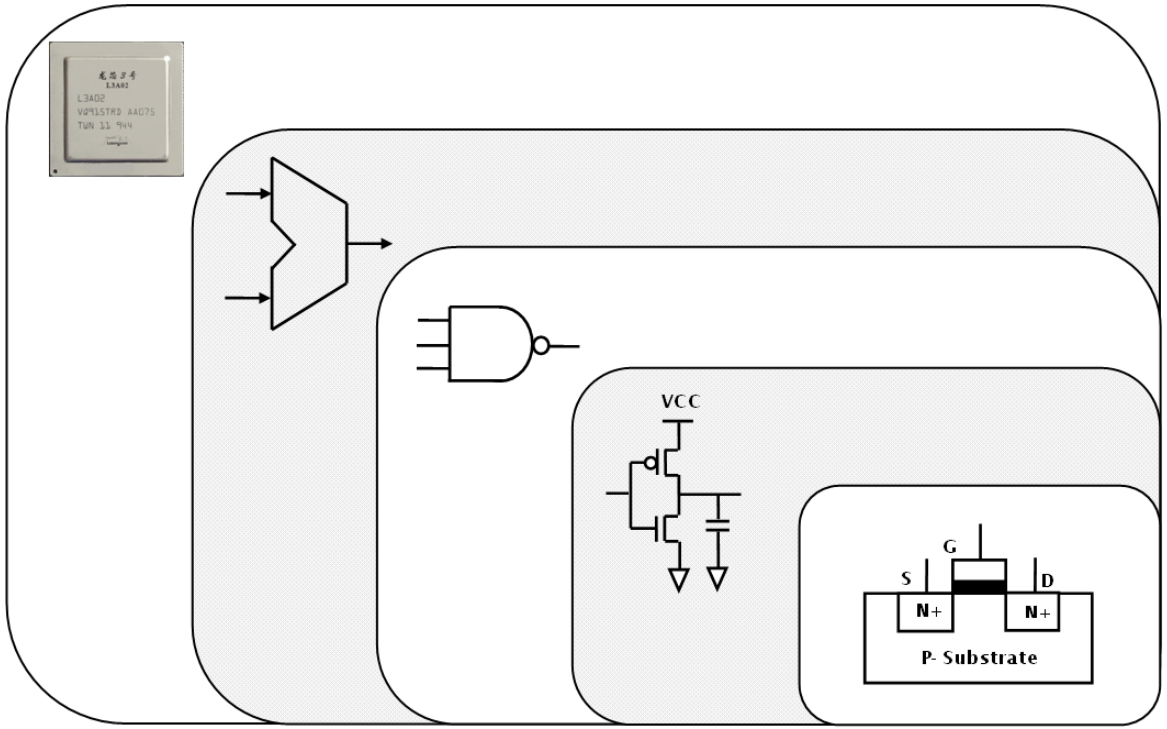
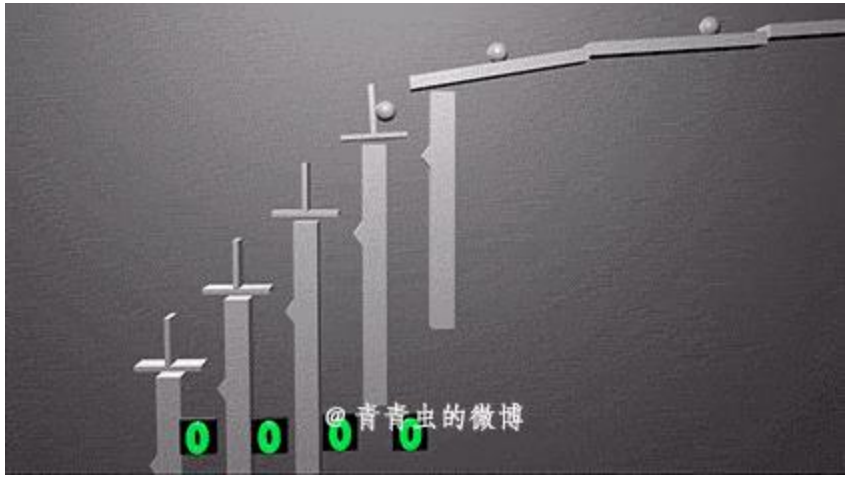
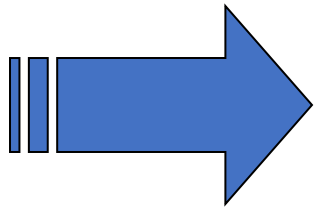
Chapter 1

Fundamentals of computer design

Why use binary in the computers?







- The computer is composed of electronic components, and binary is the easiest to realize.

The history of binary

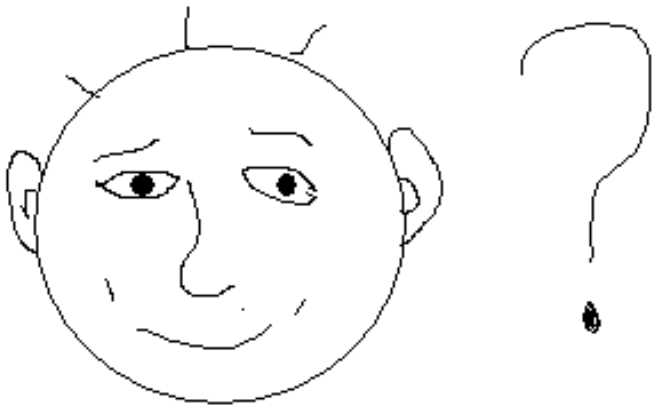
- Leibnitz is the first mathematician to use binary in Europe.
- John von Neumann first introduced binary into computer application, in which binary was used for data and program.
- Even over 2000 B.C. in China, the eight trigrams were invented, which were also binary with two symbols — and --.

Leibniz's binary Medallion for the Duke of Auguste

The history of binary

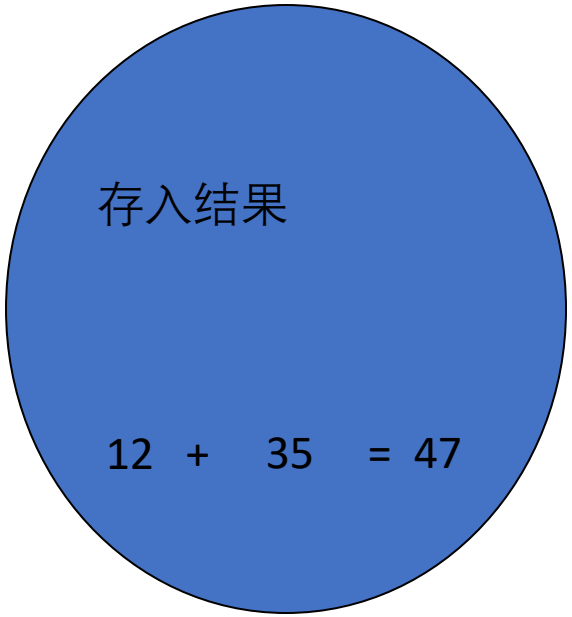
(3×4) + (5×7)?

- 3 ×4 =12
- 5 ×7=35
- 12+35 = 47

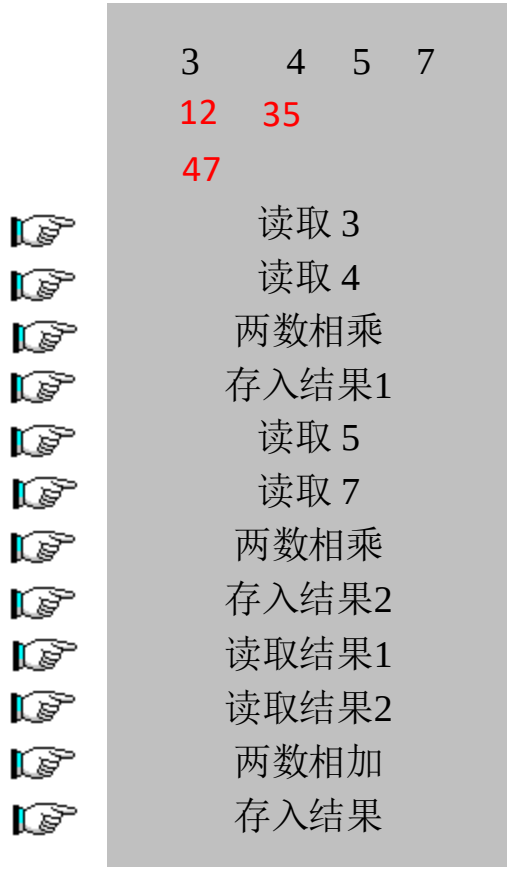


The history of binary

CPU



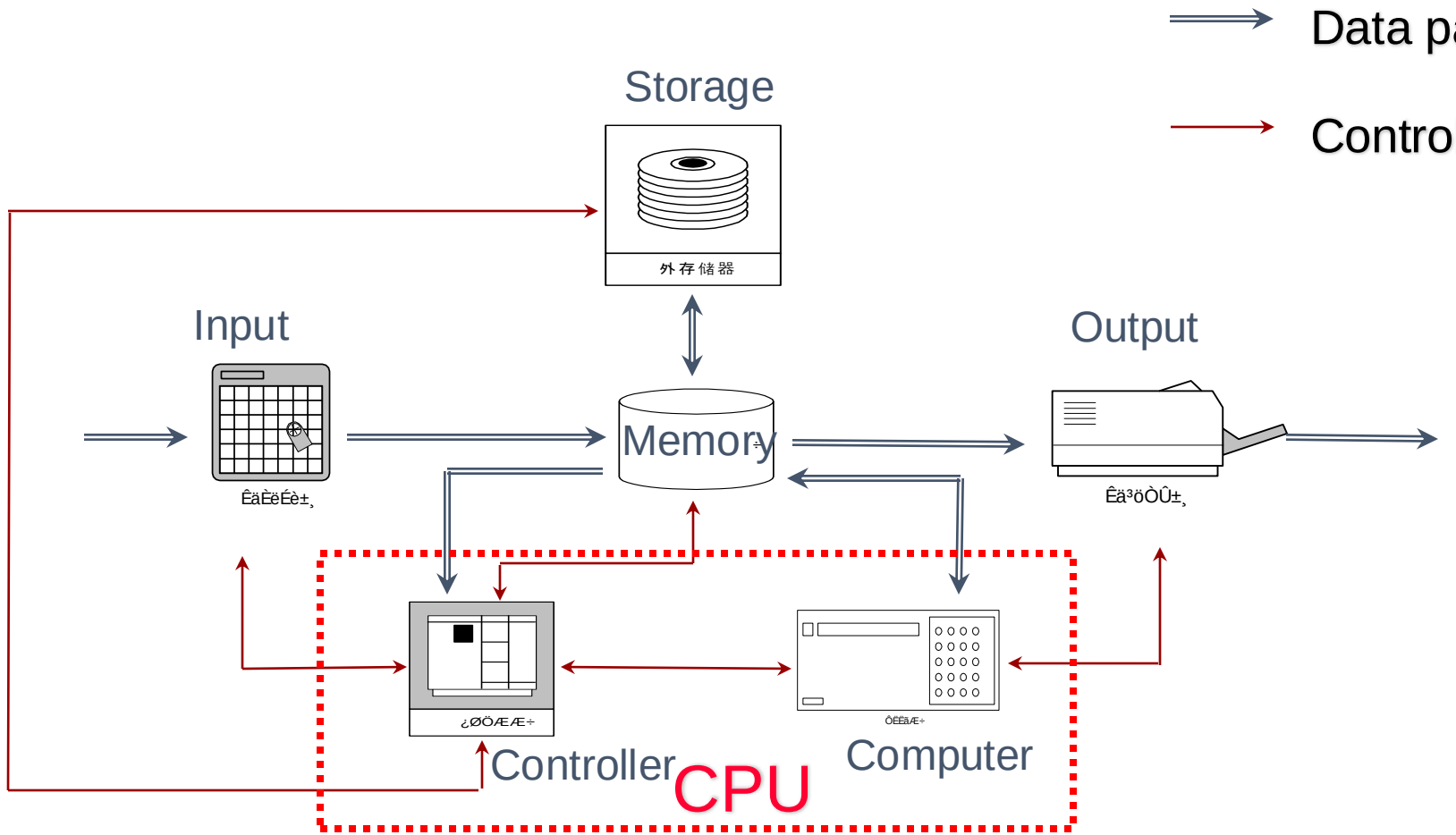
Memory



Von Neumann Structure

- Von Neumann structure: data and programs are in memory.
- CPU takes instructions and data from memory for operation and puts the results into memory

Von Neumann Structure



The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore's Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
  - Large Language Model
- Computers are pervasive

How you will learn?

- Possibility:
  - standing on the shoulder of giants.
- Concepts, Ideas and Principles
- Quantitative approaches
- Hit the problem and right way to solve problem

- As a man sows, so he shall reap.

Big Men in Architecture(1)

- Mateo Valero
  - http://personals.ac.upc.edu/mateo/

- For important contributions to instruction level parallelism and superscalar processor design.

Big Men in Architecture(2)

- John Leroy Hennessy
- Hennessy is one of the founders of MIPS Computer Systems Inc. as well as Atheros and served as the tenth President of Stanford University.
- Hennessy announced that he would step down in the summer of 2016. He was succeeded as President by Marc Tessier-Lavigne.
- Marc Andreessen called him "the godfather of Silicon Valley."

Big Men in Architecture(3)

- David Andrew Patterson

- An American computer pioneer and academic who has held the position of professor of computer science at the University of California, Berkeley since 1976.

- He currently is Vice Chair of the Board of Directors of the RISC-V Foundation, and the Pardee Professor of Computer Science, Emeritus at UC Berkeley.

one big thing at a time

Big Men in Architecture(3)

JOHN L HENNESSY

United States – 2017

CITATION

For pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.

RESEARCH

DAVID PATTERSON

United States – 2017

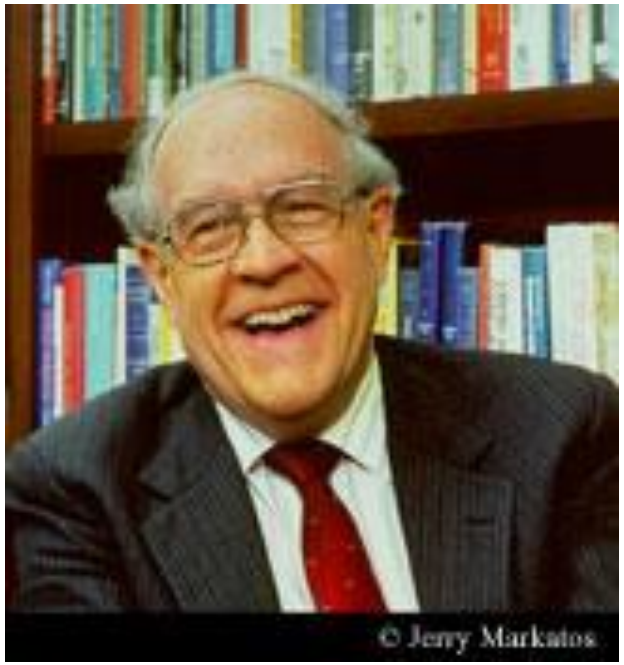
CITATION

For pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.

RESEARCH

- Turing Award for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.
- Hennessy and Patterson created a systematic and quantitative approach to designing faster, lower power, and reduced instruction set computer (RISC) microprocessors.

