CS/EE 320 Computer Organization and Assembly Language Spring 2024

Shahid Masud

Lecture 9

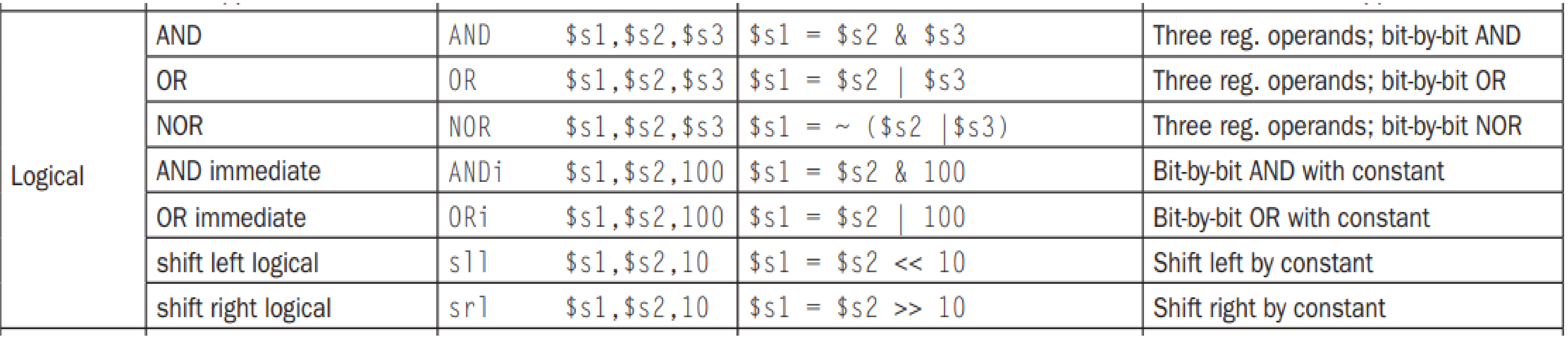
Topics

- Logic and Arithmetic Instructions in Assembly Language
- Binary Numbers, Logic Design and Boolean Algebra
- Basic digital logic functions Invert, AND, OR, NAND, NOR
- Half-Adder (without Carry-In)
- 1-bit Full Adder (with Carry-In)
- Making 2’s Complement Subtractor from Full-Adder
- Design of 1-bit ALU containing AND gate, OR gate, Full Adder
- Quiz 2

MIPS Arithmetic Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow detected
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three operands; overflow detected
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow detected
	add unsigned	`addu $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands; overflow undetected
	subtract unsigned	`subu $s1,$s2,$s3`	`$s1 = $s2 – $s3`	Three operands; overflow undetected
	add immediate unsigned	`addiu $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant; overflow undetected
	move from coprocessor register	`mfc0 $s1,$epc`	`$s1 = $epc`	Copy Exception PC + special regs
	multiply	`mult $s2,$s3`	Hi, Lo = `$s2 × $s3`	64-bit signed product in Hi, Lo
	multiply unsigned	`multu $s2,$s3`	Hi, Lo = `$s2 × $s3`	64-bit unsigned product in Hi, Lo
	divide	`div $s2,$s3`	Lo = `$s2 / $s3`, Hi = `$s2` mod `$s3`	Lo = quotient, Hi = remainder
	divide unsigned	`divu $s2,$s3`	Lo = `$s2 / $s3`, Hi = `$s2` mod `$s3`	Unsigned quotient and remainder
	move from Hi	`mfhi $s1`	`$s1` = Hi	Used to get copy of Hi
	move from Lo	`mflo $s1`	`$s1` = Lo	Used to get copy of Lo

MIPS Logical Instructions



Logical	AND	`AND $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	OR	`OR $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	NOR	`NOR $s1,$s2,$s3`	`$s1 = ~ ($s2 |$s3)`	Three reg. operands; bit-by-bit NOR
	AND immediate	`ANDi $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND with constant
	OR immediate	`ORi $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant

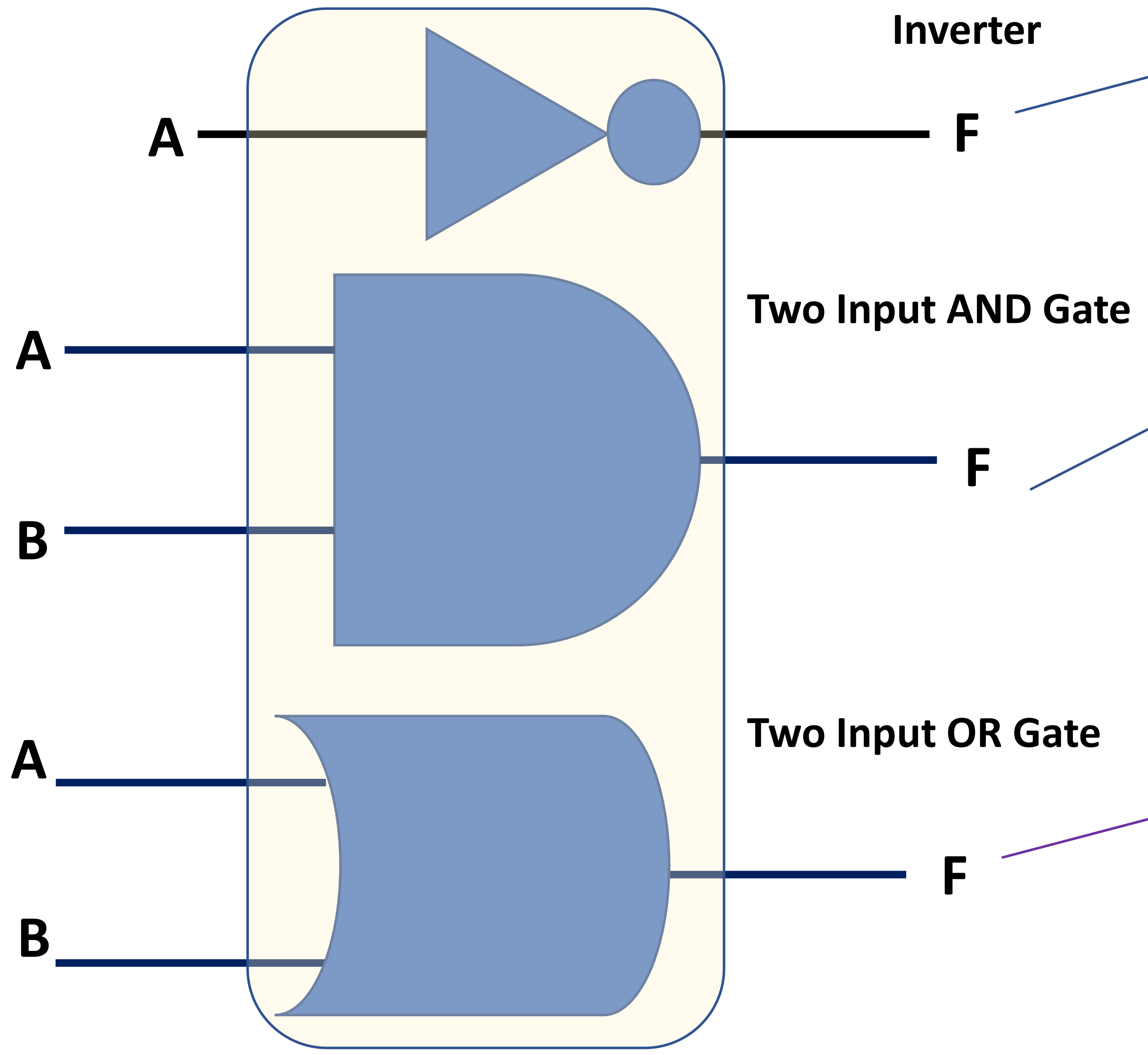
Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers (later)
  - Representation and operations

Basic Logic Operations in CPU

- Functions Through Logic Gates
  - NOT
  - AND
  - OR
  - NOR
  - NAND
  - XOR

Gates (Primary Gates)



