CS 31007 Autumn 2020

COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors
Rajat Subhra Chakraborty
Bhargab B. Bhattacharya

Lecture 02

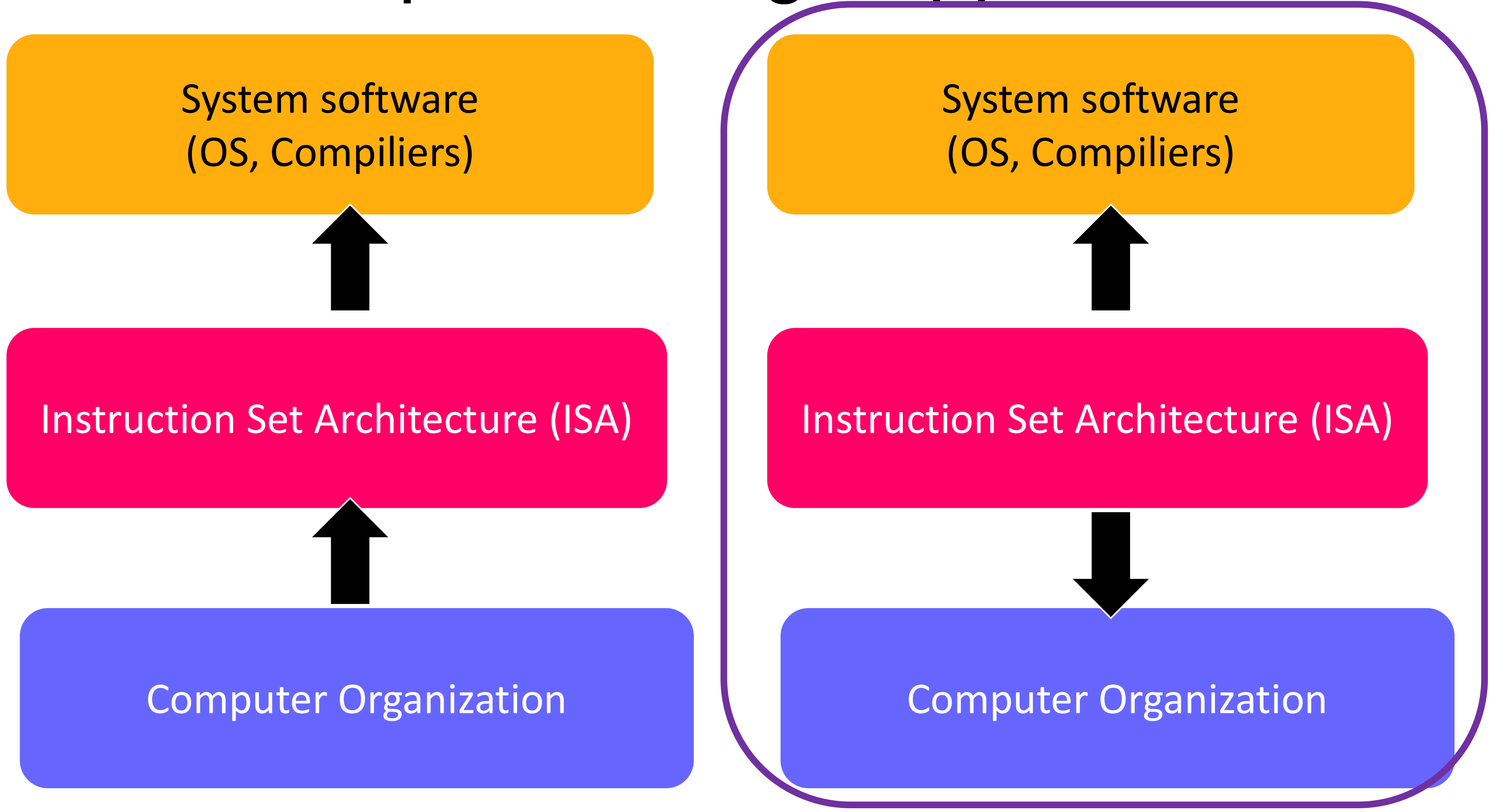
Indian Institute of Technology Kharagpur
Computer Science and Engineering

Previous Class

- ❖ Overview of the course
- ❖ Evolution and history of computer design
- ❖ Moore’s law
- ❖ Basic components of a computer
- ❖ Instruction Set Architecture (ISA)
- ❖ Computer organization and computer architecture: Bottom-up and Top-down view

Acknowledgement: Patterson and Hennessy; V. D. Agrawal; S. R. Sarangi

Computer Design Approach



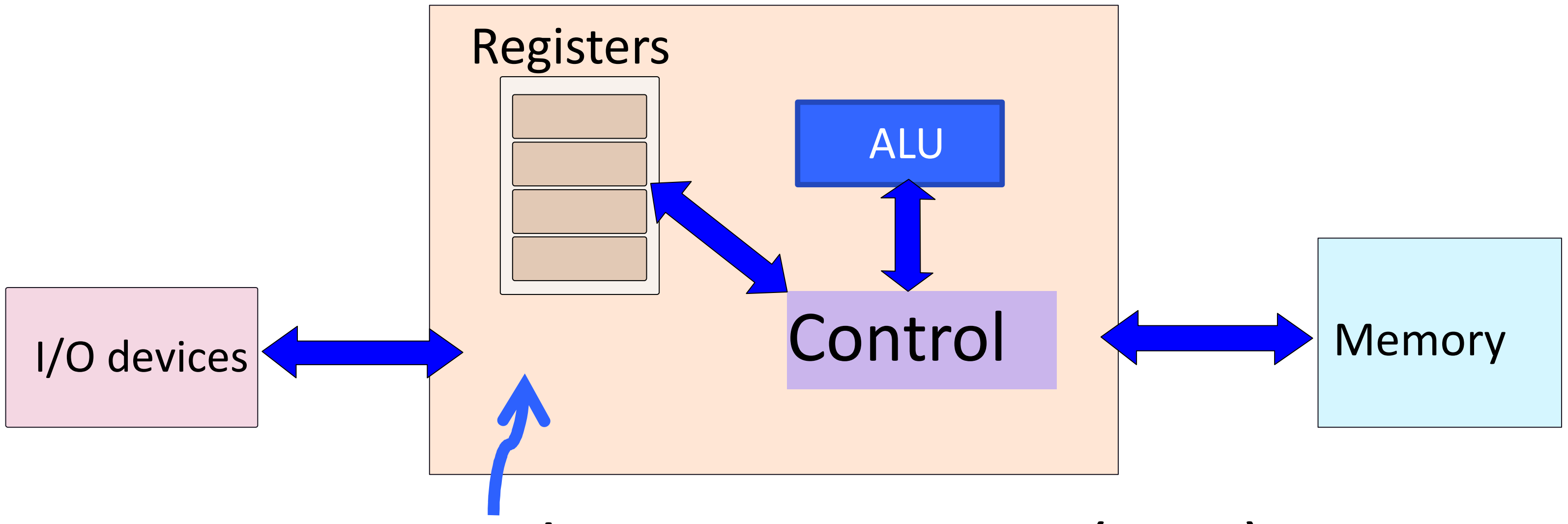
Bottom-up approach:
Hardware first, ISA later

Top-down approach to hardware design:
ISA first, hardware later

Instruction Set Architecture (ISA)

- A set of assembly language instructions that separates the interface between software and hardware
- In top-down design, given an ISA, an appropriate hardware platform is built to support it
- Based on ISA, OS and compilers are to be developed accordingly further going up
- Once ISA is fixed, software and hardware engineers can work independently
- ISA is designed to optimize the performance supported by the available hardware technology

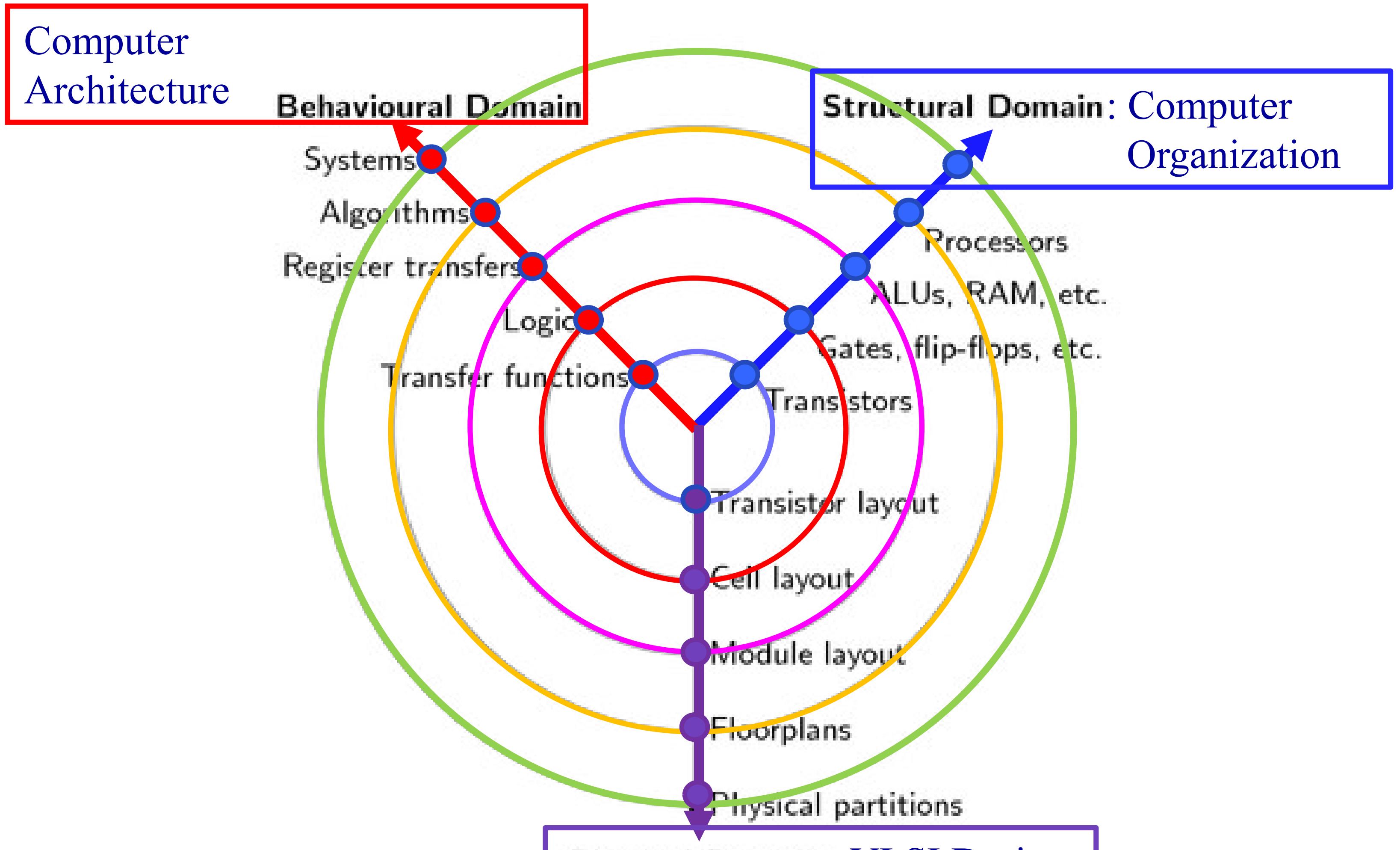
The Hardware of a Computer



Central Processing Unit (CPU)

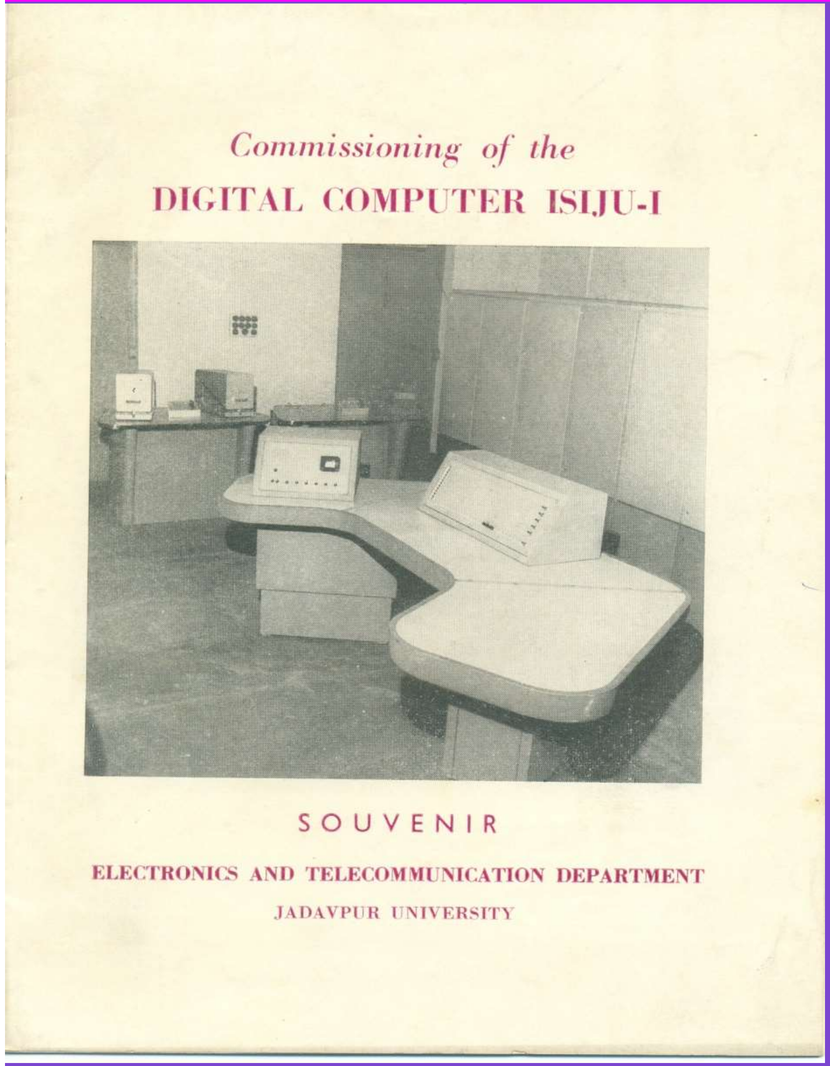
Five easy pieces:
Control, Arithmetic Logic Unit (ALU), Memory,
Input/Output, Datapath (Bus)

Digital System Design from Three Perspectives



Gajski-Kuhn Y-Chart (1983)

First Digital Computer Built in India


Commissioning of the

DIGITAL COMPUTER ISIJU-I

SOUVENIR

ELECTRONICS AND TELECOMMUNICATION DEPARTMENT

JADAVPUR UNIVERSITY

In 1966, a digital computer named ISIJU is designed and commissioned, with joint collaboration between Indian Statistical Institute (ISI) and Jadavpur University (JU), Kolkata

Factors behind the exponential growth of computing paradigms

- Evolution of computer architecture
- Growth of semiconductor technology and IC design processes
- Memory technologies
- Algorithms and data structures
- Programming languages, compilers, OS
- Test, verification, error management techniques
- Bootstrapping: using present computers to design the next generation of computers (design automation)
- Parallel and distributed computing
- Networking

Agenda

- Model for computation and Turing Machine
- von Neumann Architecture
- Basic Features of Instruction Set Architecture (ISA)
- Die yield
- CPU Performance Equation
- Amdahl’s Law; Gustavson-Barsis Law
- RISC *versus* CISC

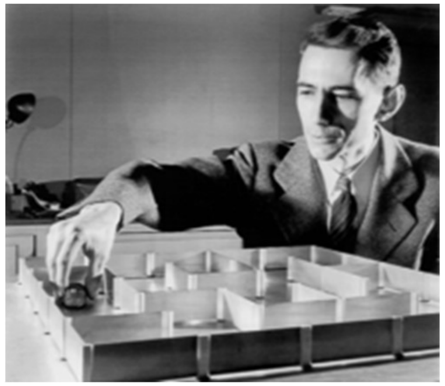
Three Challenges

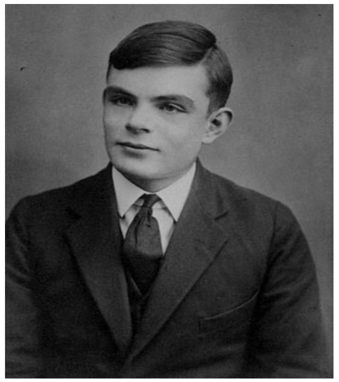
1. How to design efficient hardware (logic)?

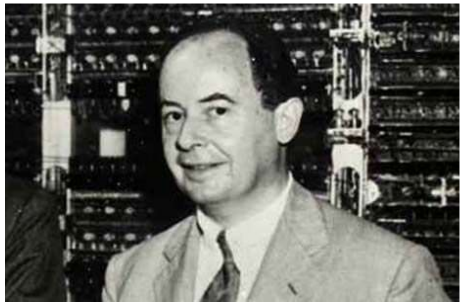
2. What is the simplest, yet all powerful computer (computability)?

3. How should the basic computer architecture be conceived?

Pioneers who answered these three questions


Claude E. Shannon
(1916-2001)


Alan Turing
(1912-1954)


John von Neumann
(1903-1957)

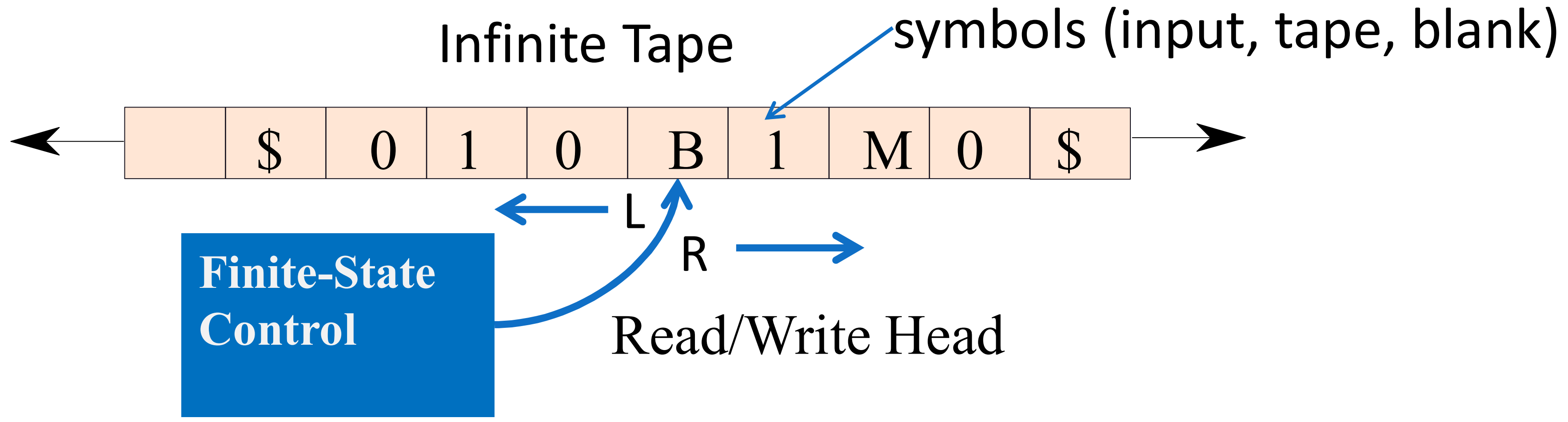
Logic design (basis of computer organization; hardware cost/ circuit delay/power optimization)

Theory of computability (basis for the fundamental requirement in computation)

Blueprint for basic computer architecture

What is the simplest yet all powerful computer?

