
12	3600	Pentium D	1814	1810
13	2167	Core Duo	1804	1796
14	3600	Pentium 4	1774	1772
15	3466	Pentium 4 EE	1772	1701
16	2700	PowerPC 970MP	1706	1623
17	2600	Athlon 64	1706	1612
18	2000	Pentium 4 Xeon LV	1668	1663
19	2160	SPARC64 V	1620	1501
20	1600	Itanium 2	1590	1590

Top 20 SPECfp2000

MHz	Processor	fp peak	fp base
2300	POWER5+ RISC	3642	3369
1600	DC Itanium 2	3098	3098
3000	Xeon 51xx	3056	2811
2933	Core 2 Duo EE	3050	3048
2660	Xeon 30xx	3044	2763
1600	Itanium 2	3017	3017
2667	Core 2 Duo	2850	2847
1900	POWER5	2796	2585
3000	Opteron CISC	2497	2260
2800	Opteron AM2	2462	2230
3733	Pentium 4 E	2283	2280
2800	Athlon 64 FX	2261	2086
2700	PowerPC 970MP	2259	2060
2160	SPARC64 V	2236	2094
3730	Pentium 4 Xeon	2150	2063
3600	Pentium D	2077	2073
3600	Pentium 4	2015	2009
2600	Athlon 64	1829	1700
1700	POWER4+	1776	1642
3466	Pentium 4 EE	1724	1719

Performance relative to a Sun Ultra5_10 (300 MHz) which is given a score of SPECint2000 = SPECfp2000 = 100

Source: http://www.aceshardware.com/SPECmine/top.jsp

SPEC CPU2006 Programs

	Benchmark	Language	Descriptions
CINT2006 (Integer) 12 programs	400.perlbench	C	PERL Programming Language
	401.bzip2	C	Compression
	403.gcc	C	C Compiler
	429.mcf	C	Combinatorial Optimization
	445.gobmk	C	Artificial Intelligence: go
	456.hmmer	C	Search Gene Sequence
	458.sjeng	C	Artificial Intelligence: chess
	462.libquantum	C	Physics: Quantum Computing
	464.h264ref	C	Video Compression
	471.omnetpp	C++	Discrete Event Simulation
	473.astar	C++	Path-finding Algorithms
	483.Xalancbmk	C++	XML Processing
CFP2006 (Floating Point) 17 programs	410.bwaves	Fortran	Fluid Dynamics
	416.gamess	Fortran	Quantum Chemistry
	433.milc	C	Physics: Quantum Chromodynamics
	434.zeusmp	Fortran	Physics/CFD
	435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
	436.cactusADM	C/Fortran	Physics/General Relativity
	437.leslie3d	Fortran	Fluid Dynamics
	444.namd	C++	Biology/Molecular Dynamics
	447.dealII	C++	Finite Element Analysis
	450.soplex	C++	Linear Programming, Optimization
	453.povray	C++	Image Ray-tracing
	454.calculix	C/Fortran	Structural Mechanics
	459.GemsFDTD	Fortran	Computational Electromagnetics
	465.tonto	Fortran	Quantum Chemistry
	470.lbm	C	Fluid Dynamics
	481.wrf	C/Fortran	Weather Prediction
	482.sphinx3	C	Speech recognition

Target Programs application domain: Engineering and scientific computation

Source: http://www.spec.org/cpu2006/

Example Integer SPEC CPU2006 Performance Results

For 2.5 GHz AMD Opteron X4 model 2356 (Barcelona)

I x CPI x C = T

Base Machine Time

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^9)	Execution Time (seconds)	Reference Time (seconds)	Speedup SPEC Ratio
Interpreted string processing	perl	2,118	0.75	0.4	637	9,770	15.3
Block-sorting compression	bzip2	2,389	0.85	0.4	817	9,650	11.8
GNU C compiler	gcc	1,050	1.72	0.4	724	8,050	11.1
Combinatorial optimization	mcf	336	10.00	0.4	1,345	9,120	6.8
Go game (AI)	go	1,658	1.09	0.4	721	10,490	14.6
Search gene sequence	hmmer	2,783	0.80	0.4	890	9,330	10.5
Chess game (AI)	sjeng	2,176	0.96	0.4	837	12,100	14.5
Quantum computer simulation	libquantum	1,623	1.61	0.4	1,047	20,720	19.8
Video compression	h264avc	3,102	0.80	0.4	993	22,130	22.3
Discrete event simulation library	omnetpp	587	2.94	0.4	690	6,250	9.1
Games/path finding	astar	1,082	1.79	0.4	773	7,020	9.1
XML parsing	xalancbmk	1,058	2.70	0.4	1,143	6,900	6.0
Geometric Mean							11.7

Performance relative to a Sun Ultra Enterprise 2 workstation with a 296-MHz UltraSPARC II processor which is given a score of SPECint2006 = SPECfp2006 = 1

$T = I \times CPI \times C$

Computer Performance Measures : MIPS (Million Instructions Per Second) Rating

- For a specific program running on a specific CPU the MIPS rating is a measure of how many millions of instructions are executed per second:

MIPS Rating = Instruction count / (Execution Time x 10^6)

= Instruction count / (CPU clocks x Cycle time x 10^6)

= (Instruction count x Clock rate) / (Instruction count x CPI x 10^6)

➔ = Clock rate / (CPI x 10^6)

- Major problem with MIPS rating: As shown above the MIPS rating does not account for the count of instructions executed (I).
  - A higher MIPS rating in many cases may not mean higher performance or better execution time. i.e. due to compiler design variations.
- In addition the MIPS rating:
  - Does not account for the instruction set architecture (ISA) used.
    - Thus it cannot be used to compare computers/CPUs with different instruction sets.
  - Easy to abuse: Program used to get the MIPS rating is often omitted.
    - Often the Peak MIPS rating is provided for a given CPU which is obtained using a program comprised entirely of instructions with the lowest CPI for the given CPU design which does not represent real programs.

(From 350)

$T = I \times CPI \times C$

Computer Performance Measures : MIPS (Million Instructions Per Second) Rating

- **Under what conditions can the MIPS rating be used to compare performance of different CPUs?**

- **The MIPS rating is <u>only valid</u> to compare the performance of different CPUs <u>provided that the following conditions are satisfied</u>:**

  1 **<u>The same high level language program is used</u>**

  (actually this applies to all performance metrics)

  2 **<u>The same ISA is used</u>**

  3 **<u>The same compiler is used</u>**

  ⇒ (Thus the resulting programs used to run on the CPUs and obtain the MIPS rating are <u>identical</u> at the machine code (binary) level including the <u>same instruction count</u>)

  ↖ I

(From 350)

Compiler Variations, MIPS, Performance: An Example

- For the machine (CPU) with instruction classes:

Instruction class	CPI
A	1
B	2
C	3

- For a given high level language (HLL) program two compilers produced the following instruction counts:

	Instruction counts (in millions) for each instruction class		
Code from:	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

- The machine is assumed to run at a clock rate of 100 MHz

(From 350)

Compiler Variations, MIPS, Performance: An Example (Continued)

MIPS = Clock rate / (CPI x 10^6) = 100 MHz / (CPI x 10^6)

CPI = CPU execution cycles / Instructions count

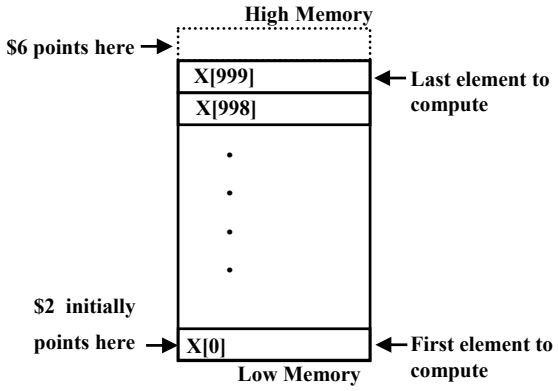
$$CPU\ clock\ cycles = \sum_{i=1}^{n} \left(CPI_i \times C_i \right)$$

CPU time = Instruction count x CPI / Clock rate

- For compiler 1:
  - CPI_1 = (5 x 1 + 1 x 2 + 1 x 3) / (5 + 1 + 1) = 10 / 7 = 1.43
  - MIP $Rating_1$ = 100 / (1.428 x 10^6) = 70.0
  - CPU $time_1$ = ((5 + 1 + 1) x 10^6 x 1.43) / (100 x 10^6) = 0.10 seconds
- For compiler 2:
  - CPI_2 = (10 x 1 + 1 x 2 + 1 x 3) / (10 + 1 + 1) = 15 / 12 = 1.25
  - MIPS $Rating_2$ = 100 / (1.25 x 10^6) = 80.0
  - CPU $time_2$ = ((10 + 1 + 1) x 10^6 x 1.25) / (100 x 10^6) = 0.15 seconds