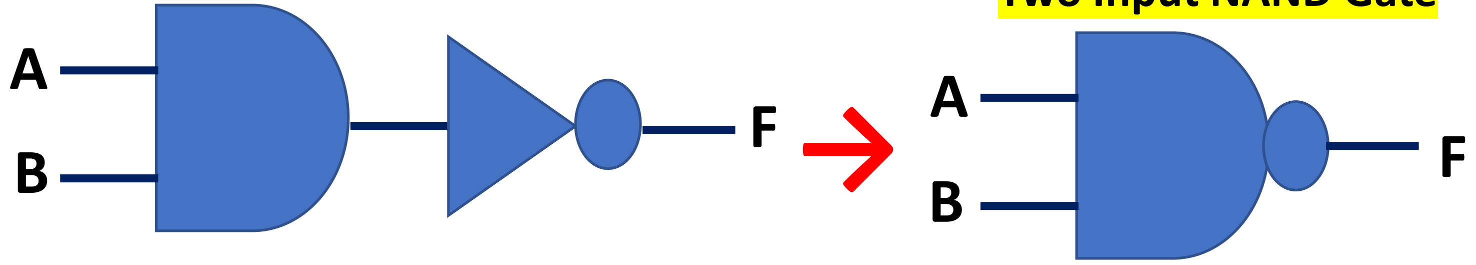
A	F
0	1
1	0

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

Gates (Compound Gates)

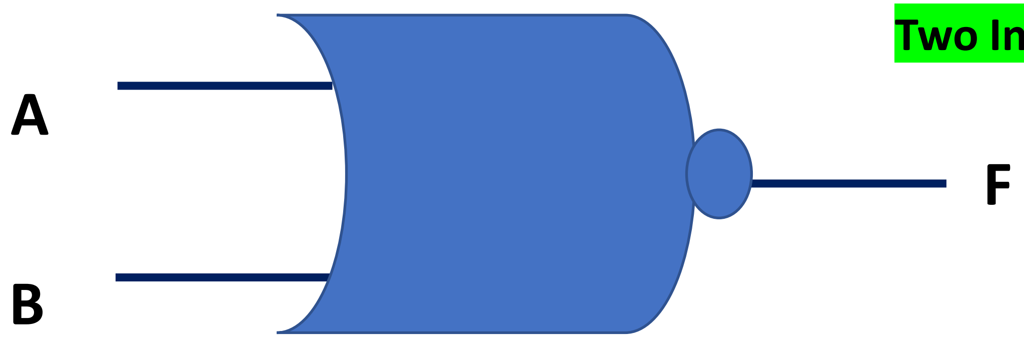
Two Input NAND Gate



AND Gate followed by Inverter

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Two Input NOR Gate

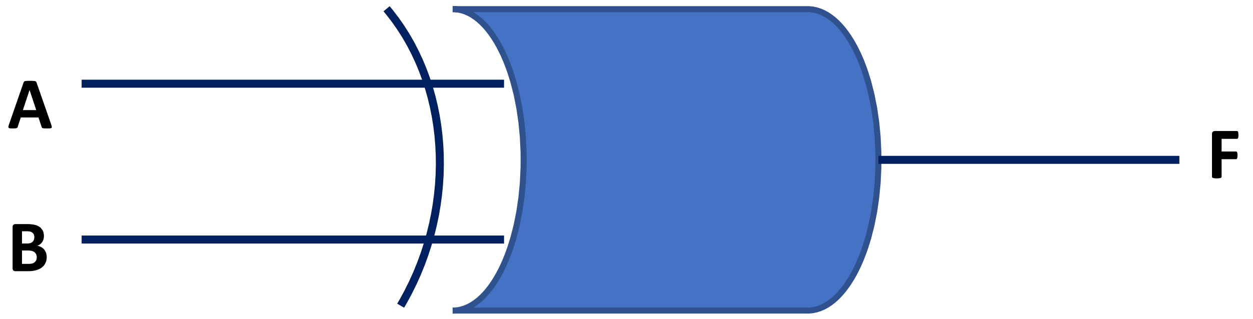


OR Gate followed by Inverter

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Complex Gates – the XOR and XNOR