Alan Turing (1936): Conceived a machine that introduces a model for computation (Turing Machine)



The tape head can only move left or right

Actions: (present state, current symbol) →
  (new state, write symbol, move one cell left/right);

-- The machine halts when an "accept"/"reject" state is reached

Example

- Start with a blank tape and create a pattern 0b1b0b1b . . .
- Define symbols: b (blank), 0, 1

Present state	Symbol on tape	Operation	Next state
S0 (begin)	blank	Write 0 and move right	S1
S1	blank	Move right	S2
S2	blank	Write 1 and move right	S3
S3	blank	Move right	S0

http://en.wikipedia.org/wiki/Turing_machine_examples

Turing Machine

❖ Very simple mechanism:
-- memory, control, read/write, shift, accept/reject

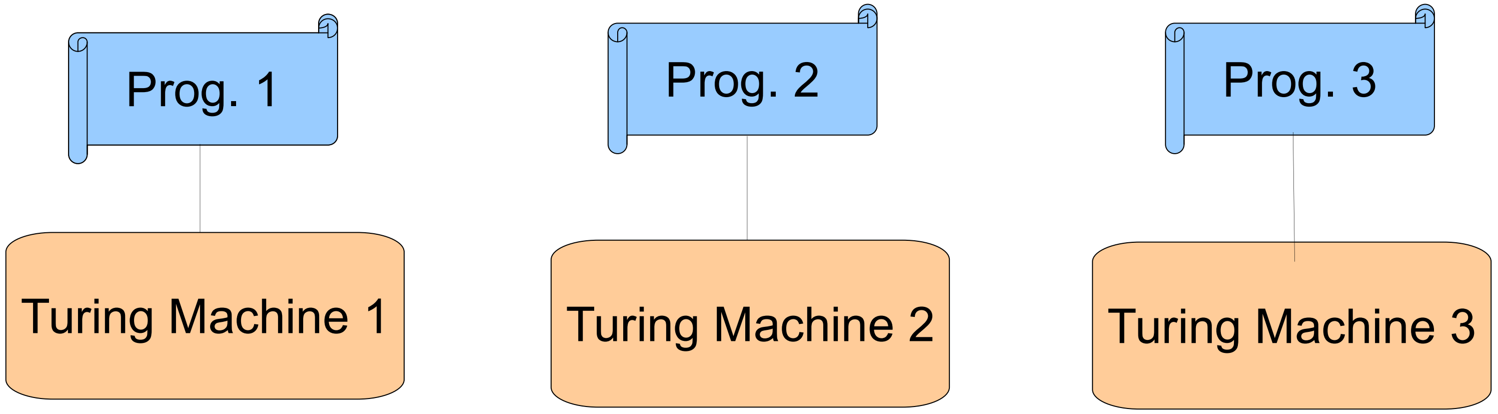
A. M. Turing (1936) , On Computable Numbers with an Application to the Entscheidungsproblem, *Proc. Royal Math. Soc.*, Ser. 2, Vol. 42, pp. 230-265, 1936.

❖ Extremely powerful

Church-Turing Conjecture (1936): A function on natural numbers can be calculated by an effective method *if and only if* it is computable by a Turing machine
Any procedure that is computable by paper-and-pencil methods (algorithm) can be solved by a Turing machine

Universal Turing Machine

Church-Turing Conjecture (1936): Any procedure that is computable by paper-and-pencil method (algorithm) can be solved by a Turing machine.

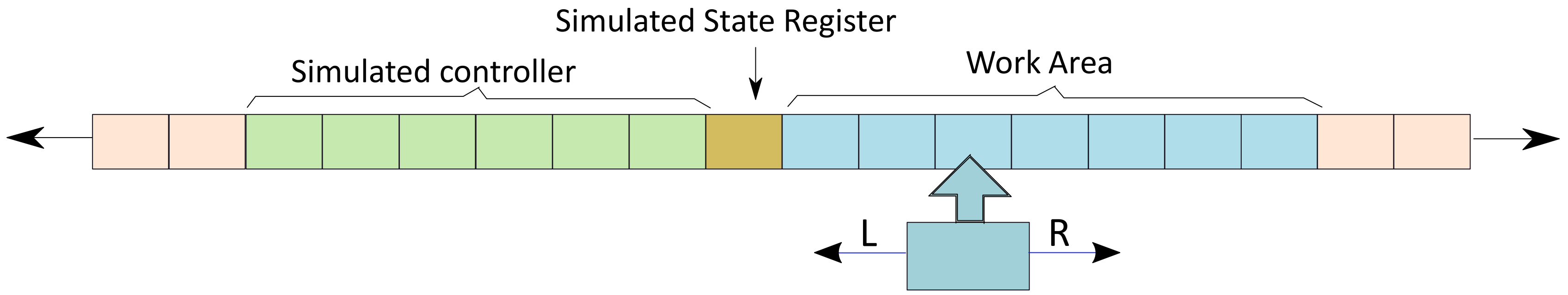


More general question:

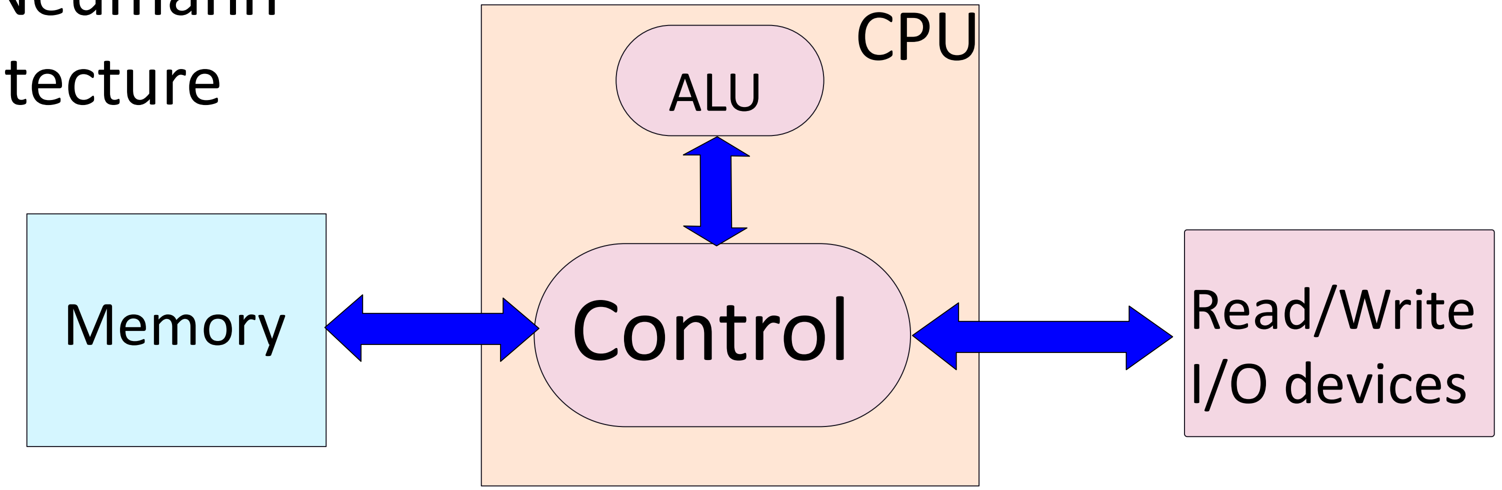
Can we design a Universal Turing machine (UTM) that can simulate any Turing machine?

Universal Turing Machine

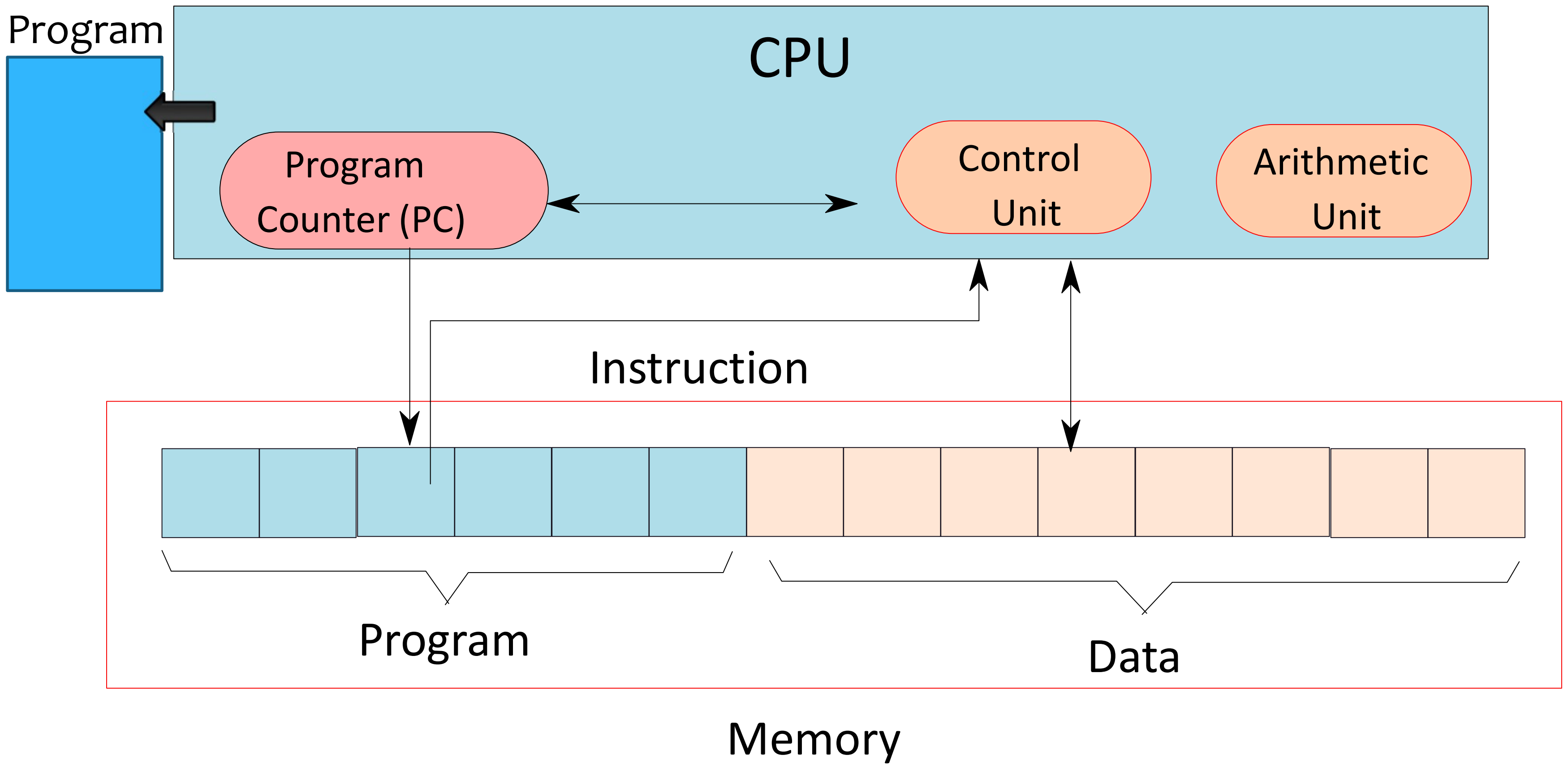
Controller and states are simulated on tape



Von Neumann Architecture



Computer mimicing Turing machine



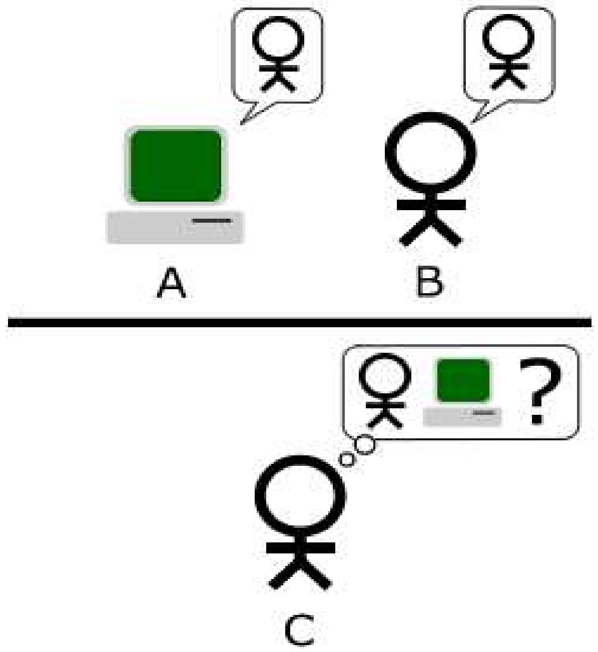
Turing Undecidability?

Halting Problem: Can we write a program *Q*, which given any arbitrary program *P* as input, will decide whether on not *P* terminates or falls into an infinite loop on input data?

Halting problem is Turing undecidable

Turing Test

- Can a computer think? (Turing, 1950)
- A. P. Saygin, I. Cicekli and V. Akman, “Turing Test: 50 Years Later,” *Minds and Machines*, vol. 10, no. 4, pp. 463-518, 2000.



Can a person *C* discriminate between a machine *A* and human *B* by asking questions and getting written answers?

http://www.engadget.com/2011/01/13/ibms-watson-supercomputer-destroys-all-humans-in-jeopardy-pract/

Turing Test

- In 2014, The 65 year-old Turing Test was passed for the very first time by computer program Eugene Goostman during Turing Test held at the Royal Society in London
- 'Eugene' simulates a 13 year old boy and was developed in Saint Petersburg, Russia. The development team includes Vladimir Veselov and Eugene Demchenko
- A program wins the Turing Test if it is mistaken for a human more than 30% of the time.

http://www.reading.ac.uk/news-and-events/releases/PR583836.aspx

CS 31007 Autumn 2020

COMPUTER ORGANIZATION AND ARCHITECTURE

Instructors
Rajat Subhra Chakraborty
Bhargab B. Bhattacharya

Lecture 03: September 07, 2020

Indian Institute of Technology Kharagpur
Computer Science and Engineering

Today’s Class

- von Neumann Architecture
- Instruction cycle of CPU
- Basic components of a computer
- IC manufacturing process
- Die yield and chip-cost
- CPU Performance Characterization

Acknowledgement: Patterson and Hennessy; V. D. Agrawal; S. R. Sarangi

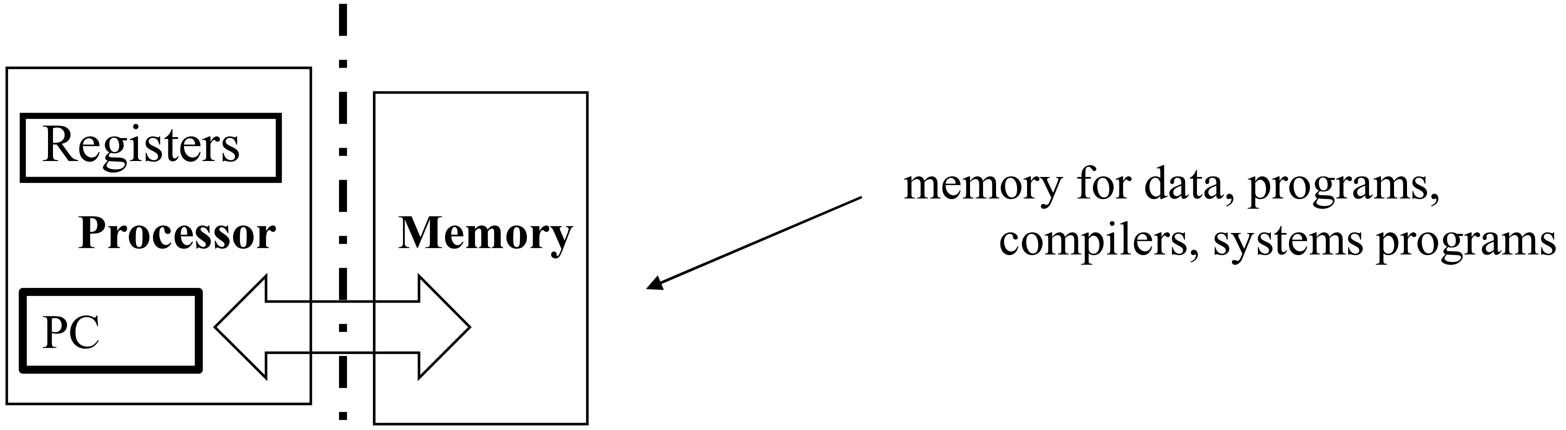
von Neumann Architecture (1945): Princeton Architecture

° Stored program concept

° Serves as the basis for almost all modern computers

° Instructions and data are just bits

° Programs (sequence of machine instructions) are stored in memory to be read or written just like data

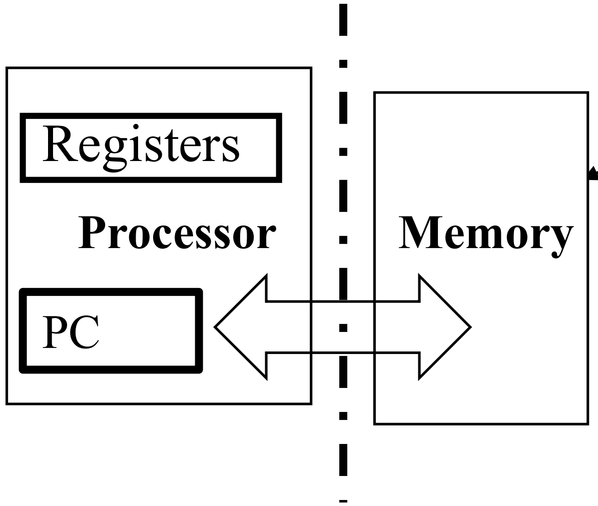


° Fetch & Execute Cycle (Instruction Cycle)

- Program Counter (PC) points to the next Instruction to be fetched
- Specific bits (opcode field) in the Instruction "control" subsequent actions
- Fetch the “next” instruction and continue

Binary String Stored in Computer: Who am I?

- X = - 1,880,113,152 (if X is a 2’s complement binary number)
- X = 2,414,854,144 (if X is an unsigned binary number)
- X = - 1.873 × 2^{-96} (if X is a floating-point number)
- X $\Rightarrow$ lw $t7, -16384 ($ra) (if X is a MIPS instruction)



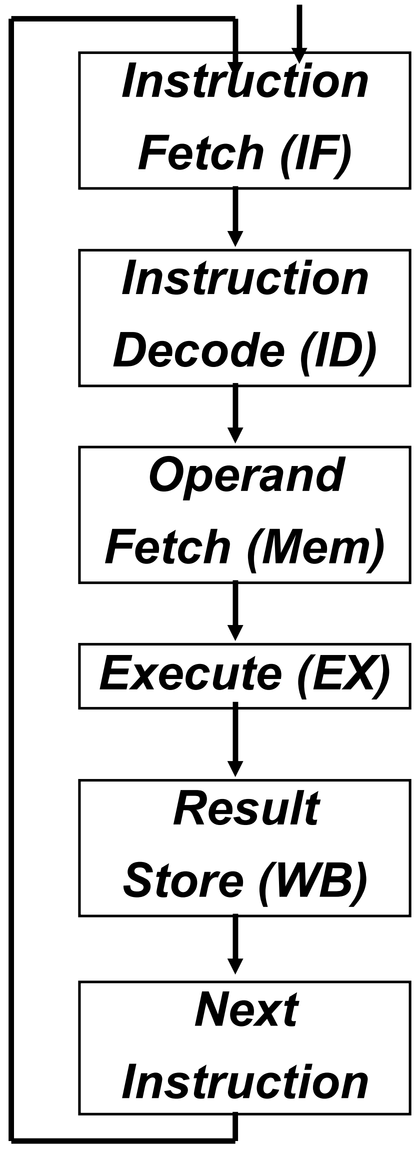
Let the content (X) of a memory location be:

X: 0x 8FEFC000

X: 1000 1111 1110 1111 1100 0000 0000 0000

- Need execution cycles to interpret the binary strings properly – whether it is instruction or data, and if data, what type?