Big Men in Architecture(4)



Frederick P. Brooks

http://www.cs.unc.edu/~brooks/

- 1999 ACM Turing Award
- landmark contributions to computer architecture, operating systems, and software engineering."

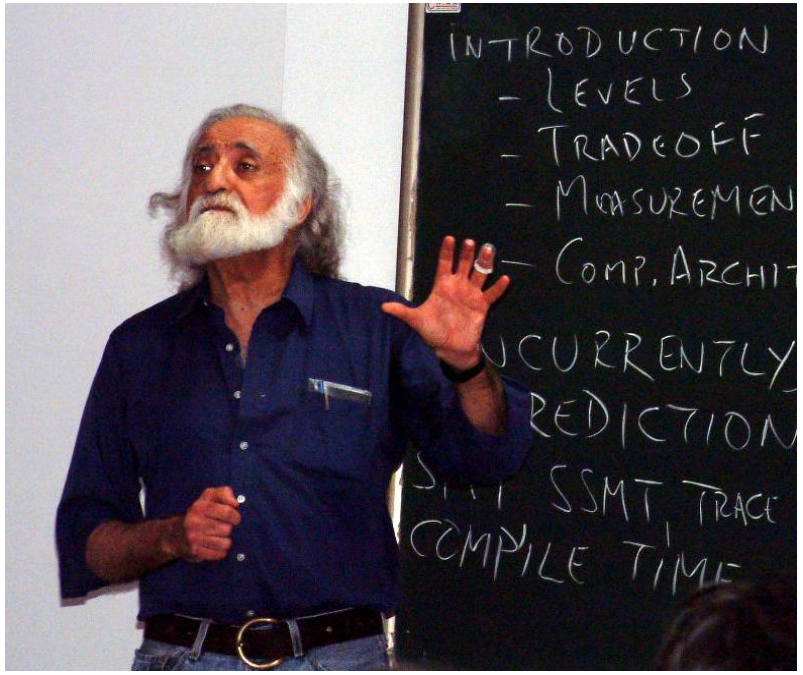
2004 Eckert-Mauchly Award

"For the definition of computer architecture and contributions to the concept of computer families and to the principles of instruction set design; for seminal contributions in instruction sequencing, including interrupt systems and execute instructions; and for contributions to the **IBM 360 instruction set architecture**."

Big Men in Architecture(5)

- Robert Marco Tomasulo
- A computer scientist, and the inventor of the Tomasulo algorithm.
- Tomasulo was the recipient of the 1997 Eckert–Mauchly Award "or the ingenious Tomasulo algorithm, which enabled out-of-order execution processors to be implemented."

Big Men in Architecture(6)



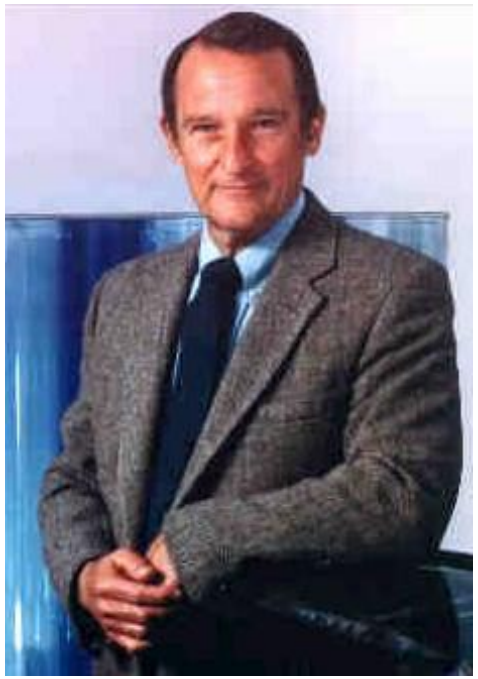
- Yale Patt
- For important contributions to instruction level parallelism and superscalar processor design.

Big Men in Architecture(7)

- Michael J. Flynn

- http://www.cpe.calpoly.edu/IAB/flynn.html
- For his important and seminal contributions to processor organization and classification, computer arithmetic and performance evaluation.

Big Men in Architecture(8)

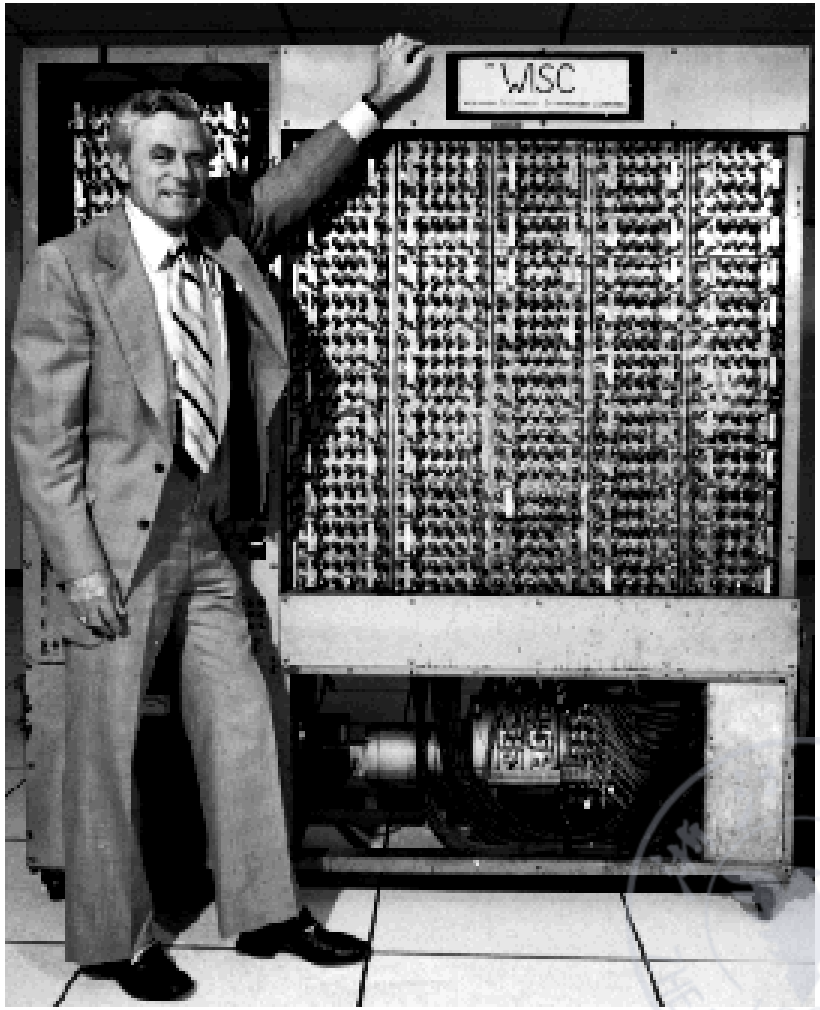


- Seymour Cray
- For a career of achievements that have advanced supercomputing design.
- In 1958, the world's first transistor-based supercomputer was designed and built.
- At the same time, he has made significant contributions to the production of reduced instruction (RISC) high-end microprocessors, and is one of the most important figures in the field of high-performance computers.

Big Men in Architecture(9)

- Amdahl, Gene M.

- For outstanding innovations in computer architecture, including pipelining, instruction look-ahead, and cache memory.



In 1975, Dr. Amdahl stands beside the Wisconsin Integrally Synchronized Computer (WISC), which he designed in 1950. It was built in 1952. *(Image courtesy of Dr. Gene M. Amdahl.)*

Classes of Computers

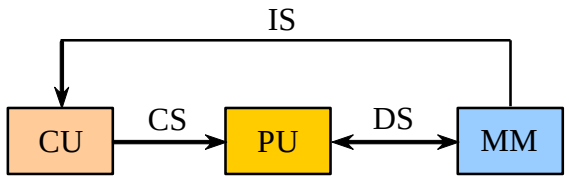
- Desktop computers (or Personal Computers)
  - General purpose, variety of software
  - Emphasize good performance for a single user at relatively low cost.
  - Mostly execute third-party software
- Server computers
  - Emphasize great performance for a few complex applications.
  - Or emphasize reliable performance for many users at once.
  - Greater computing, storage, or network capacity than personal computers.
- Embedded computers
  - Largest class and most diverse.
  - Hidden as components of systems.
  - Stringent power/performance/cost constraints.

Classes of Computers

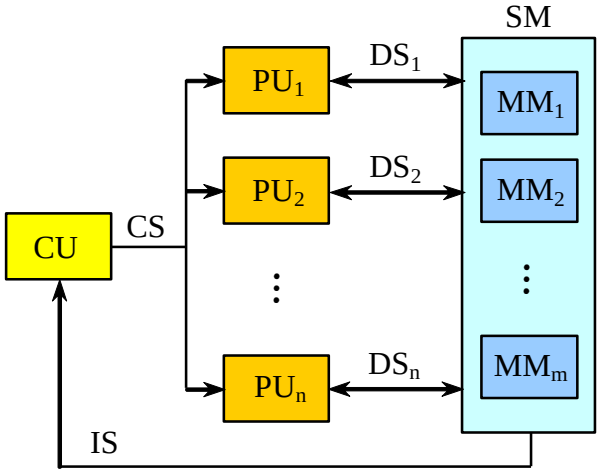
- Personal Mobile Devices
  - Smartphones
  - Tablet/iPad
  - generally have the same design requirements as PCs

- Supercomputer
  - Computer cluster
  - High capacity, performance, reliability
  - Range to building sized

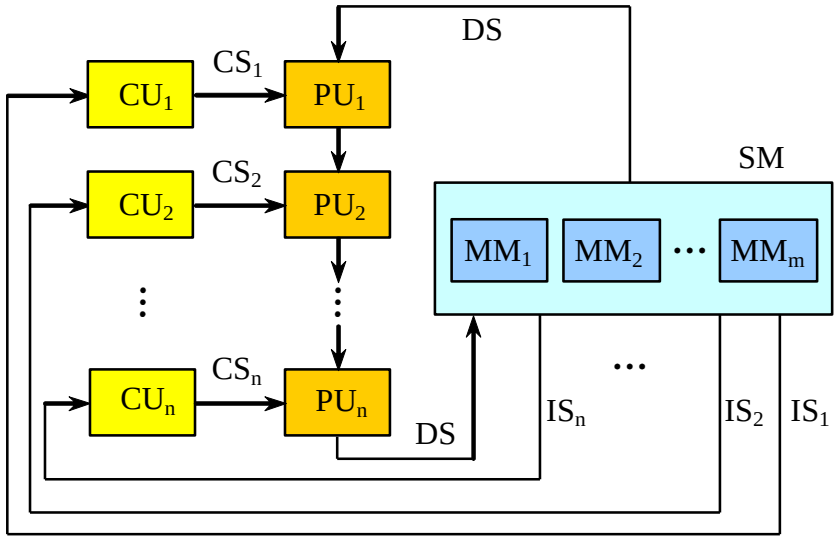
Classed By Flynn



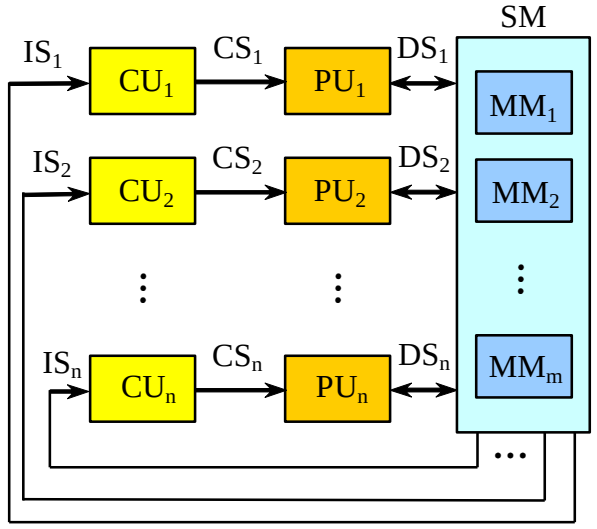
（a）SISD 计算机



（b）SIMD 计算机



（c）MISD 计算机



（d）MIMD 计算机

IS：Instruction stream

DS：Data stream

CS：Control stream

CU：Control unit

PU：Process unit

MM&SM：Memory

What You Will Learn

- 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
- What is parallel processing

Understanding Performance

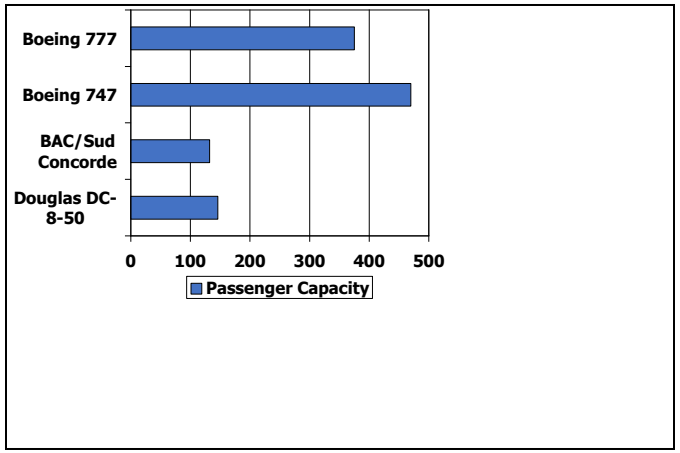
- Algorithm
  - Determines number of operations executed
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (including OS)
  - Determines how fast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

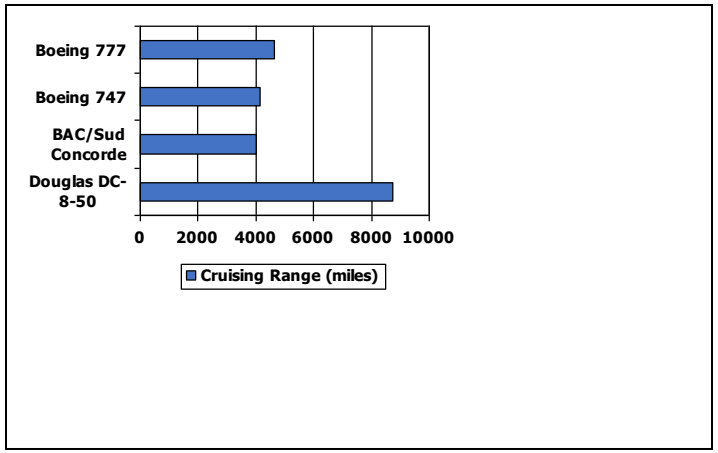
Summarize

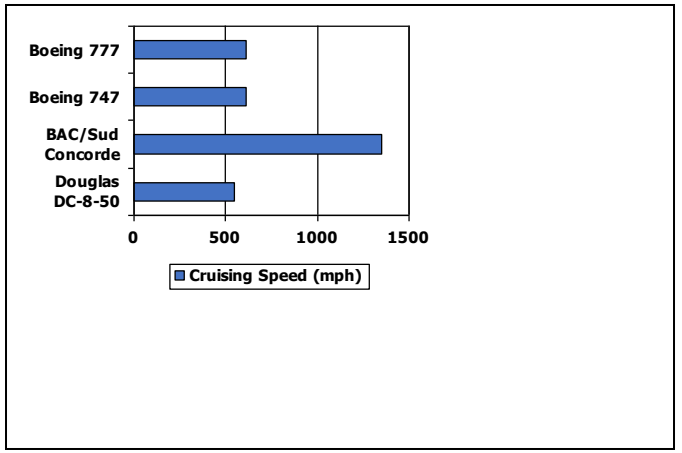
- According to the process of using data, computers are developing in three fields:
  - Speed up processing (parallel)
  - Speed up transmission (accuracy)
  - Increase storage capacity and speed up storage (reliability)
- The central issue discussed and studied in this course is
  - Processing
  - Storage
  - transmission

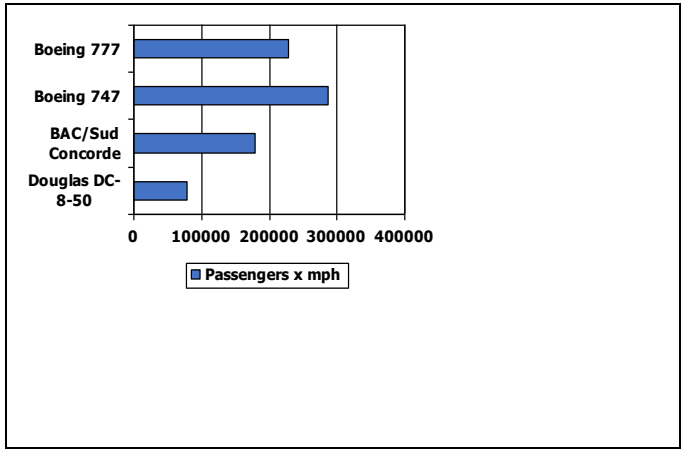
Defining Performance

- Which airplane has the best performance?