Two Input Exclusive OR Gate



$$F = \overline{A}B + A\overline{B}$$
$$F = A \oplus B$$

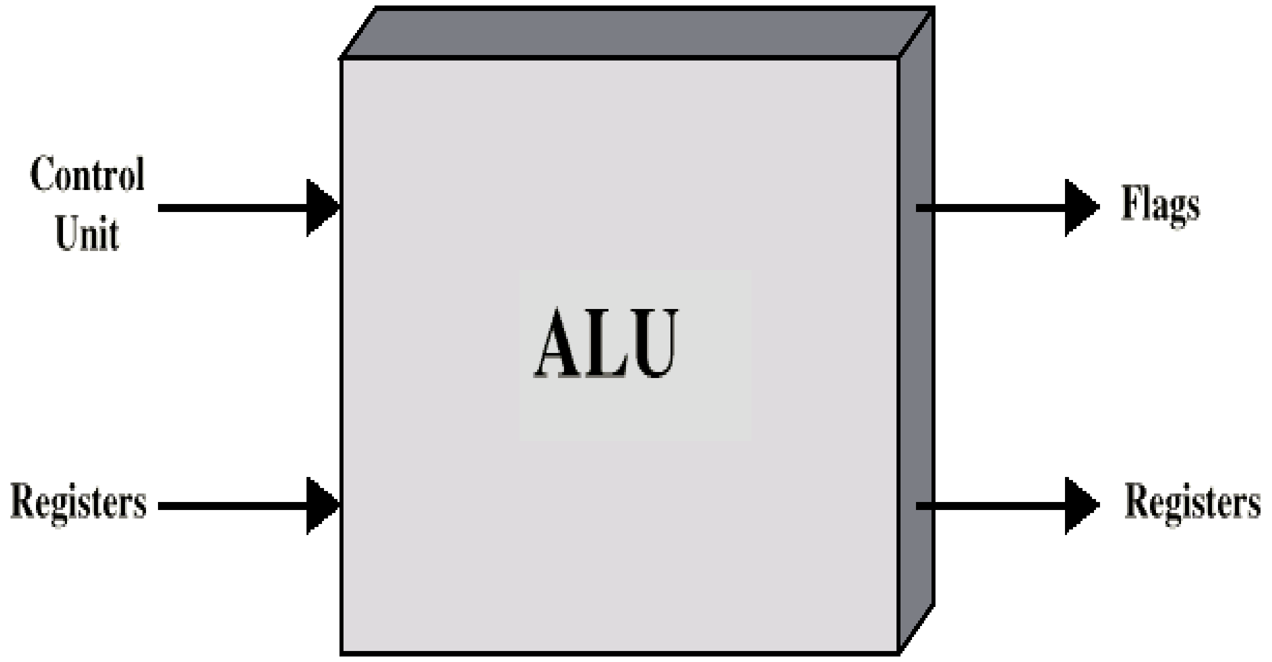
A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Arithmetic & Logic Unit

