COMPUTER ARCHITECTURE (CSL7070)

Palash Das, CSE, IIT Jodhpur

SYLLABUS / BOOKS

Contents

CSL7XX0 Multi System Architecture 1-0-0 [1]

Introduction: Defining Computer Architecture, Flynn's Classification of Computers, Metrics for Performance Measurement. (4 lectures)

Memory Hierarchy: Introduction, Advanced Optimizations of Cache Performance, Memory Technology and Optimizations, Virtual Memory and Virtual Machines, The Design of Memory Hierarchies, Introduction to Pin Instrumentation and Cache grind, Case Study: Intel Core i7 (10 lectures)

CSL7XX0 Multicore Processing 1-0-0 [1]

Instruction-Level Parallelism: Instruction-level Parallelism: Concepts and Challenges, Basic Compiler Techniques for Exposing ILP, Reducing Branch Costs with Advanced Branch Prediction, Dynamic Scheduling, Superscalar, Limitations of ILP, Case Study: Dynamic Scheduling in Intel Core i7. (9 lectures)

Multicore Processor: Introduction, CPU Interconnections, Network on Chip (NoC), Routing Protocols, Quality of Service on NoC. (5 lectures)

CSL7XX0 Fundamentals of Parallel Programming 1-0-0 [1]

Data Level Parallelism: Introduction, Vector Architecture, SIMD Instruction Set Extensions for Multimedia, Graphics Processing Units, GPU Memory Hierarchy, Detecting and Enhancing Loop- Level Parallelism, CUDA Programming, Case Study: Nvidia Maxwell. (14 lectures)

Textbook

1. Hennessy,J.L. and Patterson,D.A., (2012), *Computer Architecture: A Quantitative Approach*, 5th Edition, Morgan Kaufmann Publishers

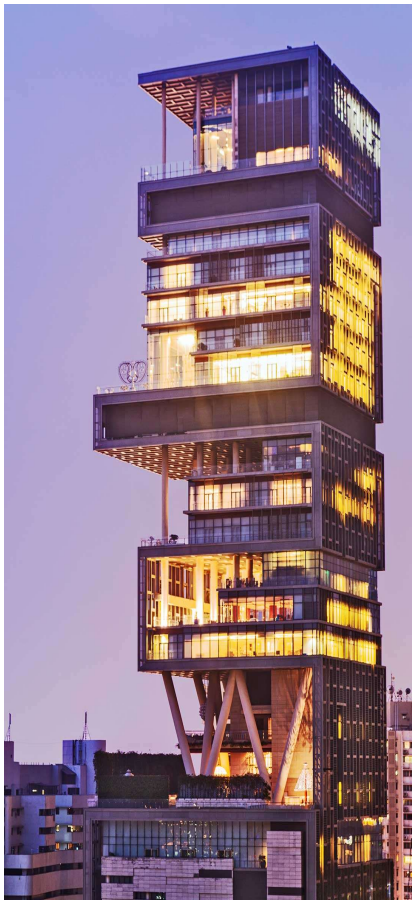
2. Shen,J.P. and Lipasti,M.H., (2005), *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill Publishers

TENTATIVE EVALUATION POLICY (I MAY CHANGE IN REAL-TIME)

- Continuous Evaluation – 40%
  - Assignments – 20%
  - Projects - 20%

- Primary Evaluation - 60%
  - Minor – 20%
  - Major – 40%

THIS OR THAT?



- Architecture is all about trade-offs
  - Strengths, weaknesses
- You need to think the most innovative design within the budget.
- Be creative and visionary for sustainability
- Understanding of the past design and build on it
- Good judgement and intuition.

HIGH-LEVEL GOALS

Understand the principles
Understand the precedents

Based on such understanding:

- Enable you to evaluate tradeoffs of different designs and ideas
- Enable you to develop principled designs
- Enable you to develop novel, out-of-the-box designs

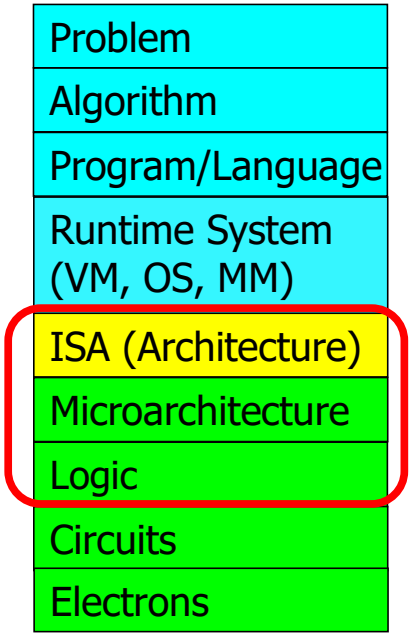
The focus is on:

- Principles, precedents, and how to use them for new designs

ROLE OF THE (COMPUTER) ARCHITECT

- Look backward (to the past)
  - Understand tradeoffs and designs, upsides/downsides, past workloads. Analyze and evaluate the past.
- Look forward (to the future) **Predict future requirements**
  - Be the dreamer and create new designs. Listen to dreamers.
  - Push the state of the art. Evaluate new design choices.
- Look up (towards problems in the computing stack)
  - Understand important problems and their nature.
  - Develop architectures and ideas to solve important problems.
- Look down (towards device/circuit technology)
  - Understand the capabilities of the underlying technology.
  - Predict and adapt to the future of technology (you are designing for N years ahead). Enable the future technology.

HOW DO WE ENSURE PROBLEMS ARE SOLVED BY ELECTRONS?



Levels of Transformation

- Being an architect is not easy
- You need to consider **many** things in designing a new system + have good intuition/insight into ideas/tradeoffs

- But, it is fun and can be very technically rewarding
- And, enables a great future
  - E.g., many scientific and everyday-life innovations would not have been possible without architectural innovation that enabled very high performing systems
    - E.g., your mobile phones

WHY DO WE NEED LEVELS OF TRANSFORMATION

Levels of transformation create abstractions

Abstraction: A higher level only needs to know about the interface to the lower level, not how the lower level is implemented

E.g., high-level language programmer does not really need to know what the ISA is and how a computer executes instructions

Abstraction improves productivity

No need to worry about decisions made in underlying levels

E.g., programming in Java vs. C vs. assembly vs. binary vs. by specifying control signals of each transistor every cycle

SHOULD WE THINK BEYOND THIS ABSTRACTION LAYERS??

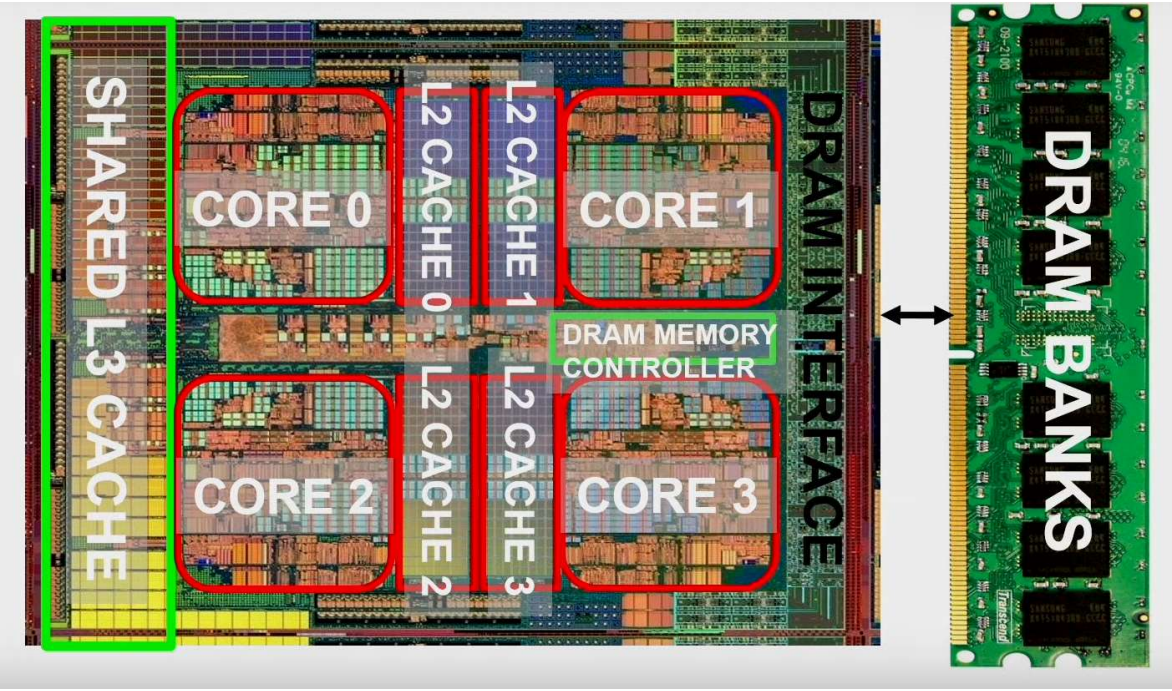
- As long as everything goes well, not knowing what happens in the underlying level (or above) is not a problem.

- What if (You are a software person)
  - The program you wrote is running slow?
  - The program you wrote does not run correctly?
  - The program you wrote consumes too much energy?

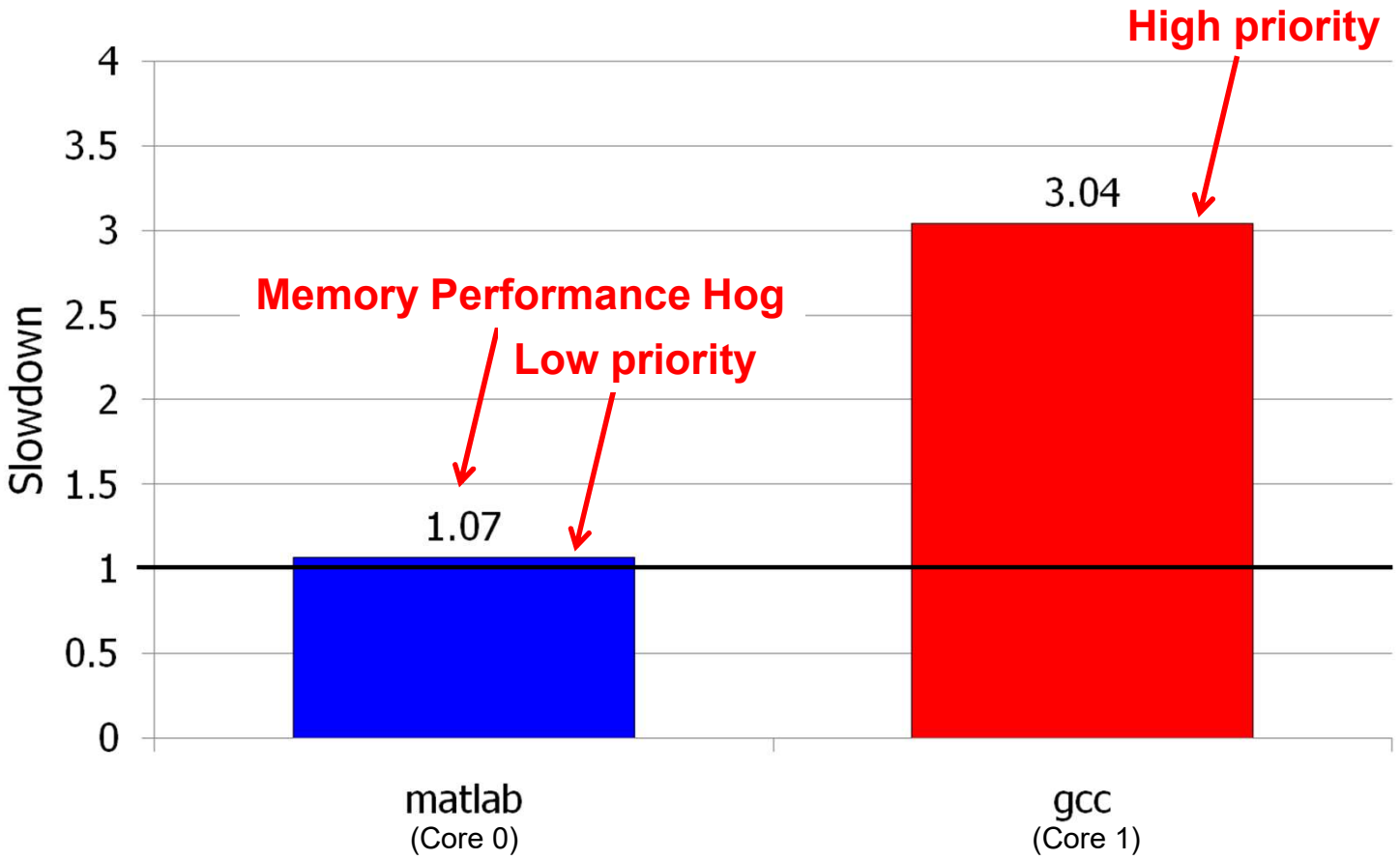
- What if (You are a hardware person)
  - The hardware you designed is too hard to program?
  - The hardware you designed is too slow because it does not provide the right primitives to the software?

- What if
  - You want to design a much more efficient and higher performance system?

WHAT ALL PARTS WE ARE GOING TO COVER?

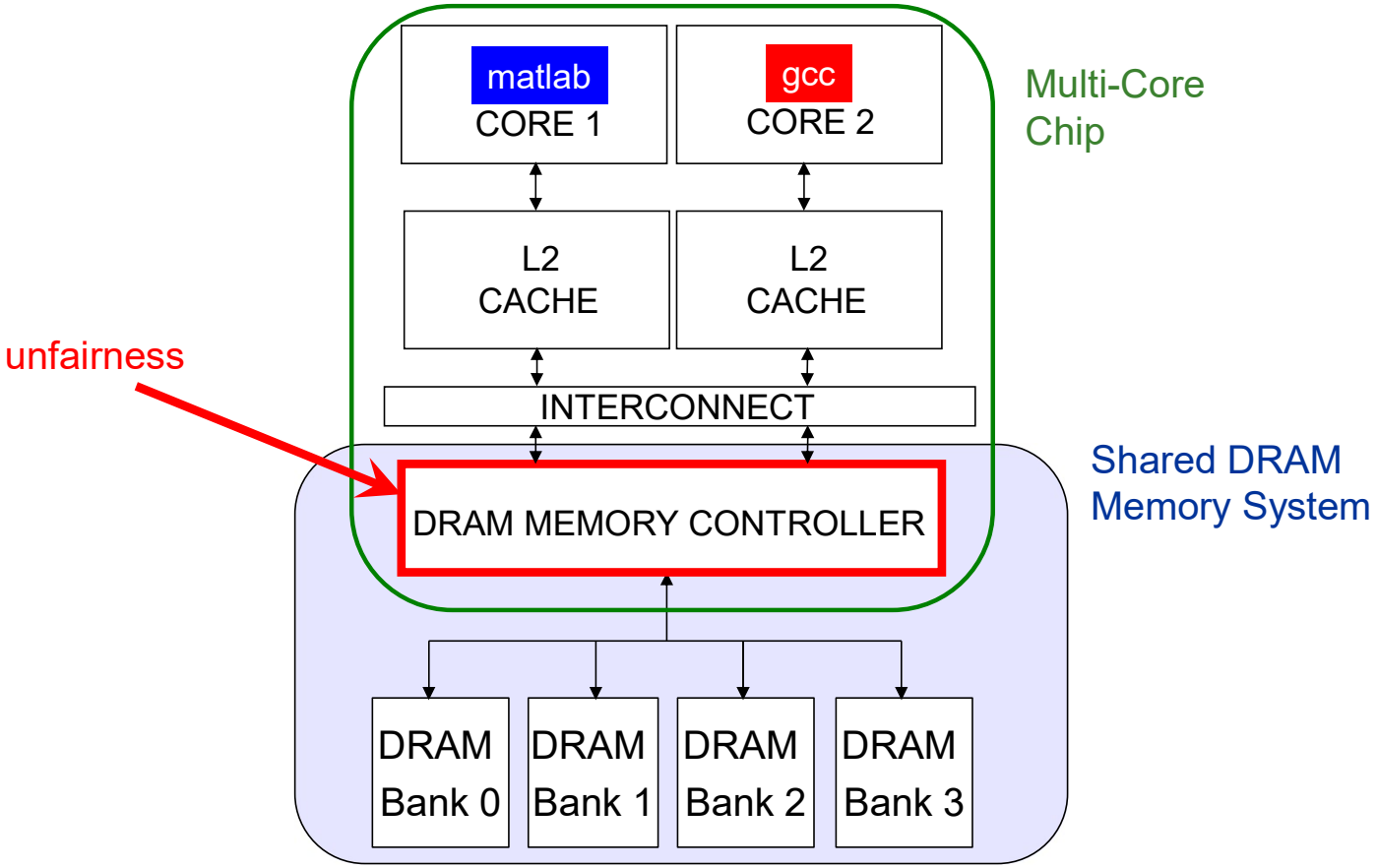


UNEXPECTED SLOWDOWNS IN MULTI-CORE



Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

TO ANSWER THE QUESTIONS...