MIPS rating indicates that compiler 2 is better while in reality the code produced by compiler 1 is faster

MIPS32 (The ISA not the metric) Loop Performance Example



For the loop:

for (i=0; i<1000; i=i+1){

x[i] = x[i] + s; }

MIPS32 assembly code is given by:

```
        lw     $3,  8($1)        ; load s in $3
        addi   $6,  $2,  4000    ; $6 = address of last element + 4
loop:   lw     $4,  0($2)        ; load x[i] in $4
        add    $5,  $4, $3       ; $5 has x[i] + s
        sw     $5,  0($2)        ; store computed x[i]
        addi   $2,  $2,  4       ; increment $2 to point to next x[ ] element
        bne    $6,  $2,  loop    ; last loop iteration reached?
```

The MIPS code is executed on a specific CPU that runs at 500 MHz (clock cycle = 2ns = 2×10^{-9} seconds) with following instruction type CPIs :

Instruction type	CPI
ALU	4
Load	5
Store	7
Branch	3

For this MIPS code running on this CPU find:

1- Fraction of total instructions executed for each instruction type
2- Total number of CPU cycles
3- Average CPI
4- Fraction of total execution time for each instructions type
5- Execution time
6- MIPS rating , peak MIPS rating for this CPU

X[ ] array of words in memory, base address in $2 ,
s a constant word value in memory, address in $1

From 350

MIPS32 (The ISA) Loop Performance Example (continued)

- The code has 2 instructions before the loop and 5 instructions in the body of the loop which iterates 1000 times,
- Thus: Total instructions executed, I = 5x1000 + 2 = 5002 instructions

1 Number of instructions executed/fraction F_i for each instruction type:

- ALU instructions = 1 + 2x1000 = 2001 $CPI_{ALU} = 4$ $Fraction_{ALU} = F_{ALU} = 2001/5002 = 0.4 = 40\%$
- Load instructions = 1 + 1x1000 = 1001 $CPI_{Load} = 5$ $Fraction_{Load} = F_{Load} = 1001/5002 = 0.2 = 20\%$
- Store instructions = 1000 $CPI_{Store} = 7$ $Fraction_{Store} = F_{Store} = 1000/5002 = 0.2 = 20\%$
- Branch instructions = 1000 $CPI_{Branch} = 3$ $Fraction_{Branch} = F_{Branch} = 1000/5002 = 0.2 = 20\%$

2 $$CPU\ clock\ cycles = \sum_{i=1}^{n} (CPI_i \times C_i)$$

$$= 2001x4 + 1001x5 + 1000x7 + 1000x3 = 23009 \text{ cycles}$$

3 Average CPI = CPU clock cycles / I = 23009/5002 = 4.6

Instruction type	CPI
ALU	4
Load	5
Store	7
Branch	3

4 Fraction of execution time for each instruction type:

- Fraction of time for ALU instructions = CPI_{ALU} x F_{ALU} / CPI= 4x0.4/4.6 = 0.348 = 34.8%
- Fraction of time for load instructions = CPI_{load} x F_{load} / CPI= 5x0.2/4.6 = 0.217 = 21.7%
- Fraction of time for store instructions = CPI_{store} x F_{store} / CPI= 7x0.2/4.6 = 0.304 = 30.4%
- Fraction of time for branch instructions = CPI_{branch} x F_{branch} / CPI= 3x0.2/4.6 = 0.13 = 13%

5 Execution time = I x CPI x C = CPU cycles x C = 23009 x $2x10^{-9}$ =

= $4.6x\ 10^{-5}$ seconds = 0.046 msec = 46 usec

6 MIPS rating = Clock rate / (CPI x 10^6) = 500 / 4.6 = 108.7 MIPS

- The CPU achieves its peak MIPS rating when executing a program that only has instructions of the type with the lowest CPI. In this case branches with $CPI_{Branch} = 3$
- Peak MIPS rating = Clock rate / (CPI_{Branch} x 10^6) = 500/3 = 166.67 MIPS

(From 350)

Computer Performance Measures :
MFLOPS (Million FLOating-Point Operations Per Second)

- A floating-point operation is an addition, subtraction, multiplication, or division operation applied to numbers represented by a single or a double precision floating-point representation.
- MFLOPS, for a specific program running on a specific computer, is a measure of millions of floating point-operation (megaflops) per second:

MFLOPS = Number of floating-point operations / (Execution time x 10^6)

- MFLOPS rating is a better comparison measure between different machines (applies even if ISAs are different) than the MIPS rating.
  - Applicable even if ISAs are different
- Program-dependent: Different programs have different percentages of floating-point operations present. i.e compilers have no floating- point operations and yield a MFLOPS rating of zero.
- Dependent on the type of floating-point operations present in the program.
  - Peak MFLOPS rating for a CPU: Obtained using a program comprised entirely of the simplest floating point instructions (with the lowest CPI) for the given CPU design which does not represent real floating point programs.

Current peak MFLOPS rating: 8,000-20,000 MFLOPS (8-20 GFLOPS) per processor core

(From 350)

Quantitative Principles of Computer Design

- **Amdahl's Law:**

The performance gain from improving some portion of a computer is calculated by:

i.e. using some enhancement

Or CPU

Or Program

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement}}{\text{Performance for the entire task without using the enhancement}}$$

$$\text{or Speedup} = \frac{\text{Execution time without the enhancement}}{\text{Execution time for entire task using the enhancement}}$$

(From 350)

Performance Enhancement Calculations: Amdahl's Law

- The overall performance enhancement possible due to a given design improvement is limited by the amount that the improved feature is used
- Amdahl's Law:

Performance improvement or speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without E}}{\text{Execution Time with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}$$

– Suppose that enhancement E accelerates a fraction F of the (original) execution time by a factor S and the remainder of the time (1-F) is unaffected then:

$$\text{Execution Time with E} = ((1-F) + F/S) \times \text{Execution Time without E}$$

Hence speedup is given by:

$$\text{Speedup(E)} = \frac{\text{Execution Time without E}}{((1 - F) + F/S) \times \text{Execution Time without E}} = \frac{1}{(1 - F) + F/S}$$

F (Fraction of execution time enhanced) refers to original execution time before the enhancement is applied

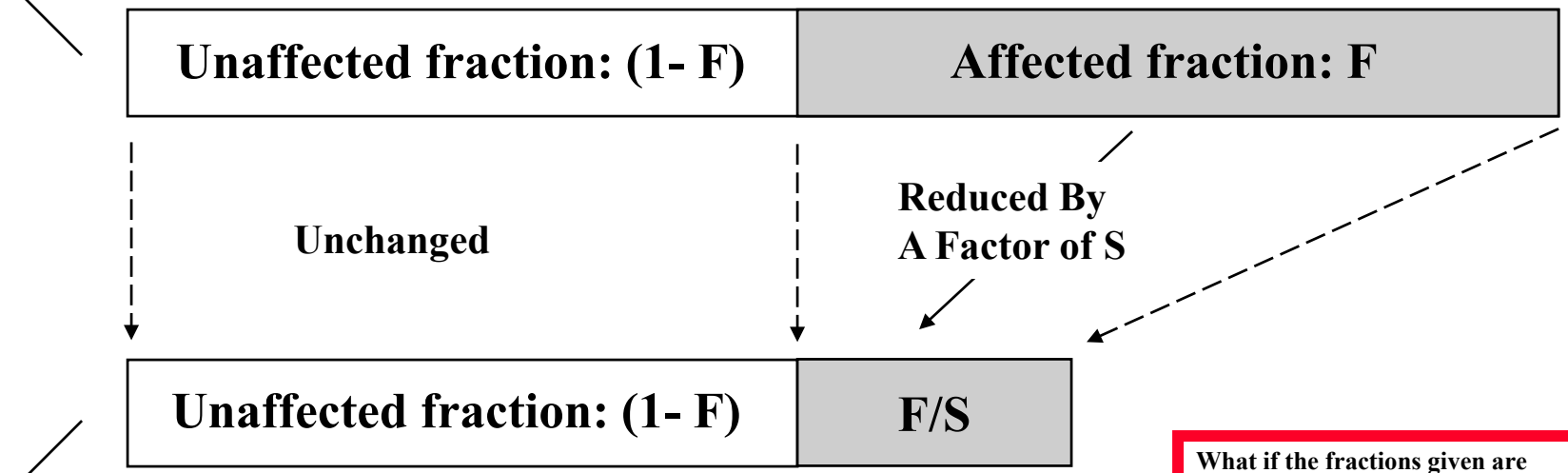
Pictorial Depiction of Amdahl's Law

Enhancement E accelerates fraction F of original execution time by a factor of S

Before:

Execution Time without enhancement E: (Before enhancement is applied)

- **shown normalized to 1 = (1-F) + F =1**



What if the fractions given are after the enhancements were applied? How would you solve the problem?