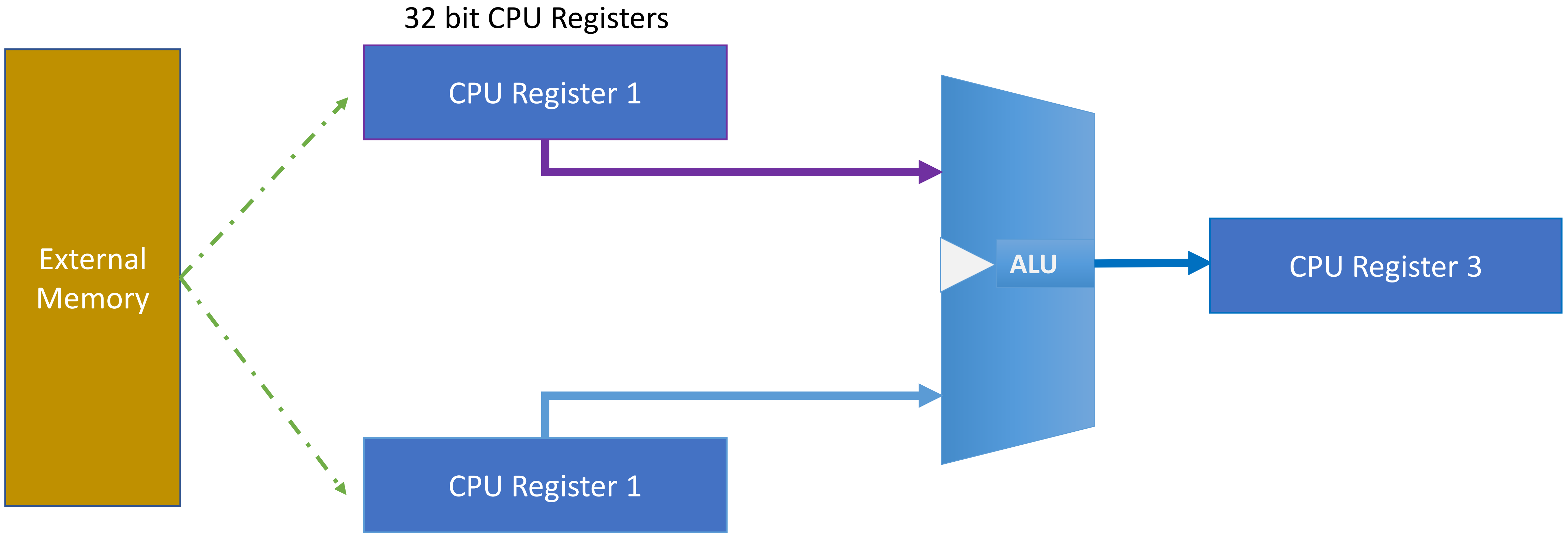
- Does the calculations
- Everything else in the computer is there to service this unit
- Handles integers
- May handle floating point (real) numbers
- May be separate FPU (maths co-processor)
- May be on chip separate FPU (Eg. 486DX)

ALU – Arithmetic and Logic Unit

- Does the calculations
- Everything else in the computer is there to service this unit
- Handles integers
- May handle floating point (real) numbers
- May be separate FPU (maths co-processor)
- May be on chip separate FPU (Eg. 486DX)



ALU in Load/Store Architecture



Number Systems

- ALU does calculations with binary numbers

- Decimal number system
  - Uses 10 digits (0,1,2,3,4,5,6,7,8,9)
  - In decimal system, a number 84, e.g., means
    84 = (8x10) + 4
  - 4728 = (4x1000)+(7x100)+(2x10)+8
  - Base or radix of 10: each digit in the number is multiplied by 10 raised to a power corresponding to that digit's position
  - E.g. 83 = $(8 \times 10^1) + (3 \times 10^0)$
  - 4728 = $(4 \times 10^3)+(7 \times 10^2)+(2 \times 10^1)+(8 \times 10^0)$

Binary Number System

- Uses only two digits, 0 and 1
- It is base or radix of 2
- Each digit has a **value** depending on its **position**:
  - $10_2 = (1x2^1)+(0x2^0) = 2_{10}$
  - $11_2 = (1x2^1)+(1x2^0) = 3_{10}$
  - $100_2 = (1x2^2)+ (0x2^1)+(0x2^0) = 4_{10}$
  - $1001.101_2 = (1x2^3)+(0x2^2)+ (0x2^1)+(1x2^0)$
    $+(1x2^{-1})+(0x2^{-2})+(1x2^{-3}) = 9.625_{10}$

Decimal to Binary conversion

- Integer and fractional parts are handled separately,
  - Integer part is handled by repeating division by 2
  - Factional part is handled by repeating multiplication by 2
- E.g. convert decimal 11.81 to binary
  - Integer part 11
  - Factional part .81

Decimal number to Binary

- Fractional values, e.g.
  - 472.83 = $(4\times10^{2})+(7\times10^{1})+(2\times10^{0})+(8\times10^{-1})+(3\times10^{-2})$