DRAM BANK OPERATION

A few important timing Parameters:

t_{RAS}: row access strobe

t_{CAS}: column access strobe

t_{RP}: Precharge time

Access Address:

(Row 0, Column 0)

(Row 0, Column 1)

(Row 0, Column 85)

(Row 1, Column 0)

Row address 0

Row address 1

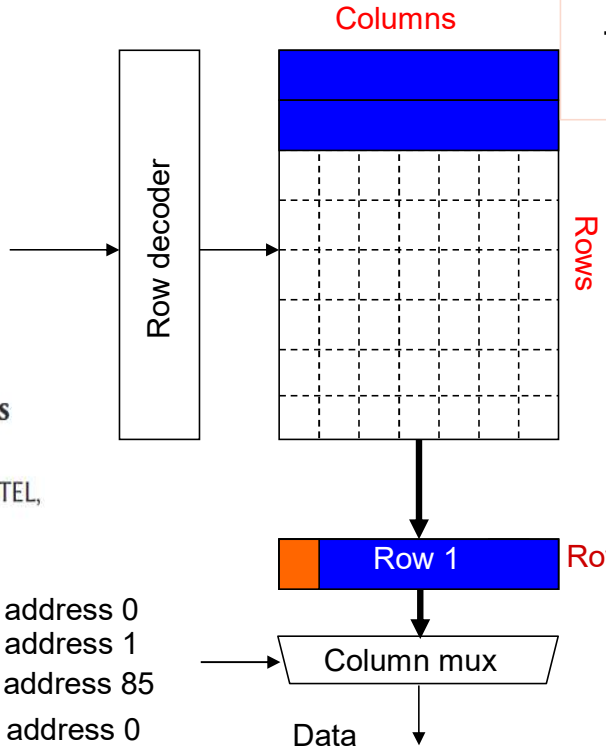


Table 1. JEDEC Timing Constraints [12]

JEDEC Specifications (cycles)			
Parameters	Description	DDR3-1600H	DDR4-1600K
t_{RCD}	ACT to RD/WR delay	9	11
t_{RL}	RD to Data Start	9	11
t_{RP}	PRE to ACT Delay	9	11
t_{WL}	WR to Data Start	8	9
t_{RTW}	RD to WR Delay	7	8
t_{WTR}	WR to RD Delay	6	6
t_{RTP}	Read to PRE Delay	6	6
t_{WR}	Data End of WR to PRE	12	12
t_{RAS}	ACT to PRE Delay	28	28
t_{RC}	ACT-ACT (same bank)	37	39
t_{RRD}	ACT-ACT (diff bank)	5	4
t_{FAW}	Four ACT Window	24	20
t_{BUS}	Data bus transfer	4	4
t_{RTR}	Rank to Rank Switch	2	2

A Comparative Study of Predictable DRAM Controllers

DANLU GUO, MOHAMED HASSAN, RODOLFO PELLIZZONI, and HIREN PATEL,
University of Waterloo

Row-hit First---> Biased Mem. CTRL.

A MEMORY PERFORMANCE HOG

```
// initialize large arrays A, B

for (j=1; j<=N; j++) {
    index = j;                    streaming
    A[index] = B[index];
    …
}
```

```
// initialize large arrays A, B

for (j=1; j<=N; j++) {
    index = rand();               random
    A[index] = B[index];
    …
}
```

STREAM

- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

RANDOM

- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.

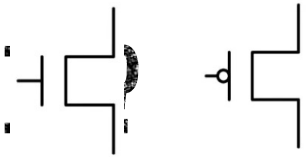
COMPUTER ARCHITECTURE

- **Computer Architecture:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.

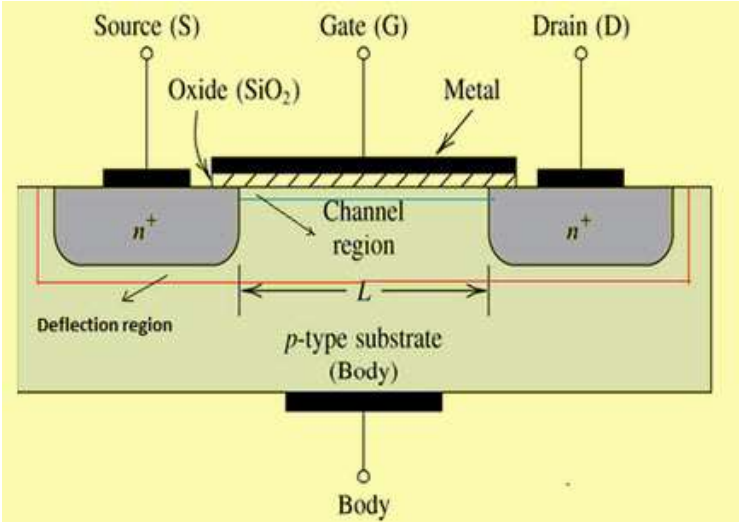
- **Traditional definition:** "The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior as distinct from the organization of the dataflow and controls, the logic design, and the physical implementation." *Gene Amdahl*, IBM Journal of R&D, April 1964

Dr. Amdahl holding a 100gate LSI air-cooled chip. On his desk is a circuit board with the chips on it. This circuit board was for an Amdahl 470 V/6 (photograph dated March 1973).

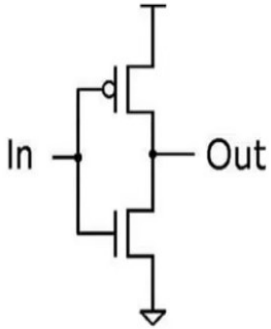
WHY DO WE CARE FOR TRANSISTOR?

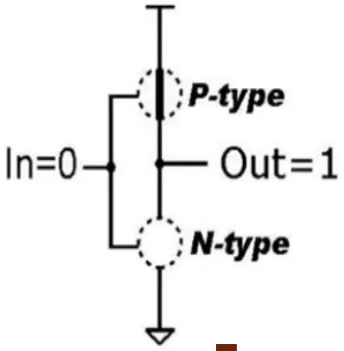


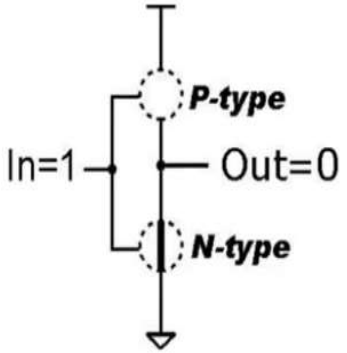
NMOS transistor PMOS transistor

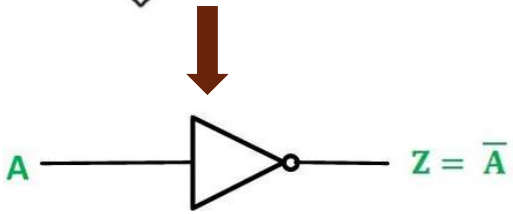


MOSFET: metal–oxide–semiconductor field-effect transistor

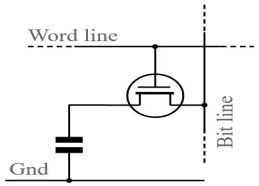




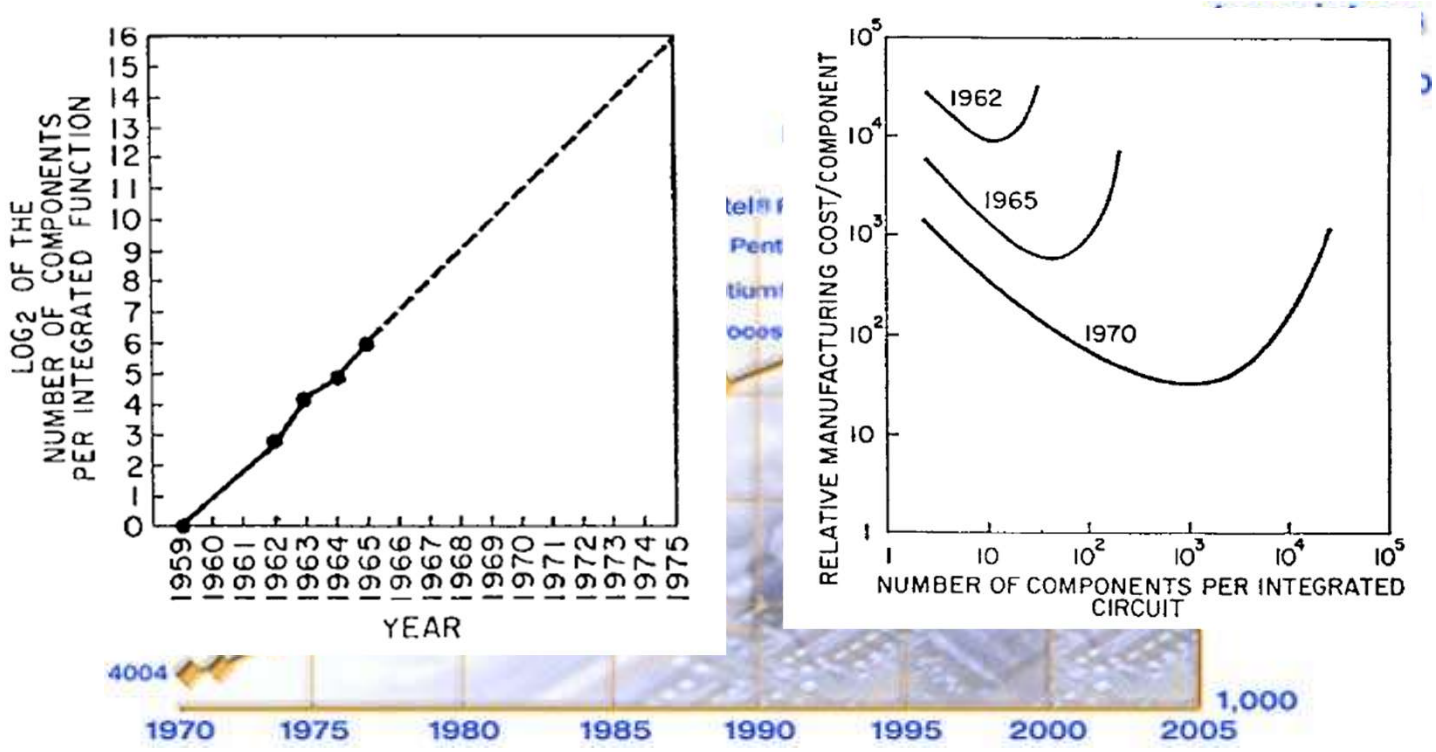




*** Even in DRAMs, we have transistors (1T-1C cell) → stores a bit



MOORE'S LAW



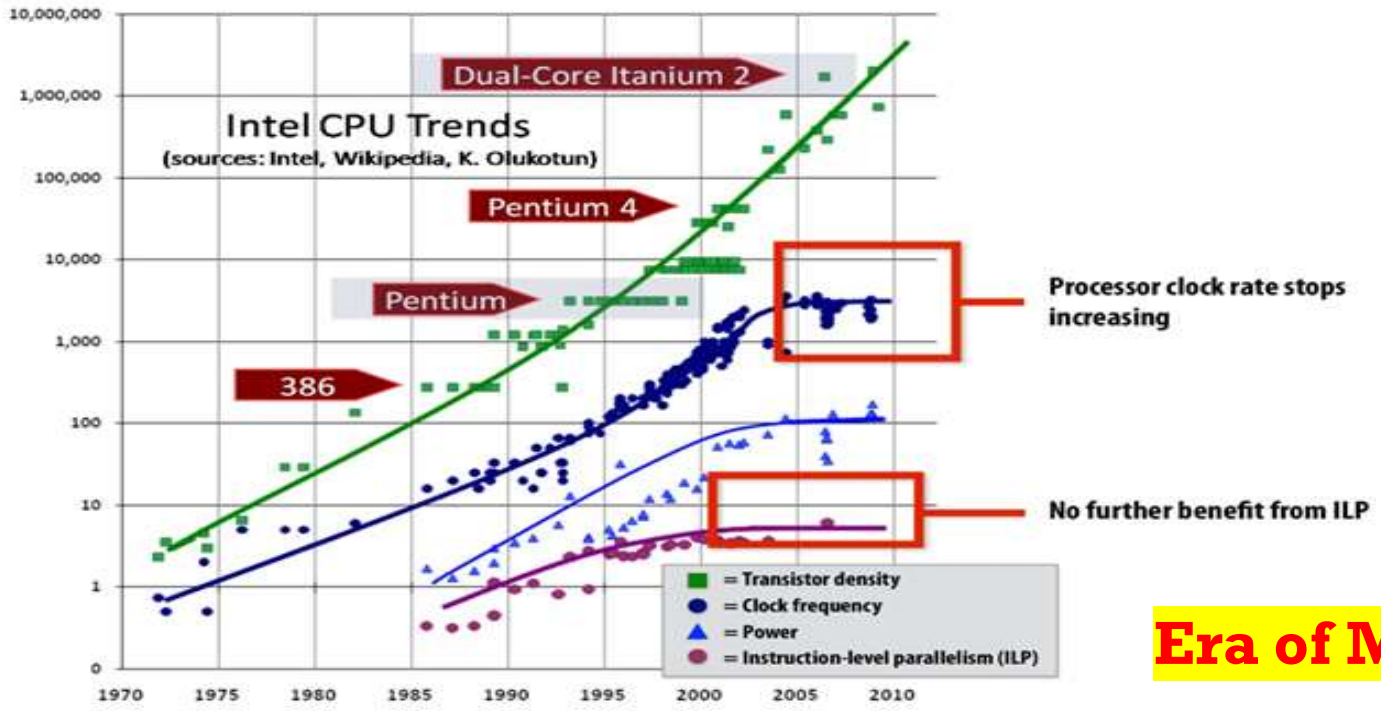
Moore, "Cramming more components onto integrated circuits," Electronics Magazine, 1965.