After:

Execution Time with enhancement E:

$$\text{Speedup(E)} = \frac{\text{Execution Time without enhancement E}}{\text{Execution Time with enhancement E}} = \frac{1}{(1 - F) + F/S}$$

(From 350)

Performance Enhancement Example

- **For the RISC machine with the following instruction mix given earlier:**

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%

CPI = 2.2

- **If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:**

Fraction enhanced = F = 45% or .45

Unaffected fraction = 100% - 45% = 55% or .55

Factor of enhancement = 5/2 = 2.5

Using Amdahl's Law:

$$\text{Speedup(E)} = \frac{1}{(1 - F) + F/S} = \frac{1}{.55 + .45/2.5} = 1.37$$

(From 350)

An Alternative Solution Using CPU Equation

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%

CPI = 2.2

- **If a CPU design enhancement improves the CPI of load instructions from 5 to 2, what is the resulting performance improvement from this enhancement:**

Old (original) CPI = 2.2

New CPI = .5 x 1 + .2 x 2 + .1 x 3 + .2 x 2 = 1.6

$$\text{Speedup(E)} = \frac{\text{Original Execution Time}}{\text{New Execution Time}} = \frac{\text{Instruction count x old CPI x clock cycle}}{\text{Instruction count x new CPI x clock cycle}}$$

$$= \frac{\text{old CPI}}{\text{new CPI}} = \frac{2.2}{1.6} = 1.37$$

Which is the same speedup obtained from Amdahl's Law in the first solution.

(From 350)

T = I x CPI x C

Performance Enhancement Example

- **A program runs in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program four times faster?**

$$\text{Desired speedup} = 4 = \frac{100}{\text{Execution Time with enhancement}}$$

→ Execution time with enhancement = 100/4 = 25 seconds

25 seconds = (100 - 80 seconds) + 80 seconds / S

25 seconds = 20 seconds + 80 seconds / S

→ 5 = 80 seconds / S

→ S = 80/5 = 16

Alternatively, it can also be solved by finding enhanced fraction of execution time:

$$F = 80/100 = .8$$

and then solving Amdahl's speedup equation for desired enhancement factor S

$$\text{Speedup(E)} = \frac{1}{(1 - F) + F/S} = 4 = \frac{1}{(1 - .8) + .8/S} = \frac{1}{.2 + .8/s}$$

Solving for S gives S= 16

Hence multiplication should be 16 times faster to get an overall speedup of 4.

(From 350)

Performance Enhancement Example

- For the previous example with a program running in 100 seconds on a machine with multiply operations responsible for 80 seconds of this time. By how much must the speed of multiplication be improved to make the program five times faster?

$$\text{Desired speedup} = 5 = \frac{100}{\text{Execution Time with enhancement}}$$

$\rightarrow$ Execution time with enhancement = 20 seconds

20 seconds = (100 - 80 seconds) + 80 seconds / n

20 seconds = 20 seconds + 80 seconds / n

$\rightarrow$ 0 = 80 seconds / n

No amount of multiplication speed improvement can achieve this.

(From 350)

Extending Amdahl's Law To Multiple Enhancements

- **Suppose that enhancement E_i accelerates or improves a fraction F_i of the original execution time by a factor S_i and the remainder of the time is unaffected then:**

$$Speedup = \frac{\text{Original Execution Time}}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right) X \text{Original Execution Time}}$$

Unaffected fraction

$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right)}$$

What if the fractions given are after the enhancements were applied? How would you solve the problem?

Note: All fractions F_i refer to original execution time before the enhancements are applied.

(From 350)

Amdahl's Law With Multiple Enhancements: Example

- Three CPU performance enhancements are proposed with the following speedups and percentage of the code execution time affected:

$\text{Speedup}_1 = S_1 = 10$	$\text{Percentage}_1 = F_1 = 20\%$
$\text{Speedup}_2 = S_2 = 15$	$\text{Percentage}_1 = F_2 = 15\%$
$\text{Speedup}_3 = S_3 = 30$	$\text{Percentage}_1 = F_3 = 10\%$

These fractions are from before enhancements are applied

- While all three enhancements are in place in the new design, each enhancement affects a different portion of the code and only one enhancement can be used at a time.
- What is the resulting overall speedup?

$$Speedup = \frac{1}{\left((1 - \sum_i F_i) + \sum_i \frac{F_i}{S_i}\right)}$$

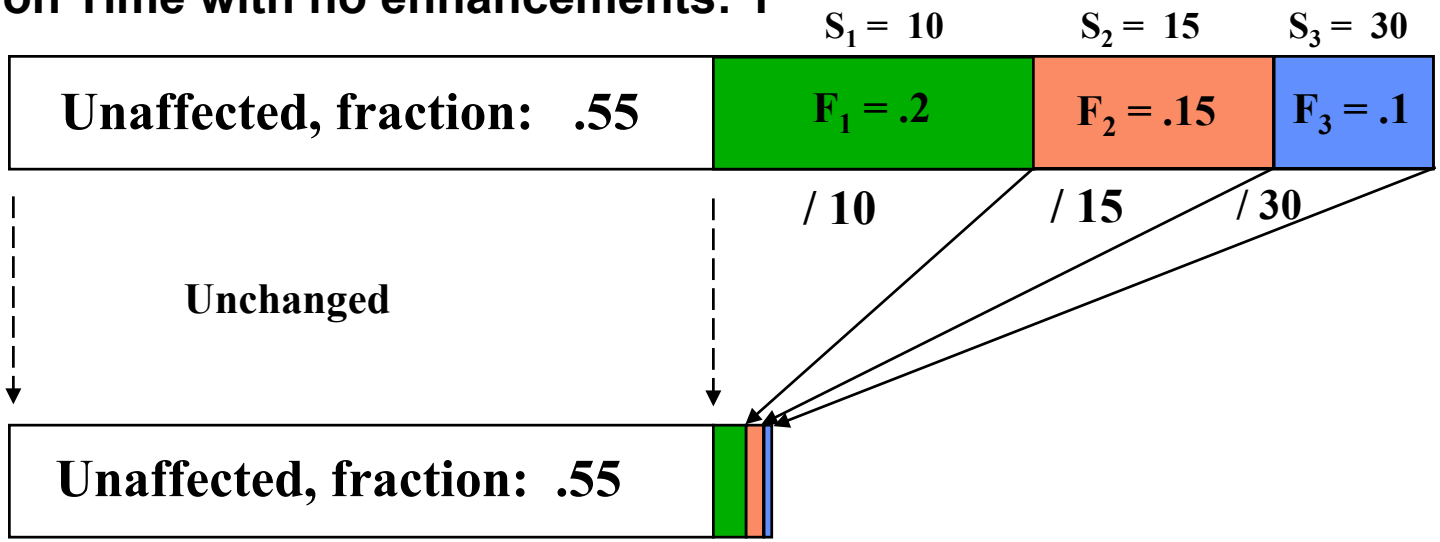
- Speedup = 1 / [(1 - .2 - .15 - .1) + .2/10 + .15/15 + .1/30)]
  = 1 / [.55 + .0333]
  = 1 / .5833 = 1.71

(From 350)

Pictorial Depiction of Example

Before:

Execution Time with no enhancements: 1



After:

Execution Time with enhancements: .55 + .02 + .01 + .00333 = .5833

Speedup = 1 / .5833 = 1.71

Note: All fractions (F_i , i = 1, 2, 3) refer to original execution time.

What if the fractions given are after the enhancements were applied? How would you solve the problem?