  - In general, for the decimal representation of
    $X = \{\ldots x_2 x_1 x_0 . x$

    Whole Number Part

    Fractional Part

    $X = \sum_i x_i 10^i$

Decimal to Binary conversion example

- e.g. 11.81 to 1011.11001 (approx)
  - 11/2 = 5 remainder 1
  - 5/2 = 2 remainder 1
  - 2/2 = 1 remainder 0
  - 1/2 = 0 remainder 1
  - Binary number 1011
  - .81x2 = 1.62 integral part 1
  - .62x2 = 1.24 integral part 1
  - .24x2 = 0.48 integral part 0
  - .48x2 = 0.96 integral part 0
  - .96x2 = 1.92 integral part 1
  - Binary number .11001 (approximate)

Hexadecimal Notation

- Compact representation of bus information
- Use 16 digits, (0,1,3,…9,A,B,C,D,E,F)
- $1A_{16} = (1_{16} \times 16^1)+(A_{16} \times 16^o)$
  $= (1_{10} \times 16^1)+(10_{10} \times 16^0)=26_{10}$
- Convert group of four binary digits to/from one hexadecimal digit,
  - 0000=0; 0001=1; 0010=2; 0011=3; 0100=4; 0101=5; 0110=6; 0111=7; 1000=8; 1001=9; 1010=A; 1011=B; 1100=C; 1101=D; 1110=E; 1111=F;
- e.g.
  - 1101 1110 0001. 1110 1101 = DE1.DE

Integer Representation +/- numbers

- Only have 0 & 1 to represent everything
- Positive numbers stored in binary
  - e.g. 41=00101001
- No minus sign
- No period
- **How to represent negative number?**
  - Sign-Magnitude
  - Two’s compliment

Sign-Magnitude Representation

- Add one extra bit on MSB to represent sign
- Left most bit is sign bit
- 0 means positive
- 1 means negative
- +18 = **0**0010010
- -18 = **1**0010010
- Problems
  - Need to consider both sign and magnitude in arithmetic
  - Two representations of zero (+0 and -0)

Two’s Compliment Representation

- +3 = 00000011
- +2 = 00000010
- +1 = 00000001
- +0 = 00000000
- -1 = 11111111
- -2 = 11111110
- -3 = 11111101

2’s Compliment Number System

- One representation of zero
- Arithmetic works easily, use same hardware as positive numbers
- Negating is fairly easy (2’s compliment operation)
  - 3 = 00000011
  - Boolean **complement** gives 11111100
  - Add +1 to LSB 11111101

Range of 2’s Compliment Numbers

- 8 bit 2’s compliment
  - +127 = 01111111 = 2^7 -1
  - -128 = 10000000 = -2^7
- 16 bit 2’s compliment
  - +32767 = 011111111 11111111 = 2^{15} - 1
  - -32768 = 100000000 00000000 = -2^{15}

Different word lengths in 2’s Compliment

- Positive number pack with leading zeros
- +18 =   00010010 (in 8 bits)
- +18 = 00000000 00010010 (in 16 bits)
- Negative numbers pack with leading ones
- -18 =   10010010 (in 8 bits)
- -18 = 11111111 10010010 (in 16 bits, notice sign bit replication)
- i.e. pack higher bits with MSB (sign bit)

Negation Special Case 1

- 0 = 00000000
- Bitwise not 11111111
- Add 1 to LSB +1
- Result $\underline{1}$ 00000000
- **Overflow** is ignored, so:
- - 0 = 0 OK!