Component counts double every other year

Image source: Intel

END OF DENNARD SCALING

ILP tapped out + end of frequency scaling



- Transistor scaled by 30%
- Delay reduced by 30%
- Frequency increased by 40%
- Reduced power consumption by 50%

Single-Core System

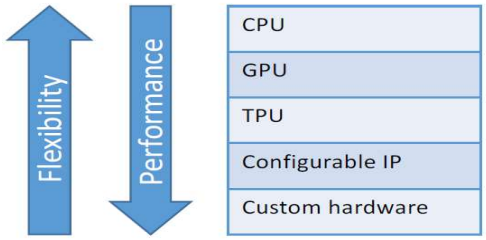
↓

Multi-Core System

Era of Multicore Started

PROCESSING OPTIONS

NMP Options	Area efficiency	Power Efficiency	Flexibility
Programmable Core	✗	✗	✓
GPUs	✗	✗	✓
FPGA	✗	✓	✓
ASICs	✓	✓	✗

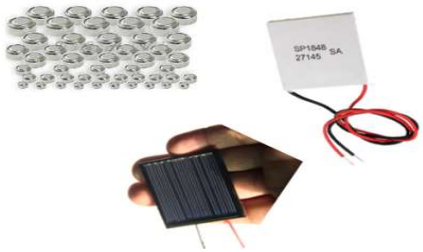


If your power cable looks like one of these:

Use CPUs and GPUs

(a)

If your power supply looks like these:



Need to look for efficient inferencing alternatives

(b)

COMPUTER ARCHITECTURE TODAY (I)

- Today is a very exciting time to study computer architecture

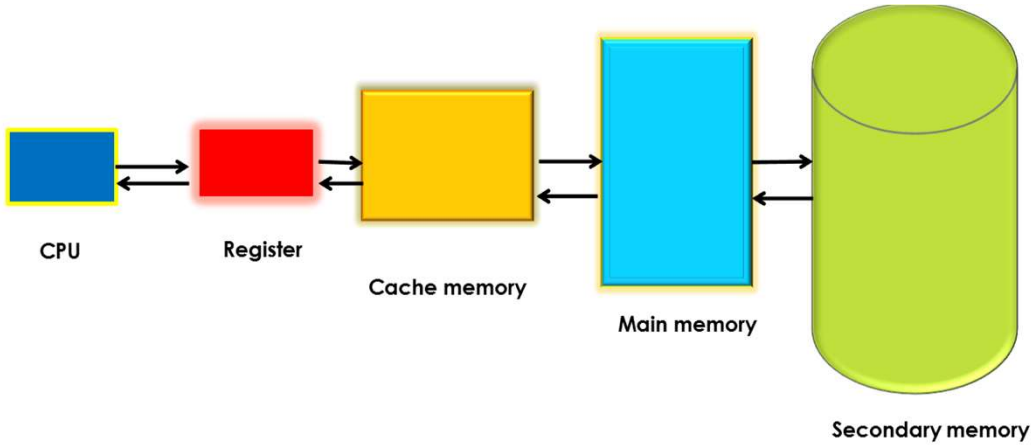
- Industry is in a large paradigm shift (to multi-core and beyond) – many different potential system designs possible, problems increased for programmer with multicore

- Many difficult problems *motivating* and *caused by* the shift
  - Power/energy constraints → multi-core?
  - Complexity of design → multi-core?
  - Difficulties in technology scaling → new technologies?
  - Memory wall/gap
  - Reliability wall/issues
  - Huge hunger for data and new data-intensive applications

Complexity and power management from single to multi core have become different

- No clear, definitive answers to these problems

MEMORY HIERARCHY (CONT....)



Level	Typical Access Time	Typical Capacity	Other Features
Register	300-500 ps	500-1000 B	On-chip
Level-1 cache	1-2 ns	16-64 KB	On-chip
Level-2 cache	5-20 ns	256 KB – 2 MB	On-chip
Level-3 cache	20-50 ns	1-32 MB	On or off chip
Main memory	50-100 ns	1-16 GB	
Magnetic disk	5-50 ms	100 GB – 16 TB	

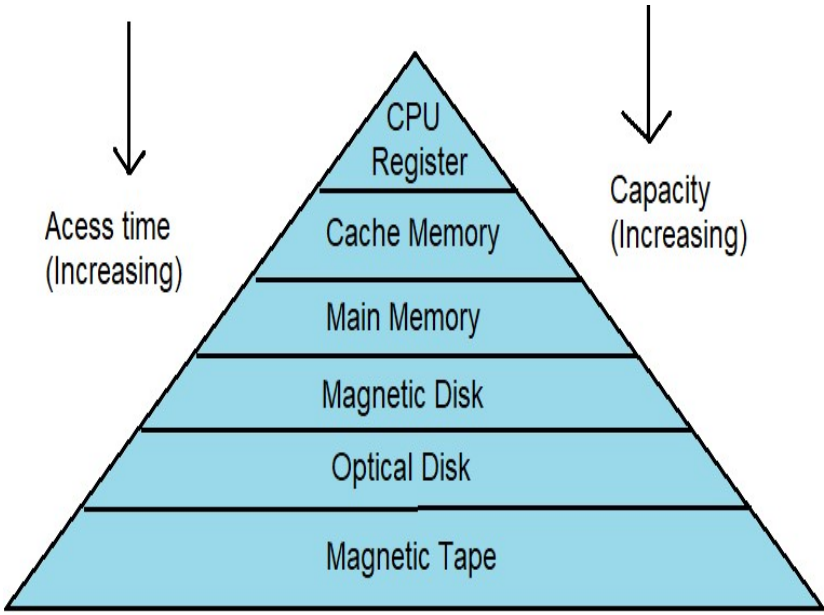
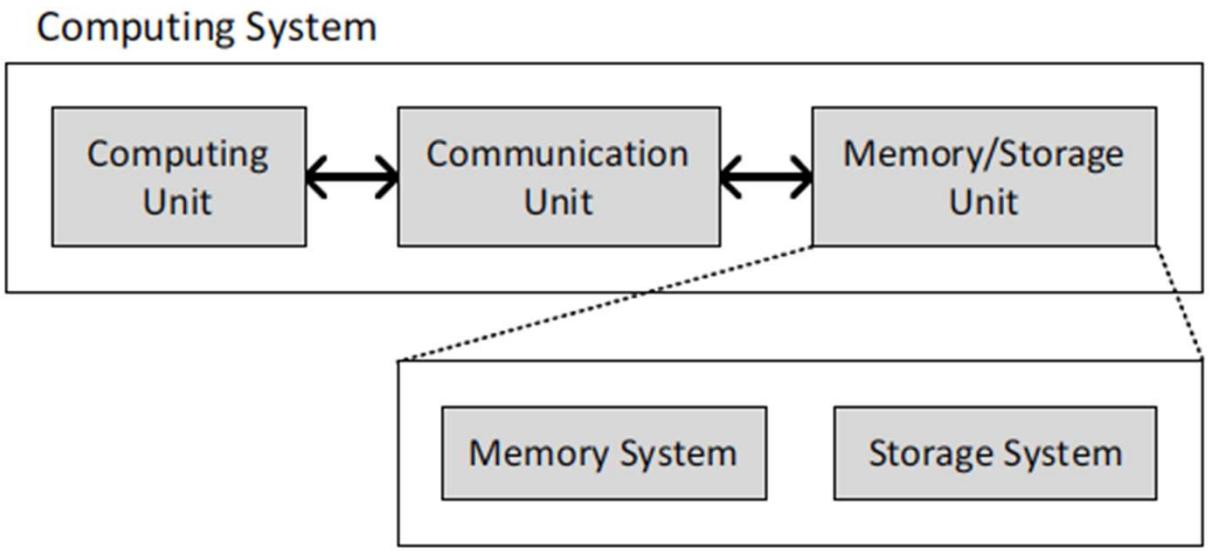


Fig:- Memory Hierarchy

WHAT IS A COMPUTER?

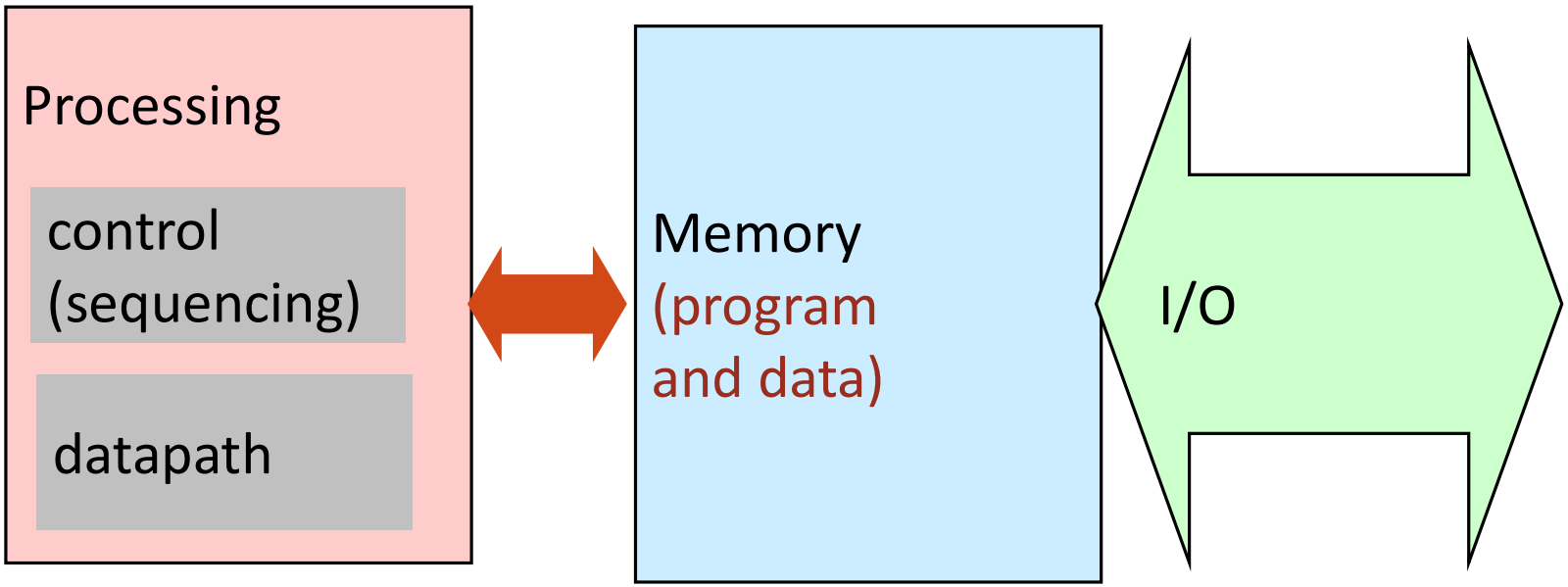
- Three key components

- Computation
- Communication
- Storage (memory)



WHAT IS A COMPUTER?

- We will cover all three components



THE VON NEUMANN MODEL/ARCHITECTURE

- Also called *stored program computer* (instructions in memory). Two key properties:

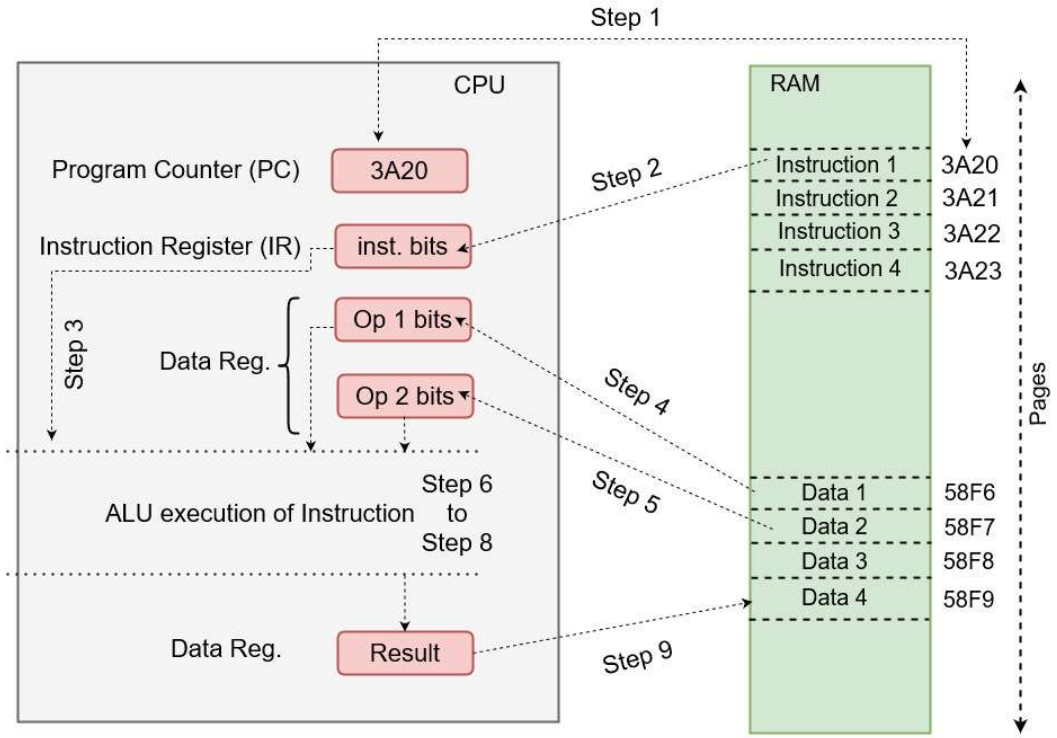
- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals

When is a value interpreted as an instruction?