Aircraft type	Passenger Capacity	Cruising Range(miles)	Cruishing Speed(mph)	Passengers *mph
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Co ncorde	132	400 0	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

Performance

Being able to gauge the relative performance of a computer is an important but tricky task. There are a lot of factors that can affect performance.

- Architecture
- Hardware implementation of the architecture
- Compiler for the architecture
- Operating system

Furthermore, we need to be able to define a measure of performance.

- Single users on a PC → a minimization of response time.
- Large data → a maximization of throughput

Response Time and Throughput

- Latency (Response time)
  - is the time between the start and completion of an event
  - How long it takes to do a task
- Throughput (bandwidth)
  - is the total amount of work done in a given period of time
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
- How are response time and throughput affected by
  - Replacing the processor in a computer with a faster processor.
  - Adding more processors?
- We'll focus on program response time for now…

Performance

- Define Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y = \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time$_B$ / Execution Time$_A$ = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Performance

Performance has an inverse relationship to execution time.

$$Performance = \frac{1}{Execution\ Time}$$

Comparing the performance of two machines can be accomplished by comparing execution times.

$$Performance_X > Performance_Y$$

$$\rightarrow \quad \frac{1}{Execution_X} > \frac{1}{Execution_Y}$$

$$\rightarrow \quad Execution_Y > Execution_X$$

Measuring Execution Time

- Elapsed time
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises user CPU time and system CPU time
    - User CPU time : CPU time spent in the program itself
    - System CPU time: CPU time spent in the OS, performing tasks on behalf of the program.
  - Different programs are affected differently by CPU and system performance

The main goal of architecture improvement is to improve the performance of the system

How can computers run fast?

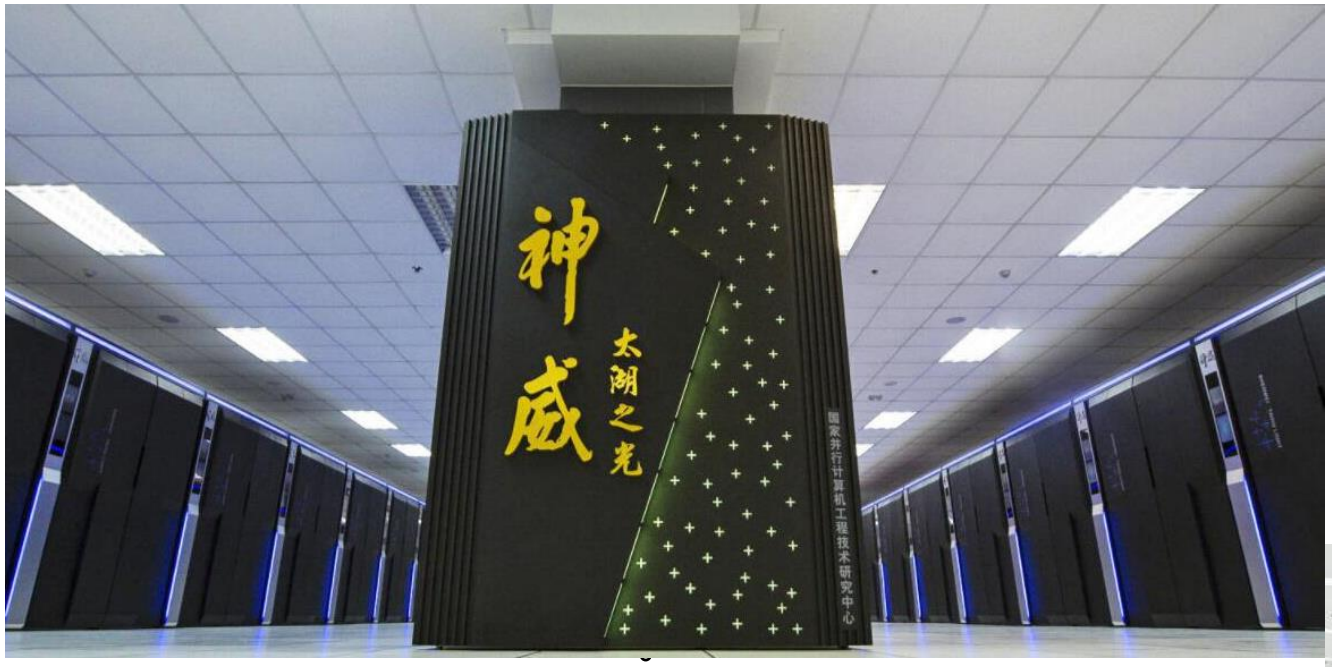
Describe a thing with the least instructions
--Algorithms, compiling

Do more things in a Clock cycle
--Architecture

Increase the speed of "core"
-- main frequency

Is everything OK to run fast?

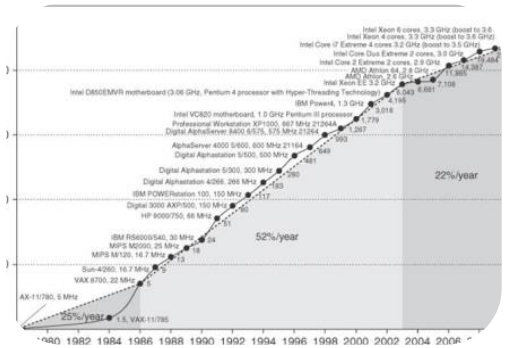
These big guys are strategic.



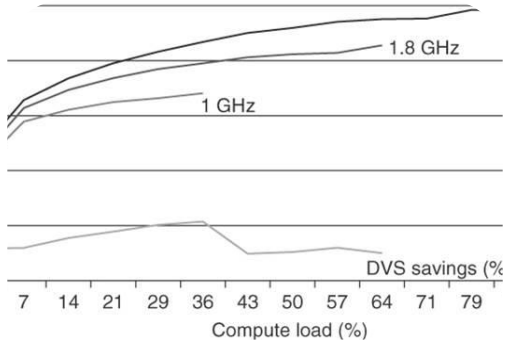
Every minute counts.
The goal is to be as fast as possible!

Measuring Execution Time

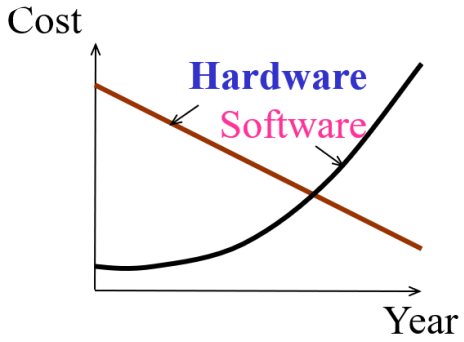
- Computers are becoming more and more common.
- Ubiquitous CPU.
- How can batteries last long?
- To make it more affordable, price matters.