Execution Cycle *a.k.a.* Instruction Cycle



Obtain instruction from memory

Determine required actions and instruction size

Locate and obtain operand data from memory/register (Mem)

Compute result value or status (Execute)

Deposit results in register/memory for later use (Write Back)

Determine successor instruction

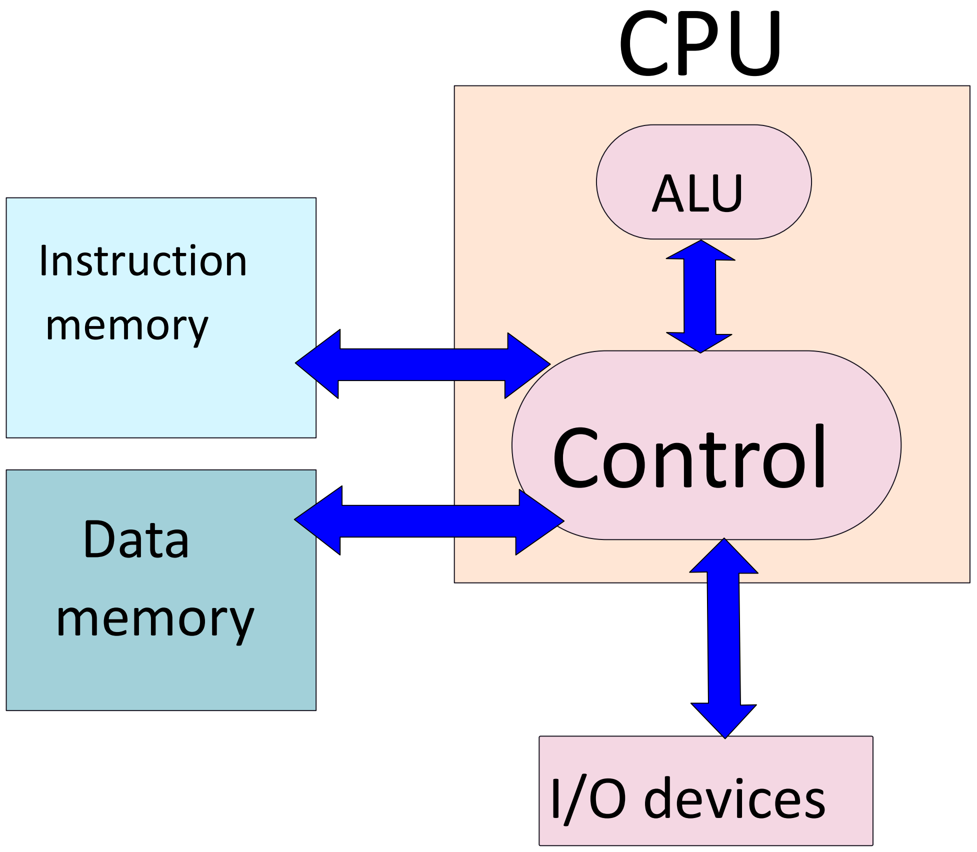
(Program Counter points to next instruction; next sequential memory address or branch targets)

Features of von Neumann Architecture

- Same physical memory to save instructions and data
- Instruction fetch and data transfer cannot be done concurrently; they need separate clock cycles
- Simple architecture

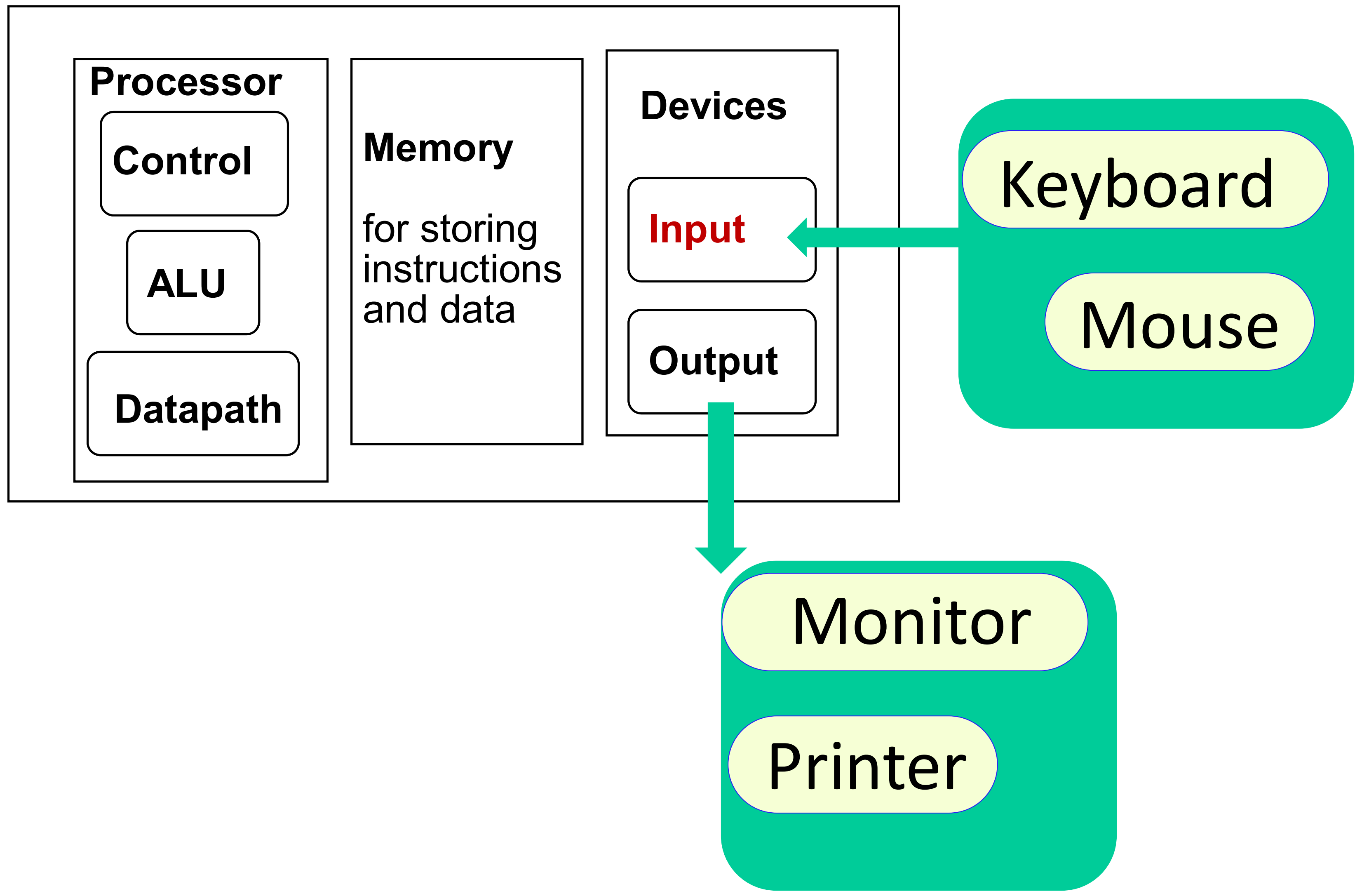
- Harvard Architecture: Separate instruction and data memory

Harvard Architecture



- Based on Harvard Mark I relay-based computer model
- Separate signal pathways for instruction and data; can be accessed concurrently

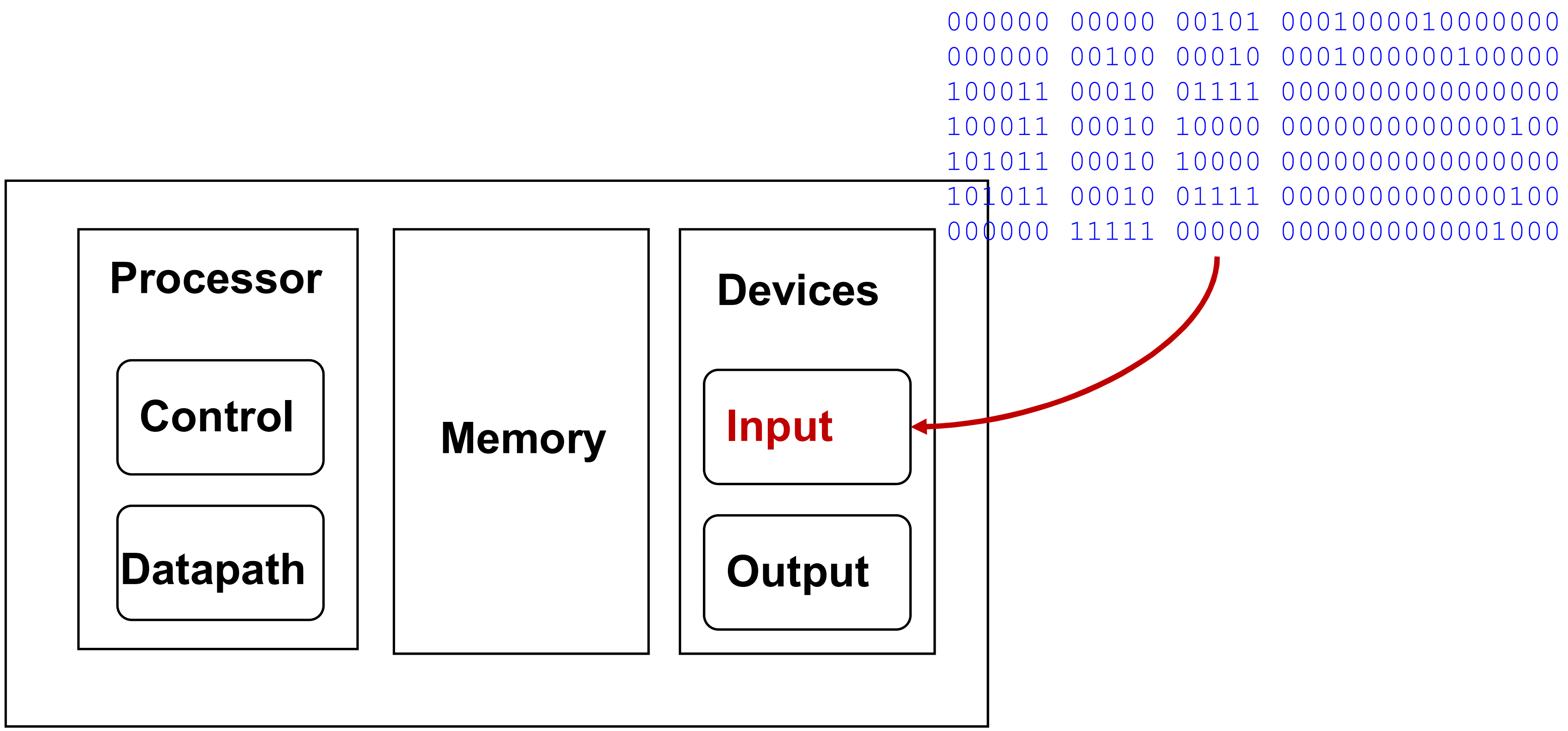
Full Picture



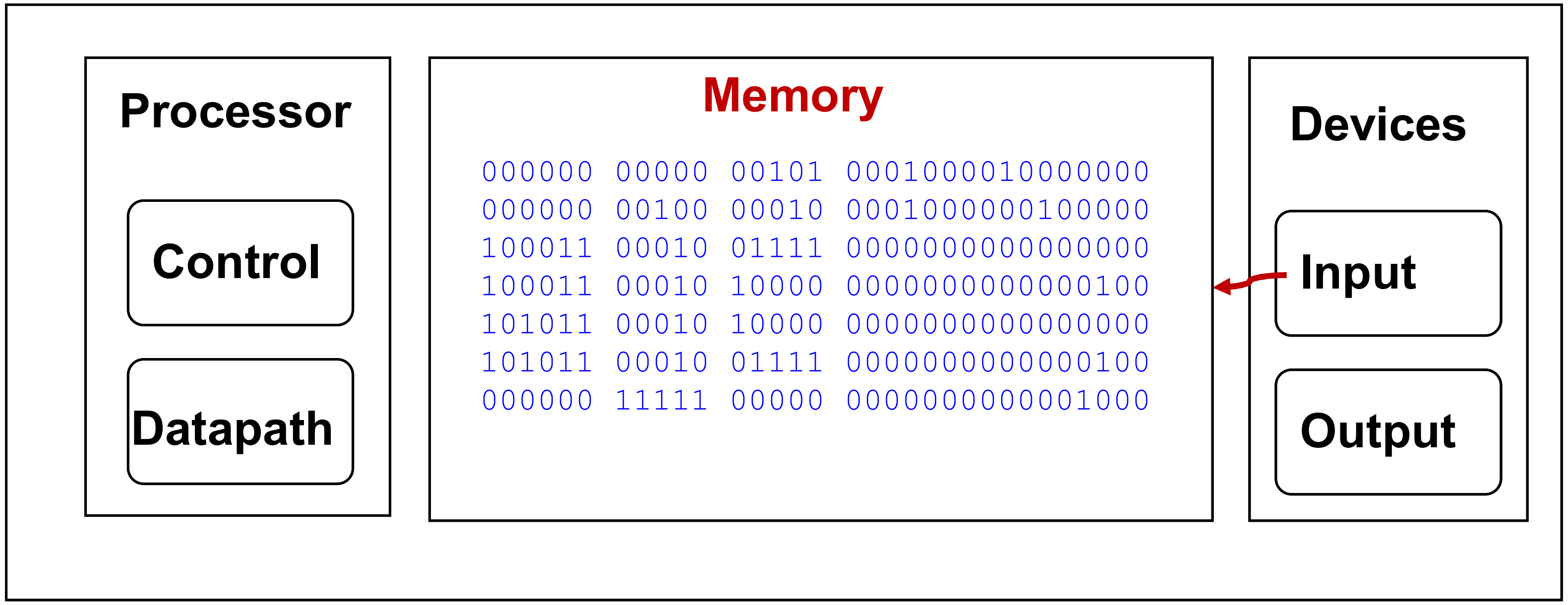
Next ..

- How programs are translated into the machine language
  - And how the hardware executes them
- The hardware/software interface
- What determines program performance
  - And how it can be improved
- How hardware designers improve performance
- How parallel processing helps