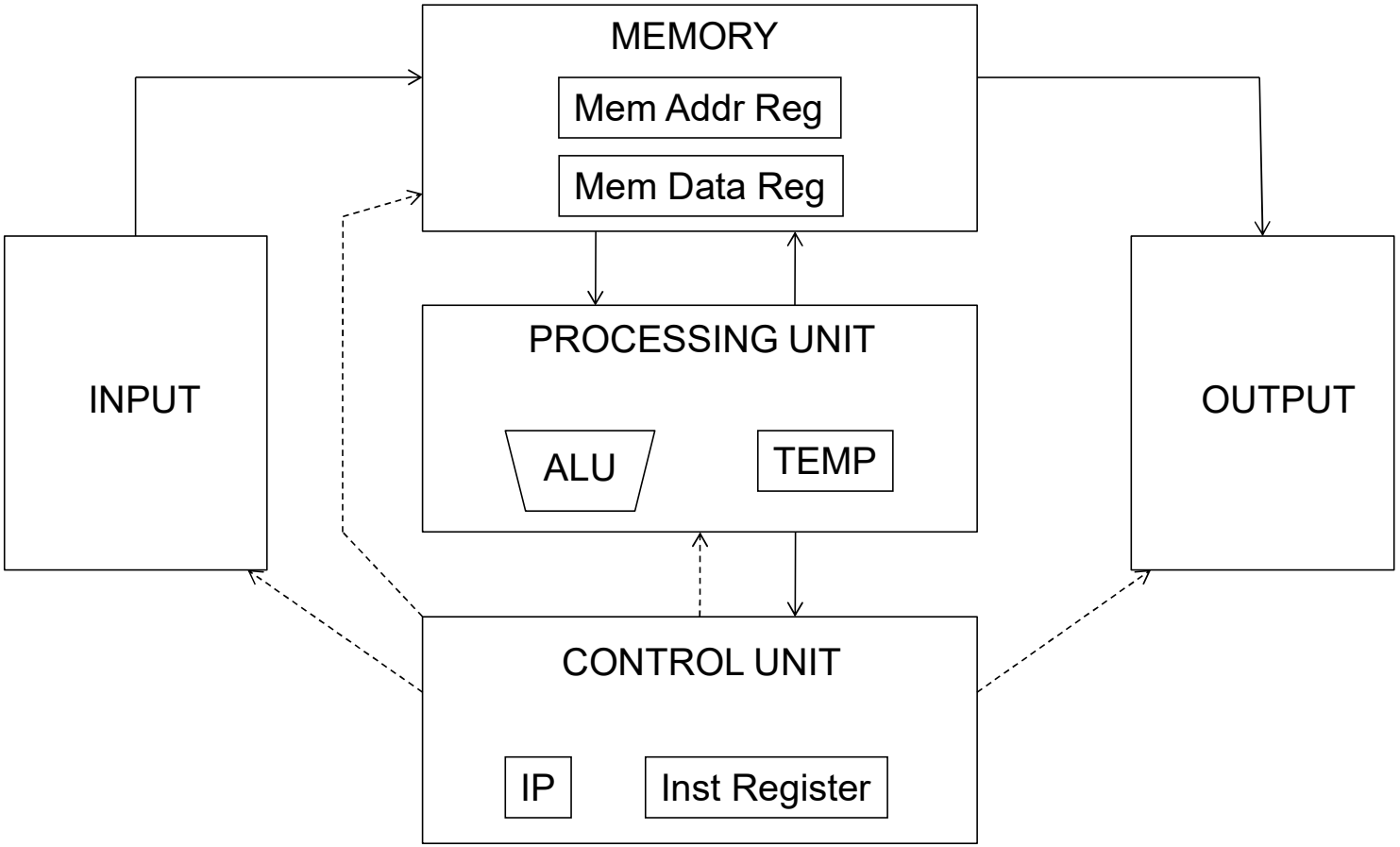
- Sequential instruction processing
  - One instruction processed (fetched, executed, and completed) at a time
  - Program counter (instruction pointer) identifies the current instr.
  - Program counter is advanced sequentially except for control transfer instructions

*** *Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.*

A HIGH-LEVEL VIEW OF PROGRAM EXECUTION

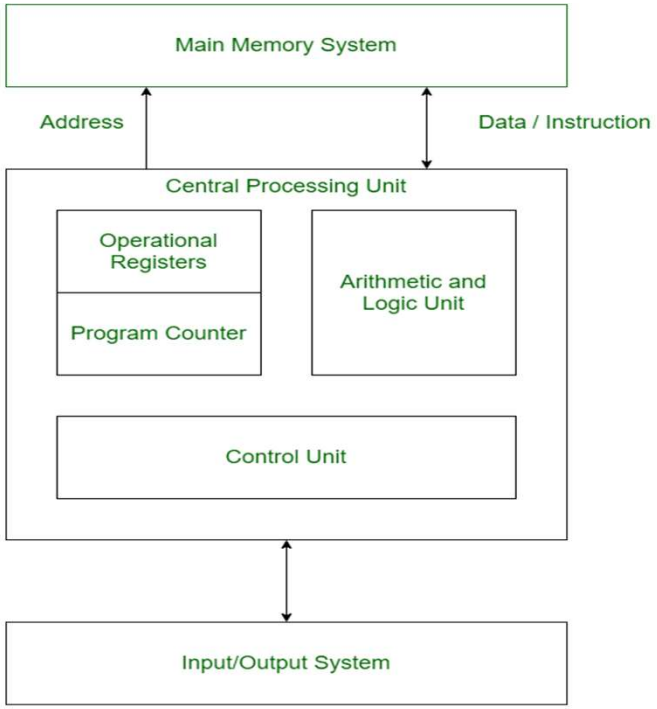


THE VON NEUMANN MODEL (OF A COMPUTER)

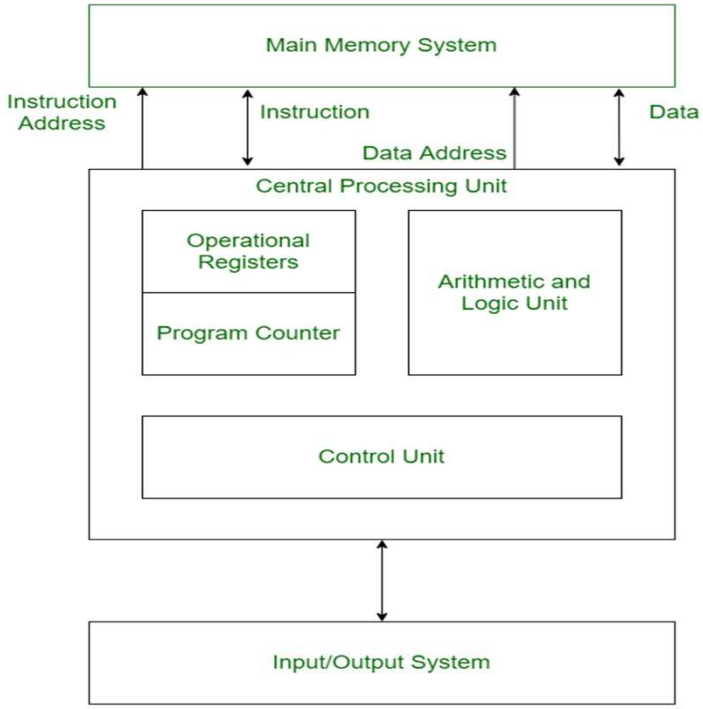


If we have muti-core, is it von neumann?

HIGH-LEVEL VIEW OF A SYSTEM (ARCHITECTURE)



Von Neumann Architecture



Harvard Architecture

Feature	Von Neumann Architecture	Harvard Architecture
Memory	Shared memory for instructions and data	Separate memory for instructions and data
Buses	Single bus for both instructions and data	Separate buses for instructions and data
Speed	Slower due to sequential memory access	Faster due to parallel access
Design Complexity	Simpler and cheaper to implement	More complex and expensive
Flexibility	High (self-modifying programs possible)	Limited

EXAMPLES

- Classic Von Neumann Processors
  - Intel 8086
  - Intel 8088
  - Motorola 6800
- Neumann (Inspired)
  - Intel Pentium Series
  - AMD Athlon Series
  - Intel Core Series (i3, i5, i7, i9)
  - AMD Ryzen Processors

- Harvard architecture
  - GPU Cores
    - Graphics Processing Units (e.g., NVIDIA, AMD) often use Harvard-inspired designs with separate instruction and data memory.
  - DSP Cores in SoCs (System on Chips)
    - Many modern SoCs (like Qualcomm Snapdragon and Apple A-series chips) have Harvard-based DSP cores for multimedia tasks.

Traditional Approach

DATAFLOW approach

Modern Data Flow-Inspired Processors: Tensor Processing Units (TPUs), Intel Nervana NNPs (Neural Network Processors), Neuromorphic Processors (Intel Loihi), FPGA-based designs (Xilinx Versal AI Core).

THE VON NEUMANN MODEL (OF A COMPUTER)

- Q: Is this the only way that a computer can operate?

- A: No.
- Qualified Answer: But, it has been the dominant way
  - i.e., the dominant paradigm for computing
  - for N decades

DATA FLOW MODEL

- Data Flow Architecture is a computer architecture model where,

- The flow of data, rather than the sequential execution of instructions, determines the execution of operations.

- Unlike traditional architectures like Von Neumann or Harvard, which rely on a program counter, **data flow architecture executes instructions as soon as the required input data becomes available**.

THE DATAFLOW MODEL (OF A COMPUTER)

- Von Neumann model: An instruction is fetched and executed in control flow order
  - As specified by the instruction pointer or program counter
  - Sequential unless explicit control flow instruction

- Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - As a result, there is no need of instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies "who" should receive the result
    - An instruction can "fire" whenever all operands are received
  - Potentially many instructions can execute at the same time
    - Inherently more parallel

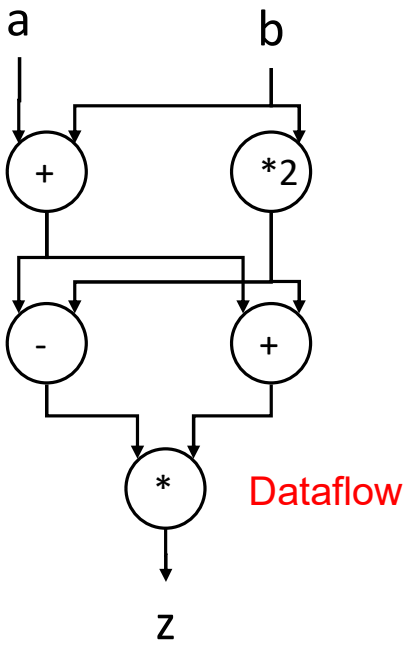
Reading suggestion ☺

***Dennis and Misunas, "A preliminary architecture for a basic data-flow processor," ISCA 1974.

VON NEUMANN (CONTROLFLOW) VS DATAFLOW

```
v = a + b;
w = b * 2;
x = v - w
y = v + w
z = x * y
```

Sequential



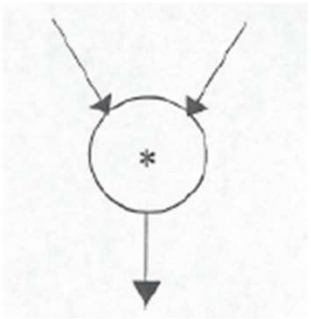
Dataflow

- Which model is more natural to you as a programmer? → Could be problem dependant ?

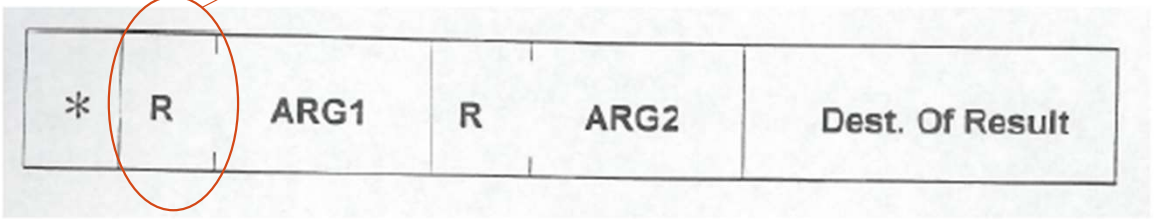
MORE ON DATA FLOW

- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all it inputs are ready
    - i.e. when all inputs have tokens

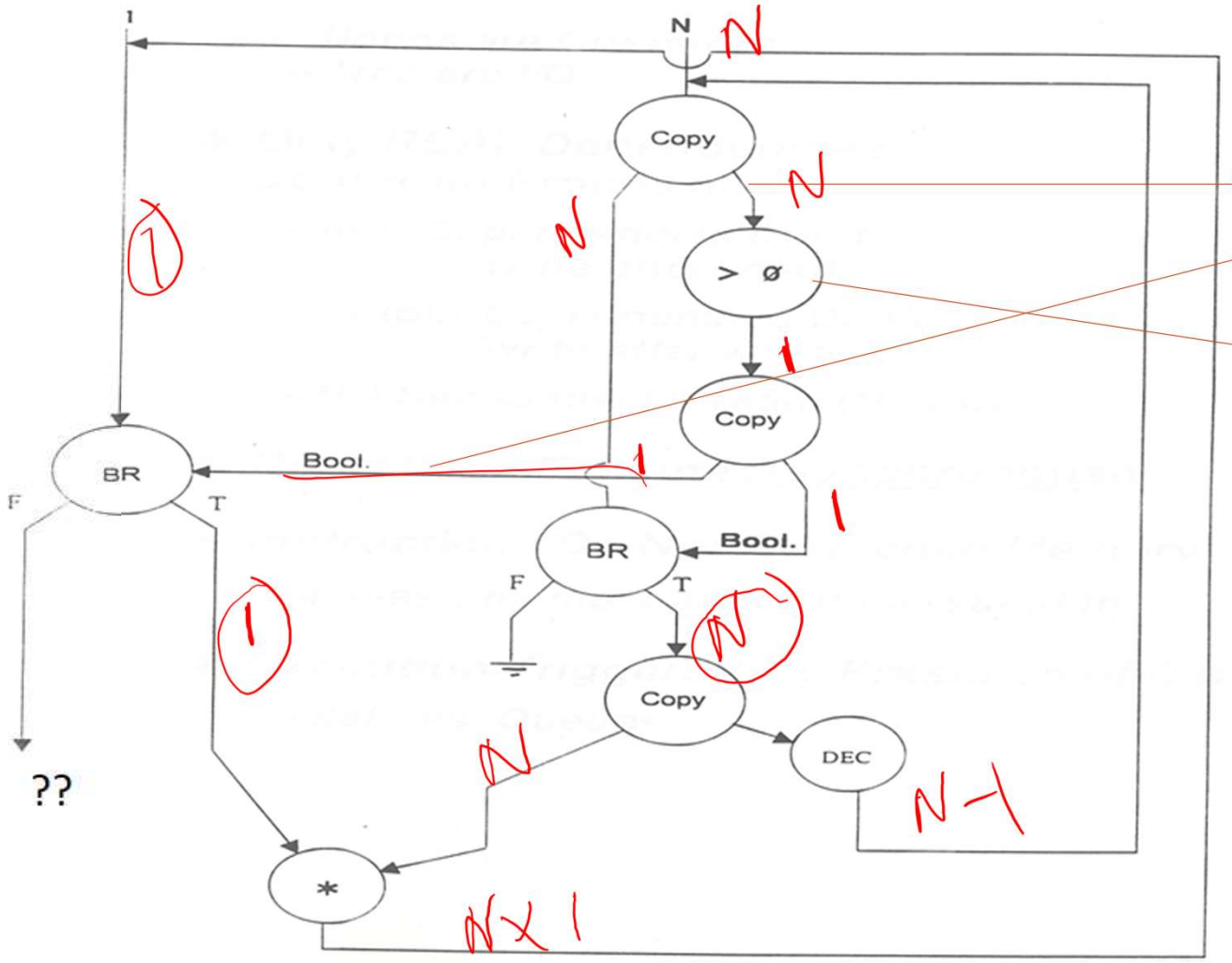
- Data flow node and its ISA representation



Argument is ready to be taken.
Someone may have produced it.



AN EXAMPLE DATA FLOW PROGRAM



Data Input

Control Input

Node represents Operations

VLSI DESIGN PROCESS

- Design Complexity increasing rapidly
  - Increased size and complexity
  - Fabrication technology improving
  - CAD tools are essential
  - Support from AI is coming **(AI for Systems and Systems of AI)**
  - Conflicting requirements like area, speed, and energy consumption
- The present trend
  - Standardize the design flow
  - Emphasis on low-power design, and increased performance

VLSI DESIGN FLOW

- Standardized Design Process----> Starting from the design idea down to actual implementation.
- It has many steps:
  - Specification
  - Synthesis
  - Simulation
  - Layout
  - Testability analysis
  - ……

WE NEED CAD TOOLS

- CAD tool support various Hardware Description Languages (HDL).

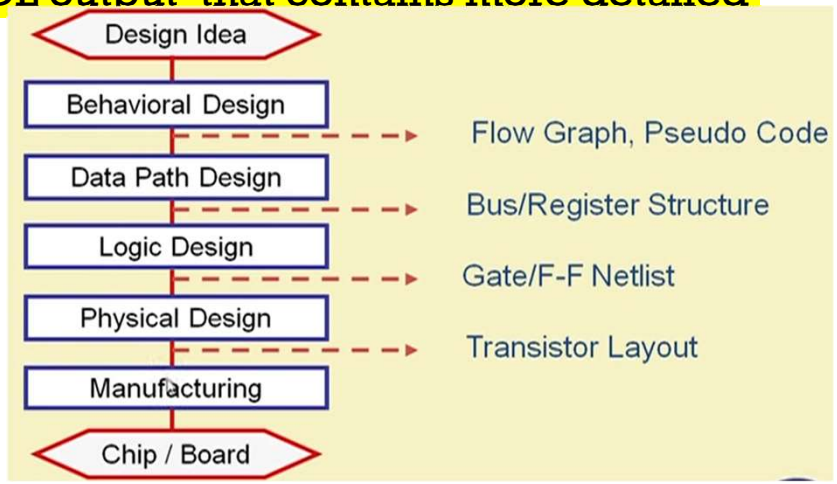
- A CAD tool transforms its HDL input into HDL output that contains more detailed information about the hardware.