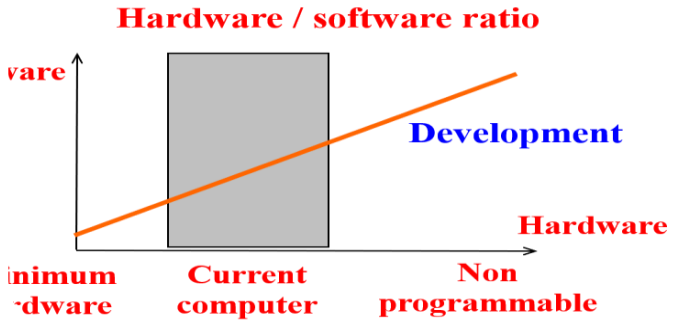
Performance trend



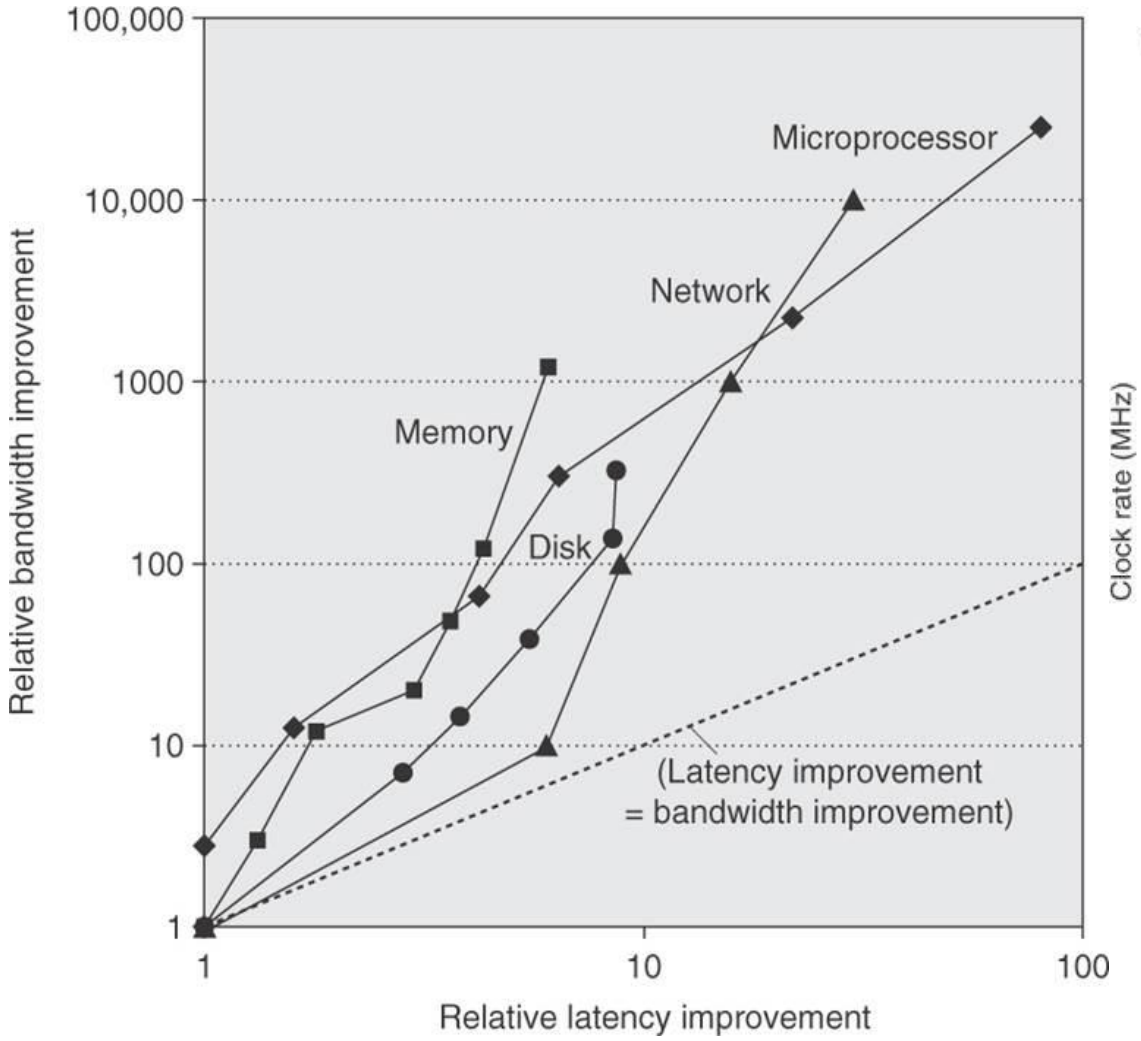
Power and energy consumption

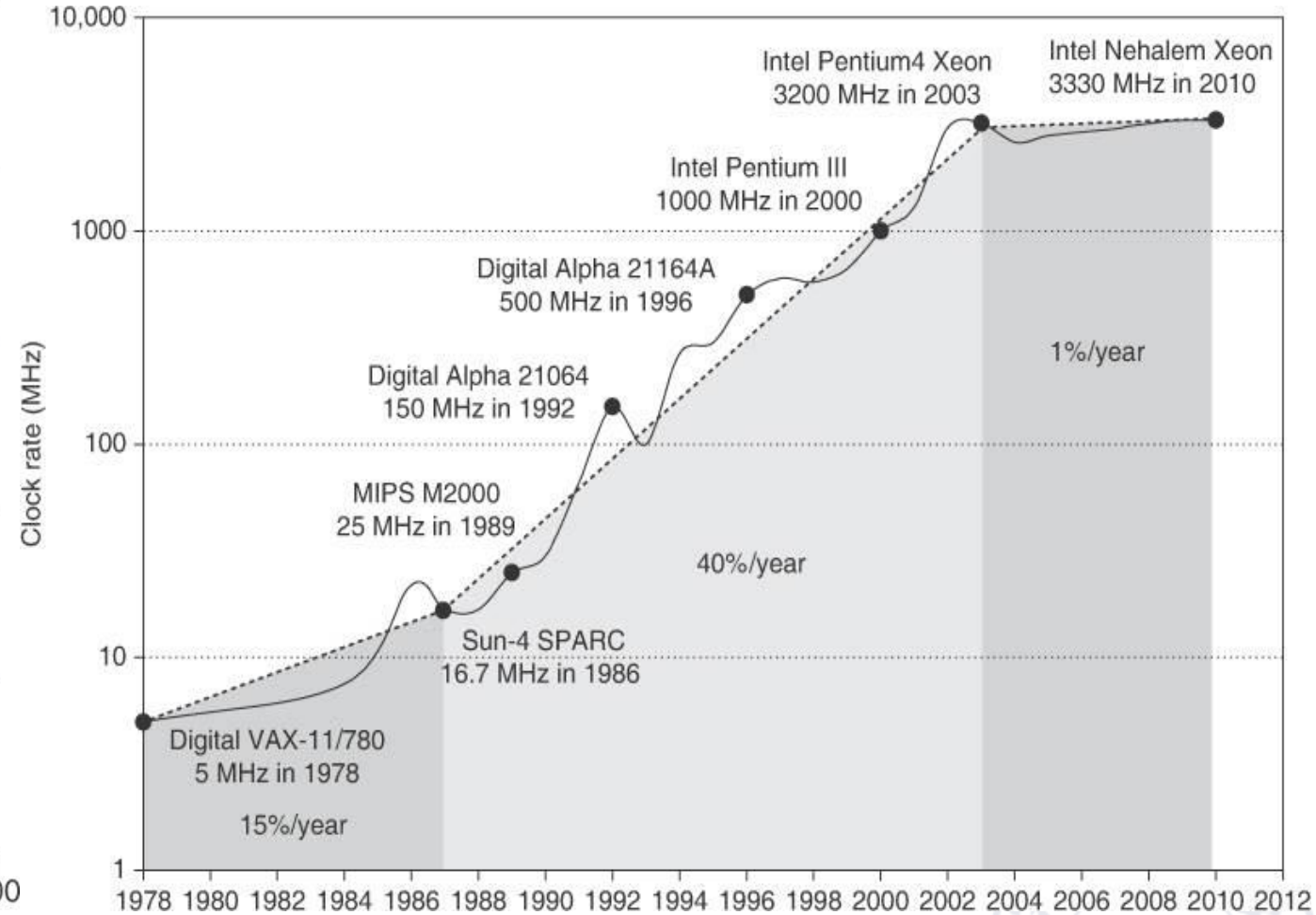


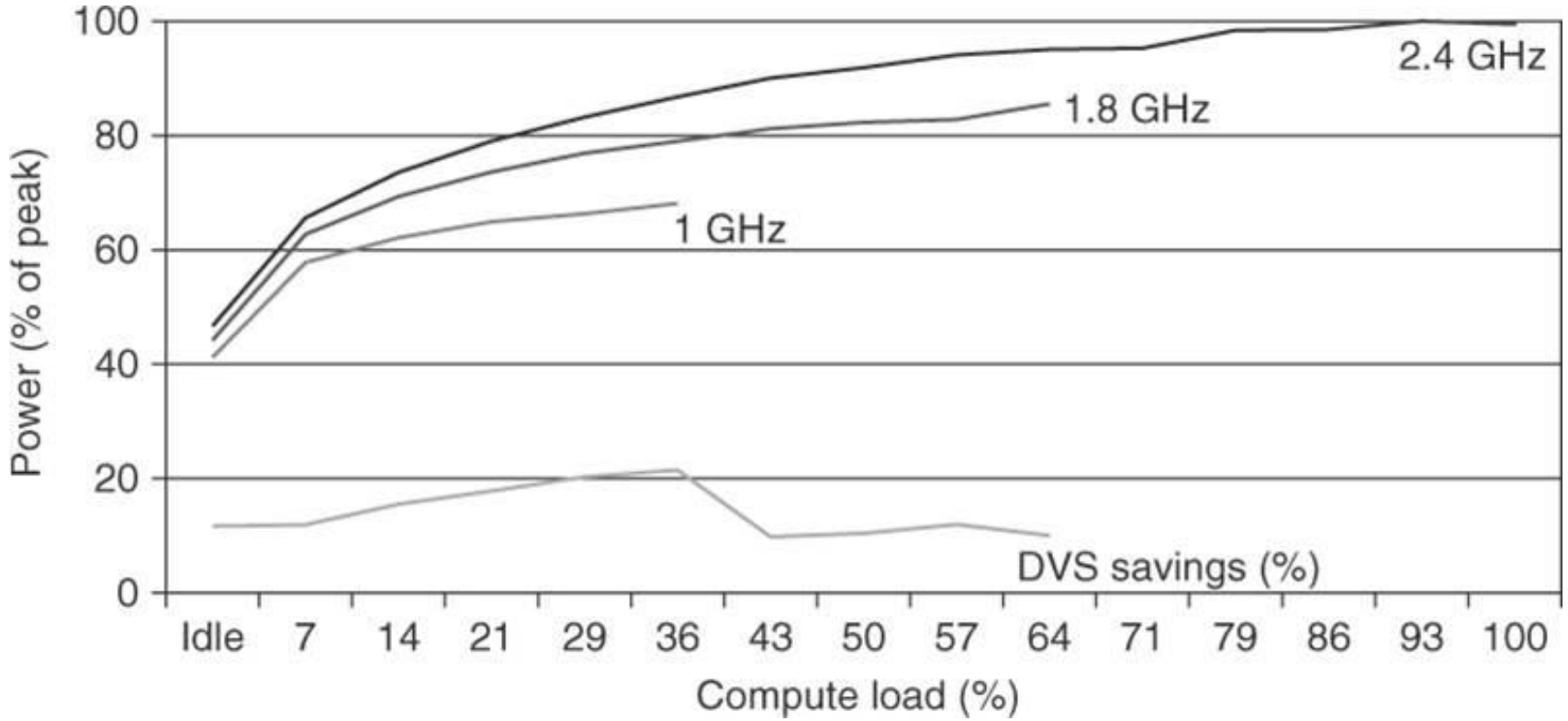
Cost trend

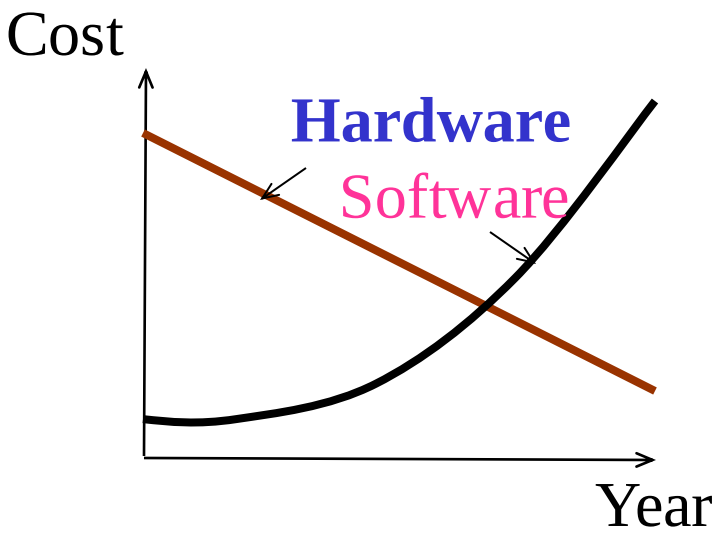


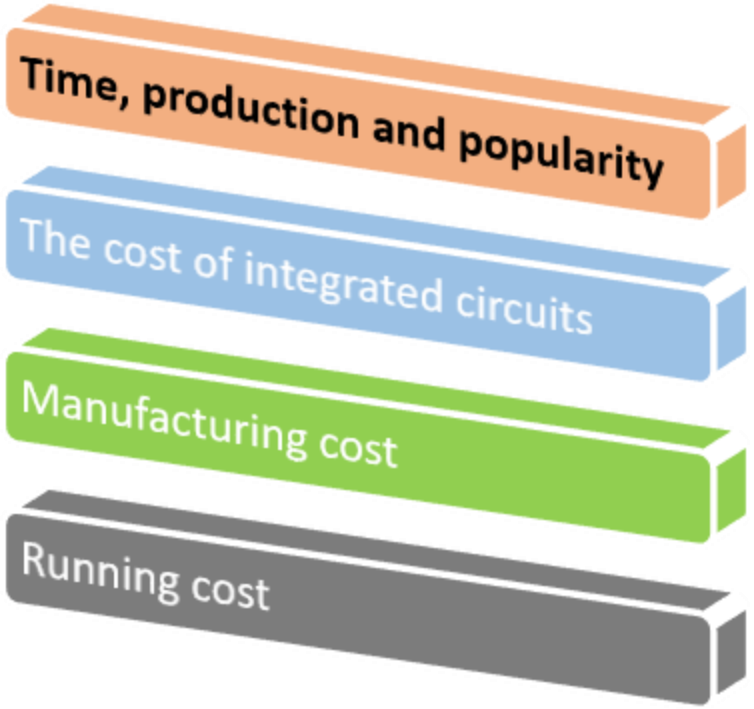
Software and hardware trend

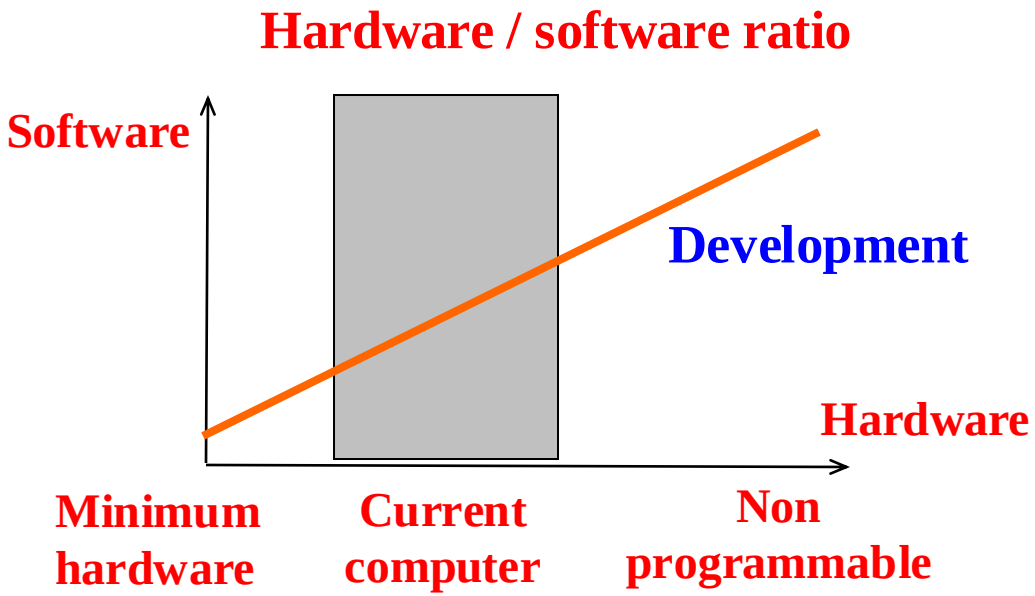

100,000
10,000
1000
100
10
1
Relative bandwidth improvement
Microprocessor
Network
Memory
Disk
(Latency improvement
= bandwidth improvement)
1
10
100
Relative latency improvement


10,000
1000
100
10
1
Clock rate (MHz)
Intel Pentium4 Xeon
3200 MHz in 2003
Intel Nehalem Xeon
3330 MHz in 2010
Intel Pentium III
1000 MHz in 2000
Digital Alpha 21164A
500 MHz in 1996
Digital Alpha 21064
150 MHz in 1992
MIPS M2000
25 MHz in 1989
Sun-4 SPARC
16.7 MHz in 1986
Digital VAX-11/780
5 MHz in 1978
15%/year
40%/year
1%/year
1978 1980 1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004 2006 2008 2010 2012


100
80
60
40
20
0
Power (% of peak)
2.4 GHz
1.8 GHz
1 GHz
DVS savings (%)
Idle
7
14
21
29
36
43
50
57
64
71
79
86
93
100
Compute load (%)


Cost
Hardware
Software
Year


Time, production and popularity
The cost of integrated circuits
Manufacturing cost
Running cost



- The proportion of hardware implementation is more higher, and the cost of hardware is much more lower.

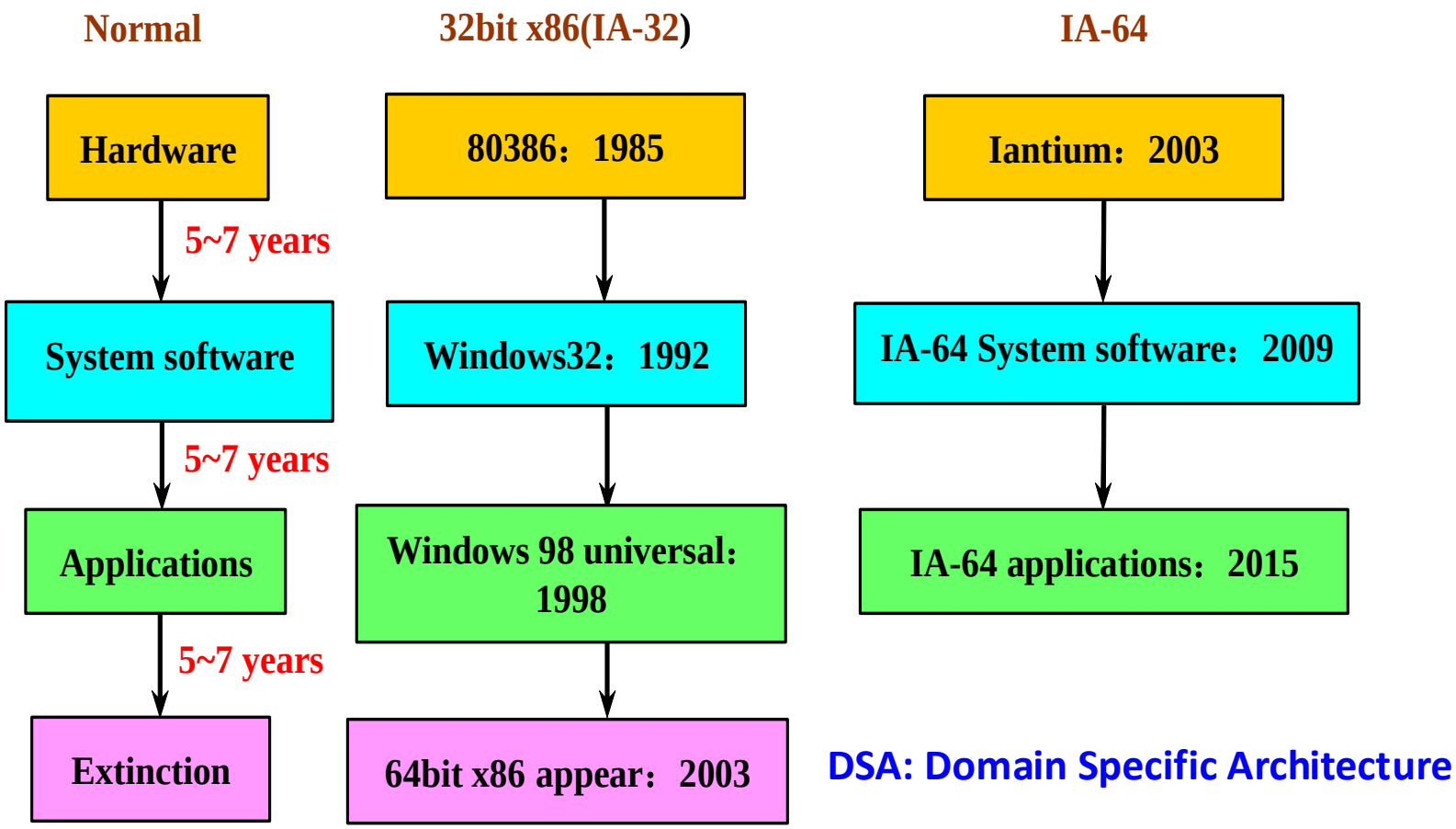
  For a computer system with the same functions, the proportion of software and hardware functions can be changed within a certain range.

The improvement of computer architecture

- Improvement of input / output
- The development of memory organization structure
  - Associative memory and associated processor
  - General register group
  - Cache
- Two directions of instruction set development：
  - CISC
  - RISC
- Parallel processing technology
  - How to develop parallelism in traditional machines?
  - Develop parallel technologies at different levels.
  - For example, micro operation level, instruction level, thread level, process level, task level, etc.