Load Input Binary


```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

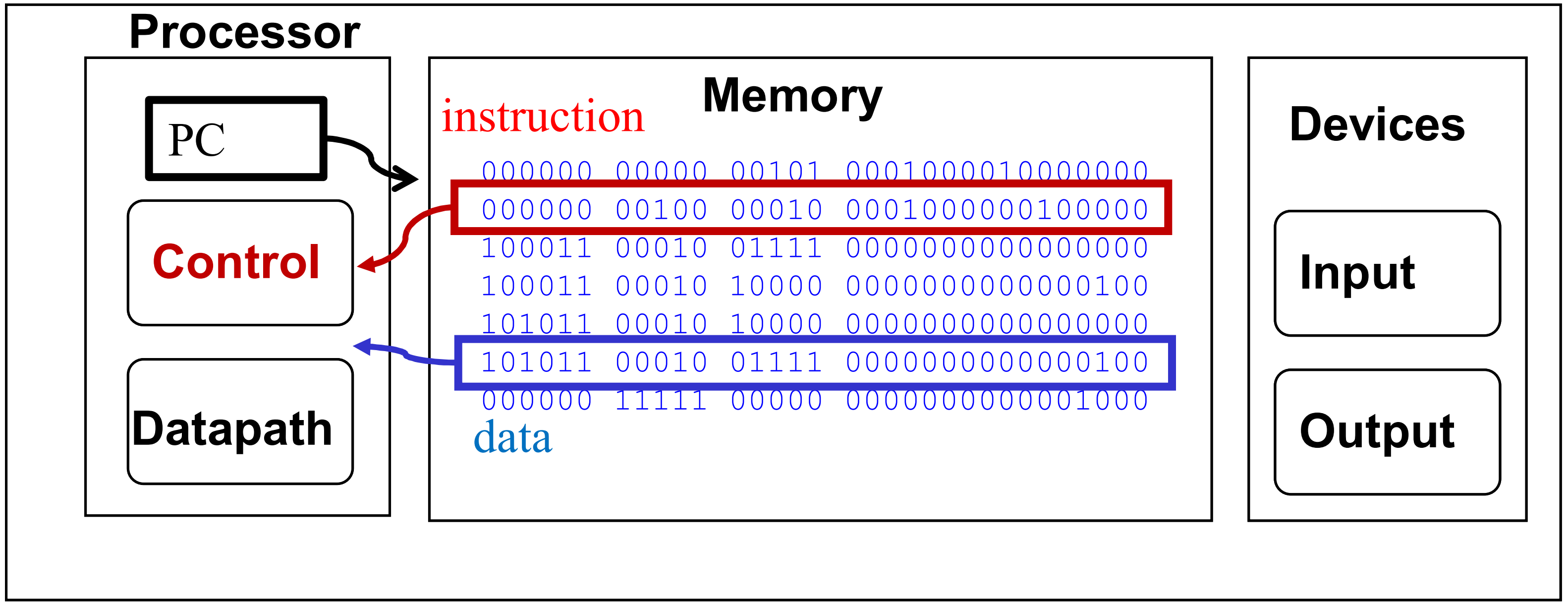
Code Stored in Memory



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

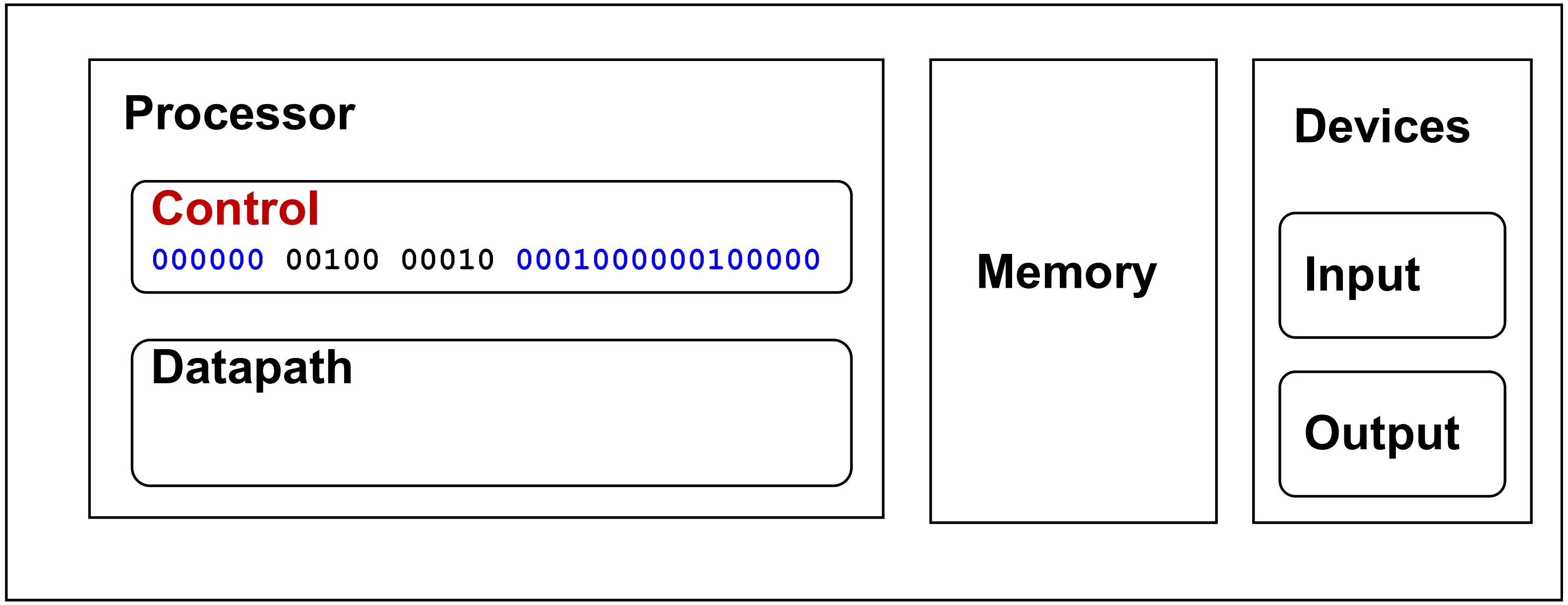
Processor Fetches an Instruction

Processor fetches an instruction from memory pointed by Program Counter PC



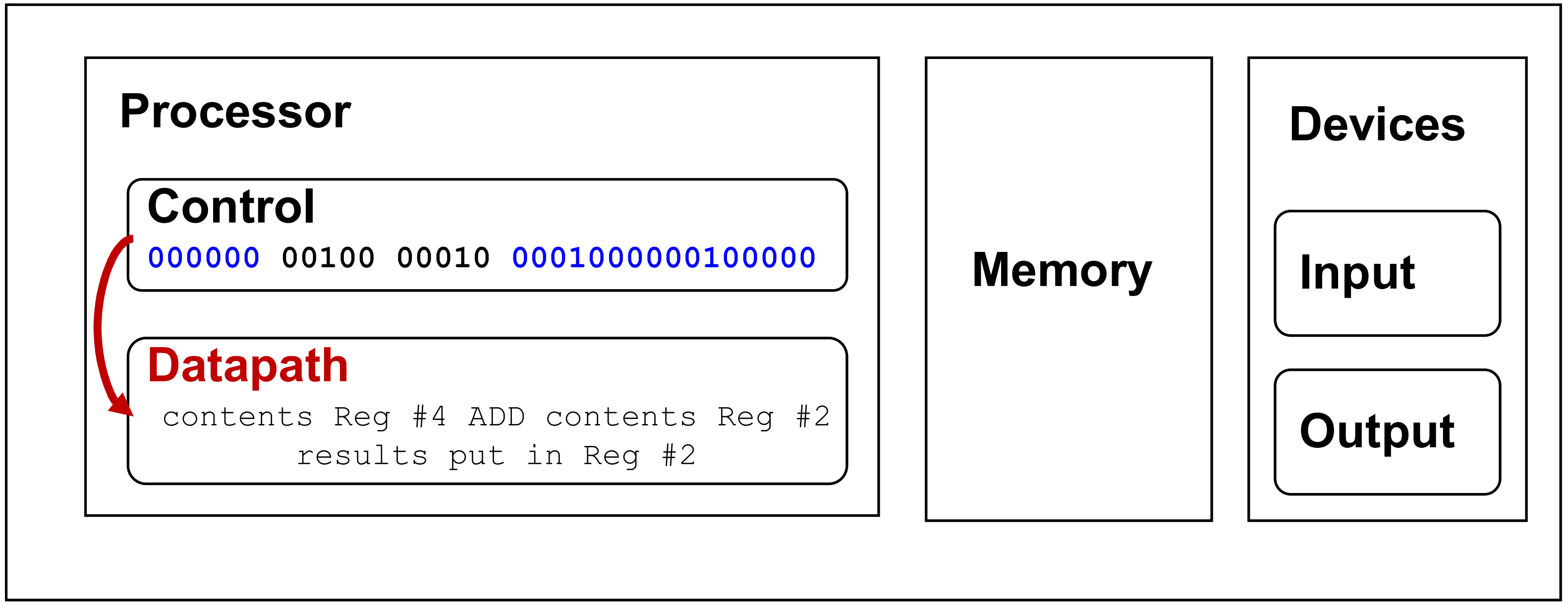
Where does it fetch from?

Control Decodes the Instruction



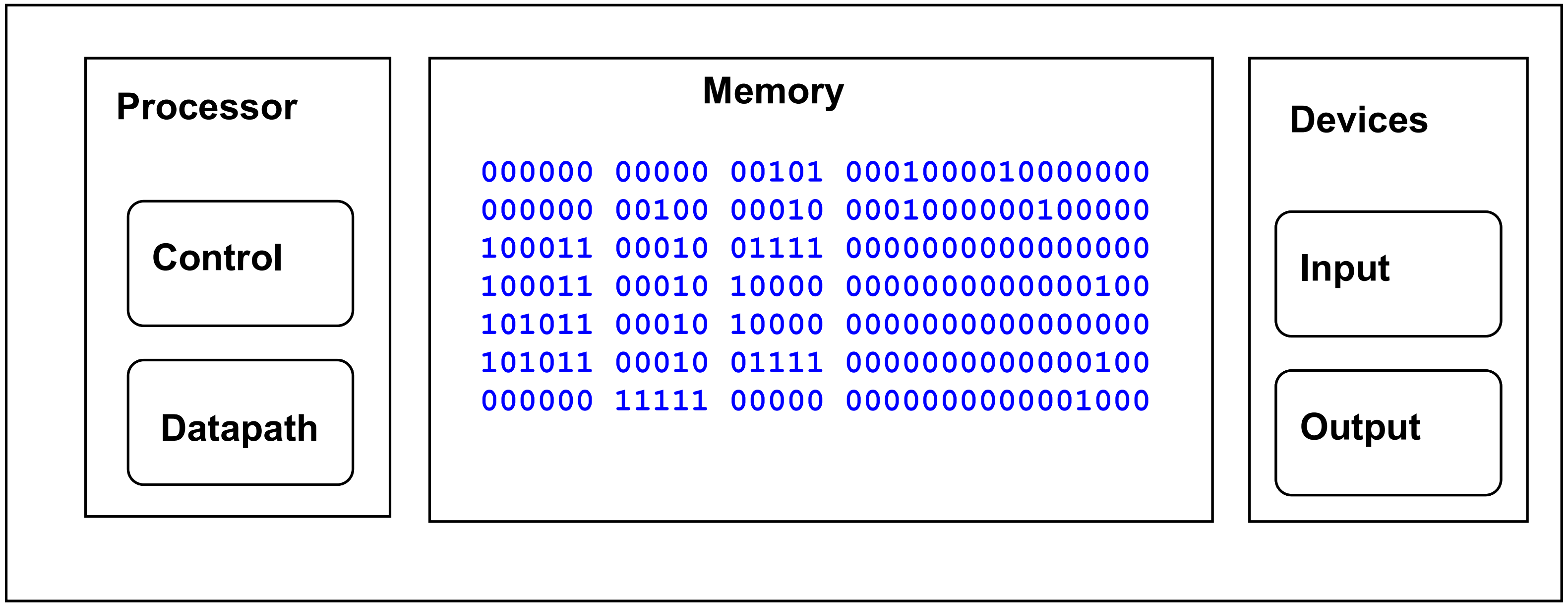
Control decodes the instruction to determine what to execute

Datapath Executes the Instruction

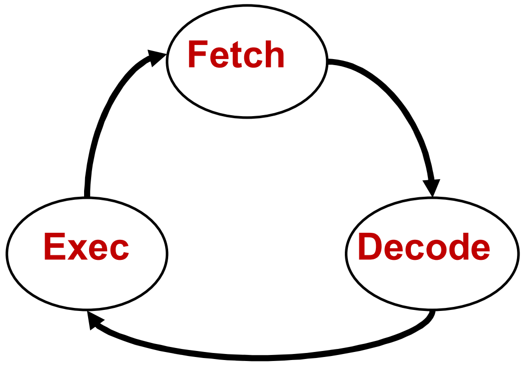


Datapath executes the instruction as directed by control

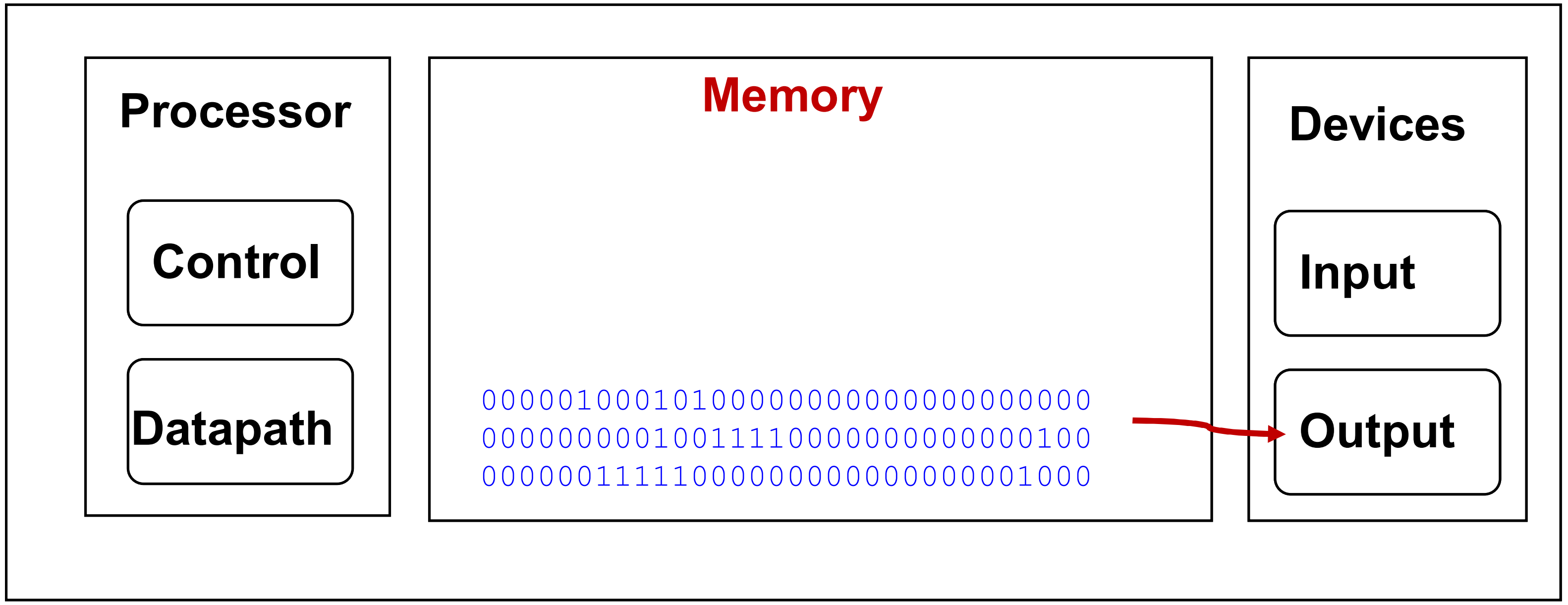
What Happens Next?



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

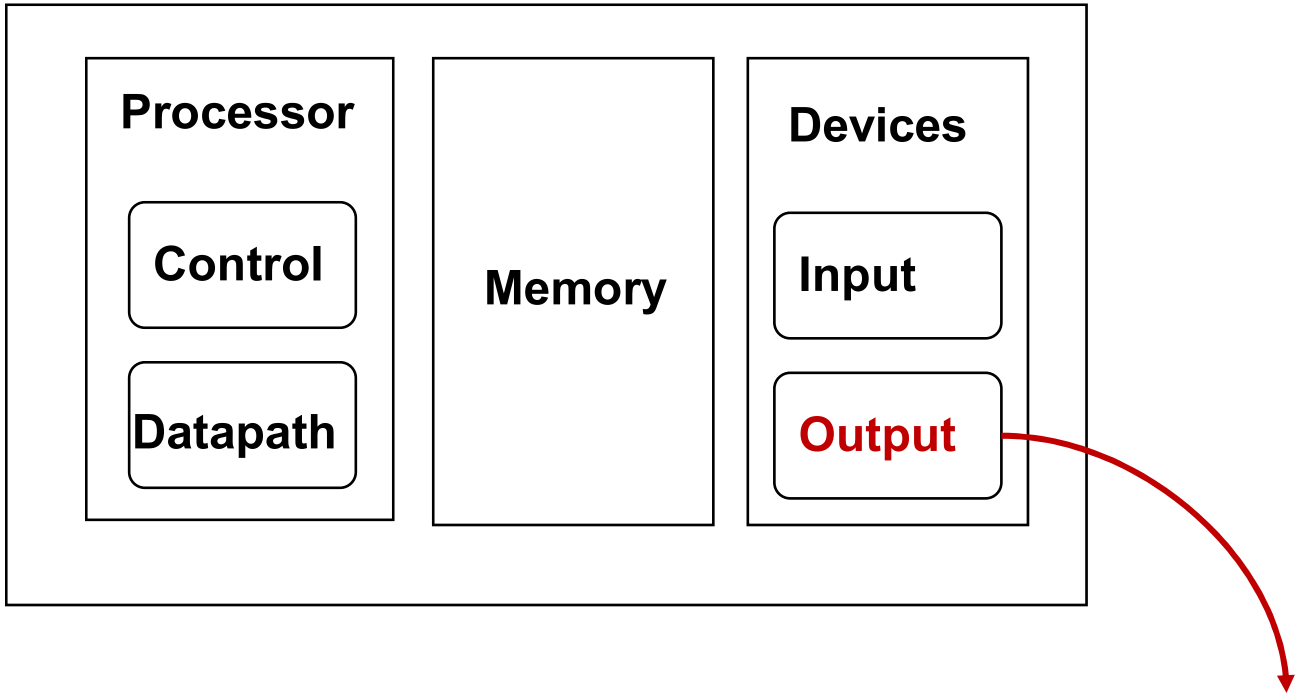


Output Data Stored in Memory



On program completion, results reside in memory

Output Device Outputs Data

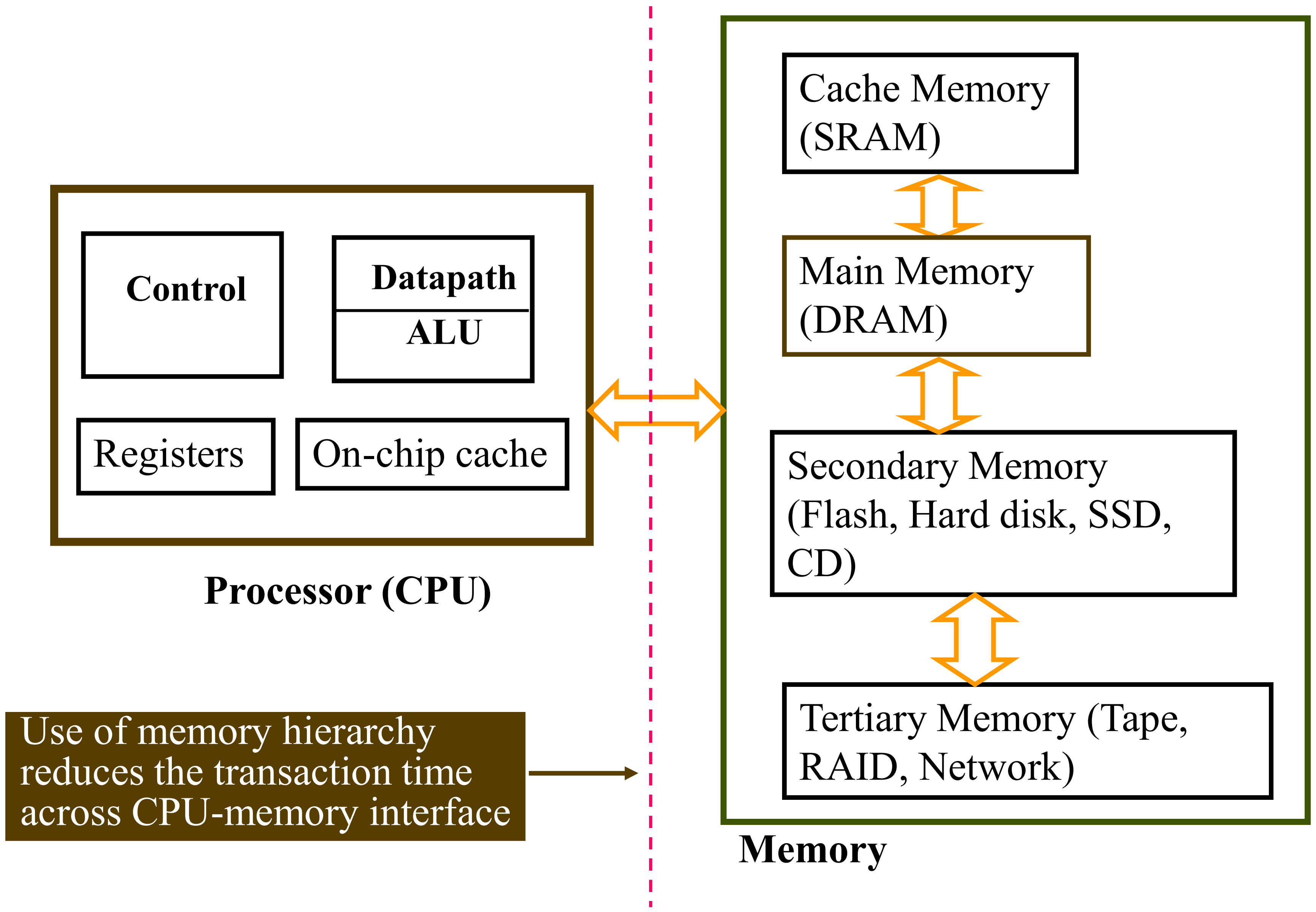