✓ Behavioral level to register transfer level

✓ Register transfer level to gate level

✓ Gate level to transistor level

✓ Transistor level to the layout level



Two popular HDLs

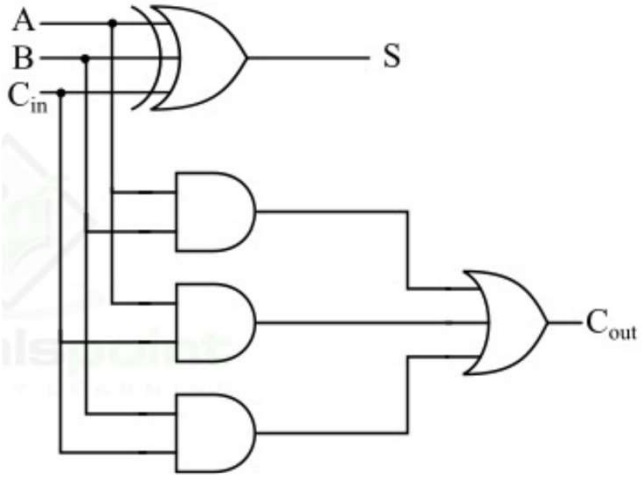
- Verilog
- VHDL

There are other HDLs like SystemC, SystemVerilog,….

BEHAVIORAL VS. STRUCTURAL CODE

Aspect	Behavioral	Structural
Focus	Describes functionality	Describes interconnection of components
Abstraction Level	High	Low
Constructs Used	Procedural statements (`always` , `if`)	Module instantiation, wiring
Ease of Writing	Easier	More complex
Simulation	More abstract, faster	Can reflect actual timing and layout
Synthesis	Can be synthesized (limited to RTL)	Directly synthesizable

FULL ADDER



Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

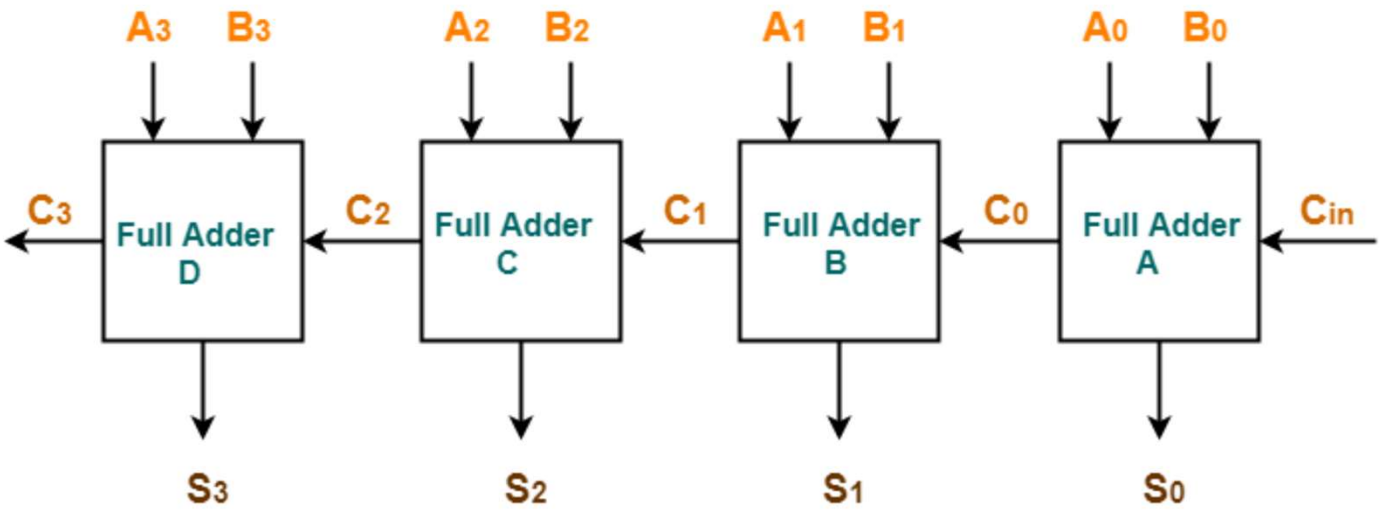
```
module full_adder_behavioral (
    input  A, B, Cin,   // Inputs: two bits and carry-in
    output Sum, Cout    // Outputs: Sum and carry-out
);
    // Behavioral description using assign statements
    assign Sum = A ^ B ^ Cin;           // Sum = XOR of inputs
    assign Cout = (A & B) | (B & Cin) | (A & Cin);  // Carry-out logic
endmodule
```

```
module full_adder_structural (
    input  A, B, Cin,   // Inputs: two bits and carry-in
    output Sum, Cout    // Outputs: Sum and carry-out
);
    // Internal signals
    wire AxorB, AB, BCin, ACin;

    // Gate-level description
    xor (AxorB, A, B);      // XOR gate for A and B
    xor (Sum, AxorB, Cin);  // XOR gate for AxorB and Cin (Sum logic)
    and (AB, A, B);         // AND gate for A and B
    and (BCin, B, Cin);     // AND gate for B and Cin
    and (ACin, A, Cin);     // AND gate for A and Cin
    or (Cout, AB, BCin, ACin);  // OR gate for carry-out
endmodule
```

TESTBENCH

```
// Testbench for full_adder_behavioral
module full_adder_behavioral_tb;

    // Declare testbench variables
    reg A, B, Cin;       // Inputs
    wire Sum, Cout;      // Outputs

    // Instantiate the DUT (Device Under Test)
    full_adder_behavioral uut (
        .A(A),
        .B(B),
        .Cin(Cin),
        .Sum(Sum),
        .Cout(Cout)
    );

    // Testbench procedure
    initial begin
        // Display the header for the results
        $display("Time | A B Cin | Sum Cout");
        $display("-------------------------");

        // Apply specific input combinations
        A = 0; B = 0; Cin = 0; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        A = 0; B = 0; Cin = 1; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        A = 0; B = 1; Cin = 0; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        A = 0; B = 1; Cin = 1; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        A = 1; B = 0; Cin = 0; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        A = 1; B = 0; Cin = 1; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        A = 1; B = 1; Cin = 0; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        A = 1; B = 1; Cin = 1; #10;
        $display("%4d | %b  %b   %b  |  %b   %b", $time, A, B, Cin, Sum, Cout);

        // End simulation
        $stop;
    end

endmodule
```

ASSIGNMENT (HDL FOR THIS?)



4-bit Ripple Carry Adder

ISA-LEVEL TRADEOFF: INSTRUCTION POINTER

- Do we need an instruction pointer or PC in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the IP points to it
    - IP automatically changes sequentially (except for control flow instructions)
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available (data flow)

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity? → Depends

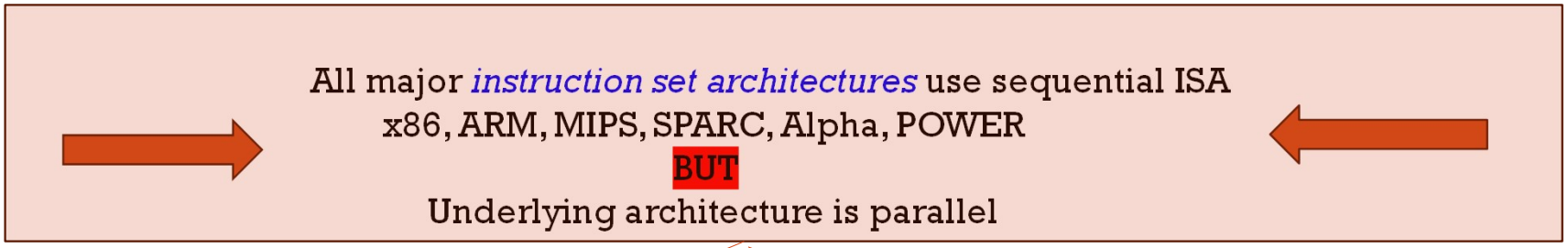
ISA VS. MICROARCHITECTURE LEVEL TRADEOFF

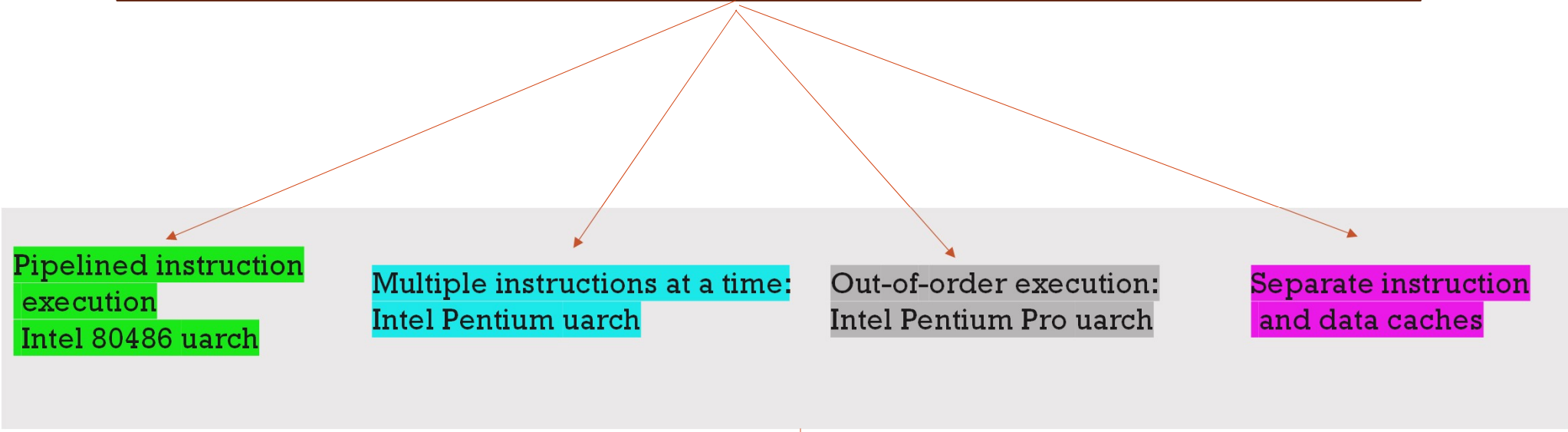
- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level

- ISA: Specifies how the programmer sees instructions to be executed
  - Programmer sees a sequential, control-flow execution order vs.
  - Programmer sees a data-flow execution order

- Microarchitecture: How the underlying implementation actually executes instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
    - Programmer should see the order specified by the ISA

O_O execution

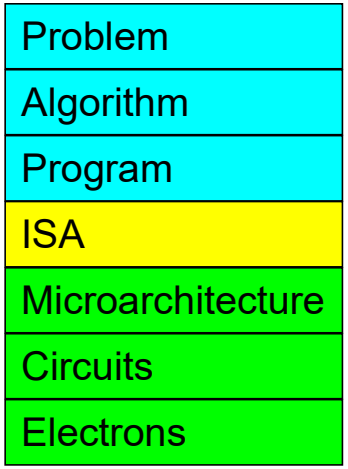




Whatever is underneath is not von-neuman, but it is also not exposed to software.

ISA VS. MICROARCHITECTURE

- ISA
  - Agreed upon interface between software and hardware
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture



*** x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, …
*** Microarchitecture usually changes faster than ISA.
*** Few ISAs (x86, ARM, SPARC, MIPS, Alpha) but many uarchs. Why so?

MICROARCHITECTURE

- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Caching? Levels, size, associativity, replacement policy
  - Prefetching?
  - Voltage and frequency scaling

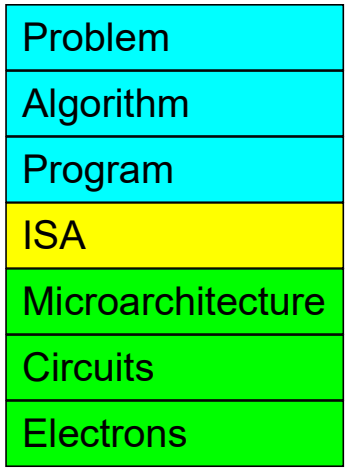
PROPERTY OF ISA VS. UARCH (CAN YOU IDENTIFY?)

- ADD instruction's opcode
- Number of general-purpose registers
- Number of ports to the register file
- Number of cycles to execute the MUL instruction
- Whether or not the machine employs pipelined instruction execution

- Remember
  - Microarchitecture: Implementation of the ISA under specific design constraints and goals

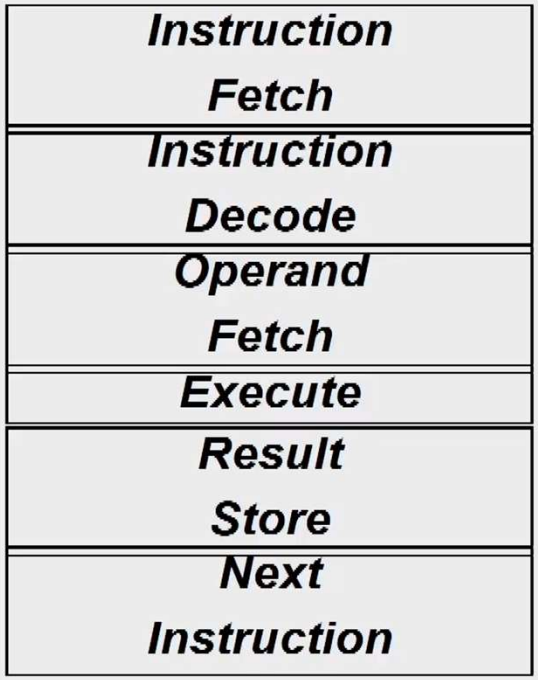
DESIGN POINT OF MICRO ARCHITECTURE

- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Considerations
  - Cost
  - Performance
  - Maximum power consumption
  - Energy consumption (battery life)
  - Reliability and Correctness



- Design point determined by the "Problem" space (application space), the intended users/*market*