A major turn in computer architecture

- From instruction level parallelism to development thread level parallelism and data level parallelism.
- The computer architecture plays an important role in the development of computer.



The life cycle of architecture: emerge, development, maturity …, extinction

Quantitative approaches

CPU performance formula

Amdahl's Law

Measuring Data Size

- bit - Binary digit
- nibble - four bits
- byte - eight bits
- word - four bytes (32 bits) on many embedded/mobile processors and eight bytes (64 bits) on many desktops and servers.
- kibibyte (KiB) [kilobyte (KB)] - 2^{10} (1,024) bytes
- mebibyte (MiB) [megabyte (MB)] - 2^{20} (1,048,576) bytes
- gibibyte (GiB) [gigabyte (GB)] - 2^{30} (1,073,741,824) bytes
- tebibyte (TiB) [terabyte (TB)] - 2^{40} (1,099,511,627,776) bytes
- pebibyte (PiB) [petabyte (PB)] - 2^{50} (1,125,899,906,842,624) bytes

CPU Performance

In order to determine the effect of a design change on the performance experienced by the user, we can use the following relation:

$$CPU\ Execution\ Time = CPU\ Clock\ Cycles \times Clock\ Period$$

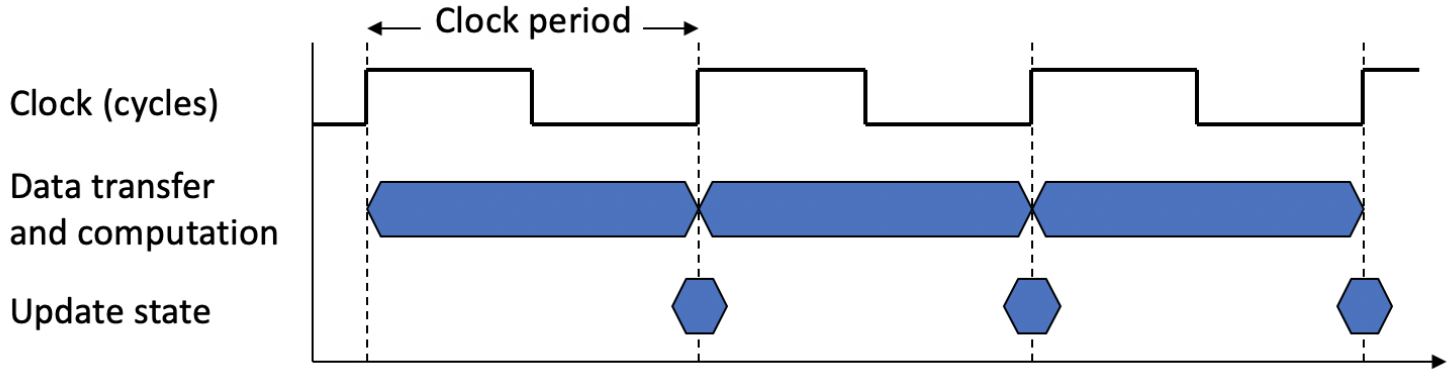
Alternatively,

$$CPU\ Execution\ Time = \frac{CPU\ Clock\ Cycles}{Clock\ Rate}$$

Clearly, we can reduce the execution time of a program by either reducing the number of clock cycles required or the length of each clock cycle.

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

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?

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

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

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

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

Instruction Count and CPI

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

$$CPI = \frac{CPU\ Clock\ Cycles}{Instruction\ Count}$$

$$CPU\ Clock\ Cycles = Instructions\ for\ a\ Program \times Average\ Clock\ Cycles\ Per\ Instruction$$

$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Period$$

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

Components of CPU performance

The basic components of performance and how each is measured.

Component	Units of Measure
CPU Execution Time for a Program	Seconds for the Program
Instruction Count	Instructions Executed for the Program
Clock Cycles per Instruction	Average Number of Clock Cycles per Instruction
Clock Cycle Time (Clock Period)	Seconds per Clock Cycle

Instruction Count, CPI, and Clock Period combine to form the three important components for determining CPU execution time. *Just analyzing one is not enough!* Performance between two machines can be determined by examining non-identical components.

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?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \\ \text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps} \\ \frac{\text{CPU Time}_B}{\text{CPU Time}_A} &= \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2\end{aligned}$$

CPI in More Detail

- If different instruction classes take different numbers of cycles

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

- Weighted average CPI

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

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

Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructio ns}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructio n}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Multiprocessors

- Multicore microprocessors
  - More than one processor per chip

- 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

Amdahl's Law

Amdahl's Law states that 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.

Amdahl's Law depends on two factors:

- The fraction of the time the enhancement can be exploited.
- The improvement gained by the enhancement while it is exploited.

$$Improved\ Execution\ Time = \frac{Affected\ Execution\ Time}{Amount\ of\ Improvement} + Unaffected\ Execution\ Time$$

Make the common case fast!

Amdahl's Law

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

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

- 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!

Amdahl's Law

- The system performance acceleration rate is limited by the percentage of the execution time of the component to the total execution time in the system.
- Amdahl's law defines the *speedup* that can be gained by using a particular feature.

$$\text{Speedup} = \frac{\text{Performance for entire task}_{\text{Using Enhancement}}}{\text{Performance for entire task}_{\text{Without Enhancement}}}$$

$$= \frac{\text{Total Execution Time}_{\text{Without Enhancement}}}{\text{Total Execution Time}_{\text{Using Enhancement}}}$$

Amdahl's Law

- $\text{Fraction}_{\text{enhanced}}$
  - It is always less than or equal to 1.
  - For example, the execution time of a whole program is 60 seconds, 20 seconds for calculation can be improved, then the fraction is 20/60.
- $\text{Speedup}_{\text{enhanced}}$
  - It is always more than 1.
  - For example, the execution time was 5 seconds before, and the execution time is 2 seconds after improvement, then the speedup is 5/2.

Amdahl's Law

- Based on the basic idea that:

Execution time $_{new}$ = Execution time $_{can\ not\ be\ enhance}$ + Execution time $_{enhanced}$

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Amdahl's Law

- Improved ratio: In the system before the improvement, the ratio of the execution time of the improvement part to the total execution time.
  - It is always less than or equal to 1.
  - For example: a program that needs to run for 60 seconds has 20 seconds of calculation that can be accelerated,
    Then the ratio is 20/60.
- Component speedup ratio: The multiple that can be improved after some improvements. It is the ratio of the execution time before the improvement to the execution time after the improvement.
  - Under normal circumstances, the component acceleration ratio is greater than 1.
  - For example: if the system is improved, the execution time of the improved part is 2 seconds, Before the improvement, its execution time was 5 seconds, and the component acceleration ratio was 5/2.

Amdahl's Law

- Example 1.1 Increasing the processing speed of a certain function in the computer system to *20 times* the original, but the processing time of this function only accounts for *40%* of the running time of the entire system. After adopting this method to improve performance, how much can the performance of the entire system improve?

Answer:

- Fraction $_{enhanced}$ = 40%
- Speedup $_{enhanced}$ = 20

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{20}} = 1.613$$

Amdahl's Law

- Example 1.2 After a computer system adopts floating-point arithmetic components, the floating-point arithmetic speed is increased by *20 times*, and the overall performance of a certain program of the system is increased by *5 times*. Try to calculate the proportion of the floating-point operations in this program.

Answer:

- Speedup $_{\text{overall}}$ = 5
- Speedup $_{\text{enhanced}}$ = 20

$$\frac{1}{(1 - \text{Fraction}) + \frac{\text{Fraction}}{20}} = 5$$

Fraction = 84.2%

Amdahl's Law

- A decreasing rule for performance improvement
  If only a part of the computing task is improved, the more the improvement, the more limited the overall performance improvement.

- Important inference: If only a part of the whole task is improved and optimized, the speedup obtained will not exceed:

1/ (1-the ratio can be improved)

Great Architecture Ideas

Great Architecture Ideas

- There are 8 great architectural ideas that have been applied in the design of computers for over half a century now.

- As we cover the material of this course, we should stop to think every now and then which ideas are in play and how they are being applied in the current context.

Great Architecture Ideas

- Design for Moore's law.
  - The number of transistors on a chip doubles every 18-24 months.
  - Architects have to anticipate where technology will be when the design of a system is completed.
- Use abstraction to simplify design.
  - Abstraction is used to represent the design at different levels of representation.
  - Lower-level details can be hidden to provide simpler models at higher levels.
- Make the common case fast.
  - Identify the common case and try to improve it.
  - Most cost efficient method to obtain improvements.
- Improve performance via parallelism.
  - Improve performance by performing operations in parallel.
  - There are many levels of parallelism – instruction-level, process-level, etc.

Great Architecture Ideas

- Improve performance via pipelining.
  - Break tasks into stages so that multiple tasks can be simultaneously performed in different stages.
  - Commonly used to improve instruction throughput.
- Improve performance via prediction.
  - Sometime faster to assume a particular result than waiting until the result is known.
  - Known as speculation and is used to guess results of branches.
- Use a hierarchy of memories.
  - Make the fastest, smallest, and most expensive per bit memory the first level accessed and the slowest, largest, and cheapest per bit memory the last level accessed.
  - Allows most of the accesses to be caught at the first level and be able to retain most of the information at the last level.
- Improve dependability via redundancy.
  - Include redundant components that can both detect and often correct failures.
  - Used at many different levels.

Why learn architecture?

Improving a program's performance is not as simple as reducing its memory usage. Modern programmers need to have an understanding of the issues "below the program":

- The parallel nature of processors.
  - How might you speed up your application by introducing parallelism via threading or multiprocessing?
  - How will the compiler translate and rearrange your own instruction-level code to perform instructions in parallel?
- The hierarchical nature of memory.
  - How can you rearrange your memory access patterns to more efficiently read data?
  - Ever heard of *page (cache) coloring*, *false sharing*, *side-channel attacks*?
- The translation of high-level languages into hardware instructions
  - What decisions are made by the compiler in the process of generating instruction-level statements?

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
- 8 great architectural ideas

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

What is ISA?

Instruction **S**et **A**rchitecture

ISA: Instruction Set Architecture

Programmer-visible instruction set

```
        LOAD    R1,&a       ; R1 <-- contents of 'a'
        LOAD    R2,&a       ; R2 <-- contents of 'a'
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JNE     @L1         ; goto L1 if not equal
        ADD     R1,R2       ; R1 <-- R1 + R2
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JGE     @L2         ; goto L2 if R1 >= R2
        JMP     @END        ; goto END
@L1     ADD     R1, R2      ; R1 <-- R1 + R2
        JMP     @END        ; goto END
@L2     ADD     R1, R2      ; R1 <-- R1 + R2
@END    SUB     R2, R3      ; R2 <-- R2 - R3
```

ISA: Instruction Set Architecture

```c
#include <stdio.h>
#include <stdlib.h>
void read(int *p);
int findmax(int *p);
#define N 10
int m,n;
```

Programmer-visible instruction set

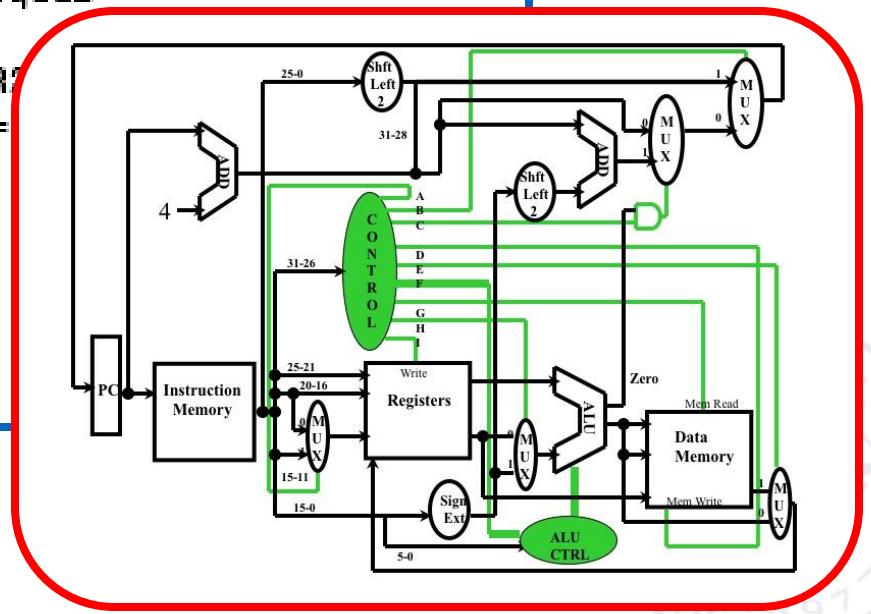
```
        LOAD    R1,&a       ; R1 <-- contents of 'a'
        LOAD    R2,&a       ; R2 <-- contents of 'a'
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JNE     @L1         ; goto L1 if not equal
        ADD     R1,R2       ; R1 <-- R1 + R2
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JGE     @L2         ; goto L2 if R1 >= R2
        JMP     @END        ; goto END
@L1     ADD     R1, R2      ; R1 <-- R1 + R2
        JMP     @END        ; goto END
@L2     ADD     R1, R2      ; R1 <-- R1 + R2
@END    SUB     R2, R3      ; R2 <-- R2 - R3
```

ISA: Instruction Set Architecture

```c
#include <stdio.h>
#include <stdlib.h>
void read(int *p);
int findmax(int *p);
#define N 10
int m,n;
```

Programmer-visible instruction set

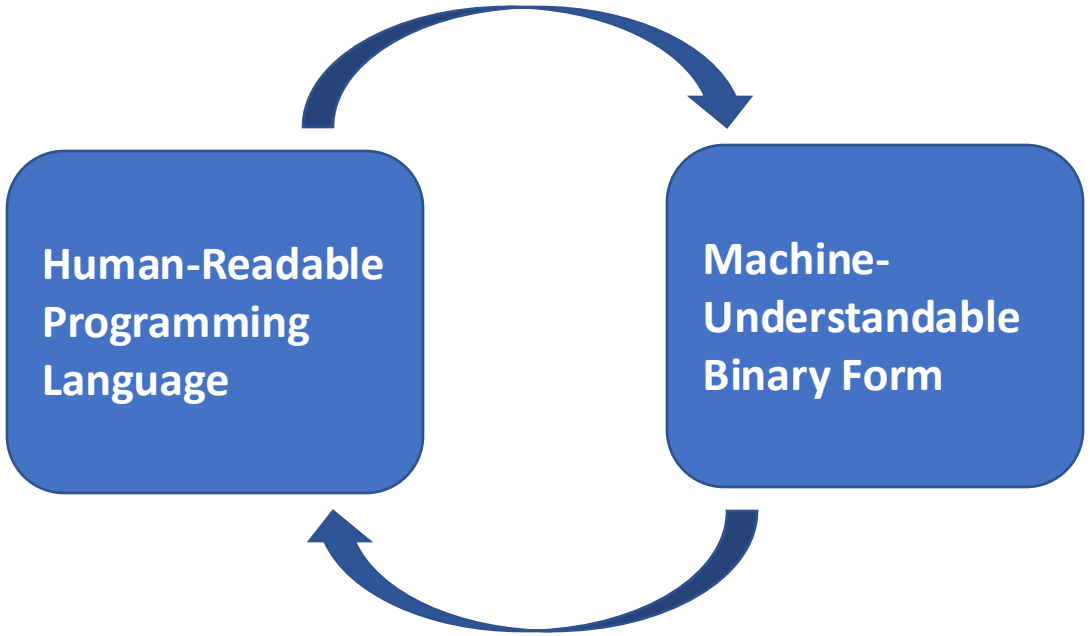
```
        LOAD    R1,&a       ; R1 <-- contents of 'a'
        LOAD    R2,&a       ; R2 <-- contents of 'a'
        TEST    R1,R2       ; compare R1 and R2, set condition code
        JNE     @L1         ; goto L1 if not equal
        ADD     R1,R2       ; R1 <-- R1 + R2
        TEST    R1,R2       ; compare R1 and R2
        JGE     @L2         ; goto L2 if R1 >=
        JMP     @END        ; goto END
@L1     ADD     R1, R2      ; R1 <-- R1 + R2
        JMP     @END        ; goto END
@L2     ADD     R1, R2      ; R1 <-- R1 + R2
@END    SUB     R2, R3      ; R2 <-- R2 - R3
```



High-level Language

⇩

Assembly language



Machine Language

Example of Translating a C Program

High-Level Language Program

```
swap(int v[], int k){
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

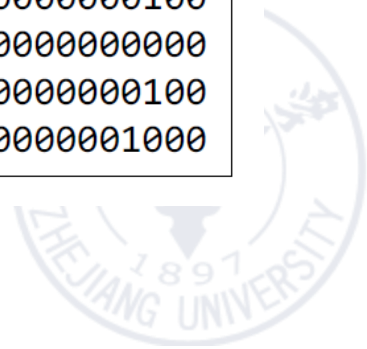
Assembly Language Program

```
swap:
        multi   $2, $5,  4
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

Assembler

Binary Machine Language Program

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```



What is instruction set

- A single human-readable high-level language instruction is generally translated into multiple assembly instructions.
- A single assembly instruction is a symbolic representation of a single machine language instruction.
  - Some assembler supports high-level assembly (HLA) code
- A single machine language instruction is a set of bits representing a basic operation that can be performed by the machine.
- The instruction set is the set of possible instructions for a given machine.