(From 350)

"Reverse" Multiple Enhancements Amdahl's Law

- **Multiple Enhancements Amdahl's Law assumes that the fractions given refer to original execution time (i.e. Before the enhancements were applied).**
- **If for each enhancement S_i the fraction F_i it affects is given as a fraction of the resulting execution time after the enhancements were applied then:**

$$Speedup = \frac{\left((1-\sum_i F_i)+\sum_i F_i \times S_i\right) X \text{Resulting Execution Time}}{\text{Resulting Execution Time}}$$

Unaffected fraction

$$Speedup = \frac{(1-\sum_i F_i)+\sum_i F_i \times S_i}{1} = (1-\sum_i F_i)+\sum_i F_i \times S_i$$

i.e as if resulting execution time is normalized to 1

- **For the previous example assuming fractions given refer to resulting execution time after the enhancements were applied (not the original execution time), then:**

Speedup = (1 - .2 - .15 - .1) + .2 x10 + .15 x15 + .1x30
= .55 + 2 + 2.25 + 3
= 7.8

Instruction Set Architecture (ISA)

Assembly Programmer Or Compiler

Or CPU

"... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation."
– Amdahl, Blaaw, and Brooks, 1964.

i.e. CPU Design

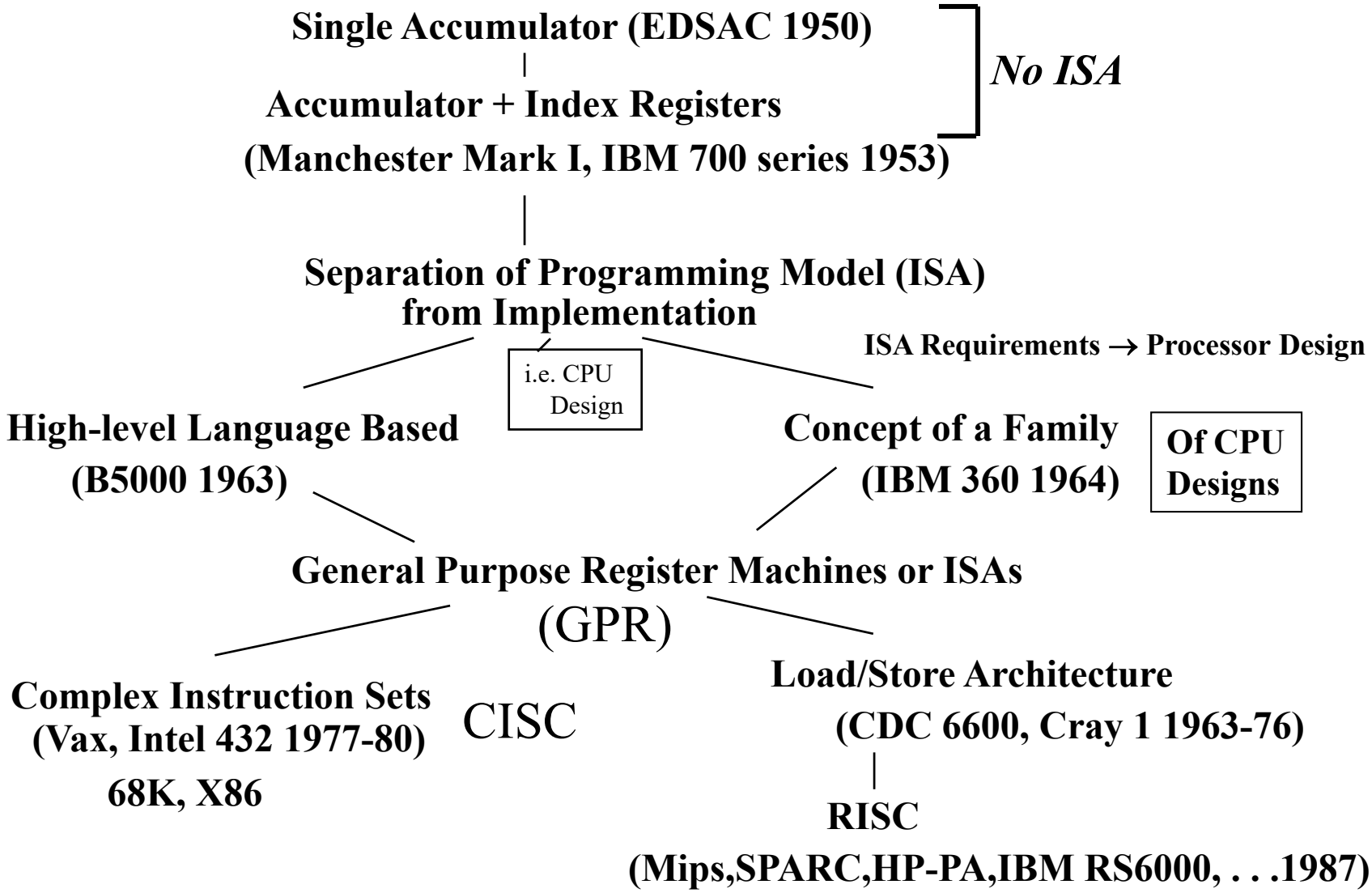
The ISA forms an abstraction layer that sets the requirements for both complier and CPU designers

➔ **The instruction set architecture is concerned with:**

- **Organization of programmable storage (memory & registers): Includes the amount of addressable memory and number of available registers.**
- **Data Types & Data Structures: Encodings & representations.**
- **Instruction Set: What operations are specified.**
- **Instruction formats and encoding.**
- **Modes of addressing and accessing data items and instructions**
- **Exceptional conditions.**

ISA in 4th Edition: Appendix B (3rd Edition: Chapter 2)

Evolution of Instruction Sets



The ISA forms an abstraction layer that sets the requirements for both complier and CPU designers

Complex Instruction Set Computer (CISC)

ISAs

- **Emphasizes doing more with each instruction:**
  - **Thus fewer instructions per program (more compact code).**
- **Motivated by the high cost of memory and hard disk capacity when original CISC architectures were proposed**

Why?

  - **When M6800 was introduced: 16K RAM = $500, 40M hard disk = $ 55, 000**
  - **When MC68000 was introduced: 64K RAM = $200, 10M HD = $5,000**

Circa 1980

- **Original CISC architectures evolved with faster more complex CPU designs but backward instruction set compatibility had to be maintained (e.g. x86)**
- **Wide variety of addressing modes:**
  - **14 in MC68000, 25 in MC68020**
- **A number instruction modes for the location and number of operands:**
  - **The VAX has 0- through 3-address instructions.**
- **Variable-length instruction encoding.**

to reduce code size

Reduced Instruction Set Computer (RISC)

~1984 ISAs

RISC: Simplify ISA → Simplify CPU Design → Better CPU Performance

- Focuses on reducing the number and complexity of instructions of the machine/ISA.
- Reduced CPI. Goal: At least one instruction per clock cycle.
- Designed with pipelining in mind.
- Fixed-length instruction encoding.
- Only load and store instructions access memory for data.
- Simplified addressing modes. (Thus more instructions executed than CISC)
  - Usually limited to immediate, register indirect, register displacement, indexed.
- Delayed loads and branches.
- Instruction pre-fetch and speculative execution.
- Examples: MIPS, ARM, POWER, PowerPC, Alpha ..

Machine = CPU or ISA

(CPI = 1 or less)

RISC Goals

Simpler CPU Design
Better CPU performance

Types of Instruction Set Architectures
According To Operand Addressing Fields

Memory-To-Memory Machines (or ISAs):

- **Operands obtained from memory and results stored back in memory by any instruction that requires operands.**
- **No local CPU registers are used in the CPU datapath.**
- **Include:**
  - **The 4 Address Machine.**
  - **The 3-address Machine.**
  - **The 2-address Machine.**

The 1-address (Accumulator) Machine:

- **A single local CPU special-purpose register (accumulator) is used as the source of one operand and as the result destination.**

The 0-address or Stack Machine:

- **A push-down stack is used in the CPU.**

General Purpose Register (GPR) Machines (or ISAs):

GPR ISAs

- The CPU datapath contains several local general-purpose registers which can be used as operand sources and as result destinations.
- A large number of possible addressing modes.
- **Load-Store or Register-To-Register Machines:** GPR machines where only data movement instructions (loads, stores) can obtain operands from memory and store results to memory.

CISC to RISC observation (load-store simplifies CPU design)

General-Purpose Register (GPR) ISAs/Machines

- Every new ISA designed after 1980 uses a load-store GPR architecture (i.e RISC, to simplify CPU design).

Why GPR?

1 • Registers, like any other storage form internal to the CPU, are faster than memory.

2 • Registers are easier for a compiler to use.