INSTRUCTION EXECUTION CYCLE



Obtain instruction from program storage

Determine required actions and instruction size

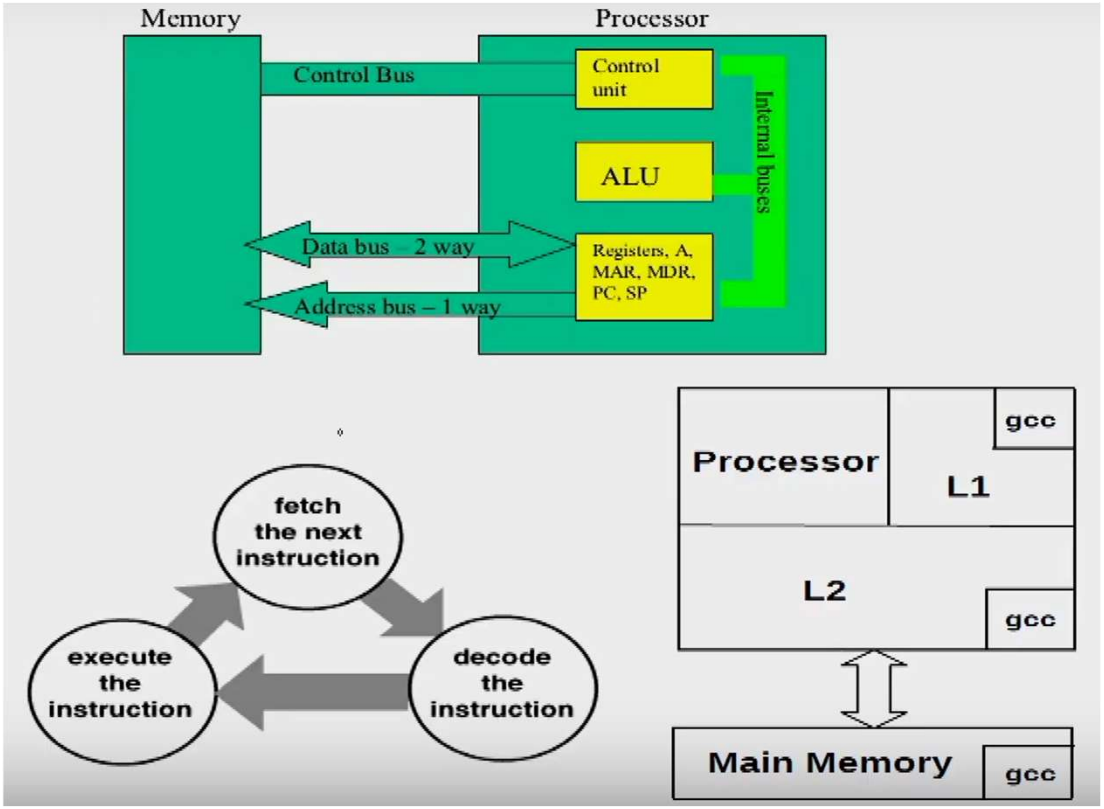
Locate and obtain operand data

Compute result value or status

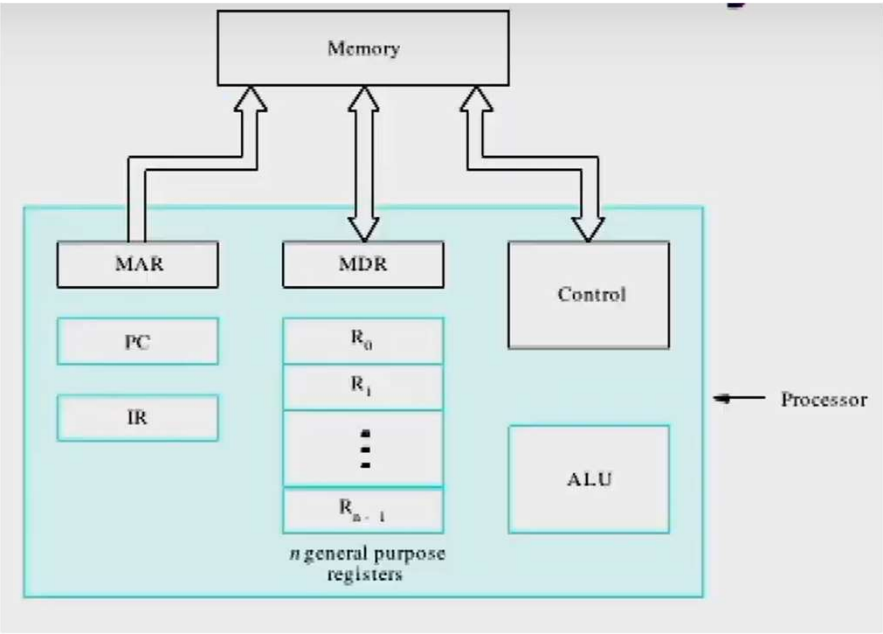
Deposit results in storage for later use

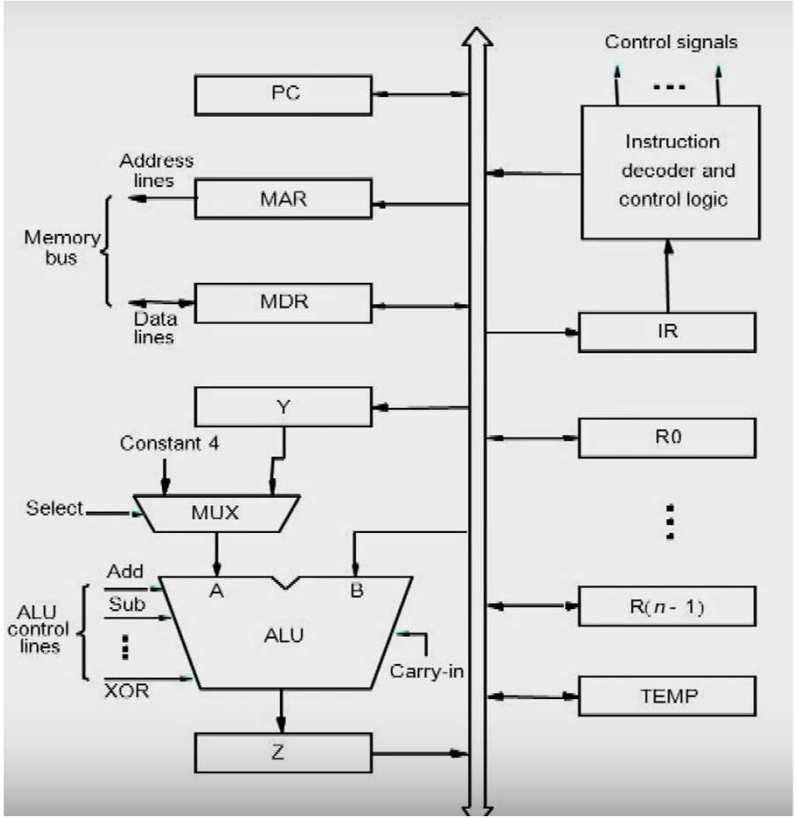
Determine successor instruction

PROCESSOR – MEMORY INTERACTION

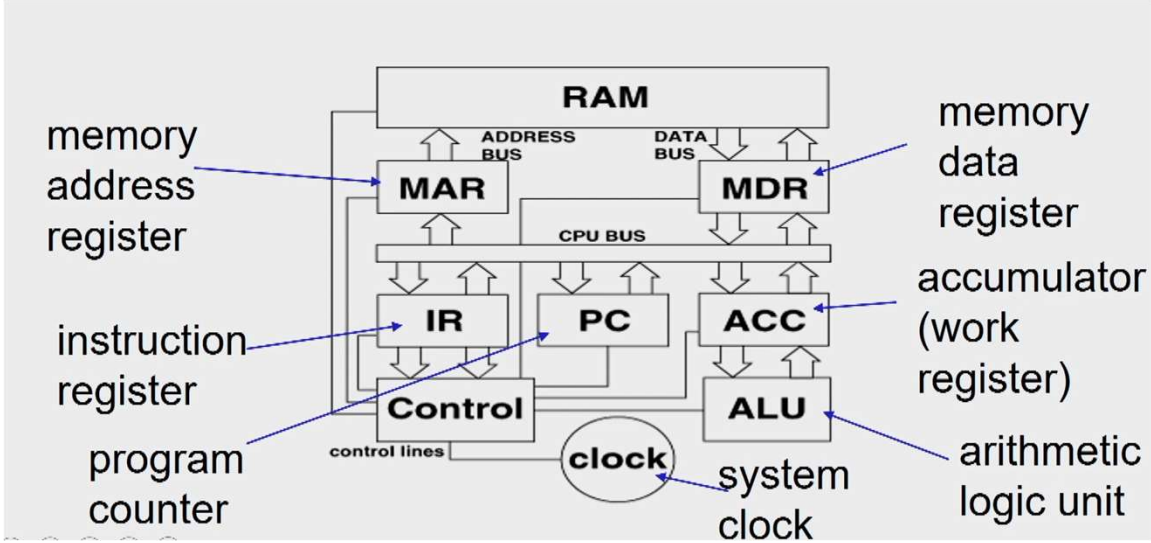


PROCESSOR-MEMORY INTERACTION





INSIDE CPU



Fetch – decode – execute – store all happens through this modules

TYPE OF INSTRUCTIONS IN TERMS OF OPERATIONS (OPERATIONS THAT A CPU CAN DO)

- Arithmetic and Logical Operations
  - integer arithmetic
  - comparing two quantities
  - shifting, rotating bits in a quantity
  - testing, comparing, and converting bits

TYPE OF INSTRUCTIONS IN TERMS OF OPERATIONS

- Data Movement Operations
  - moving data from memory to the CPU
  - moving data from memory to memory
  - input and output

TYPE OF INSTRUCTIONS IN TERMS OF OPERATIONS

- Program Control Operations
  - starting a program
  - halting a program
  - skipping to other instructions
  - testing data to decide whether to skip over some instructions

A BROAD CLASSIFICATION OF ISA

ISA: The Instruction Set Architecture (ISA) is the interface between computer hardware and software. It defines the set of instructions that a processor can execute

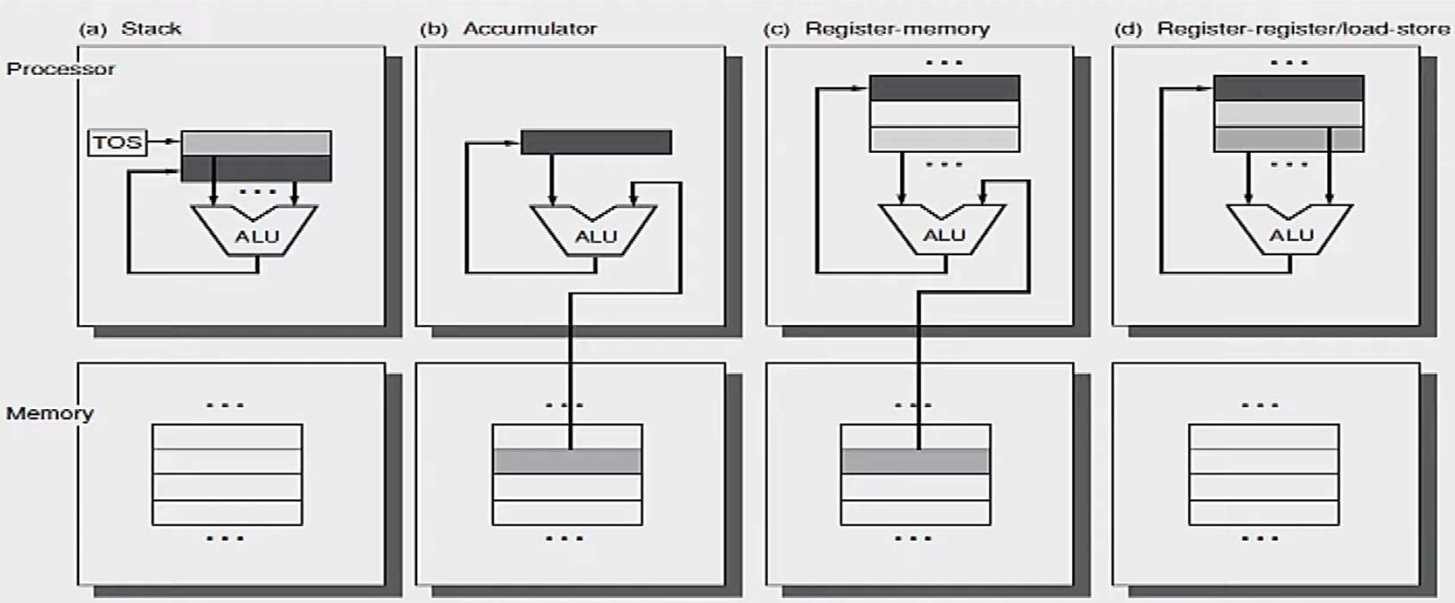
Based on

1. the context of the organization and design of the CPU
2. the positions of the operands

- Stack Architecture
- Accumulator Architecture
- Register-Memory Architecture
- Register-Register or Load Store Architecture

UNDERSTANDING THE TYPES

A simple Equation:
A = D * (B+C) -E



Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

The code sequence for C = A + B for four classes of instruction sets.

Stack Machine
Push D
Push B
Push C
Add
Mul
Push E
Sub
Pop A

Accumulator Machine
Load B
Add C
Mul D
Sub E
Store A

Addressing Modes

- One operand is kept in register for efficiency
- But both can be from memory
- Depend on architectures

Addressing mode refers to the way an instruction specifies the location of its operands (data).

– **Register**	add r1,	r2	r1 <- r1+r2
– **Immediate**	add r1,	#5	r1 <- r1+5
– **Direct**	add r1,	(0x200)	r1 <- r1+M[0x200]
– **Register indirect**	add r1,	(r2)	r1 <- r1+M[r2]
– **Displacement**	add r1,	100(r2)	r1 <- r1+M[r2+100]
– **Indexed**	add r1,	(r2+r3)	r1 <- r1+M[r2+r3]
– **Scaled**	add r1,	(r2+r3*4)	r1 <- r1+M[r2+r3*4]
– **Memory indirect**	add r1,	@(r2)	r1 <- r1+M[M[r2]]
– **Auto-increment**	add r1,	(r2)+	r1 <- r1+M[r2], r2++
– **Auto-decrement**	add r1,	-(r2)	r2--, r1 <- r1+M[r2]

Effective address?

Processor Category 1 - RISC (Based on ISA)

RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) are two different CPU architectures that define how processors execute instructions.

- Philosophy: Simplicity and efficiency—fewer, simpler instructions executed in a single clock cycle.