Instruction Set Design Issues

- Instruction set design issues include:
  - Where are operands stored?
    - registers, memory, stack, accumulator
  - How many explicit operands are there?
    - 0, 1, 2, or 3 (Classification of ISAs)
  - How is the operand location specified?
    - register, immediate, indirect, . . . (Addressing Modes)
  - What type & size of operands are supported?
    - byte, int, float, double, string, vector. . . (Data Representation)
  - What operations are supported?
    - add, sub, mul, move, compare . . . (Types of Instructions)

Instruction Set Design Basic Principles

- Compatibility
- Versatility
- High efficiency
- Security

What affect instruction sets design?

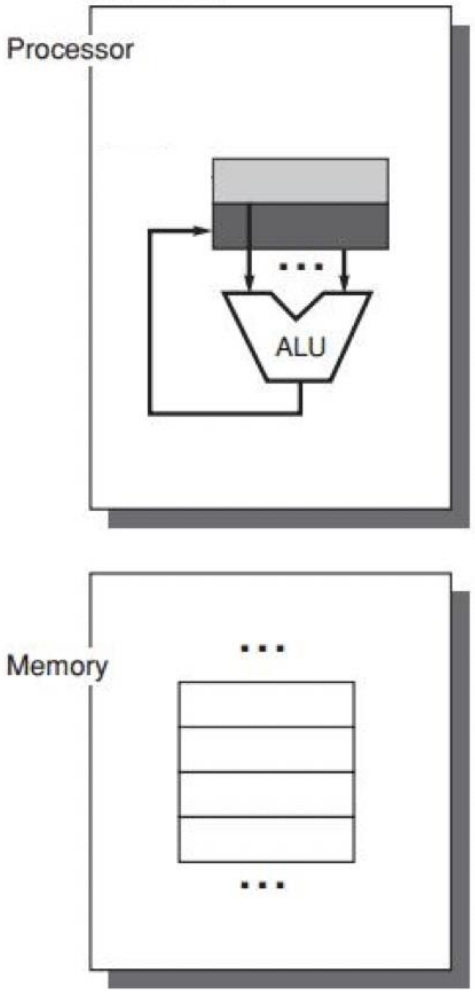
- Technology
- Computer architecture
- Operating system
- Compile
- Application

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003

ISA Classification Basis

- The types of internal storage:
  - Stack
  - Accumulator
  - Register

In processor, stores data fetched from memory or cache



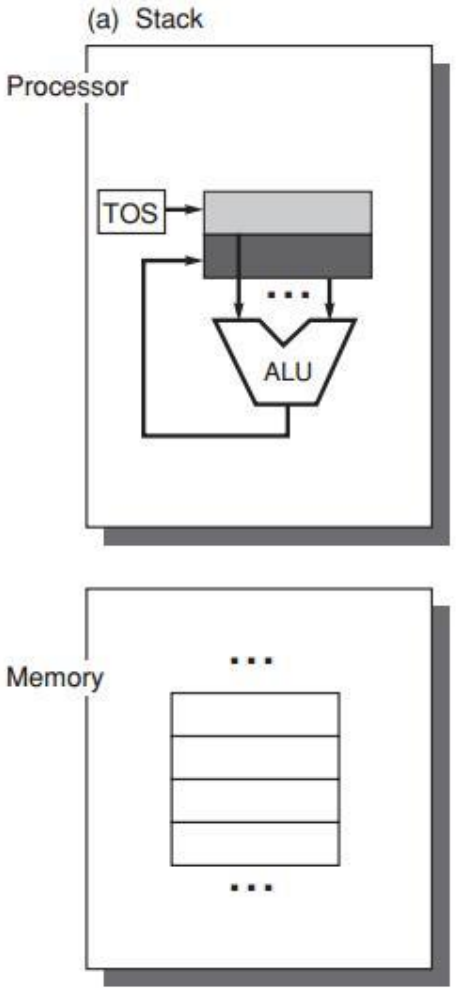
ISA Classes

- Stack architecture
- Accumulator architecture
- General-purpose register architecture (GPR)

ISA Classes: Stack Architecture

- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

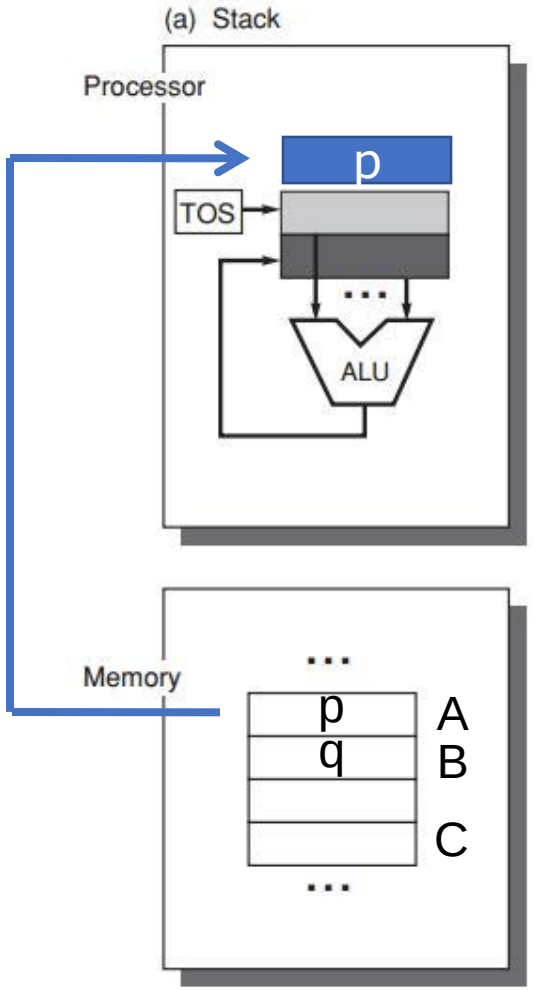
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

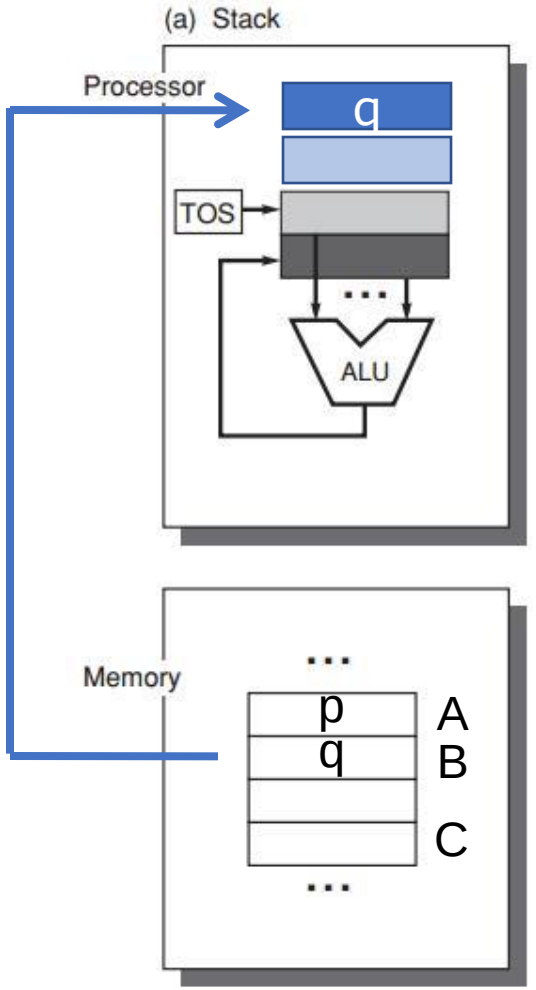
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

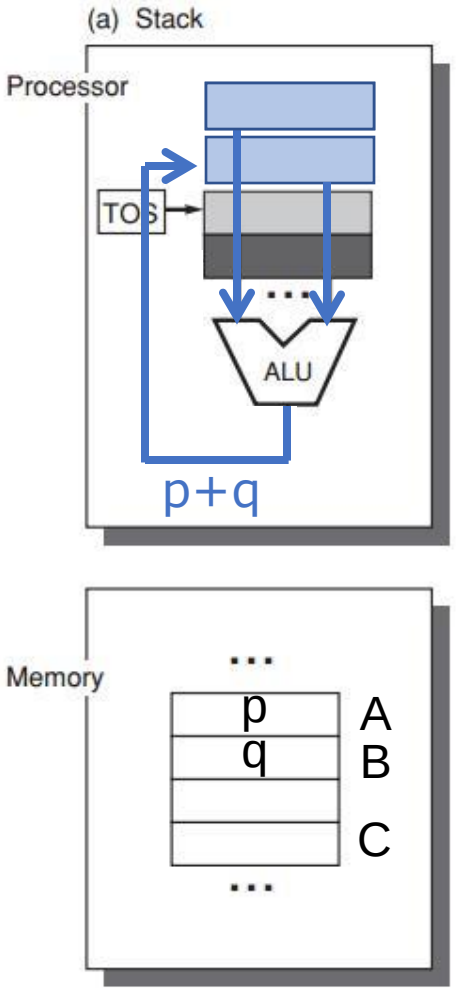
- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)

- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



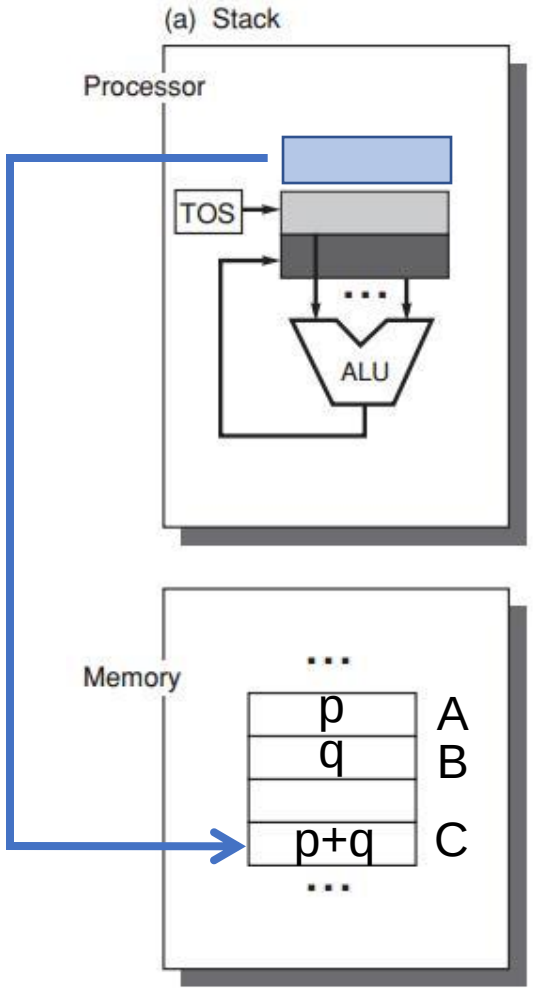
ISA Classes: Stack Architecture

- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)
- First operand removed from
  second op replaced by the result
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

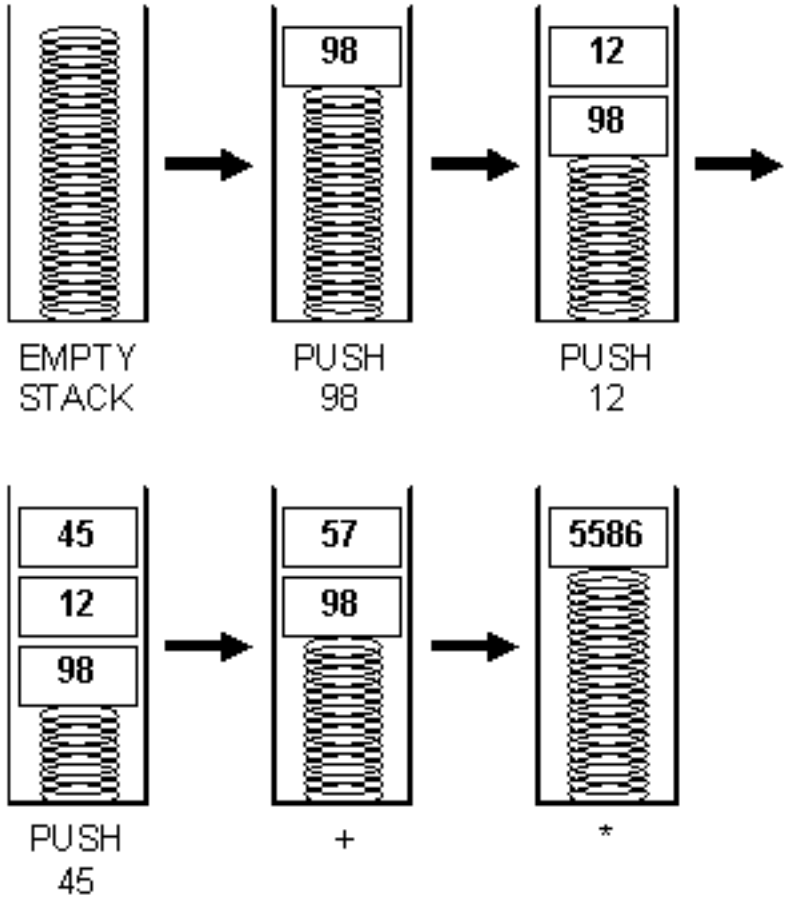
- Implicit Operands
  on the **T**op **O**f the **S**tack (TOS)
- First operand removed from
  second op replaced by the result
- C = A + B (memory locations)
  Push A
  Push B
  Add
  Pop C