3 • Shorter instruction encoding.

- GPR architectures are divided into several types depending on two factors:
  - Whether an ALU instruction has two or three operands.
  - How many of the operands in ALU instructions may be memory addresses.

ISA Examples

Machine	Number of General Purpose Registers	Architecture	year
EDSAC	1	accumulator	1949
IBM 701	1	accumulator	1953
CDC 6600	8	load-store	1963
IBM 360	16	register-memory	1964
DEC PDP-8	1	accumulator	1965
DEC PDP-11	8	register-memory	1970
Intel 8008 (8-bit)	1	accumulator	1972
Motorola 6800	1	accumulator	1974
DEC VAX	16	register-memory memory-memory	1977
Intel 8086 (16-bit)	1	extended accumulator	1978
Motorola 68000	16	register-memory	1980
Intel 80386 (32-bit)	8	register-memory	1985
MIPS	32	load-store	1985
HP PA-RISC	32	load-store	1986
SPARC	32	load-store	1987
PowerPC	32	load-store	1992
DEC Alpha	32	load-store	1992
HP/Intel IA-64	128	load-store	2001
AMD64 (EMT64) - 64-bit	16	register-memory	2003

Typical Memory Addressing Modes

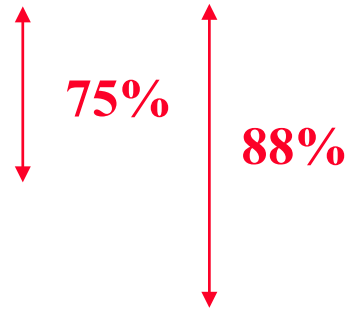
Addressing Mode	Sample Instruction	Meaning
Register	Add R4, R3	Regs [R4] ←Regs[R4] + Regs[R3]
Immediate	Add R4, #3	Regs[R4] ←Regs[R4] + 3
Displacement	Add R4, 10 (R1)	Regs[R4] ←Regs[R4]+Mem[10+Regs[R1]]
Indirect	Add R4, (R1)	Regs[R4] ←Regs[R4]+ Mem[Regs[R1]]
Indexed	Add R3, (R1 + R2)	Regs [R3] ←Regs[R3]+Mem[Regs[R1]+Regs[R2]]
Absolute	Add R1, (1001)	Regs[R1] ←Regs[R1] + Mem[1001]
Memory indirect	Add R1, @ (R3)	Regs[R1] ←Regs[R1] + Mem[Mem[Regs[R3]]]
Autoincrement	Add R1, (R2) +	Regs[R1] ←Regs[R1] + Mem[Regs[R2]] Regs[R2] ←Regs[R2] + d
Autodecrement	Add R1, - (R2)	Regs [R2] ←Regs[R2] -d Regs{R1] ←Regs[Regs[R1] +Mem[Regs[R2]]
Scaled	Add R1, 100 (R2) [R3]	Regs[R1] ←Regs[R1] + Mem[100+Regs[R2]+Regs[R3]*d]

For GPR ISAs

Addressing Modes Usage Example

For 3 programs running on VAX ignoring direct register mode:

Displacement	**42% avg, 32% to 55%**
Immediate:	**33% avg, 17% to 43%**
Register deferred (indirect):	**13% avg, 3% to 24%**
Scaled:	**7% avg, 0% to 16%**
Memory indirect:	**3% avg, 1% to 6%**
Misc:	**2% avg, 0% to 3%**



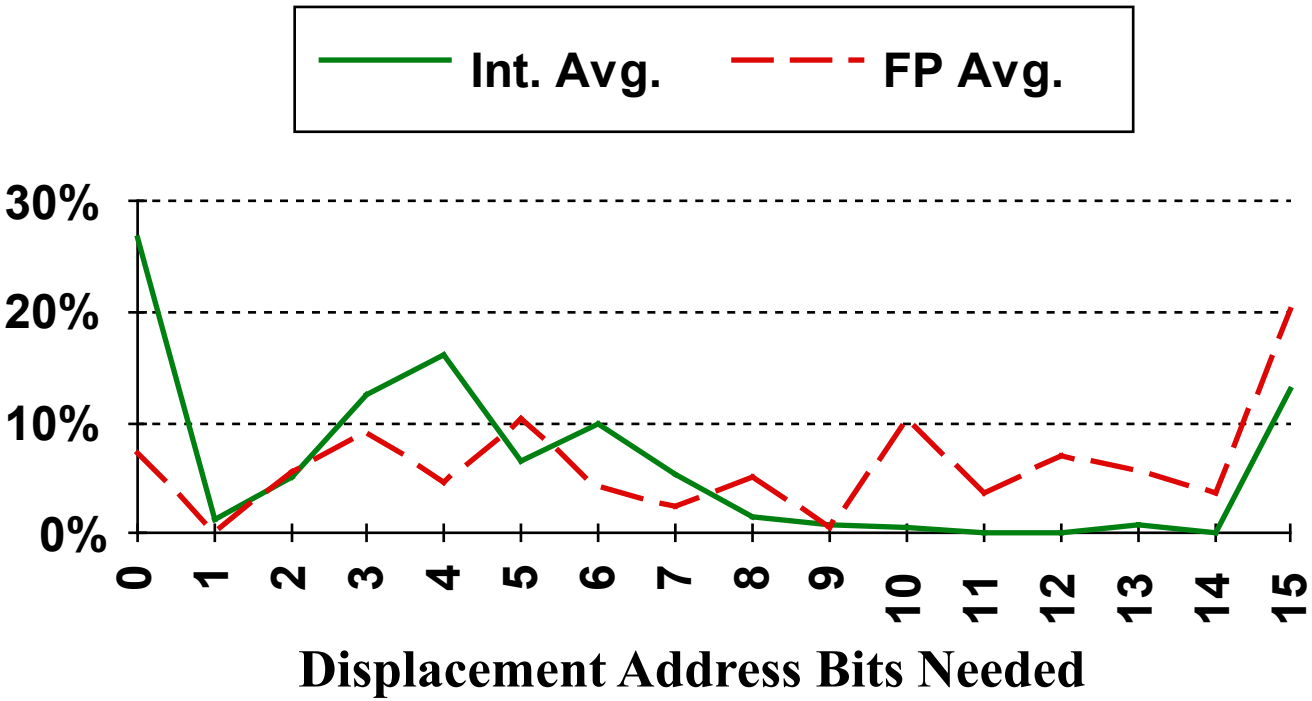
75% displacement & immediate
88% displacement, immediate & register indirect.

Observation: In addition Register direct, Displacement, Immediate, Register Indirect addressing modes are important.

CISC to RISC observation
(fewer addressing modes simplify CPU design)

Displacement Address Size Example

Avg. of 5 SPECint92 programs v. avg. 5 SPECfp92 programs



1% of addresses > 16-bits

→ 12 - 16 bits of displacement needed

→ CISC to RISC observation

Operation Types in The Instruction Set

Operator Type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, or
Data transfer 2	Loads-stores (move on machines with memory addressing)
Control 3	Branch, jump, procedure call, and return, traps.
System	Operating system call, virtual memory management instructions
Floating point	Floating point operations: add, multiply.
Decimal	Decimal add, decimal multiply, decimal to character conversion
String	String move, string compare, string search
Media	The same operation performed on multiple data (e.g Intel MMX, SSE)

1

Instruction Usage Example: Top 10 Intel X86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

→ Observation: Simple instructions dominate instruction usage frequency.

→ CISC to RISC observation

Instruction Set Encoding

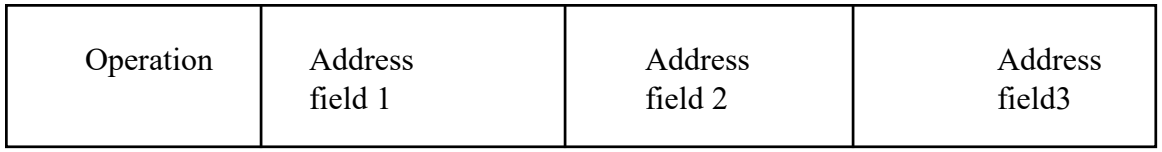
Considerations affecting instruction set encoding:

- **To have as many registers and addressing modes as possible.**
- **The Impact of of the size of the register and addressing mode fields on the average instruction size and on the average program.**
- **To encode instructions into lengths that will be easy to handle in the implementation. On a minimum to be a multiple of bytes.**
  - **Fixed length encoding:** Faster and easiest to implement in hardware. e.g. Simplifies design of pipelined CPUs
  - **Variable length encoding:** Produces smaller instructions. to reduce code size
  - **Hybrid encoding.**

CISC to RISC observation

Three Examples of Instruction Set Encoding

Variable: VAX (1-53 bytes)



→ **Fixed: MIPS, PowerPC, SPARC (Each instruction is 4 bytes, e.g RISC ISAs)**

Hybrid : IBM 360/370, Intel 80x86

Example CISC ISA:

Motorola 680X0

18 addressing modes:

- Data register direct.
- Address register direct.
- Immediate.
- Absolute short.
- Absolute long.
- Address register indirect.
- Address register indirect with postincrement.
- Address register indirect with predecrement.
- Address register indirect with displacement.
- Address register indirect with index (8-bit).
- Address register indirect with index (base).
- Memory inderect postindexed.
- Memory indirect preindexed.
- Program counter indirect with index (8-bit).
- Program counter indirect with index (base).
- Program counter indirect with displacement.
- Program counter memory indirect postindexed.
- Program counter memory indirect preindexed.