- Key Features:

- Fixed-length instructions (typically one word).
- Load/store architecture (only load/store instructions access memory, others operate on registers).
- Fewer addressing modes.
- Highly optimized for pipelining (multiple instructions executed in parallel).
- Requires more instructions to accomplish complex tasks.
  - Examples: ARM, RISC-V, MIPS, PowerPC

Processor Category 2 - CISC (Based on ISA)

- Philosophy: Rich, complex instruction set—single instructions perform multi-step operations.
- Key Features:

- Variable-length instructions.
- Memory-to-memory operations (can directly access memory without needing load/store instructions).
- Many addressing modes.
- Fewer Registers.
- Fewer instructions needed per program, but each instruction takes multiple cycles.
  - Examples: x86 (Intel & AMD), VAX, IBM System/360

MEASURING PERFORMANCE

- When can we say one computer/architecture/ design is better than other?
  - Desktop PC - Execution time of a program
  - Servers (Transaction / unit time) → Throughput

- When can we say a machine X is n times faster than Y ?
  - Execution time$_Y$ / Execution time$_x$ = n
  - Throughput$_Y$ / Throughput$_x$ = n

MEASURING PERFORMANCE

- Typical performance metrics:
  - Response Time
  - Throughput
  - CPU time
  - Wall clock time
  - Speedup
- Benchmarks (WHY??)
  - Toy programs (e.g. sorting, matrix multiply)
  - Benchmark Suits (e.g. SPEC06, SPLASH , PARSEC)

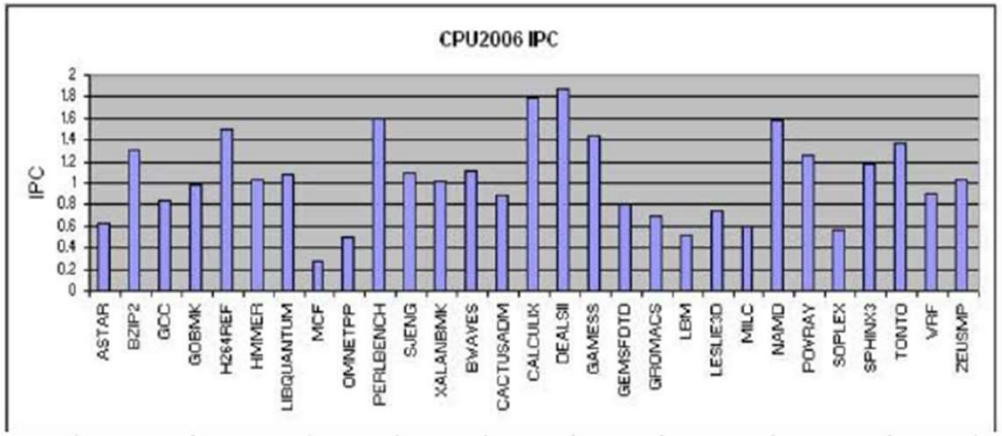
BENCHMARK SUIT

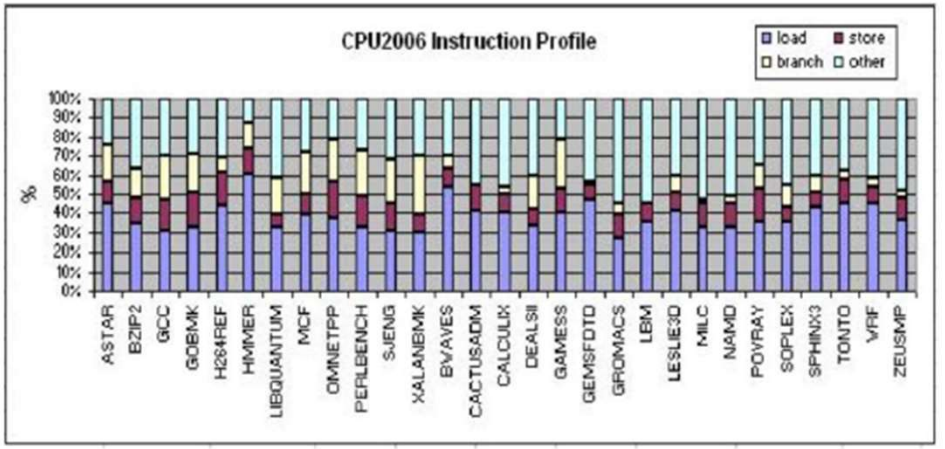
SPEC CPU2006 Programs

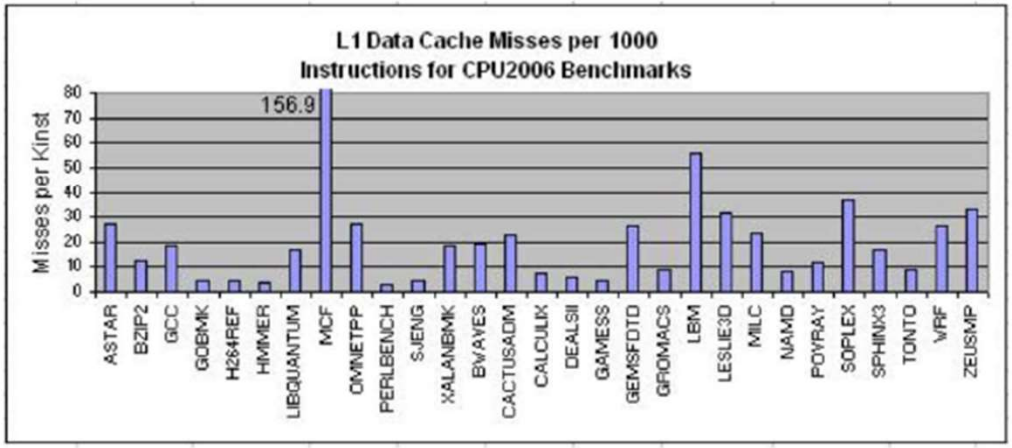
	Benchmark	Language	Descriptions
CINT2006 (Integer) 12 programs	400.perlbench	C	PERL Programming Language
	401.bzip2	C	Compression
	403.gcc	C	C Compiler
	429.mcf	C	Combinatorial Optimization
	445.gobmk	C	Artificial Intelligence: go
	456.hmmer	C	Search Gene Sequence
	458.sjeng	C	Artificial Intelligence: 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

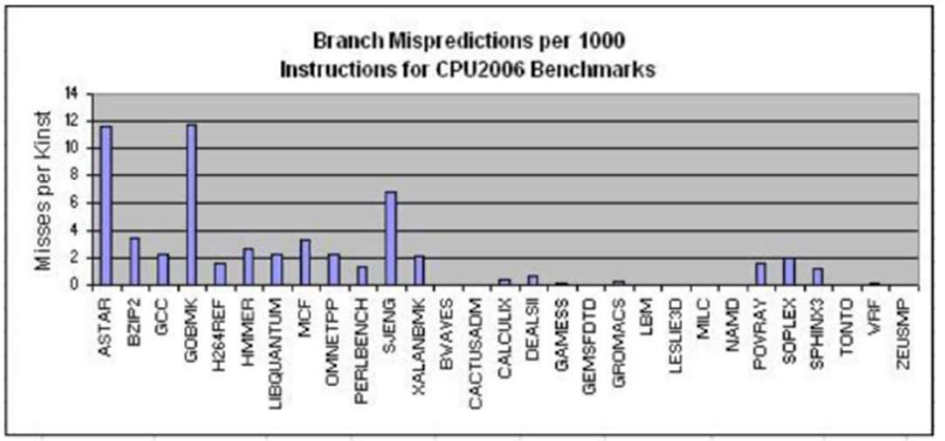
GeoMean can be a good approach

SOME BENCHMARK BASED EVALUATION









Prakash, Tribuvan Kumar, and Lu Peng. "Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor." ISAST Trans. Comput. Softw. Eng 2.1 (2008): 36-41.

AMDAHL'S LAW

- Amdahl's Law defines the speedup that can be gained by improving some portion of a computer.
- The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.

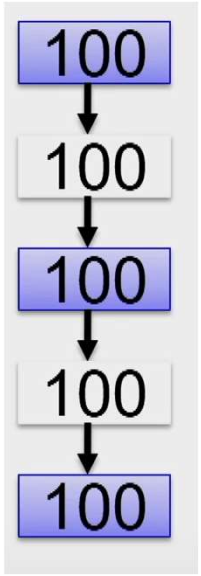
$$ET_{new} = ET_{old} \times (1 - \text{fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}$$

$$Speedup = \frac{ET_{old}}{ET_{new}} = \frac{1}{(1 - \text{fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

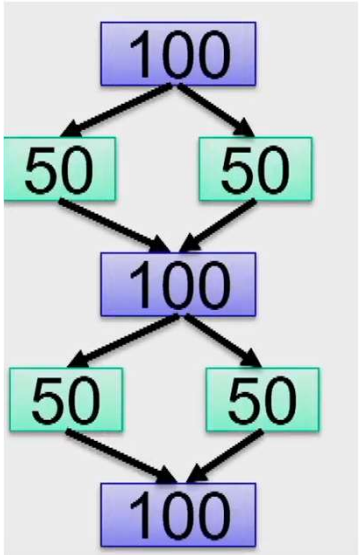
AMDAHL'S LAW ILLUSTRATION

Example: Suppose that we want to enhance the floating point operations of a processor by introducing a new advanced FPU unit. Let the new FPU is 10 times faster on floating point computations than the original processor. Assuming a program has 40% floating point operations, what is the overall speedup gained by incorporating the enhancement?

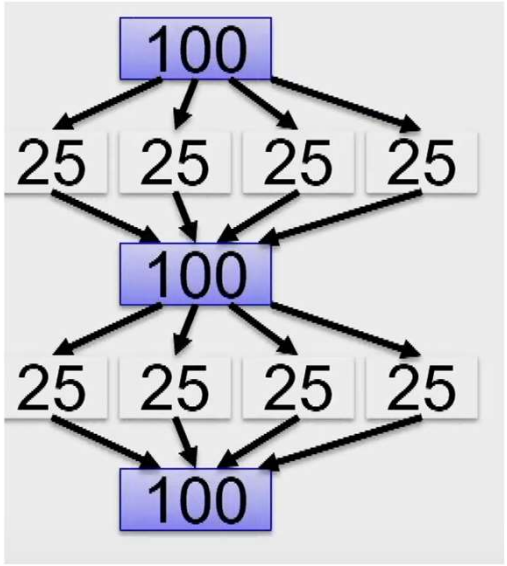
AMDAHL'S LAW FOR PARALLEL PROCESSING



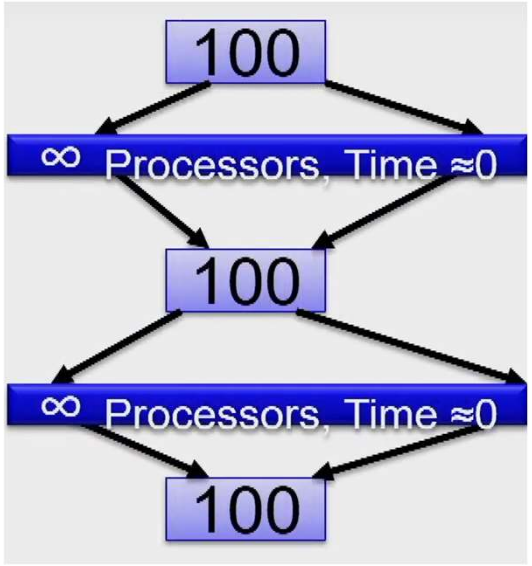
Time 500
Speedup = 1X



Time 400
Speedup = 1.25X



Time 350
Speedup = 1.4X

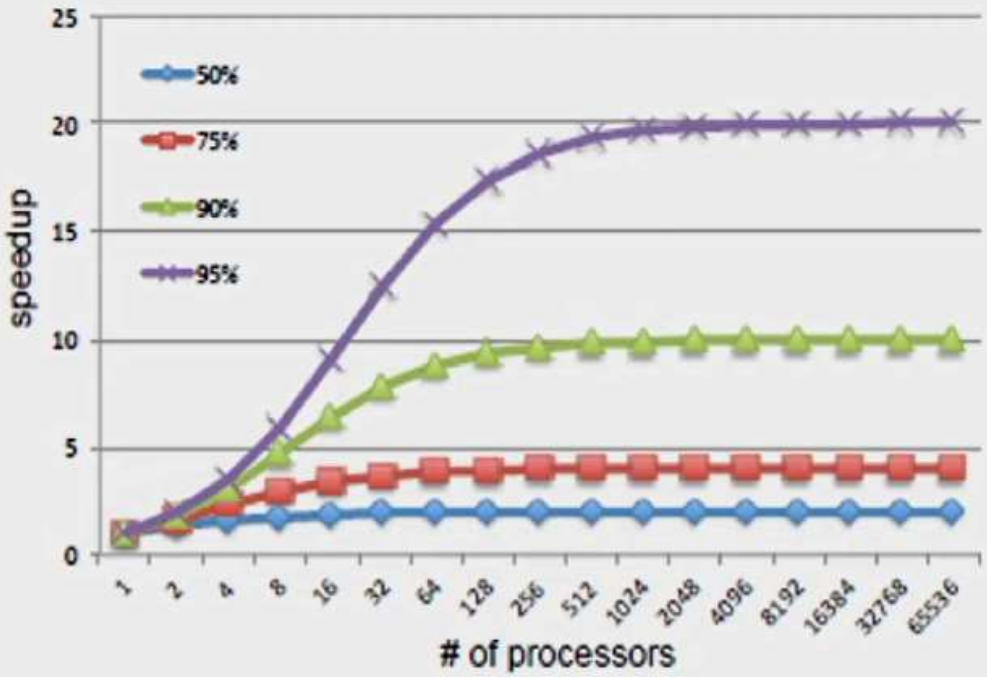


Time 300
Speedup = 1.7X

HOW MUCH SPEEDUP CAN BE ACHIEVED?

Amdahl's Law:

$$Speedup = \frac{1}{(1-\alpha)+\frac{\alpha}{n}}$$



AMDAHL'S LAW ILLUSTRATIONS

A common transformation required in graphics processors is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark.

One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for half of the execution time for the application. Compare these two design alternatives using Amdahl's Law.

DESIGN PRINCIPLES

- All processors are driven by clock.
- Expressed as clock rate in GHz or clock period in ns
- CPU Time = CPU clock cycles x clock cycle time

$$CPI = \frac{\text{CPU clock cycles for a program}}{\text{Instruction Count}}$$

$$\text{CPU Time} = IC \times CPI \times CCT$$

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

DESIGN PRINCIPLES

$$\text{CPU Time} = \text{IC} \times \text{CPI} \times \text{CCT}$$

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

Different instructions may need different clocks.

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{ICi} \times \text{CPI}_i$$

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{ICi} \times \text{CPI}_i \right) \times \text{CCT}$$

- Clock Cycle Time – Hardware Technology.
- CPI – Organization and ISA.
- IC – ISA and Compiler Technology.

PERFORMANCE ANALYSIS

Consider two programs A and B that solves a given problem. A is scheduled to run on a processor P1 operating at 1 GHz and B is scheduled to run on processor P2 running at 1.4 GHz. A has total 10000 instructions, out of which 20% are branch instructions, 40% load store instructions and rest are ALU instructions. B is composed of 25% branch instructions. The number of load store instructions in B is twice the count of ALU instructions. Total instruction count of B is 12000. In both P1 and P2 branch instructions have an average CPI of 5 and ALU instructions has an average CPI of 1.5. Both the architectures differ in the CPI of load-store instruction. They are 2 and 3 for P1 and P2, respectively. Which mapping (A on P1 or B on P2) solves the problem faster, and by how much?