```
00000100010100000000000000000000
00000000010011110000000000000100
00000011111000000000000000001000
```

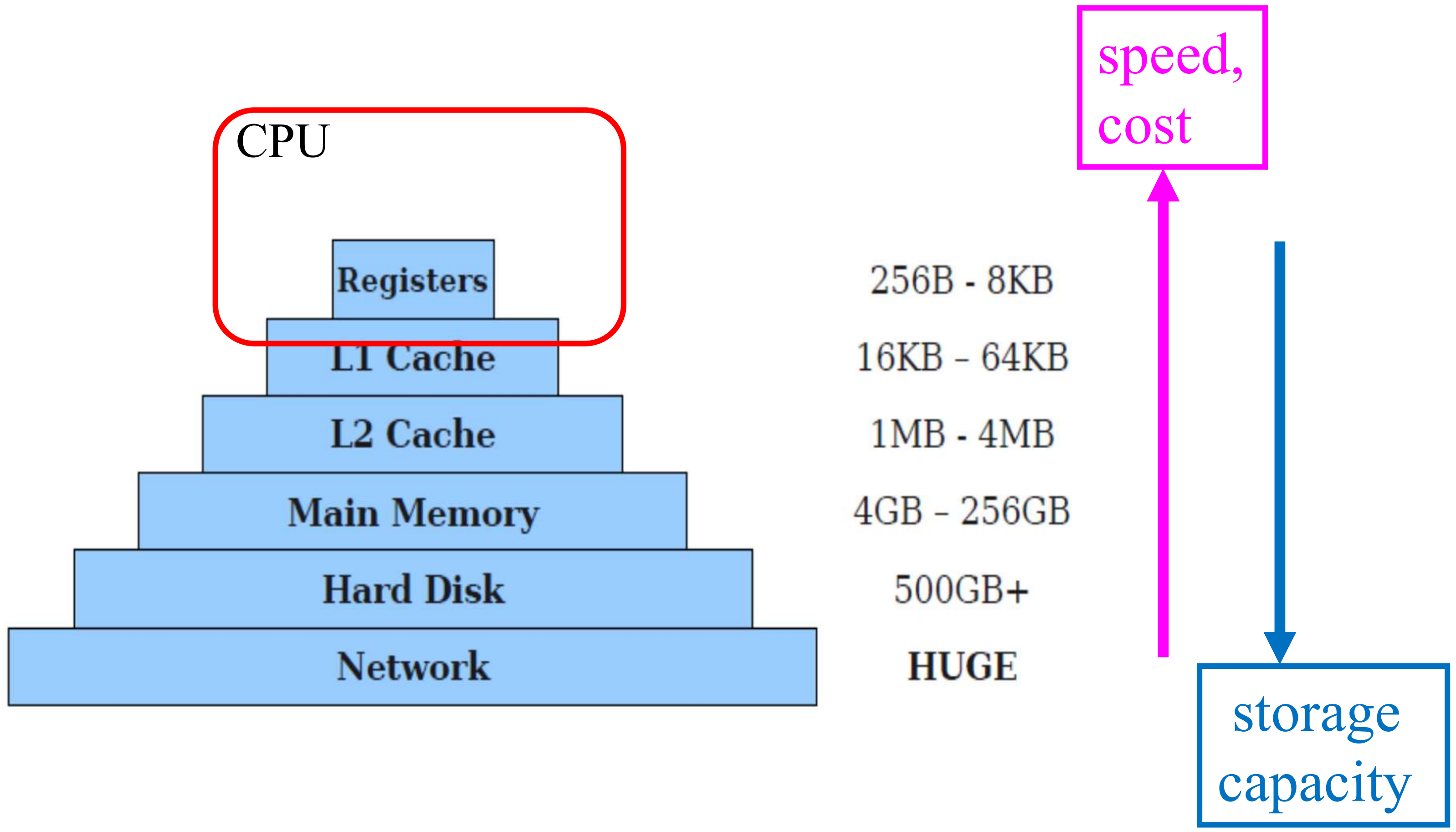
Von Neumann Bottleneck

- Von Neumann architecture uses the same memory for instructions (program) and data.
- Memory is inherently slower than logic. The time spent in memory accesses can limit the performance. This phenomenon is referred to as *von Neumann bottleneck*.
- To avoid the bottleneck, later architectures allow frequently used operands to reside in on-chip registers.

Complete View of Computer Architecture


Control
Datapath
ALU
Registers
On-chip cache
Processor (CPU)
Cache Memory (SRAM)
Main Memory (DRAM)
Secondary Memory (Flash, Hard disk, SSD, CD)
Tertiary Memory (Tape, RAID, Network)
Memory
Use of memory hierarchy reduces the transaction time across CPU-memory interface

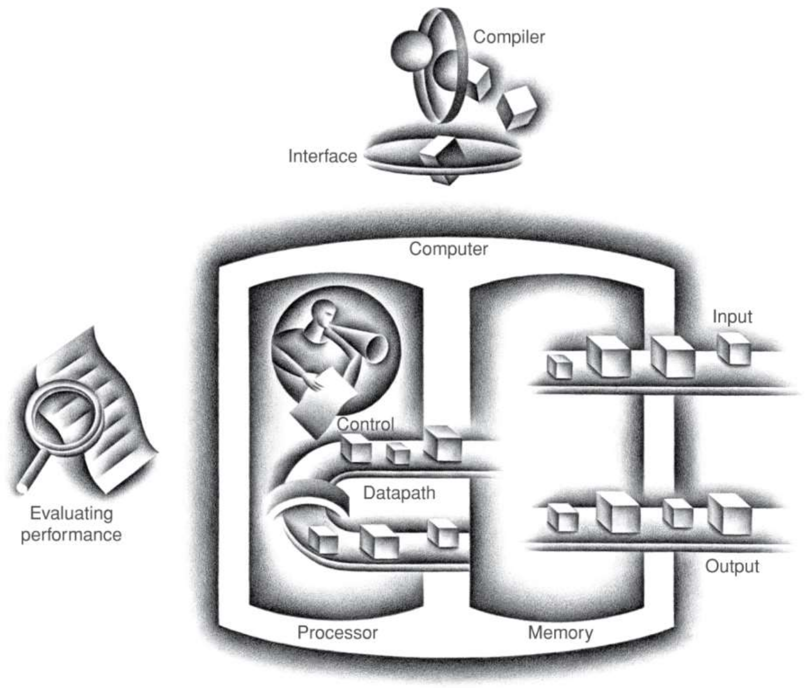
Memory Hierarchy (MH)



MH reduces average CPU-Memory transaction time;
need additional hardware, OS (memory management)

Components of a Computer

The BIG Picture



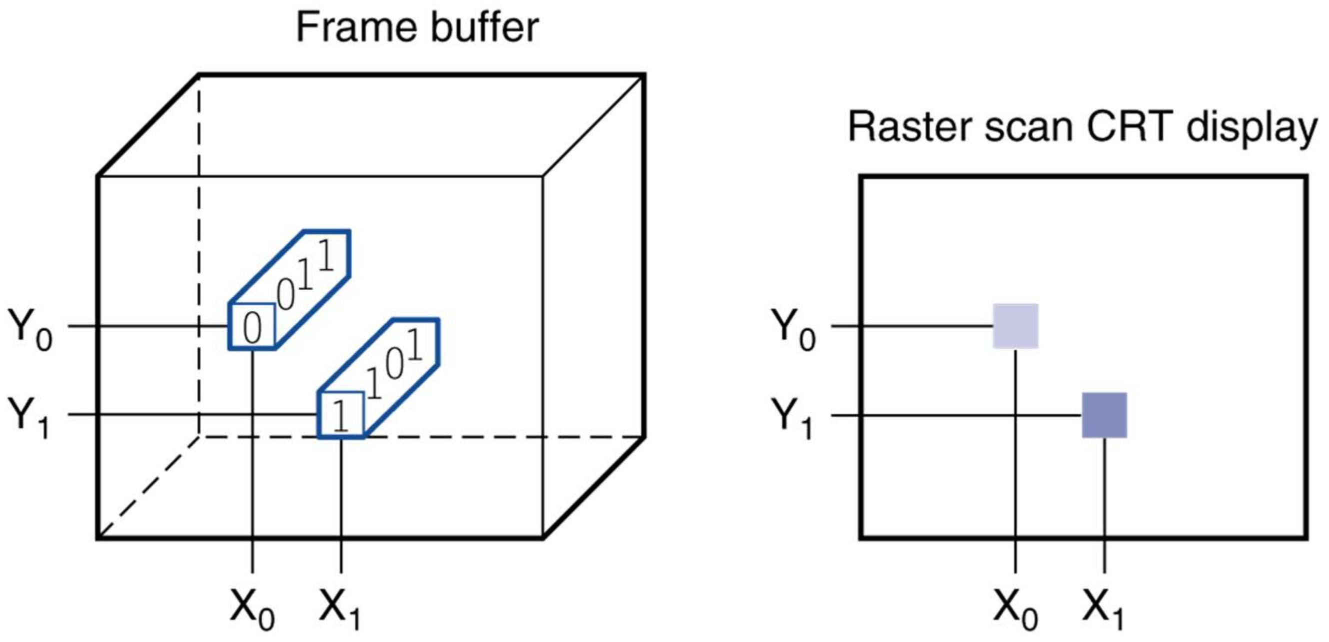
- Same components for all kinds of computer
  - Desktop, server, embedded
- Input/output includes
  - User-interface devices
    - Display, keyboard, mouse
  - Storage devices
    - Hard disk, CD/DVD, flash
  - Network adapters
    - For communicating with other computers

Touchscreen

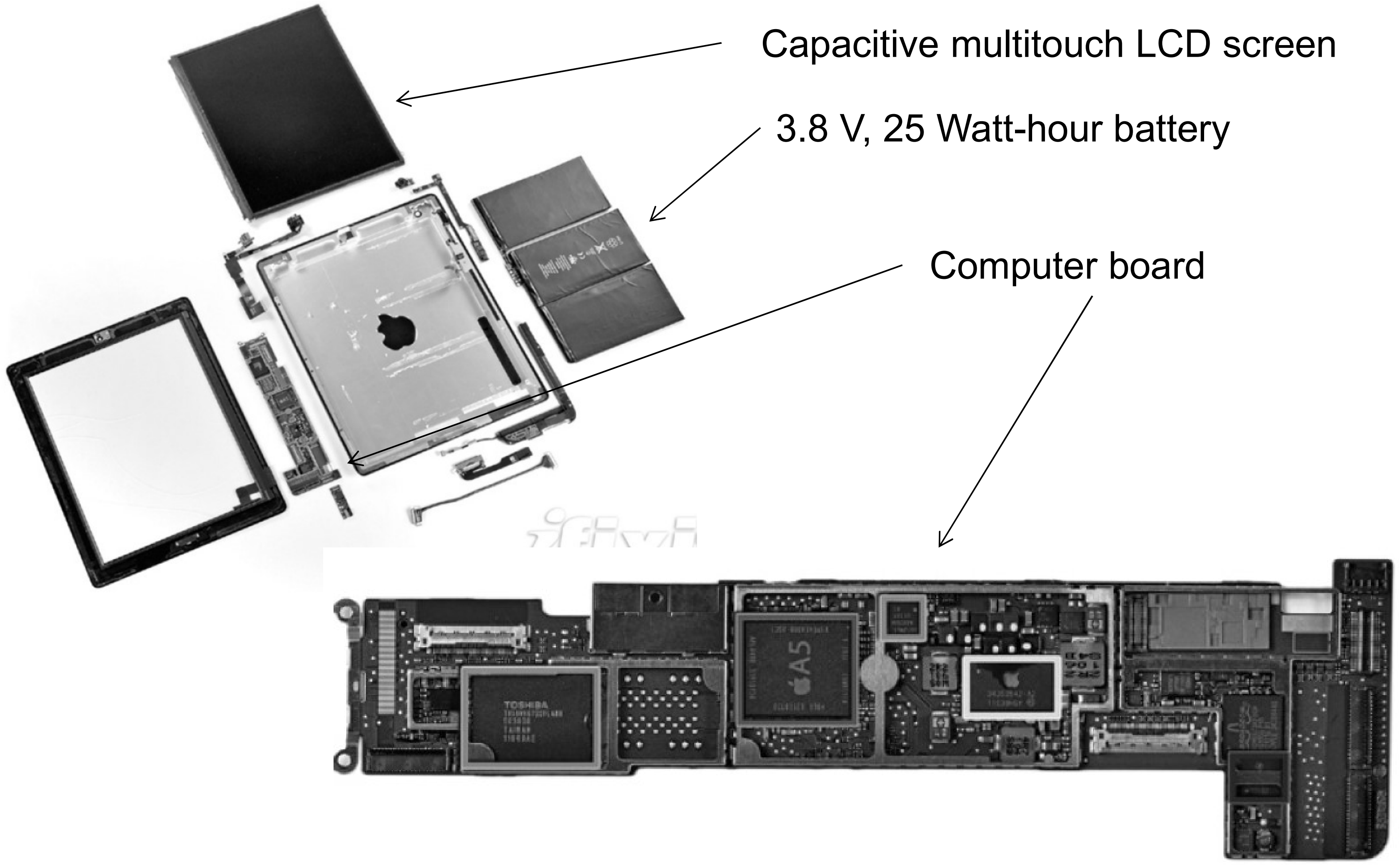
- PostPC device
- Supersedes keyboard and mouse
- Resistive and Capacitive types
  - Most tablets, smart phones use capacitive
  - Capacitive allows multiple touches simultaneously

Through the Looking Glass

- LCD screen: picture elements (pixels)
  - Mirrors content of frame buffer memory



Opening the Box

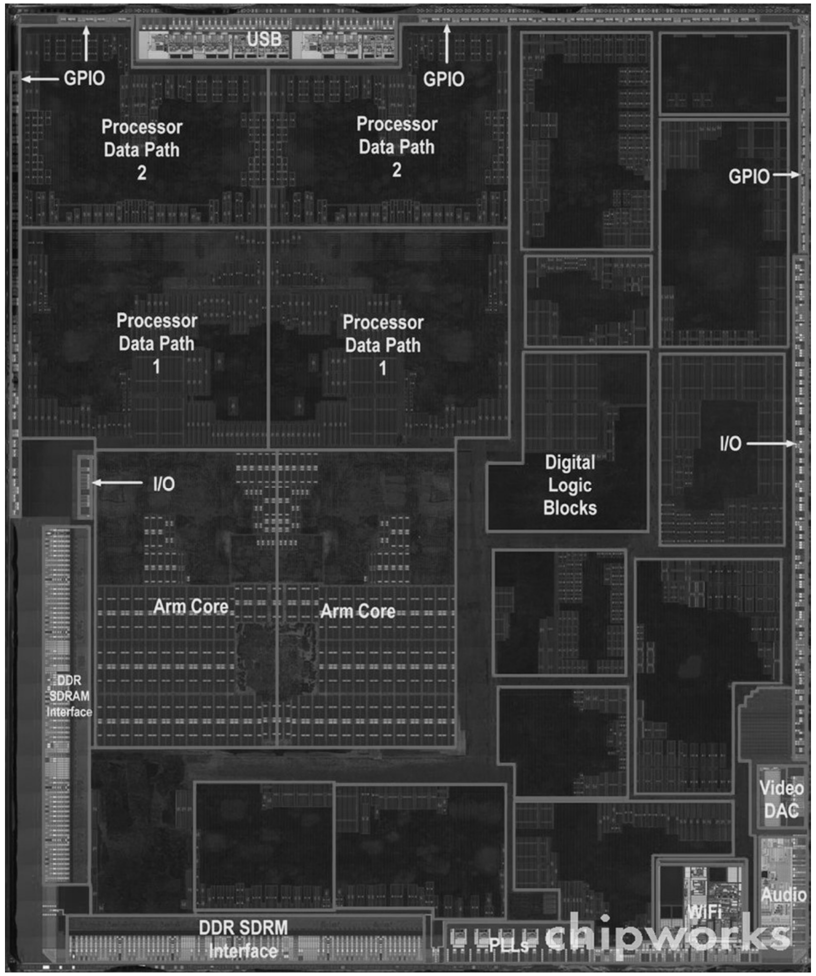


Inside the Processor (CPU)

- Datapath: performs operations on data
- Control: sequences datapath, memory, ...
- Cache memory
  - Small fast SRAM memory for immediate access to data

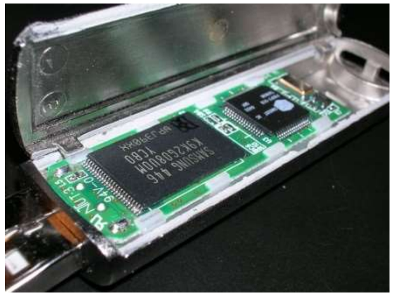
Inside the Processor

- Apple A5



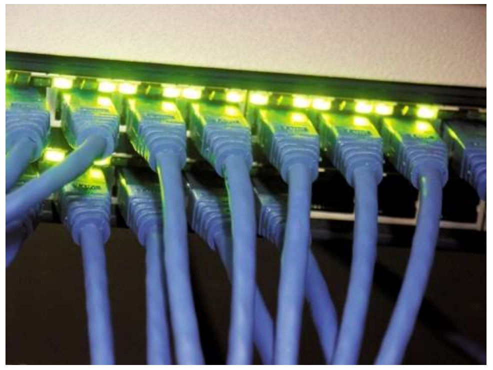
A Safe Place for Data

- Volatile main memory
  - Loses instructions and data when power off
- Non-volatile secondary memory
  - Magnetic disk
  - Flash memory, SSD
  - Optical disk (CDROM, DVD)



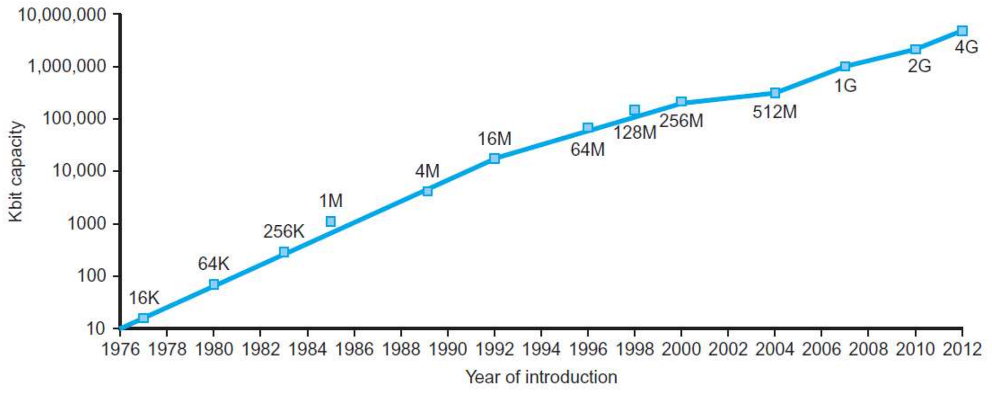
Networks

- Communication, resource sharing, nonlocal access
- Local area network (LAN): Ethernet
- Wide area network (WAN): the Internet
- Wireless network: WiFi, Bluetooth



Technology Trends

- Electronics technology continues to evolve
  - Increased capacity and performance
  - Reduced cost

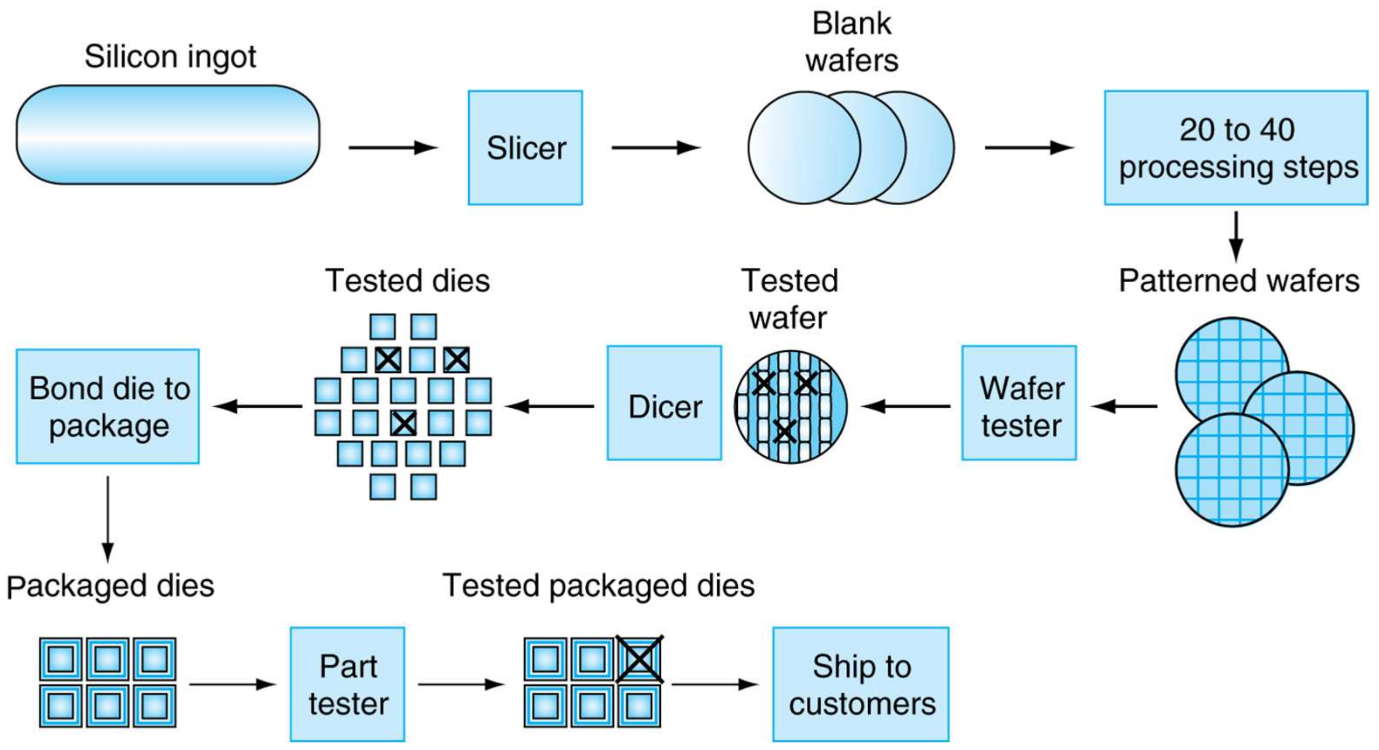


DRAM capacity

Semiconductor Technology

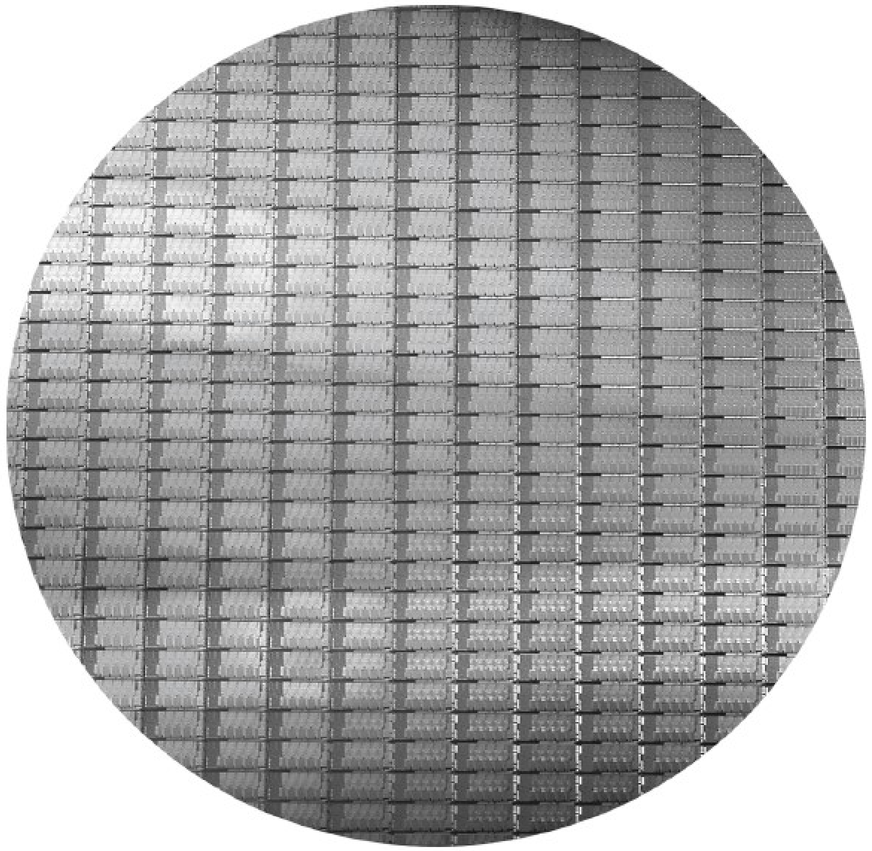
- Silicon: semiconductor
- Add materials to transform properties:
  - Conductors
  - Insulators
  - Switch

Manufacturing ICs



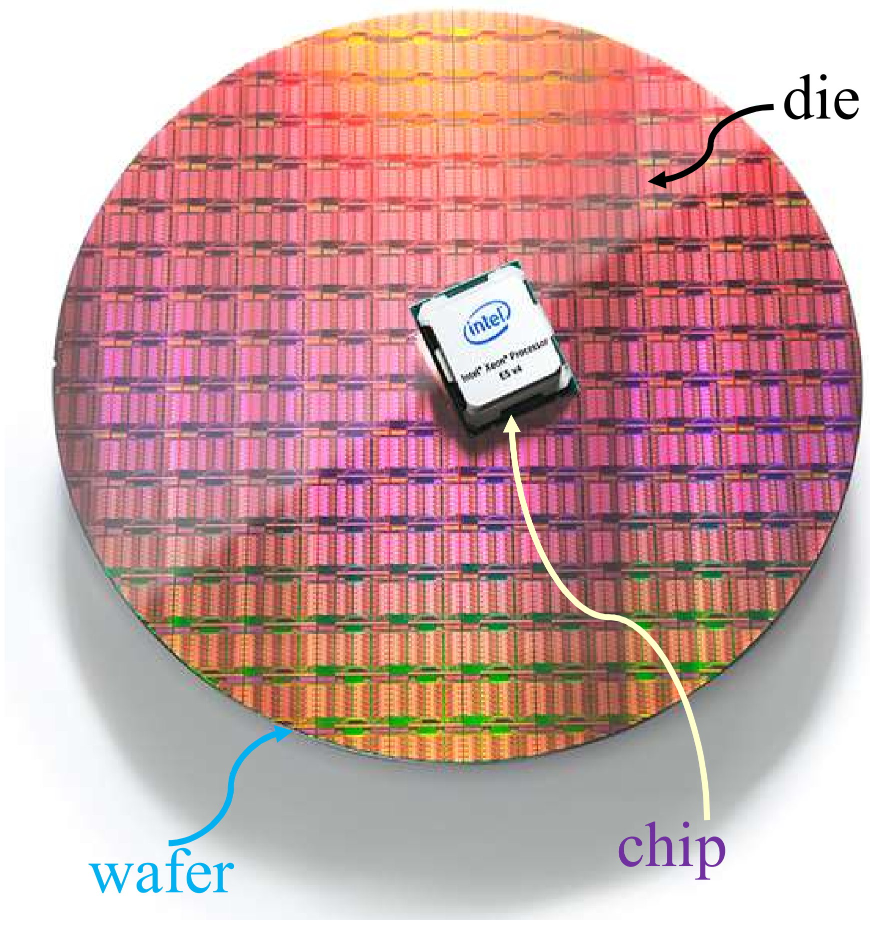
- Yield: fraction of working dies per wafer

Intel Core i7 Wafer

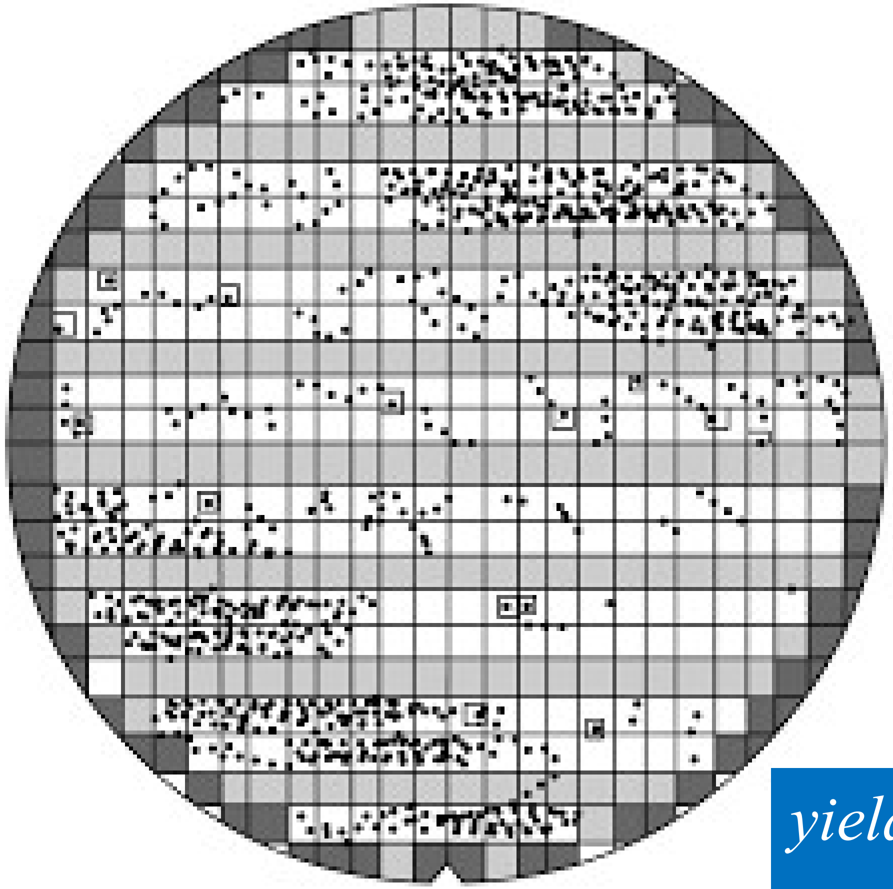


- 300mm wafer, 280 dies, 32*nm* technology, 4-8 cores per die
- Each die is 20.7 x 10.5 *mm*

Intel Xeon E5-2600 v4 chips with 22 processors, 14*nm process technology* (2016)



Defects in IC's and Yield



defects on chips make them unusable;

peripheral chips are incompletely fabricated, and hence unusable (square-peg in round-hole problem);

yield: fraction of good chips produced;

chip-cost depends on yield;

Integrated Circuit Cost

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{Yield}}$$

$$\text{Dies per wafer} \approx \text{Wafer area} / \text{Die area}$$

$$\text{Yield} = \frac{1}{[1 + \{(\text{Defects per unit area of chip} \times \text{chip area})/\alpha\}]^{\alpha}}$$

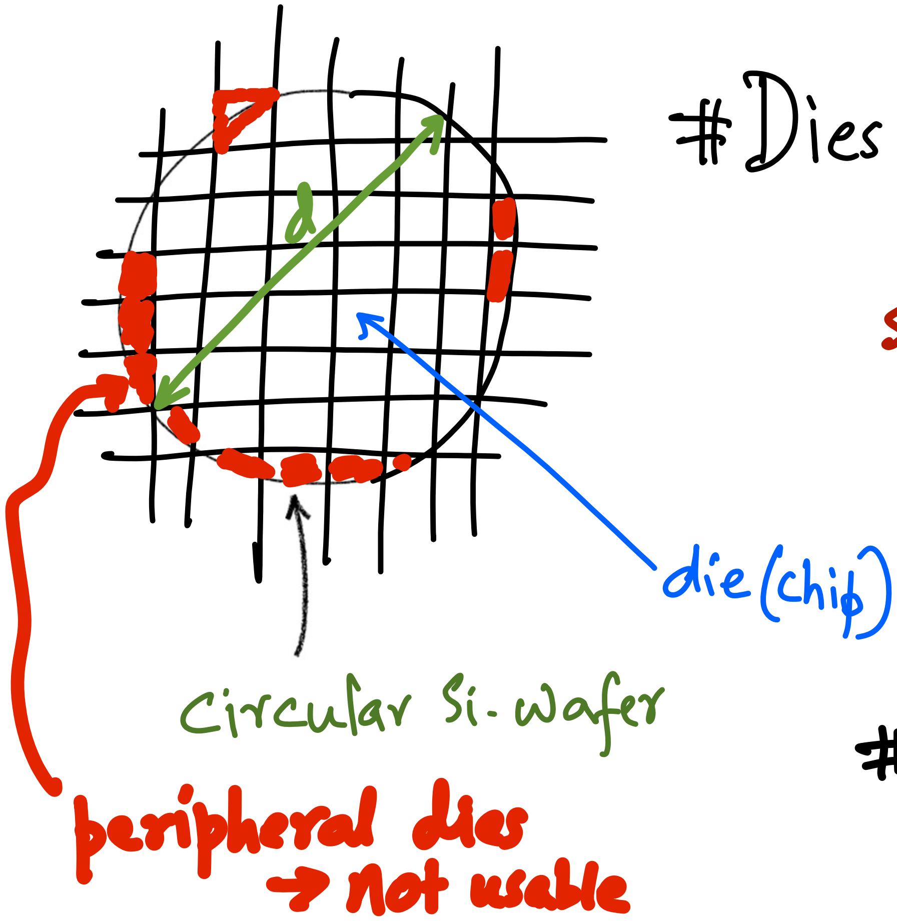
chip-cost ~ $(\text{die-area})^4$

This is an empirical estimation of yield; α is a parameter that roughly captures the complexity of the manufacturing process

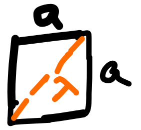
- Nonlinear relation to area and defect rate
  - Wafer cost and area are fixed
  - Defect rate determined by manufacturing process
  - Die area determined by architecture and circuit design

Dies per wafer: More accurate estimation

$$\#\text{Dies per wafer} \approx \frac{\overset{\pi\left(\frac{d}{2}\right)^2}{\text{wafer-area}}}{\text{die-area}} - \#\text{peripheral dies}$$



square-peg in round-hole problem

Assume square die 

$$\text{die-area} = a^2$$

$$\lambda = \text{diagonal length} = \sqrt{2a^2} = \sqrt{2 * \text{diearea}}$$

$$\#\text{peripheral dies} \approx \frac{\text{wafer-circumference}}{\lambda}$$

$$= \frac{\pi d}{\lambda}$$

Hence,

$$\#\text{Dies per wafer} = \frac{\pi * (d/2)^2}{\text{die.area}} - \frac{\pi \cdot \overset{\text{wafer diameter}}{d}}{\sqrt{2 * \text{die area}}}$$

Example: Estimate the number of usable dies that can be obtained from a 20-cm diameter wafer, where each die is a square with 1.5 cm on each side.