GPR ISA (Register-Memory)

Operand size:

- Range from 1 to 32 bits, 1, 2, 4, 8, 10, or 16 bytes.

Instruction Encoding:

- Instructions are stored in 16-bit words.
- the smallest instruction is 2- bytes (one word).
- The longest instruction is 5 words (10 bytes) in length.

2 Bytes → 10 Bytes

Example CISC ISA:

Intel IA-32, X86 (80386)

GPR ISA (Register-Memory)

12 addressing modes:

- Register.
- Immediate.
- Direct.
- Base.
- Base + Displacement.
- Index + Displacement.
- Scaled Index + Displacement.
- Based Index.
- Based Scaled Index.
- Based Index + Displacement.
- Based Scaled Index + Displacement.
- Relative.

Operand sizes:

- Can be 8, 16, 32, 48, 64, or 80 bits long.
- Also supports string operations.

Instruction Encoding:

- The smallest instruction is one byte.
- The longest instruction is 12 bytes long.
- The first bytes generally contain the opcode, mode specifiers, and register fields.
- The remainder bytes are for address displacement and immediate data.

One Byte → 12 Bytes

Example RISC ISA:

HP Precision Architecture, HP PA-RISC

Load-Store GPR

7 addressing modes:

- Register
- Immediate
- Base with displacement
- Base with scaled index and displacement
- Predecrement
- Postincrement
- PC-relative

Operand sizes:

- Five operand sizes ranging in powers of two from 1 to 16 bytes.

Instruction Encoding:

- Instruction set has 12 different formats.
- All are 32 bits (4 bytes) in length.

RISC ISA Example:

MIPS-I MIPS R3000 (32-bits)

Instruction Categories:

- Load/Store.
- Computational.
- Jump and Branch.
- Floating Point (using coprocessor).
- Memory Management.
- Special.

Load-Store GPR

5 Addressing Modes:

- Register direct (arithmetic).
- Immedate (arithmetic).
- Base register + immediate offset (loads and stores).
- PC relative (branches).
- Pseudodirect (jumps)

Operand Sizes:

- Memory accesses in any multiple between 1 and 4 bytes.

Registers

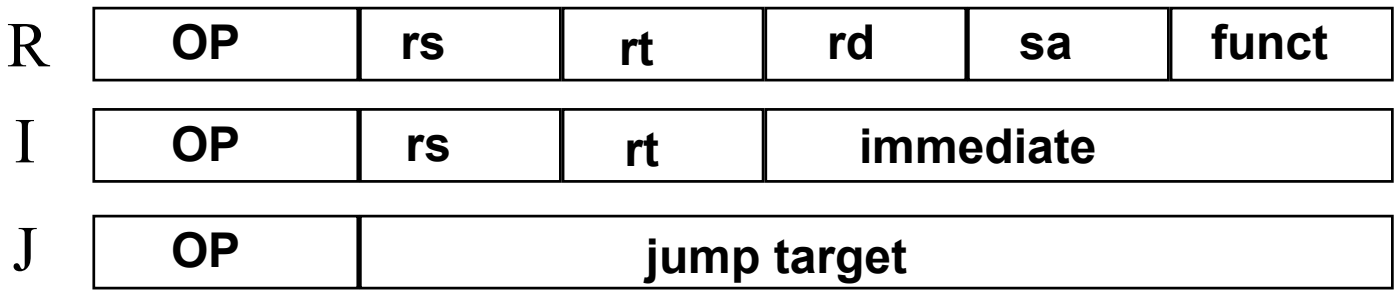
R0 - R31

PC

HI

LO

Instruction Encoding: 3 Instruction Formats, all 32 bits wide (4 bytes).



(Used as target ISA for CPU design in 350)

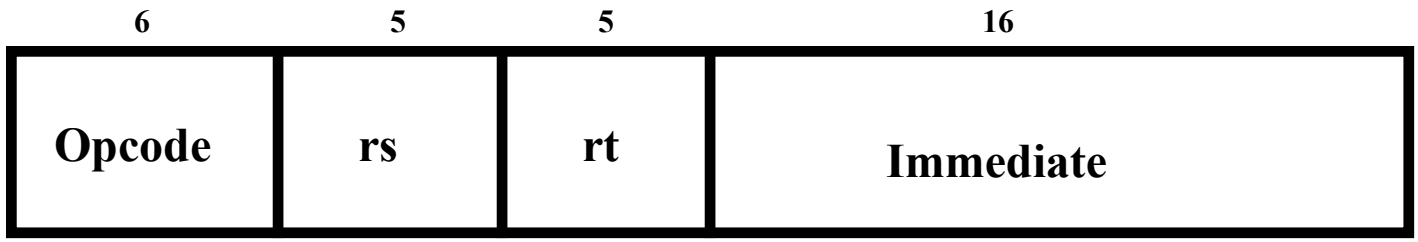
An Instruction Set Example: MIPS64

- A RISC-type 64-bit instruction set architecture based on instruction set design considerations of chapter 2:
  - Use general-purpose registers with a load/store architecture to access memory. Load/Store GPR similar to allall RISC ISAs
  - Reduced number of addressing modes: displacement (offset size of 16 bits), immediate (16 bits).
  - Data sizes: 8 (byte) Bits, 16 (half word), 32 (word) 4 Bytes, 64 (double word) 8 Bytes bit integers and 32-bit or 64-bit IEEE 754 floating-point numbers. (Single Precision FP, Double Precision FP)
  - Use fixed instruction encoding (32 bits) for performance.
  - 32, 64-bit general-purpose integer registers GPRs, R0, …., R31. R0 always has a value of zero.
  - Separate 32, 64-bit floating point registers FPRs: F0, F1 … F31 When holding a 32-bit single-precision number the upper half of the FPR is not used.

64-bit version of 32-bit MIPS ISA used in 350
4th Edition in Appendix B (3rd Edition: Chapter 2)

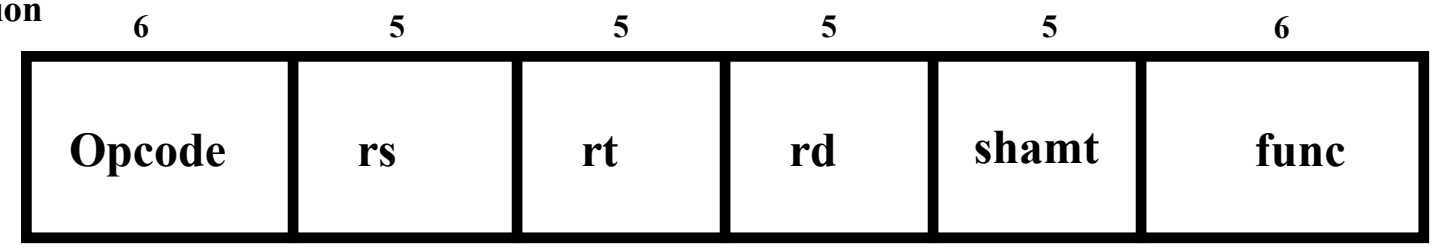
MIPS64 Instruction Format

I - type instruction



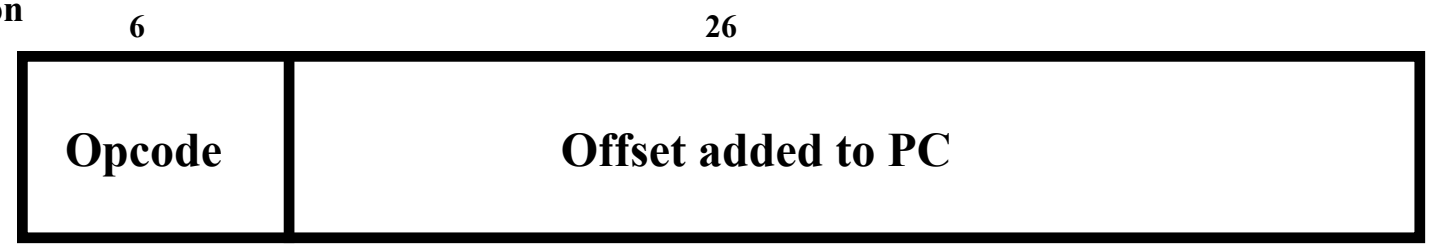
Encodes: Loads and stores of bytes, words, half words. All immediates (rt ← rs op immediate)
Conditional branch instructions
Jump register, jump and link register (rs = destination, immediate = 0)

R - type instruction



Register-register ALU operations: rd ← rs func rt Function encodes the data path operation:
Add, Sub .. Read/write special registers and moves.