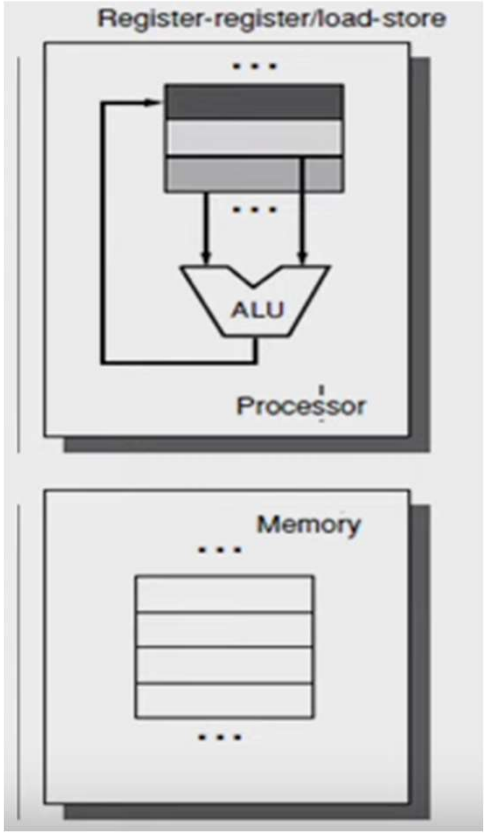
Solve it

A on P1 (1GHz → CCT = 1ns)	B on P2 (1.4 GHz→ CCT = 0.714ns)
IC=10000	IC=12000
Fraction BR: L/S: ALU = 20: 40: 40	Fraction BR: L/S: ALU = 25: 50: 25
CPI of BR: L/S: ALU = 5: 2: 1.5	CPI of BR: L/S: ALU = 5: 3 : 1.5

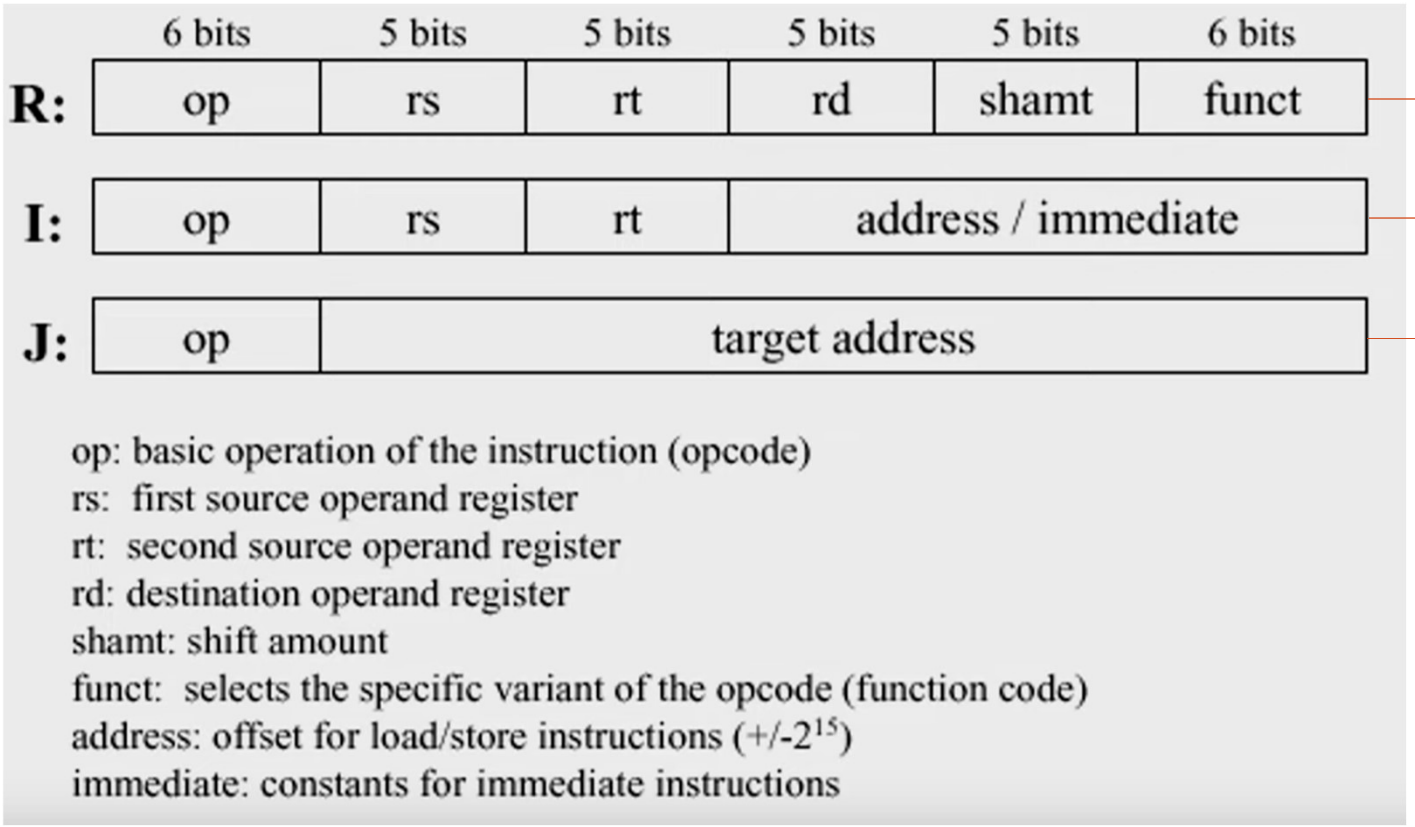
INTRODUCTION TO MIPS

Case Study: RISC Microprocessor

- Microprocessor without Interlocked (no mechanism to detect instruction dependency) Pipelined Stages
- 32 Registers (32-bit each), R0, R1,....., R31
- Uniform length instructions
- RISC- Load store architecture



HOW ARE INSTRUCTIONS REPRESENTED IN MIPS?



op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions (+/-2^{15})
immediate: constants for immediate instructions

Arithmetic and logical operations with register operands.

Data transfer operations between registers and memory

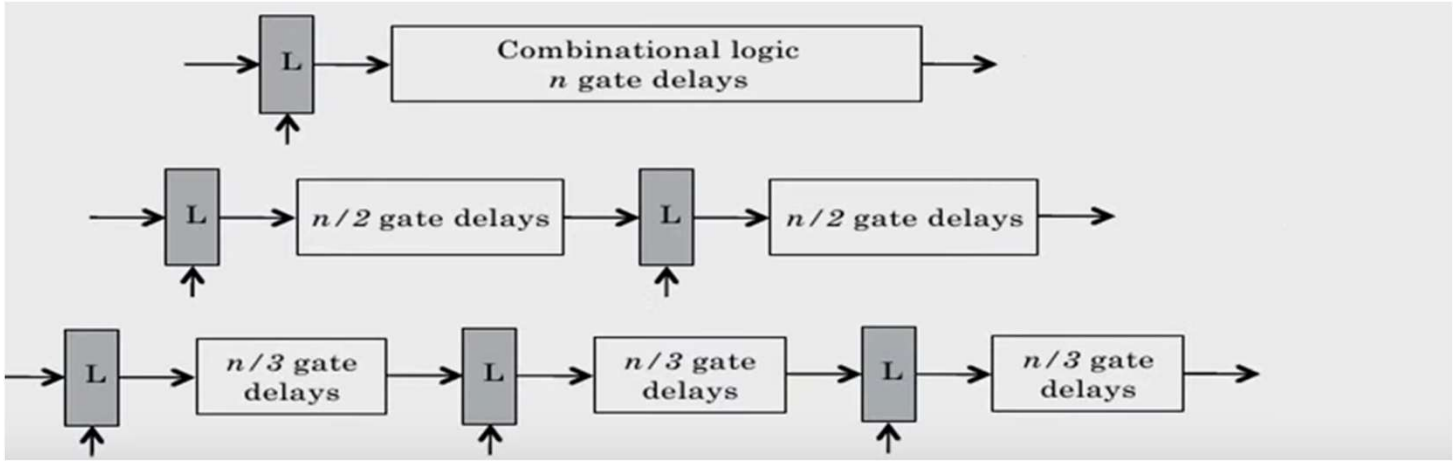
Control transfer operations, typically involving unconditional jumps or branches

BASIC CHARACTERISTICS OF PIPELINING

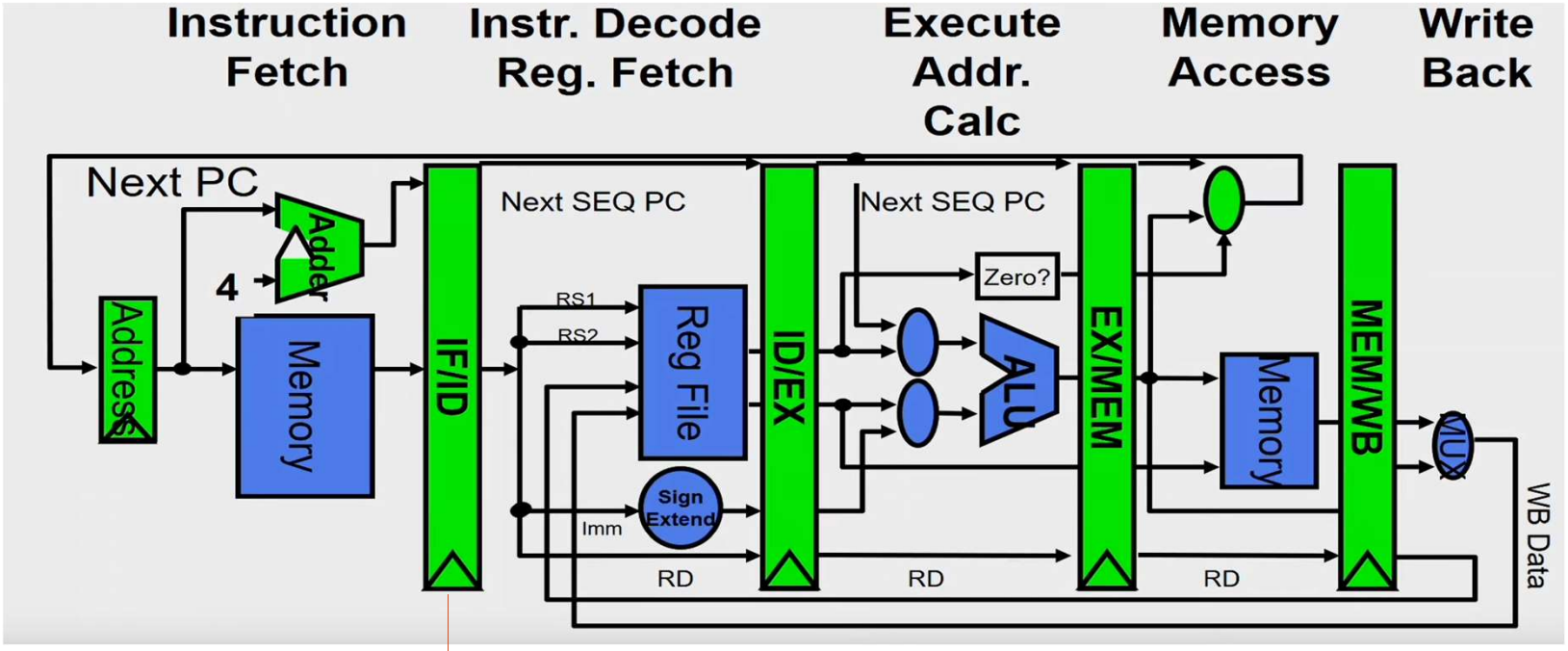
- Pipelining does not reduce the latency of a single task.
- It improves the system's throughput.
- Pipeline rate ---> limited by the slowest pipeline stage
- Potential speedup depends on the number of stages.
- Unbalanced length of pipe stages reduces speedup.
- Time to fill the pipeline and time to drain it reduces speedup.

PIPELINING IN CIRCUITS

- Pipelining partitions the system into multiple independent stages.
- It puts buffers in between the stages.



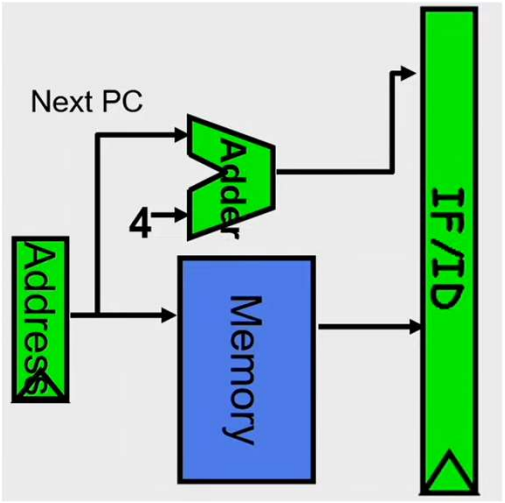
PIPELINED RISC DATA PATH



Pipeline Registers/buffers

INSTRUCTION FETCH (IF)

- Based on PC, fetch the instruction from memory
- PC++ ---->simple for RISC



INSTRUCTION DECODE/REGISTER FETCH CYCLE (ID)

- Decode the instruction + Register read operation
- Fixed field decoding

Example 1:

ADD R1, R2, R3

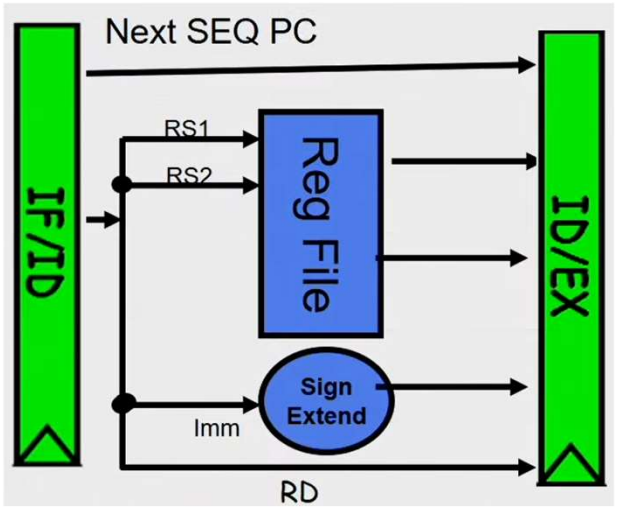
10100011 00000001 00000010 00000011

observe the difference

Example 2:

LW R1, 8(R2)

10000110 00000001 00001000 00000010

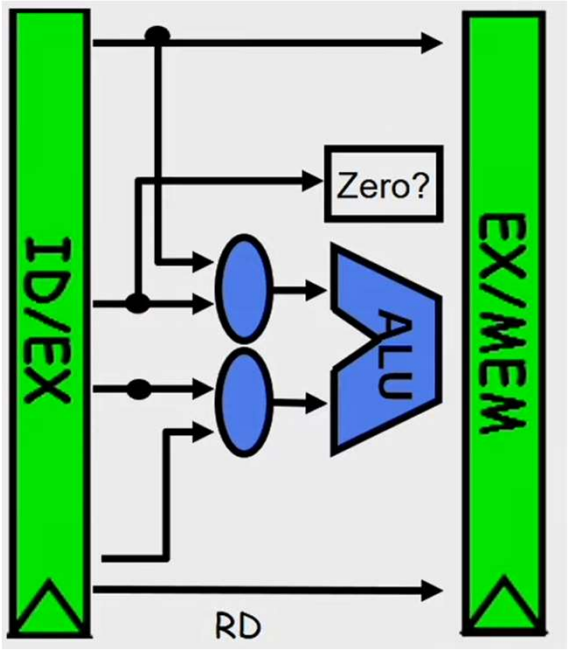


EXECUTION/EFFECTIVE ADDRESS CYCLE (EX)

- Memory Reference: Calculate the effective address
  - LW R1, 8(R2)
  - SW R1, 8(R2)

EFF ADDR = [R2] + 8

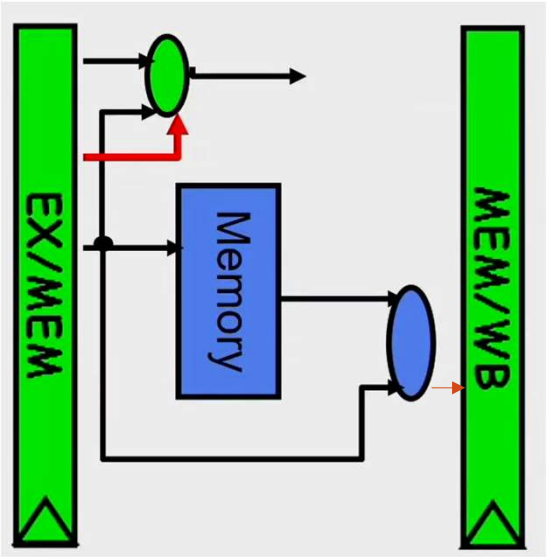
- Register-register ALU instruction
  - ADD R1, R2, R3



MEMORY ACCESS CYCLE (MEM)

- Load from memory and store in register ----------> LW R1, 8(R2)
- Store the data from the register to memory ----------> SW R3, 16(R4)

- No operation for ALU operations.



WRITE BACK CYCLE (WB)

- Register-register ALU instruction or Load instruction.
- Write to register file [LW R1, 8(R2)], [ADD R1, R2, R3]