Answer: $\#\text{Dies} = \frac{\pi * 10^2}{2.25} - \frac{\pi * 20}{\sqrt{2 * 2.25}} \approx 110$ ☒

(assuming zero defect)

Estimating good chips per wafer $\Rightarrow$ chip cost

good chips per wafer:

$$= \text{\# Dies per wafer} * \text{wafer-yield} * \beta$$

where,

$$\beta = \frac{1}{\left(1 + \frac{(\text{defect density} * \text{die area})}{\alpha}\right)^{\alpha}}$$

$\alpha \approx 2, 3$ for semiconductor processes.

Example : Die yield

Problem : Find the die yield for a chip that is of square shape with 1 cm on each side. Defect density is $0.8/cm^2$ and assume $\alpha = 3$.

Solution Die yield $= \left(1 + \frac{0.8 \times 1}{3}\right)^{-3} = 0.49$

$\Rightarrow$ 49% (i.e., 51% chips are wasted)

$$\text{Cost per chip} \propto (\text{die area})^4$$

Reduction of die area

- ISA
  (simple & regular instructions need low-cost logic
- optimize logic synthesis
- Architect's decision on functionality
  - capability
  - I/O-pins

Registers

Registers are high-speed storage units located in the CPU. They are strings of FF's.

Registers $\rightarrow$ need ID $\rightarrow$ (# registers) determine the length of instruction $\rightarrow$ ISA

Example: $\boxed{\text{add } \$t_0, \$s1, \$s2}$

($t_0 \leftarrow (s1) + (s2)$)

one H/L instruction in MIPS

$\boxed{\text{\# registers} \rightarrow \text{die area} \rightarrow \text{cost}}$

Compiler: Optimal register allocation

$f := (a+b) - (c+d)$

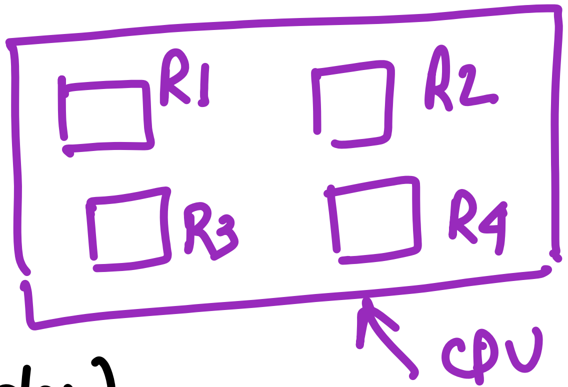
$x := f * c^2$

$y := x * a$

$z := x + y$

$w := a - b$

Bring a, b, c, d to on-chip registers



live variables → registers

Mem access is slow

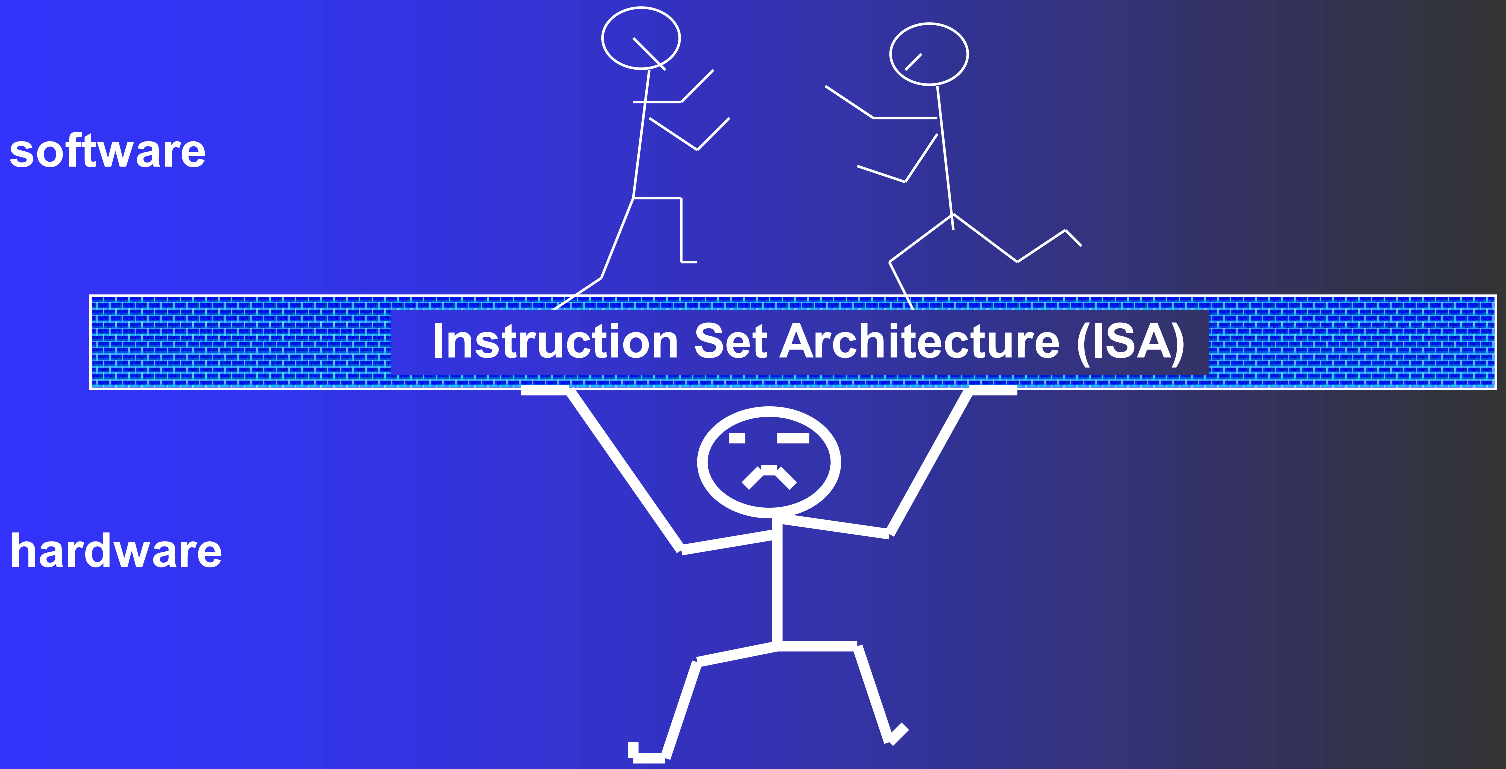
Summary: Elements of a Computer

- Memory (array of bytes) contains
  - The program, which is a sequence of instructions
  - The program data → variables and constants
- The program counter (PC) points to the memory location where the next instruction in the program is located
  - After executing an instruction, it again points to the next instruction by default
  - A branch instruction makes the PC point to another instruction (not in sequence)
- CPU (Central Processing Unit) contains
  - Program counter, instruction decoder and control logic, arithmetic logic unit (ALU), data paths
- Manufacturing defects in Integrated Circuits (IC), production yield, die cost and computer pricing

Today’s Class

- ❖ Instruction Set Architecture (ISA)
- ❖ Features of ISA
- ❖ How ISA determines chip cost
- ❖ CPU performance equation
- ❖ Improving CPU-Performance: Amdhal’s Law *versus* Gustavson-Barsis Law

ISA: Hardware-Software Interface



ISA: Collective attributes of the machine-language instruction set

Courtesy: Patterson and Hennessy

Instruction Set Architecture (ISA)

- The set of machine-level instructions in a particular CPU implementation is called *Instruction Set*

- Goals of Instruction Set:
  - Software must be able to compute anything in a reasonable number of steps using the instructions in the instruction set
- Different CPUs implement different sets of instructions

Features of Instruction Set Architecture (ISA)

Instruction format: length (how many bits – fixed or variable?), format, fields, opcodes, register specifications, how many operands/memory addresses specified?
Size of the logical address space? **Addressability** (byte/word level)?
Number of instructions? – determines #bits in op-code.
Instruction types? – arithmetic (integer/floating point), logical, data transfer, branch, procedure calls, bit-shifting
Addressing modes: immediate, direct, register, displacement, scaled, indirect; addressing granularity (word-level, byte-level?)
Others: Orthogonality, Completeness, Alignment (Big-Endian/Little-Endian)

ISA governs both hardware implementation (downward) and compiler design (upward)

Example: MIPS ISA

- The length of every instruction = 32 bits
- Memory is byte-addressable, word-organized, each word comprising 4 bytes = 32 bits
- Width of the address bus: 32 bits; size of the logical address space = 2^32 bytes =2^30 words
- 32 on-chip general purpose registers, each 32-bit wide; register-ID needs 5 bits each
- Integers, FP-numbers: 32 bits
- Op-code in each instruction: 6 bits

Example: MIPS Instruction

Collection of Assembly-level or Machine-level (M/L) instructions, which are executable by the hardware

An example of MIPS M/L instruction: `addi $t0, $s1, 5`

op	s1	t0	Immediate
6 bits	**5 bits**	**5 bits**	**16 bits**
001000	10001	01000	0000000000000101

Total length of the instruction – 32 bits; *opcode*: 6 bits; $t0, $s1 -> 32-bit CPU *registers* (ID: 5-bit each, as there are a total of 32 such registers); *immediate value* is provided as a 16-bit 2's complement integer;

Add 5 to the content of $s1 and save the result in $t0

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