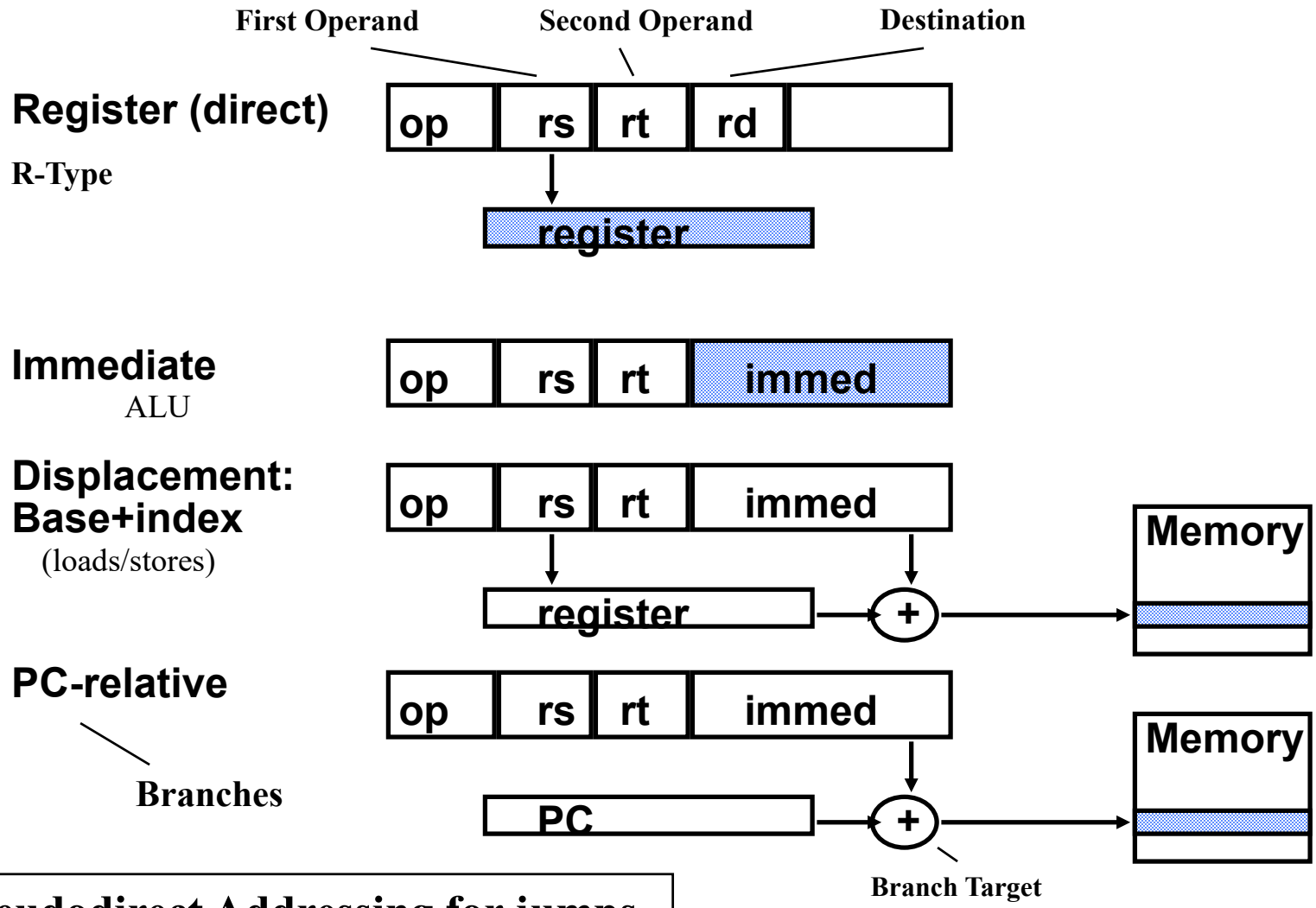
J - Type instruction



Jump and jump and link. Trap and return from exception

MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide



Pseudodirect Addressing for jumps (J-Type) not shown here

MIPS64 Instructions: Load and Store

D = Double Word = 8 Bytes

LD R1,30(R2)	**Load double word**	**Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]**
LW R1, 60(R2)	**Load word**	**Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]**
LB R1, 40(R3)	**Load byte**	**Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]**
LBU R1, 40(R3)	**Load byte unsigned**	**Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]**
LH R1, 40(R3)	**Load half word**	**Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40 + Regs[R3]] # # Mem [41+Regs[R3]]**
L.S F0, 50(R3)	**Load FP single**	**Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}**
L.D F0, 50(R2)	**Load FP double**	**Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]**
SD R3,500(R4)	**Store double word**	**Mem [500+Regs[R4]] $\leftarrow_{64}$ Reg[R3]**
SW R3,500(R4)	**Store word**	**Mem [500+Regs[R4]] $\leftarrow_{32}$ Reg[R3]**
S.S F0, 40(R3)	**Store FP single**	**Mem [40, Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0\ldots31}$**
S.D F0,40(R3)	**Store FP double**	**Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]**
SH R3, 502(R2)	**Store half**	**Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48\ldots63}$**
SB R2, 41(R3)	**Store byte**	**Mem[41 + Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56\ldots63}$**

W = Word = 4 Bytes

8 bytes = 64 bit = Double Word

MIPS64 Instructions: Integer Arithmetic/Logical

DADDU R1, R2, R3 Add unsigned Regs[R1] ← Regs[R2] + Regs[R3]

DADDI R1, R2, #3 Add immediate Regs[R1] ← Regs[R2] + 3

LUI R1, #42 Load upper immediate Regs[R1] ← 0^{32} ##42 ## 0^{16}

DSLL R1, R2, #5 Shift left logical Regs[R1] ← Regs [R2] <<5

DSLT R1, R2, R3 Set less than if (regs[R2] < Regs[R3])
Regs [R1] ← 1 else Regs[R1] ← 0

MIPS64 Instructions: Control-Flow

J name	Jump	$PC_{36..63} \leftarrow$ name
JAL name	Jump and link	Regs[31] $\leftarrow$ PC+4; $PC_{36..63} \leftarrow$ name; ((PC+4)- 2^{27}) $\leq$ name $<$ ((PC + 4) + 2^{27})
JALR R2	Jump and link register	Regs[R31] $\leftarrow$ PC+4; PC $\leftarrow$ Regs[R2]
JR R3	Jump register	PC $\leftarrow$ Regs[R3]
BEQZ R4, name	Branch equal zero	if (Regs[R4] ==0) PC $\leftarrow$ name; ((PC+4) -2^{17}) $\leq$ name < ((PC+4) + 2^{17}
+ BEQ , BNE		
BNEZ R4, Name	Branch not equal zero	if (Regs[R4] != 0) PC $\leftarrow$ name ((PC+4) - 2^{17}) $\leq$ name < ((PC +4) + 2^{17}
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3] ==0) Regs[R1] $\leftarrow$ Regs[R2]

Condition Register

Conditional instruction example

MIPS64 Instructions: Floating Point Arithmetic

Floating Point Arithmetic (R-Type) Instructions Examples:

Destination **Operands**

ADD.D F4, F0, F7

SUB.D F6, F10, F13

MUL.S F8, F18, F14

DIV.D F5, F6, F10

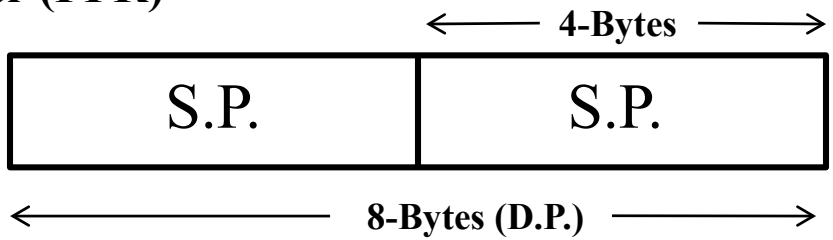
ADD.PS F7, F10, F11

FP Register (FPR)

.D = Double Precision

.S = Single Precision

.PS = Pair of Single Precision

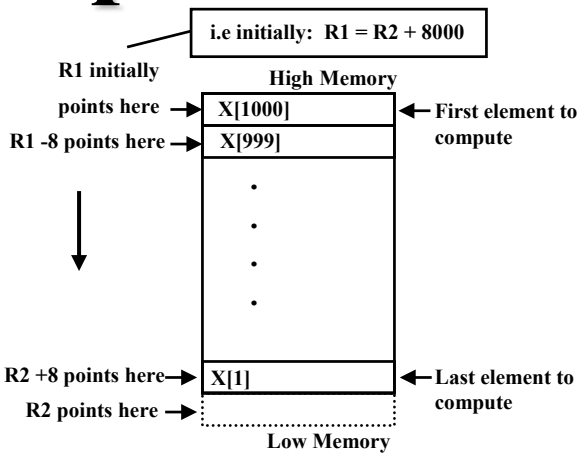


MIPS64 Loop Example

(with Floating Point Operations)

- For the loop:

```
for (i=1000; i>0; i=i-1)
x[i] = x[i] + s;
```



The straightforward MIPS64 assembly code is given by:

```
Loop:  L.D      F0, 0 (R1)      ;F0=array element
       ADD.D    F4, F0, F2      ;add scalar in F2  (constant S)
       S.D      F4, 0(R1)       ;store result
       DADDUI   R1, R1, # -8    ;decrement pointer 8 bytes
       BNE      R1, R2,Loop     ;branch R1!=R2
```

i.e done looping when R1 = R2

i.e. initially: R1 = R2 + 8000

R1 is initially the address of the element with highest address.
8(R2) is the address of the last element to operate on.

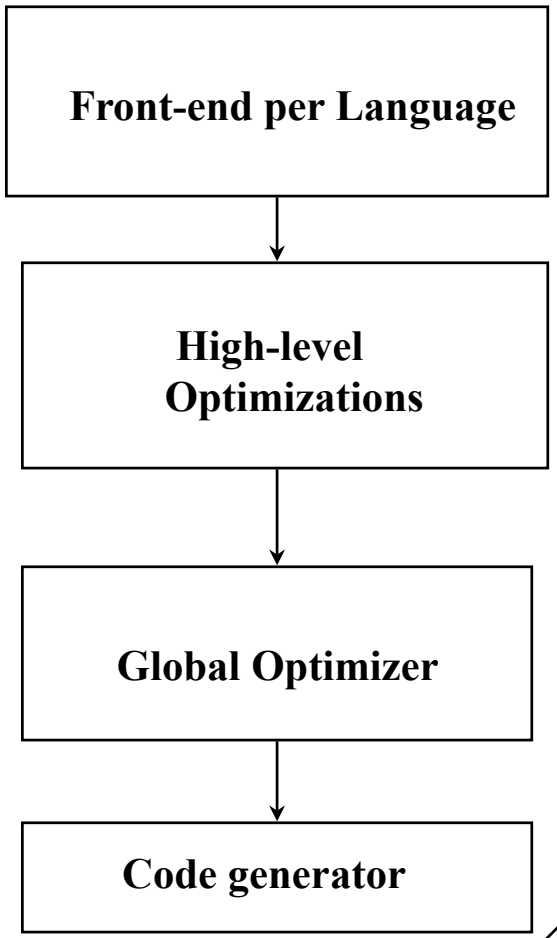
Instructions before the loop to initialize R1, R2 not shown here

X[ ] array of double-precision floating-point numbers (8-bytes each)

(Example from Chapter 2.2, will use later to illustrate loop unrolling)

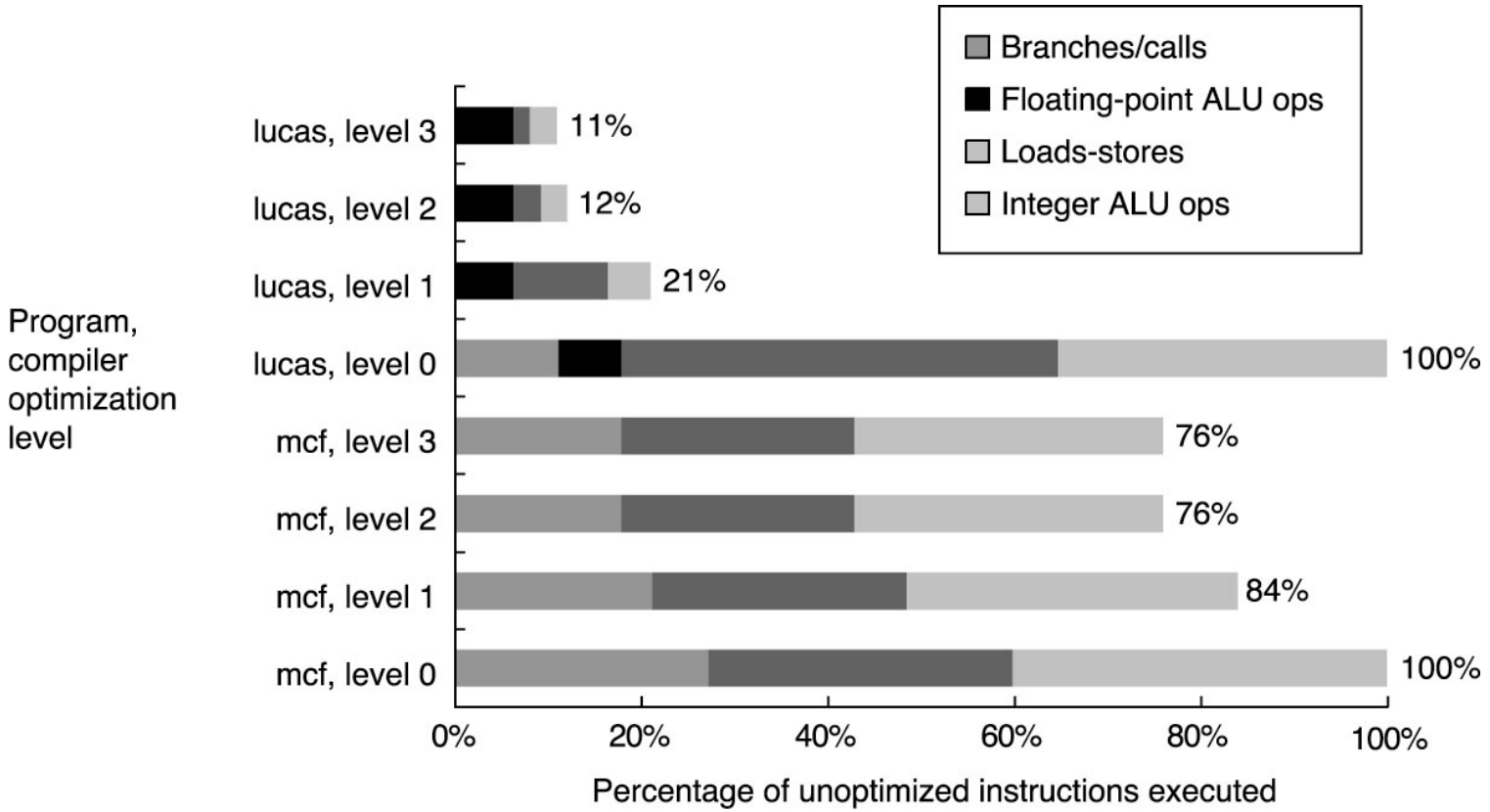
The Role of Compilers

The Structure of Recent Compilers:



$T = I \times CPI \times C$

Compiler Optimization and Executed Instruction Count



→ **Change in instruction count executed (I) for the programs lucas and mcf from SPEC2000 as compiler optimizations vary.**

Quiz #1

On Lecture Notes Set # 1

- **Monday, January 30**