ISA Classes: Stack Architecture

- Example:

98 * (12 + 45)



ISA Classes: Accumulator Architecture

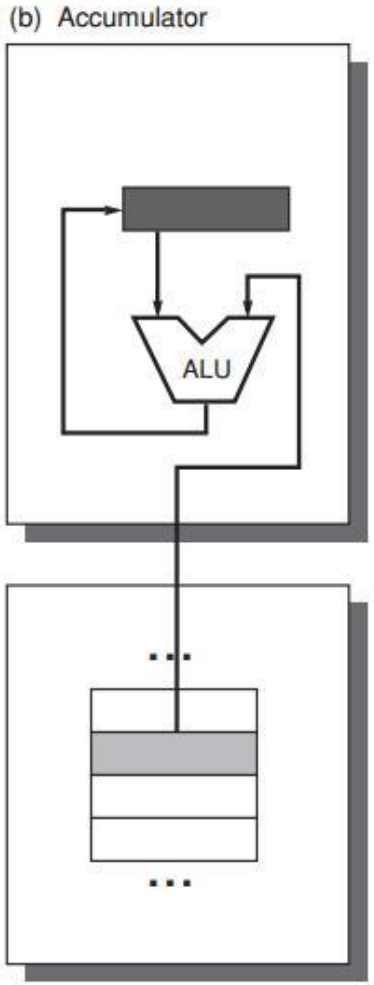
- One implicit operand: the accumulator

  one explicit operand: mem location
- C = A + B

  Load A

  Add B

  Store C
- Accumulator is both an implicit input operand and a result



ISA Classes: Accumulator Architecture

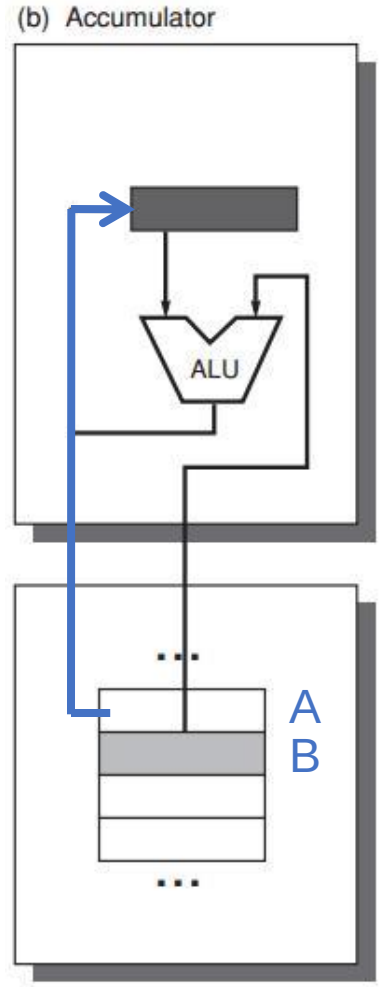
- One implicit operand: the accumulator

  one explicit operand: mem location
- C = A + B

  Load A

  Add B

  Store C
- Accumulator is both an implicit input operand and a result



ISA Classes: Accumulator Architecture

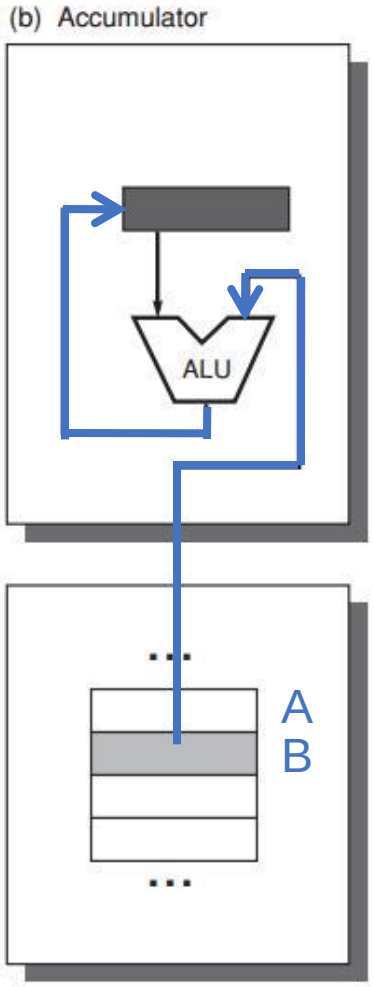
- One implicit operand: the accumulator

  one explicit operand: mem location
- C = A + B

  Load A

  Add B

  Store C
- Accumulator is both an implicit input operand and a result



ISA Classes: Accumulator Architecture

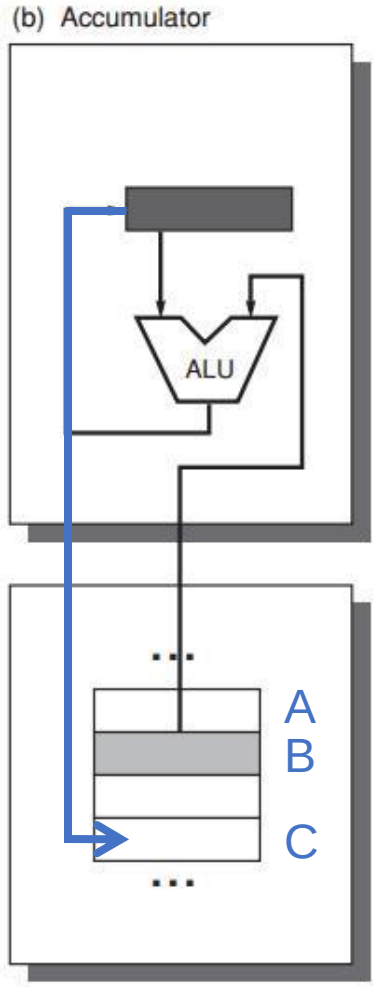
- One implicit operand: the accumulator

  one explicit operand: mem location
- C = A + B

  Load A

  Add B

  Store C
- Accumulator is both an implicit input operand and a result



ISA Classes: General-Purpose Register Arch

- Only explicit operands

  registers

  memory locations

- Operand access:

  direct memory access

  loaded into temporary storage first

ISA Classes: General-Purpose Register Arch

Two Classes:

- Register-memory architecture

  any instruction can access memory

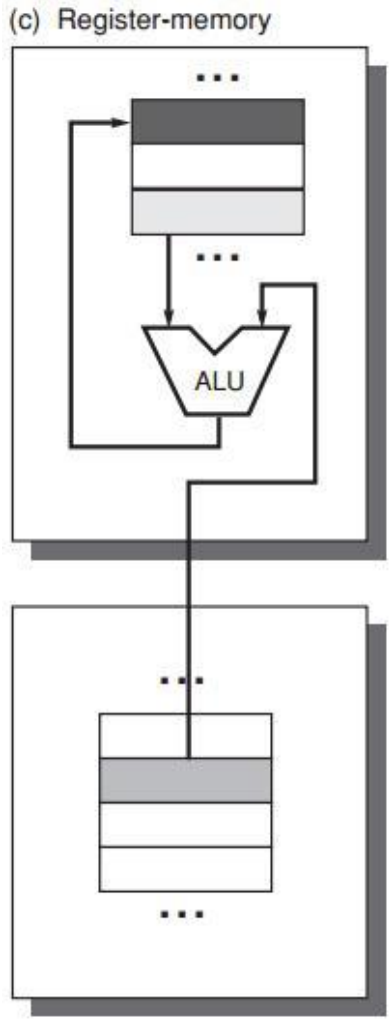
- Load-store architecture

  only load and store instructions can access memory

GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

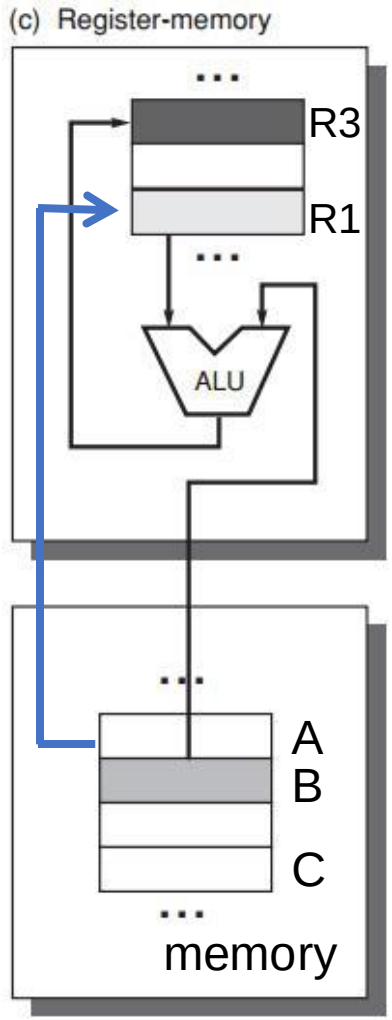
- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

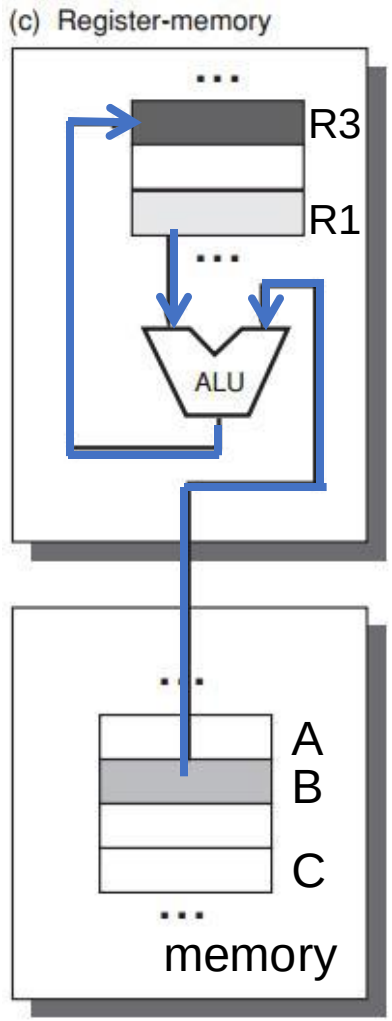
- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

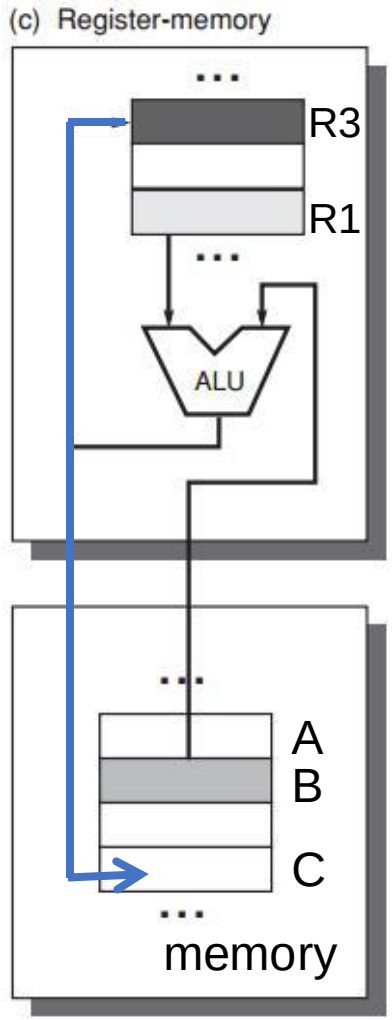
- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



GPR: Register-Memory Arch

- Register-memory architecture
  (any instruction can access memory)

- C = A + B
  Load R1, A
  Add R3, R1, B
  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

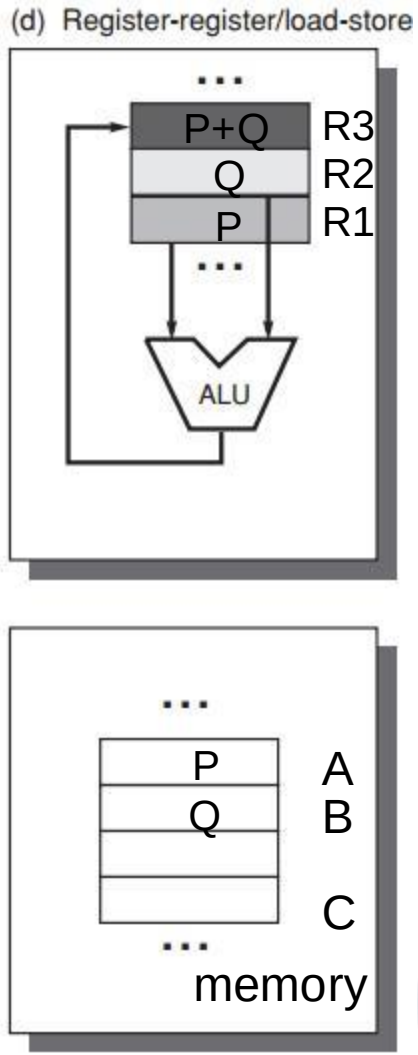
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

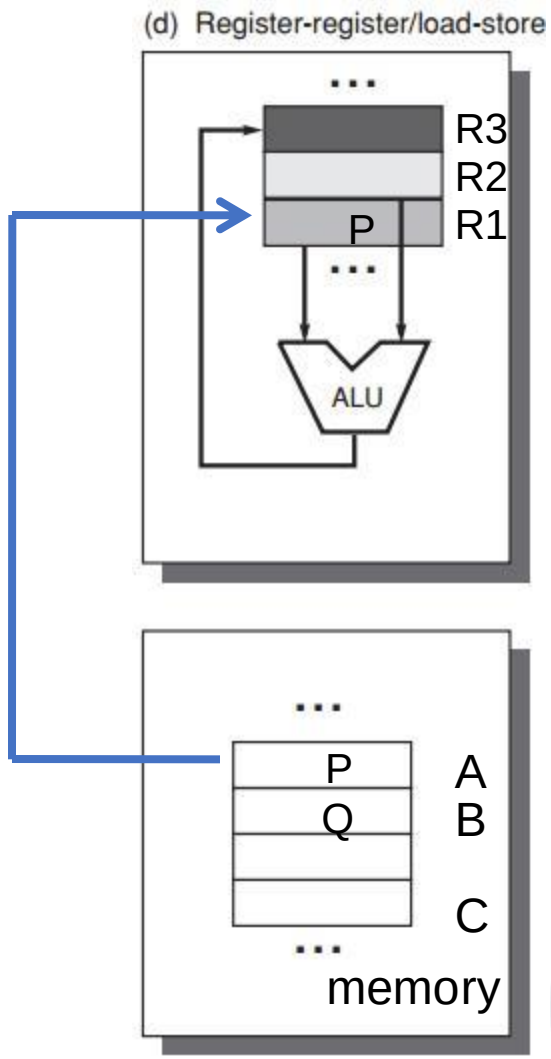
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