  ```
  addi $s2, $s1, -1
  ```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a *load* instruction, which memory access

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster

  Compare to: main memory → millions of locations

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4; PC <- PC +4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Registers

Registers are high-speed storage units located in the CPU. They are strings of FF's.

Registers → need ID → (# registers) determine the length of instruction → ISA

Example: $\boxed{\text{add \$t0, \$s1, \$s2}}$

one H/L instruction in MIPS

$(t0 \leftarrow (S1) + (S2))$

$\boxed{\text{\# registers} \rightarrow \text{die area} \rightarrow \text{cost}}$

Compiler: Optimal register allocation

$f := (a+b) - (c+d)$

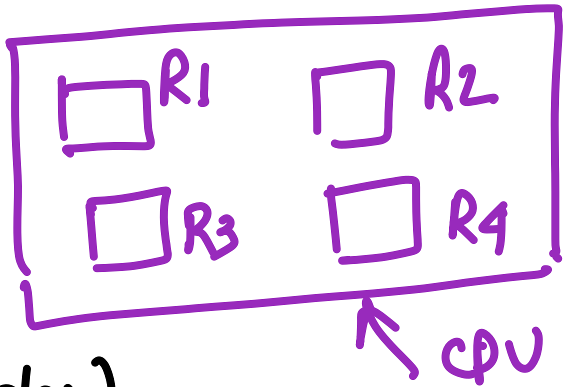
$x := f * c^2$

$y := x * a$

$z := x + y$

$w := a - b$

Bring a, b, c, d to on-chip registers



live variables $\rightarrow$ registers

Mem access is slow

Performance Issues

-- CPU-Performance Equation

-- Amdahl’s Law

What Affects Performance?

- Algorithm
  - Determines number of operations executed
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation →depends on ISA
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (including OS)
  - Determines how fast I/O operations are executed

Response Time and Throughput

- Response time
  - How long it takes to do a task
- Throughput
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?
- We'll study their estimation

- **Response time:** time between submission of a job (program P) and its completion (depends on overall system load)

This includes

--- I/O

--- Operating system time for managing programs, compile time, etc.

--- **CPU-time** includes time for executing the machine code for P, memory access time, procedure calls, and system time spent on P.

- Performance is proportional to the *inverse* of the CPU time.

What determines the execution time of a machine/assembly-level program *P* when it is run on a machine *M*?

- *P* consists of a number of machine-level instructions (IC: *instruction count*);
- Each machine instruction requires several clock cycles to complete (CPI: average number of *clock cycles per instruction*);
- Each clock cycle has certain time period (CCT: *clock cycle time*)

Thus, CPU-time = IC × CPI × CCT

(CPU Performance Equation)

Example (Compute CPI & CPU-time)

Problem Statement

A program P has 1000 M/L instructions and is being executed on a machine M running at 1 GHz clock speed.

ALU-op: 40%, Clock-cycles needed = 1
load-op: 20%, Clock-cycles needed = 2
store-op: 10%, Clock-cycles needed = 2
branch-op: 30%, Clock-cycles needed = 2

Computing CPI ⇒

Hence CPI = 1*0.4 + 2*0.2 + 2*0.1 + 2*0.3
= 1.60

CPI = 1.60

IC = 1000

CCT = 1 GHz

Hence, CPU_time

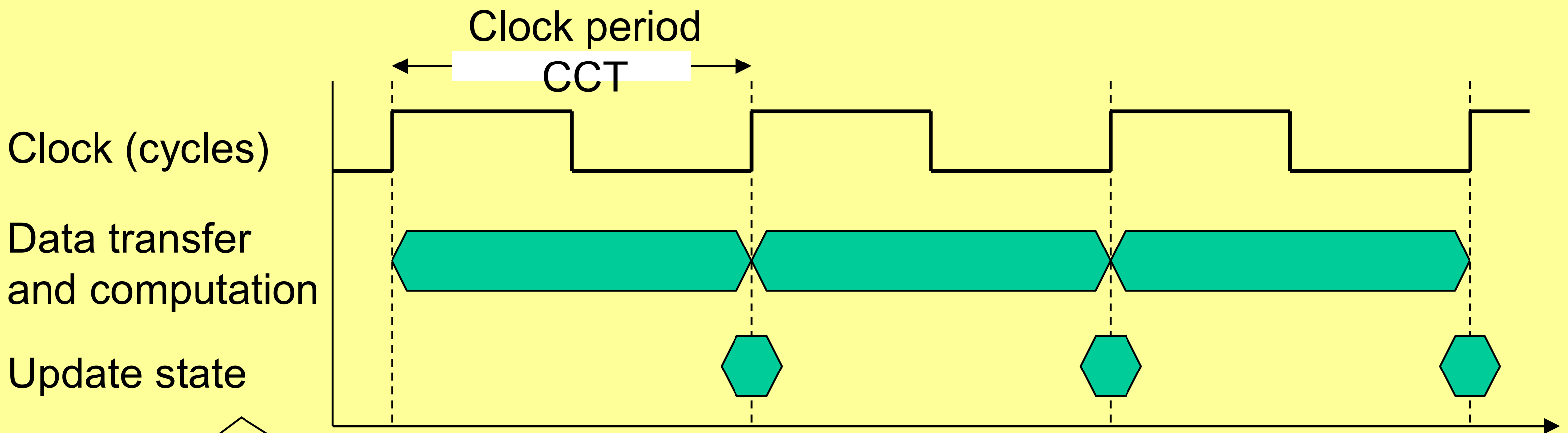
$$= IC * CPI * CCT$$

$$= 1000 * 1.60 * \frac{1}{10^9} \text{ sec}$$

$$= 1600 \times 10^{-9} \text{ sec} = \boxed{1600 \text{ ns}}$$

CPU Clocking

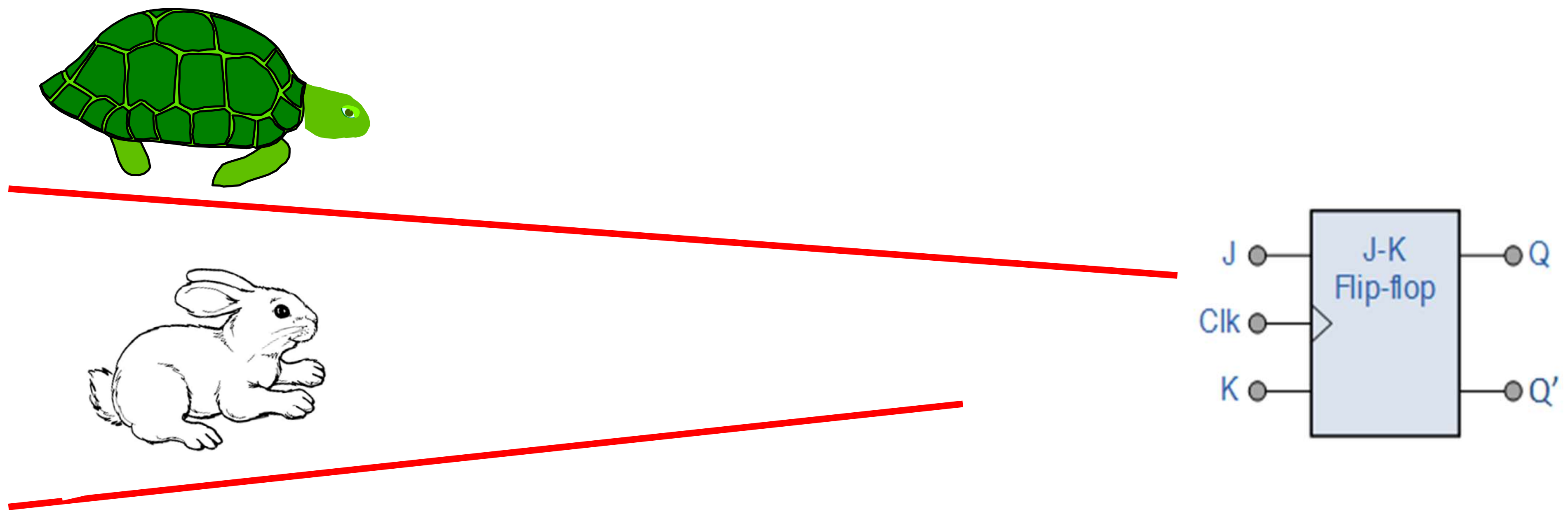
- CPUs are driven by constant-rate system clocks:
  - 100 MHz clock frequency (*f*) means the system clock ticks 100 million times every second:



One Clock Cycle Time ("Tick"), i.e., CCT
= 1/100,000,000 *sec*
=1/100 *microsec* = 10 *nanosec*

$\boxed{CCT = 1/f}$

Clock Timing

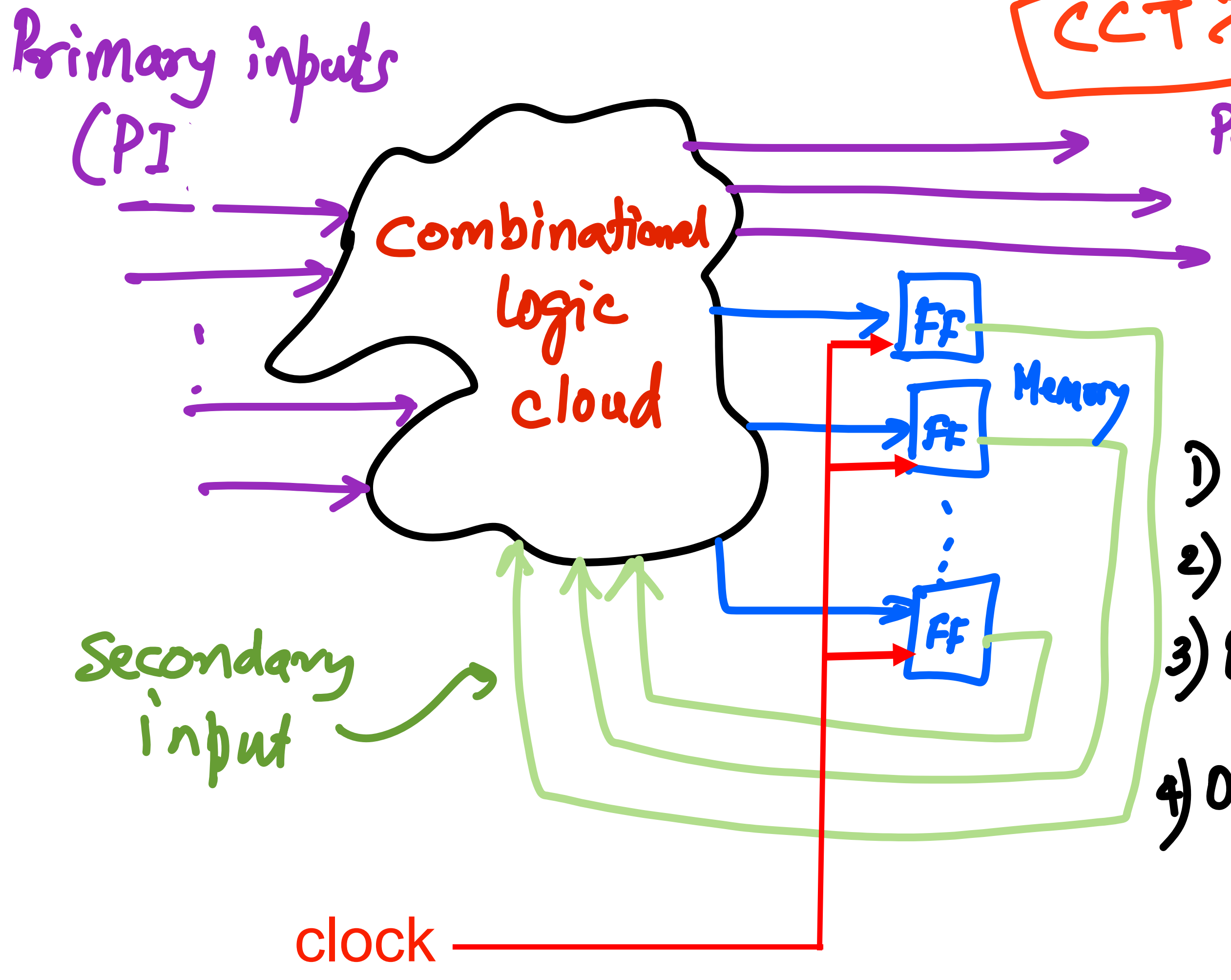


Clock period should be large enough to accommodate delays along critical paths in the circuit (longest ones); but not too large – system slows down unnecessarily

Critical delay in logic circuit

$\Longrightarrow$ CCT

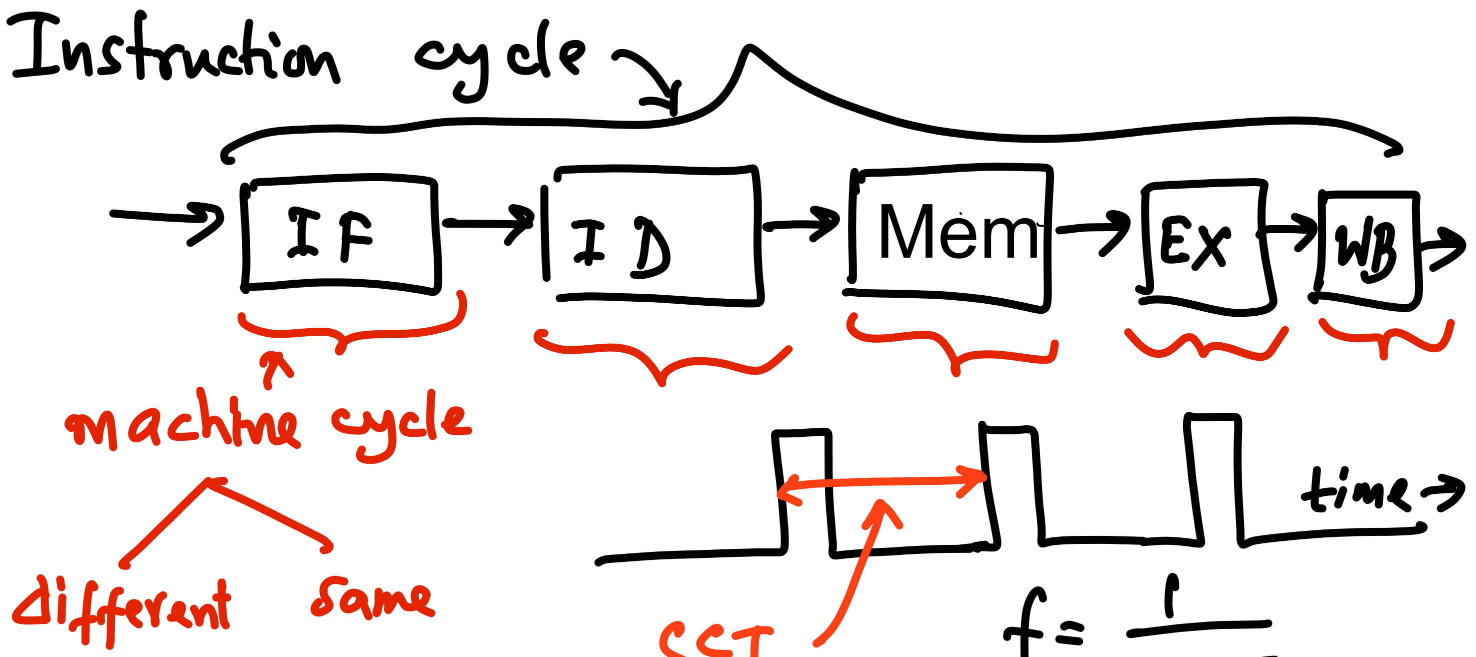
$\boxed{CCT \geq \text{max. path delay}}$

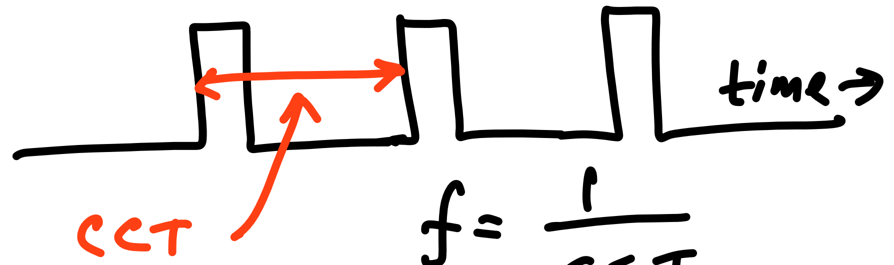


Paths

1) PI → PO
2) PI → Input(Mem)
3) Out(Mem) → Input(Mem)
4) Out(Mem) → PO

CPU_time = IC * CPI * CCT





$f = \frac{1}{CCT}$

CCT ≥ critical delay, i.e., the delay along the longest sensitizable path.

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$
$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\begin{aligned}\text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10\text{s} \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…by this much

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

$$\text{CPU\_time} = IC * CPI * CCT$$

$$\text{Performance} \propto \frac{1}{\text{CPU\_time}}$$

Reducing IC

- good ISA
- using optimizing compiler
- choice of algorithm
- by writing efficient High-level code

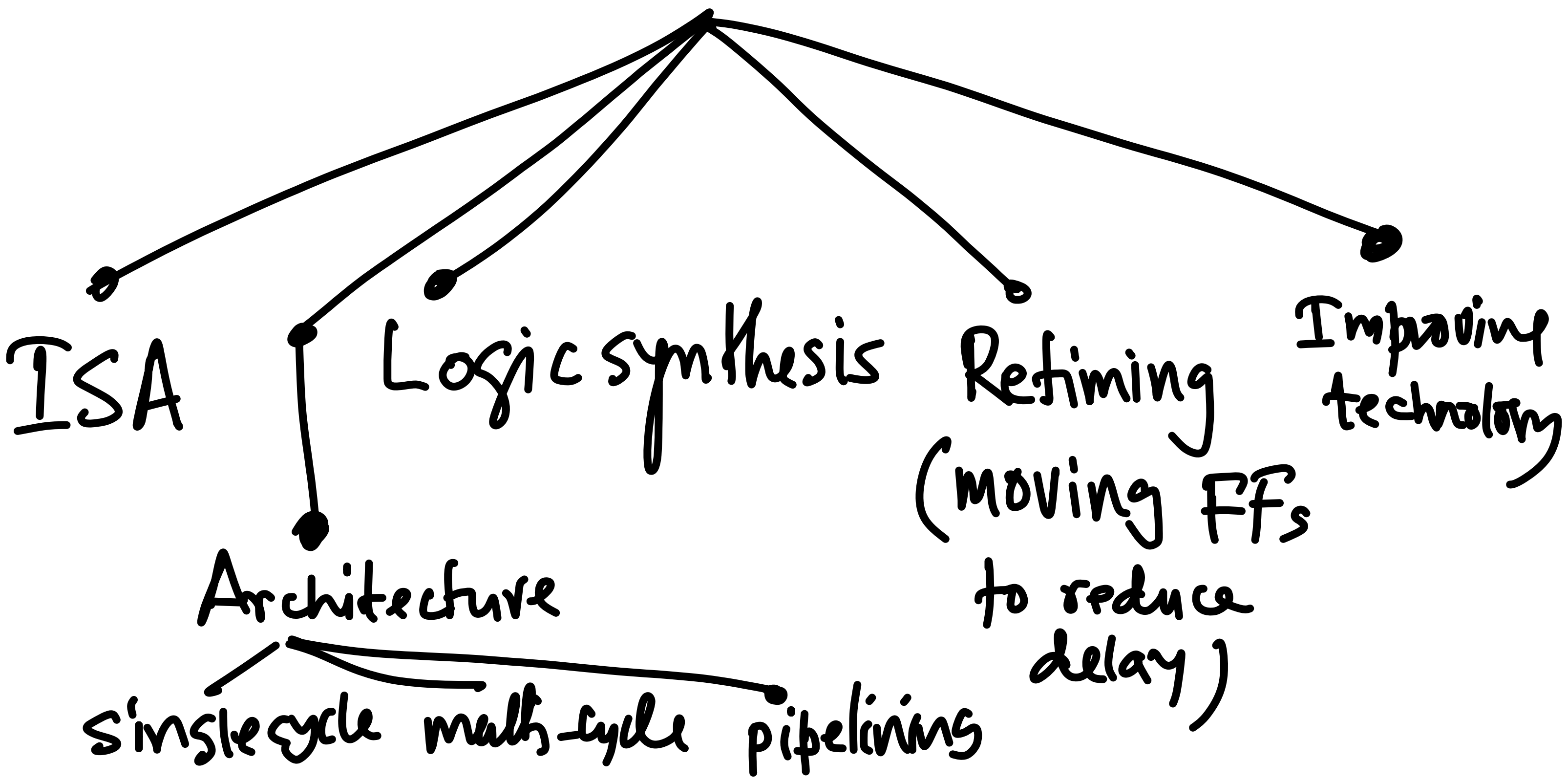
$\text{CPU\_time} = IC * CPI * CCT$

Reducing CPI

- good ISA
- pipelining
- Memory hierarchy
- Improving technology

CPU_time = IC * CPI * CCT

Reducing CCT



Performance Summary

The BIG Picture: CPU Performance Equation

$$\textbf{CPU-time} = \textbf{IC} \times \textbf{CPI} \times \textbf{CCT}$$

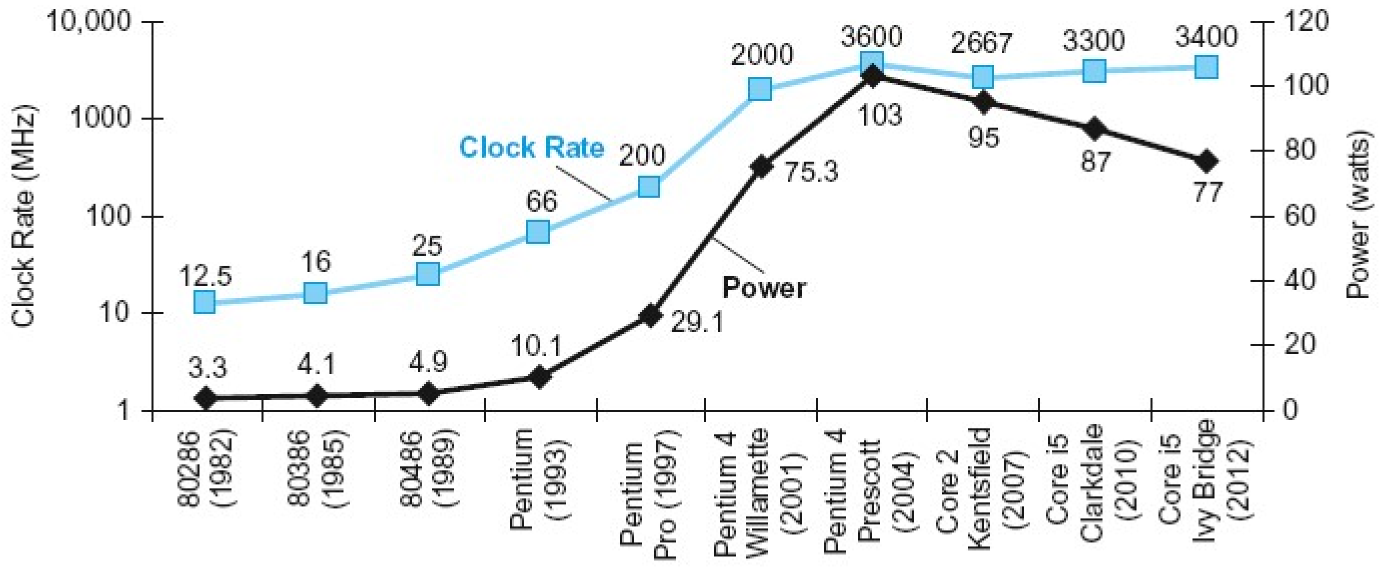
- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, CCT
  - CPI is also affected by memory hierarchy, pipelining; CCT is affected by logic design, technology

MIPS as Performance Measure

MIPS = Millions of Instructions per Second

$$= \frac{\text{Instruction Count (IC) of a program P}}{\text{Execution time of P in seconds} \times 10^6}$$

Power Trends



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30

5V → 1V

×1000

Reducing Power

- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}{}^2 \times F_{old}} = 0.85^4 = 0.52$$

- The power wall
  - We can't reduce voltage further
  - We can't remove more heat
- How else can we improve performance?

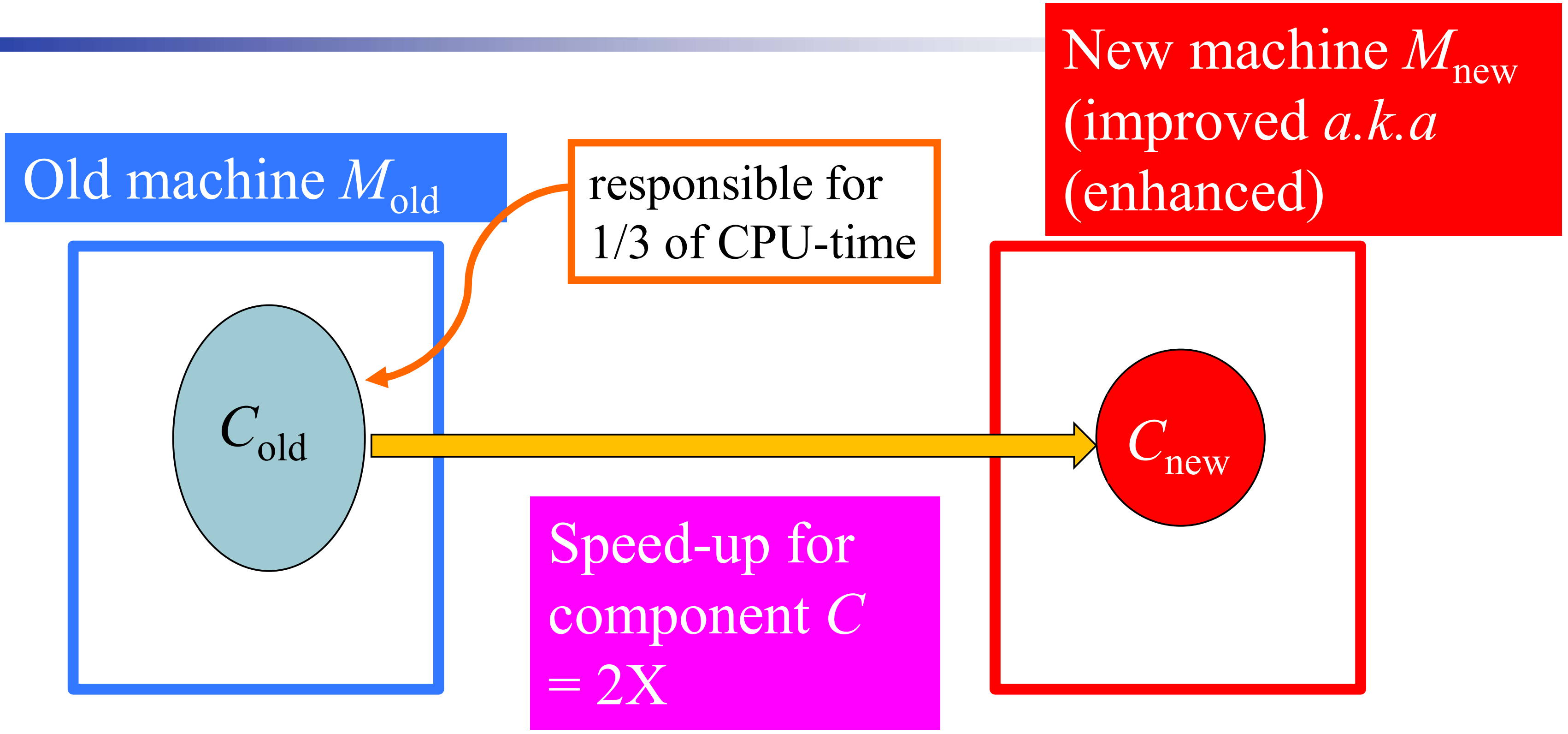
How to improve the performance of a machine by “Enhancement” of one or more components?

→ Amdahl’s Law

How to improve the performance of a computing system by adding more processors (cores)?

→ Gustavson-Barsis Law

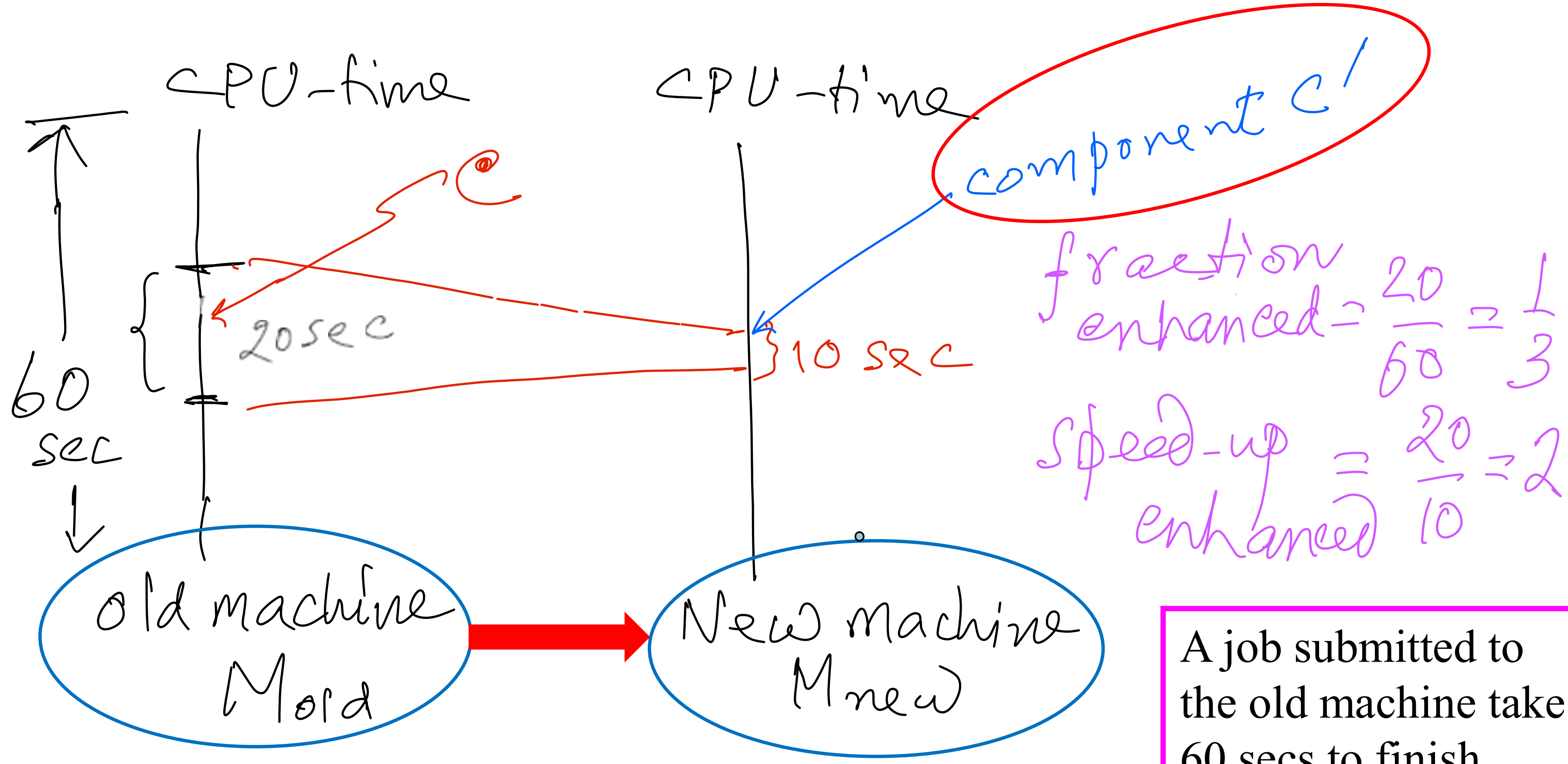
Example: Improving performance in steps


New machine M_{new} (improved *a.k.a* (enhanced)
Old machine M_{old}
responsible for 1/3 of CPU-time
C_{old}
C_{new}
Speed-up for component C = 2X

Question: What is the overall speed-up of M_{new} *w.r.t.* M_{old}?

Gradual enhancement of resources for speed-up

Let C be a component (e.g., adder) of an old machine, which is improved to C’ in a new machine



$\text{fraction enhanced} = \frac{20}{60} = \frac{1}{3}$

$\text{speed-up enhanced} = \frac{20}{10} = 2$

A job submitted to the old machine takes 60 secs to finish

Gradual enhancement of resources for speed-up

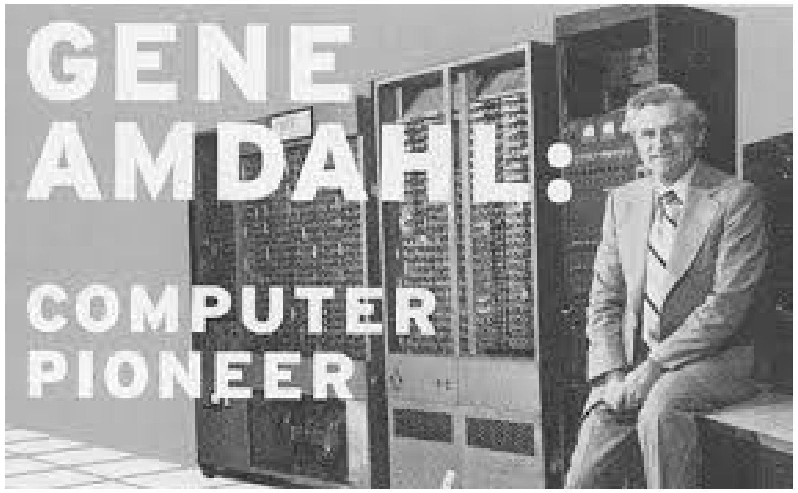
Question: what is the overall speed-up?

$\text{CPU-time-old} = 60 \text{ sec}$

$\text{CPU-time-new} = (60-20) + \frac{20}{10} = 50 \text{ sec}$

$\text{Overall speed-up} = \frac{60}{50} = 1.2$

The general result concerning the overall speed-up, when a component of the old machine is enhanced is captured in "Amdahl's Law"

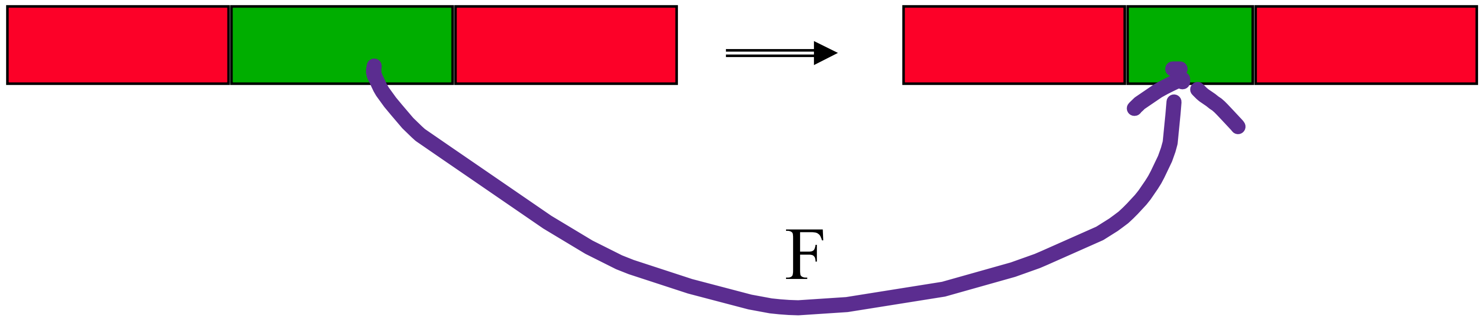


Gene Amdahl
(1922-2015)

Amdahl's Law

Speed-up due to enhancement E:

```
                    CPU-time w/o E        Performance w/E
Speed-up(E)  =  --------------    =    --------------------
                    CPU-time w/E          Performance w/o E
```



Suppose that enhancement E accelerates a fraction F of the task by a factor S, and the remainder of the task is unaffected

Amdahl's Law

$$\text{CPU-time}_{\text{new}} = \text{CPU-time}_{\text{old}} \times \left[(1 - \text{Frac}_{\text{enhanced}}) + \frac{\text{Frac}_{\text{enhanced}}}{\text{Speed-up}_{\text{enhanced}}} \right]$$

$$\text{Speed-up}_{\text{overall}} = \frac{\text{CPU-time}_{\text{old}}}{\text{CPU-time}_{\text{new}}} = \frac{1}{(1 - \text{Frac}_{\text{enhanced}}) + \frac{\text{Frac}_{\text{enhanced}}}{\text{Speed-up}_{\text{enhanced}}}}$$

- Law of diminishing return: unaffected fraction will determine the limiting case!
- Improvement of larger fraction will yield higher overall speed-up → Make the common case faster

Example: Amdahl’s Law

- Floating-point (FP) instructions are improved to run 2X in a new machine
- 10% of actual instructions are FP

Example: Amdahl’s Law

- Floating-point (FP) instructions are improved to run 2X in a new machine
- 10% of actual instructions are FP

$$\text{CPU-time}_{new} = \text{CPU-time}_{old} \times (0.9 + 0.1/2) = 0.95 \times \text{CPU-time}_{old}$$

$$\text{Speed-up}_{overall} = \frac{1}{0.95} = 1.053$$

- For FP Speed-up 100X, $\text{Speed-up}_{overall} = 1.109$

Pitfall: Amdahl's Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

  - Can't be done!

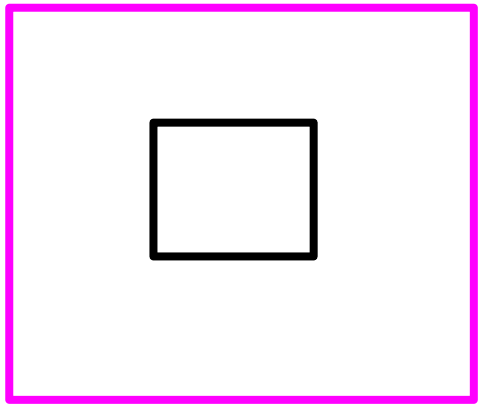
- Corollary: Make the common case fast

Multiprocessors

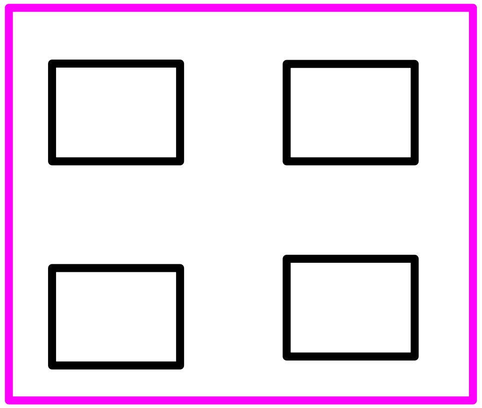
- Multicore microprocessors
  - More than one processor per chip
  - Clock frequency limited
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

Parallelism: Adding multiple processors

Pessimistic view



Single processor



Multi-processor

1. X:= A + B
2. Y:= X * C
3. D:= B + C

Statement 1 & 2 cannot be executed in parallel (serial);
1 and 3 can be executed in parallel;

What kind of *speed-up* is expected?

s: time needed by a single processor on serial parts of *P*;
p: time needed by a single processor on the parts of *P* that can be parallelized; $s + p = 1$
By **Amdahl's Law**, the speed-up in the multi-core processor
$= 1/(s + p/N) \Longrightarrow$ If $s = 10\%$, the maximum speed-up is 10X

Parallelism: Adding multiple processors
Amdahl's Law *versus* Gustavson-Barsis Law

By *Amdahl's Law*, the speed-up in the multi-core processor = $1/(s + p/N)$, i.e., it is limited by s, increasing N does not help.

In Amdahl's Law, the *size of the job* is assumed to be *constant*; however, multi-cores can solve a large job in the same time. This leads to Gustavson-Barsis Law:

Let s and p represent serial and parallel time on N-core system; The single core processor would take $s + (p \times N)$ *time* to perform the same job P.

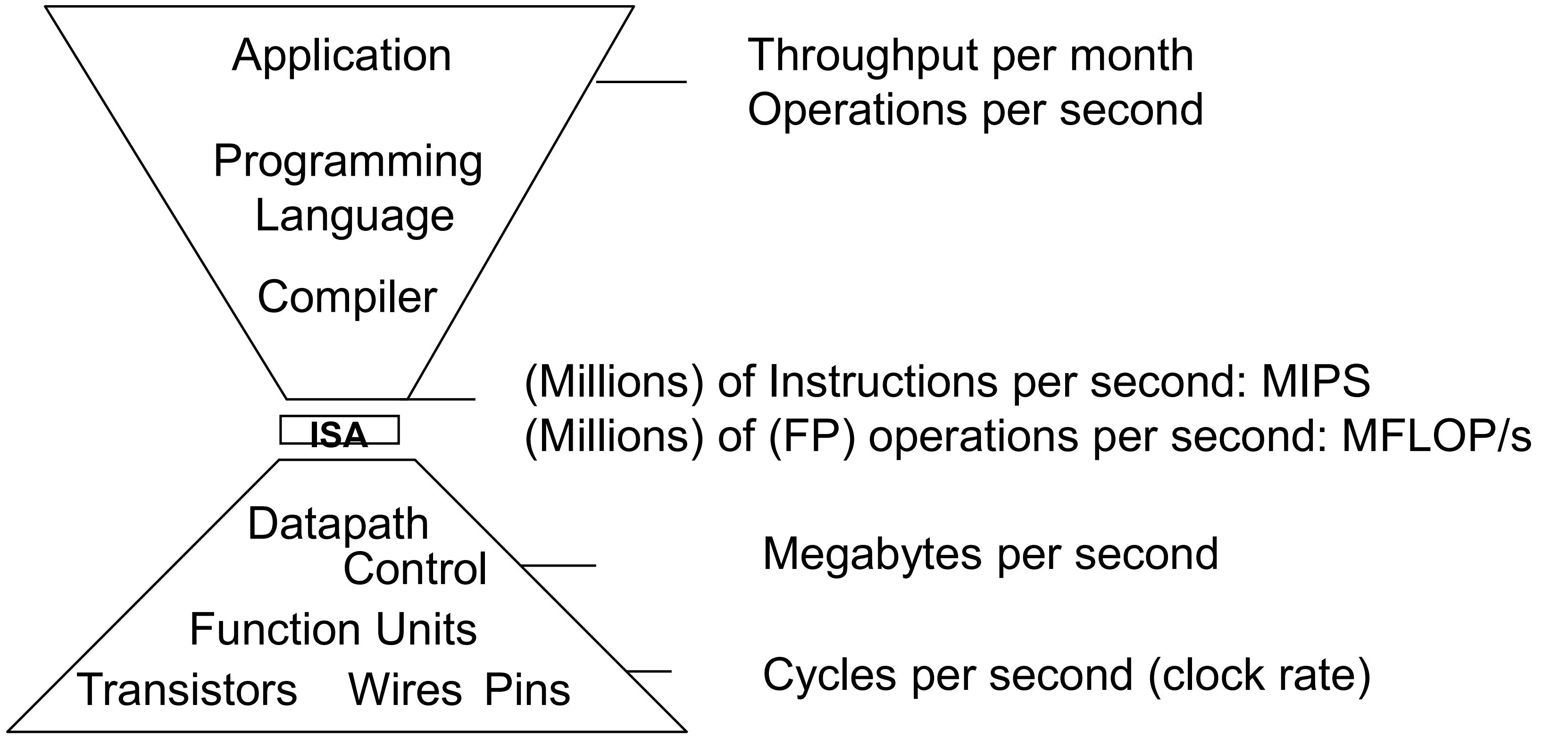
Hence, Speed-up = $(s + (p \times N))/(s + p) = (s + (p \times N))$

Thus, by *Gustavson-Barsis Law*, speed-up **grows** with N

Optimistic view

J.L. Gustavson, Reevaluating Amdahl's Law, *CACM*, May 1988

Metrics of Performance



ISA: RISC *versus* CISC

- RISC (Reduced Instruction Set Computing)
  - Keep the instruction set small and simple
  - Fixed instruction lengths, easy encoding
  - Load-store instruction sets
  - Limited addressing modes
  - Limited operations
  - IC high, CPI low

Advantage: makes the hardware simple and fast; decoding simple; pipelining easy, die-area small

Performance is optimized focused on software

RISC Example: MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Alpha, RISC-V, ARM

- **CISC - Complex Instruction Set Computer**
  - complex instructions, encoding non-uniform
  - different lengths
  - can handle multiple operands
  - complex functionalities
  - IC low, CPI high

pipelining becomes harder; hardware cost increases; compiler design becomes more involved;

- Examples of CISC processors are Intel CPUs, System/360, VAX, AMD

Eight Great Ideas

Amdahl's law

- Design for ***Moore's Law***
- Use ***abstraction*** to simplify design
- Make the ***common case fast***
- Performance *via* ***parallelism***
- Performance *via* ***pipelining***
- Performance *via* ***branch prediction***
- ***Hierarchy*** of memories
- ***Dependability*** *via* redundancy

Gustavson-Barsis law

Concluding Remarks

- Cost/performance is improving
  - Due to underlying technology development
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance

Next Class

- Introducing MIPS Assembly Language

Test 1

- Thursday, 17 September 2020, 8:00 AM – 9:00 AM
- Coverage:
  Chapter 1 $\rightarrow$ material taught in class;
  Pre-requisite topics (Boolean algebra and basic logic design)
- Questions: Multiple-Choice Q/A type
- Credit: 20%