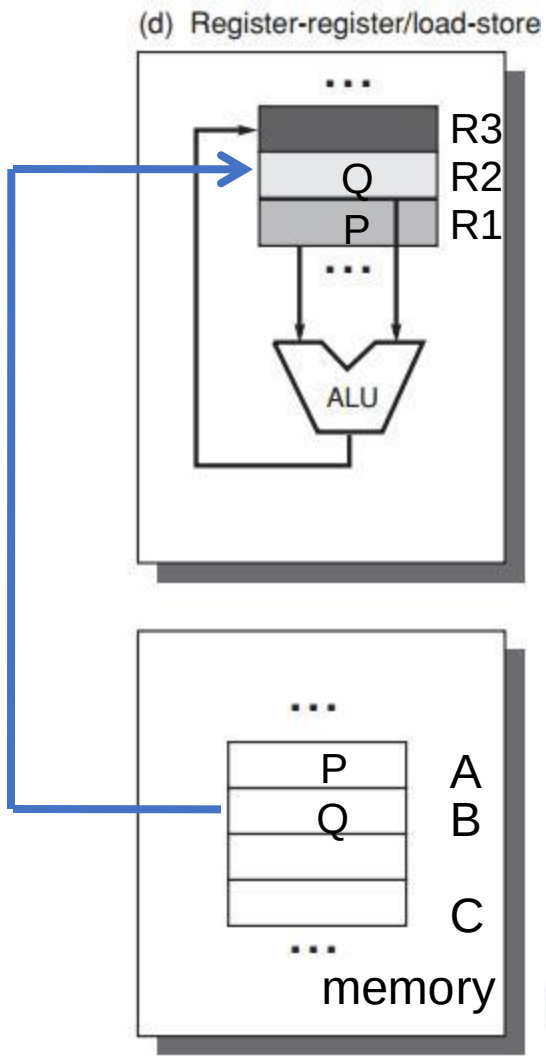
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

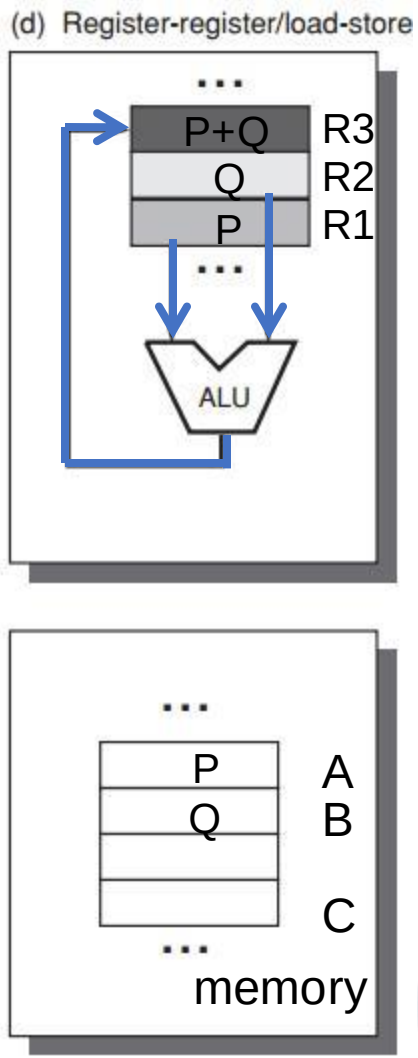
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR: Load-Store Architecture

- Load-Store Architecture

  only load and store instructions

  can access memory

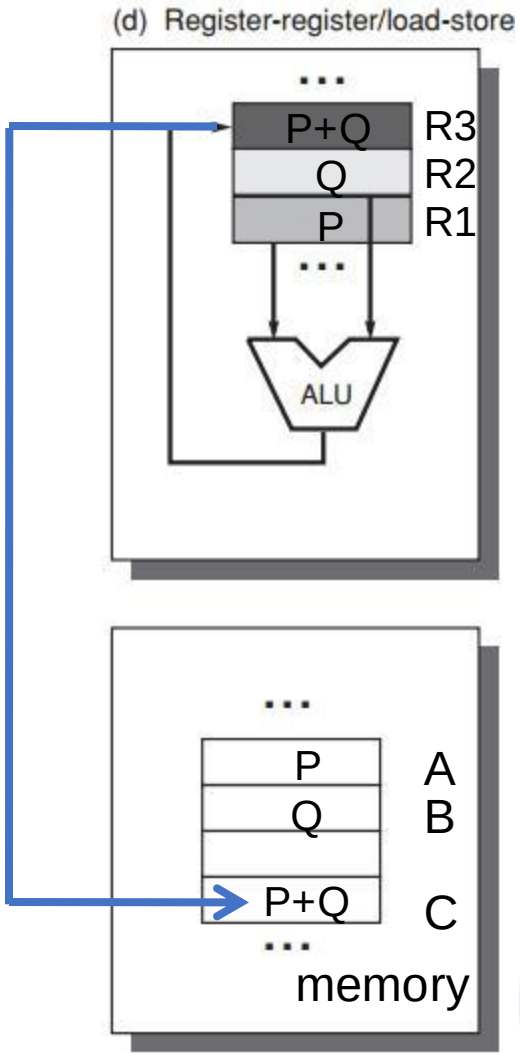
- C = A + B

  Load R1, A

  Load R2, B

  Add R3, R1, R2

  Store R3, C



GPR Classification

- ALU instruction has 2 or 3 operands?

  2 operands = 1 result & source op + 1 source op

  3 operands = 1 result op + 2 source op

- ALU instruction has 0, 1, 2, or 3 operands of memory address?

GPR Classification

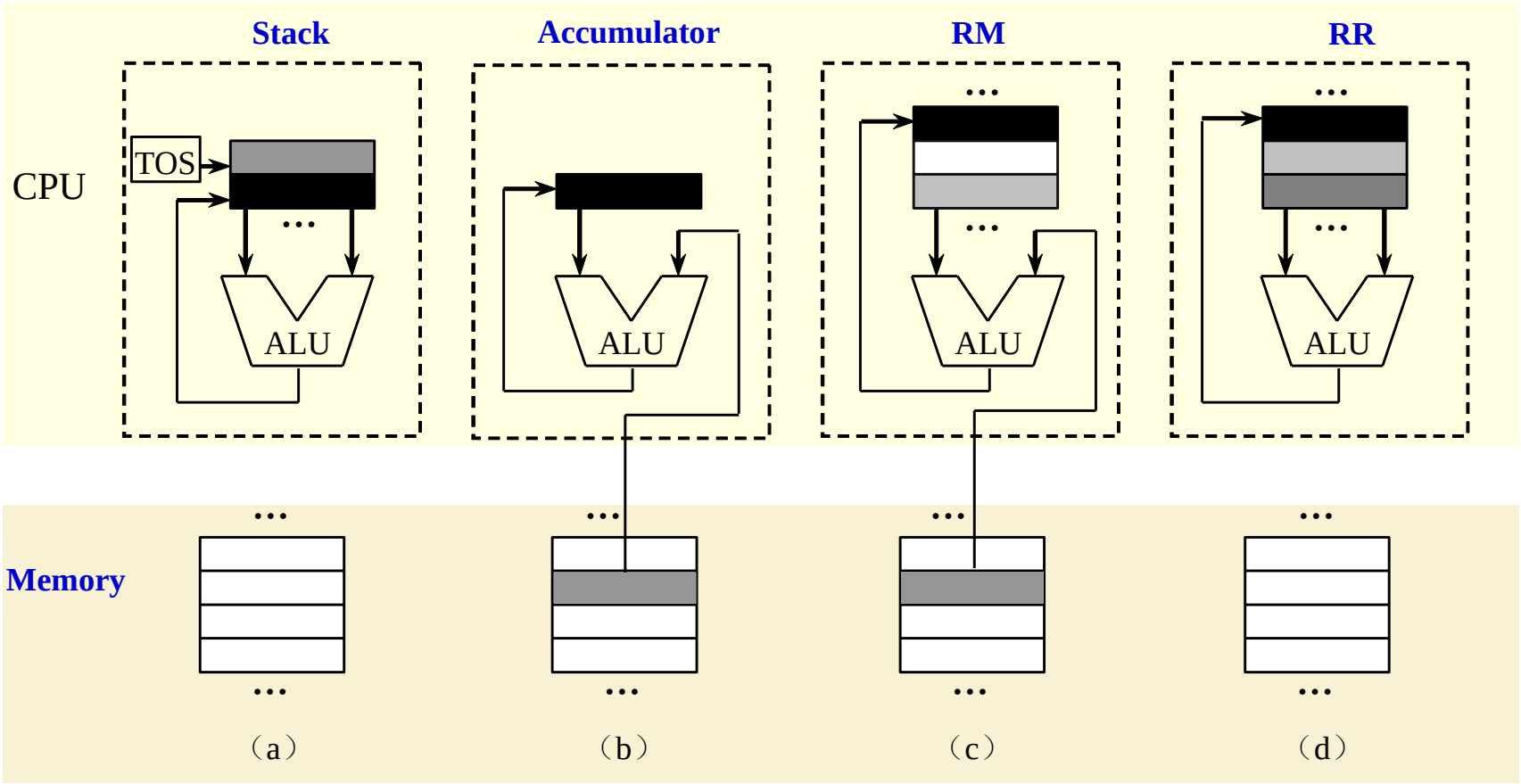
- Three major classes

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

GPR Classification

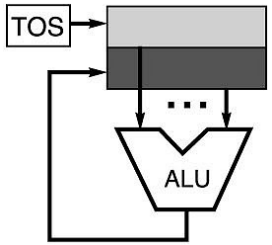
Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

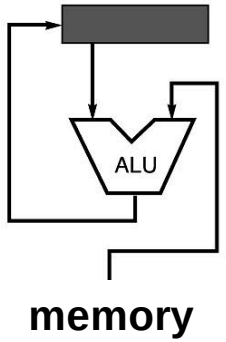
GPR Classification



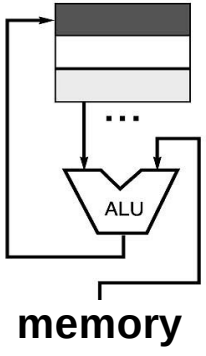
Code Sequence C = A + B

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

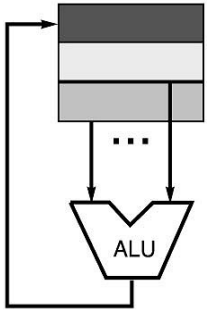




acc = acc + mem[C]



R1 = R1 + mem[C]



R3 = R1 + R2

Practice in Class

D = A * B - (A + C * B)

Stack Architectures

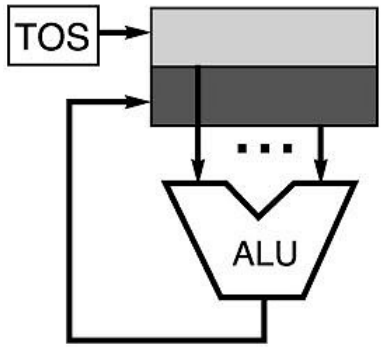
- Instruction set:

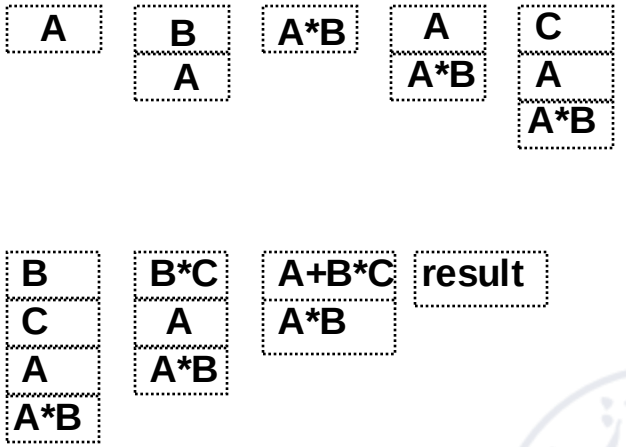
  add, sub, mult, div, . . .

  push A, pop A

- Example: D = A * B - (A + C * B)

  1. push A
  2. push B
  3. mul
  4. push A
  5. push C
  6. push B
  7. mul
  8. add
  9. sub
  10. pop D



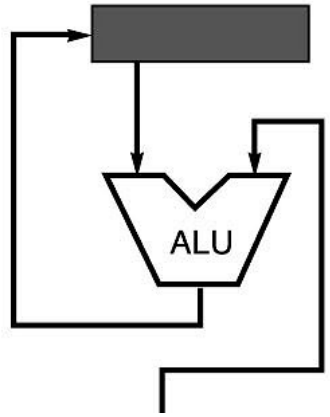


Accumulator Architectures

- Instruction set:

  add A, sub A, mult A, div A, . .

  load A, store A



acc = acc +,-,*,/ mem[A]

- Example: D = A * B - (A + C * B)

  1. load B
  2. mul C
  3. add A
  4. store D
  5. load A
  6. mul B
  7. sub D
  8. store D

B	B*C	A+B*C	A+B*C	A	A*B	result

Memory-Memory Architectures

- Instruction set:

(3 operands)	add A, B, C	sub A, B, C	mul A, B, C
(2 operands)	add A, B	sub A, B	mul A, B

- Example: D = A * B - (A + C * B)

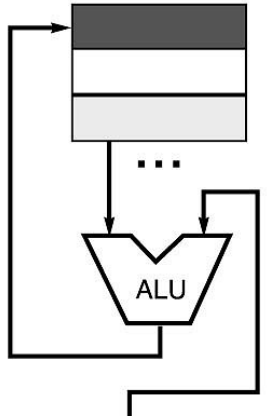
3 operands	2 operands	
mul D, A, B	mov D, A	mul E, B
mul E, C, B	mul D, B	add E, A
add E, A, E	mov E, C	sub E, D
sub E, D, E		

Register-Memory Architectures

- Instruction set:

  add R1, A sub R1, A mul R1, B

  load R1, A store R1,



R1 = R1 +,-,*,/ mem[B]

- Example: D = A * B - (A + C * B)

```
1. load R1, A
2. mul R1, B     /* A*B */
3. store R1, D
4. load R2, C
5. mul R2, B     /* C*B */
6. mul R2, B     /* C*B */
7. add R2, A     /* A + CB */
8. sub R2, D     /* AB - (A + C*B) */
9. store R2, D
```

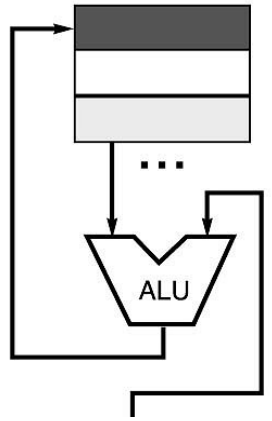
Load-Store Architectures

- Instruction set:

  add R1, R2, R3 sub R1, R2, R3

  mul R1, R2, R3

  load R1, &A store R1, &A move R1, R2



R1 = R1 +,-,*,/ mem[B]

- Example: D = A * B - (A + C * B)

1. load R1, &A	5. add R8, R7, R1 /* A+C*B */
2. load R2, &B	6. mul R9, R1, R2 /* A*B */
3. load R3, &C	7. sub R10, R9, R8 /* A*B - (A+C*B) */
4. mul R7, R3, R2 /* C*B */	8. store R10, D

Interpret Memory Address

Why do almost all new architectures use GPRs?

- Registers are much faster than memory (even cache)
  - Temporal Locality
  - Register values are available immediately
- Registers are convenient for variable storage
  - Compiler assigns some variables just to registers
  - More compact code since small fields specify registers (compared to memory addresses)