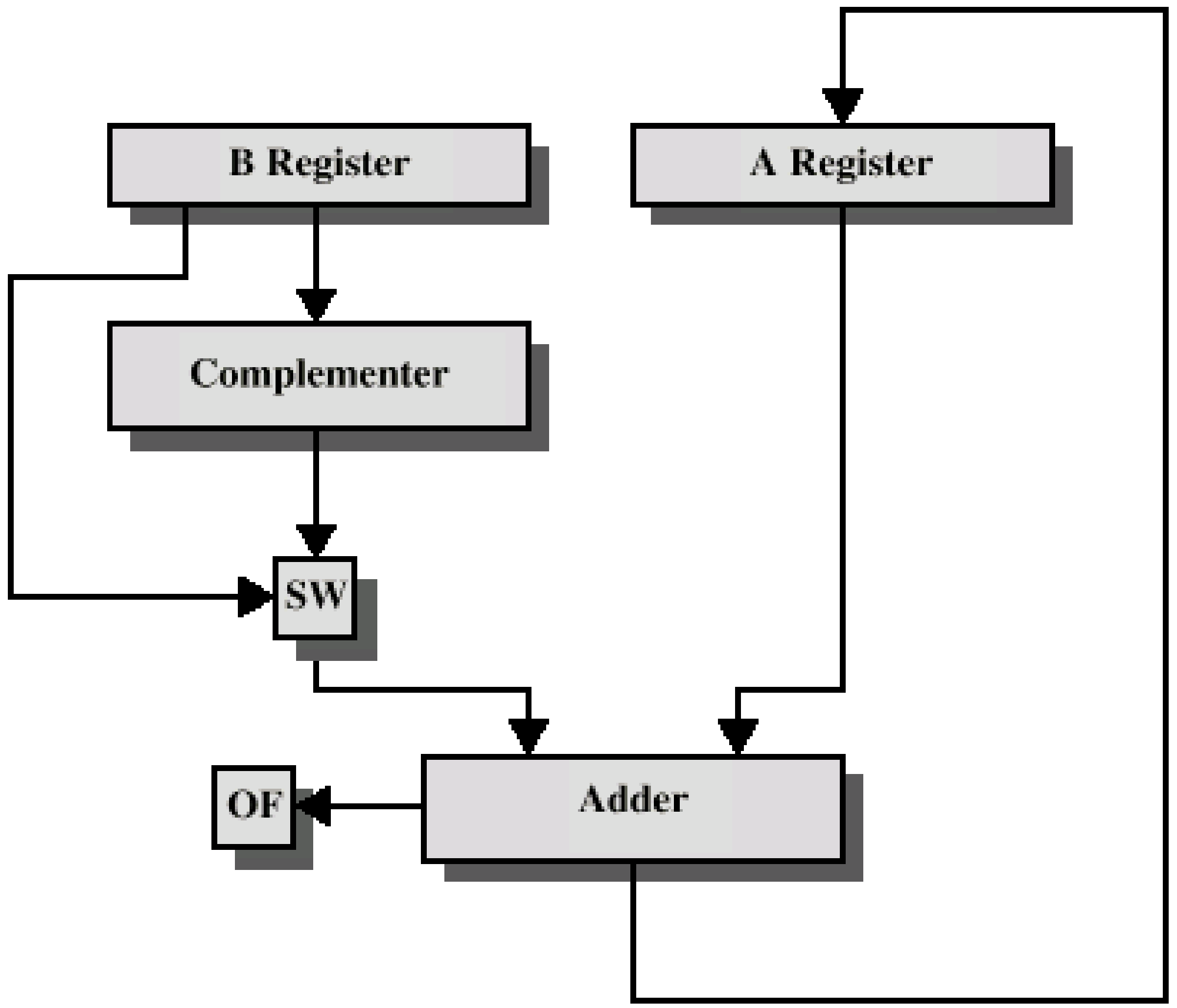
Negation Special Case 2

- -128 = 10000000
- bitwise not 01111111
- Add 1 to LSB +1
- Result 10000000
- So:
- **-(-128) = -128 Problem here!**
- Monitor MSB (sign bit)
- It should change during negation
- >> There is no representation of +128 in this case. (no $+2^n$)

Addition and Subtraction

- Normal binary addition
-   0011   0101   1100
- +0100   +0100   +1111
- --------   ----------   ------------
-   0111 $\underline{1}$001 = overflow $\underline{1}$1011
- Monitor sign bit for overflow (sign bit change as adding two positive numbers or two negative numbers.)
- Subtraction: Take twos compliment of subtrahend then add to minuend
  - i.e. a - b = a + (-b)
- **So we only need addition and complement circuits**

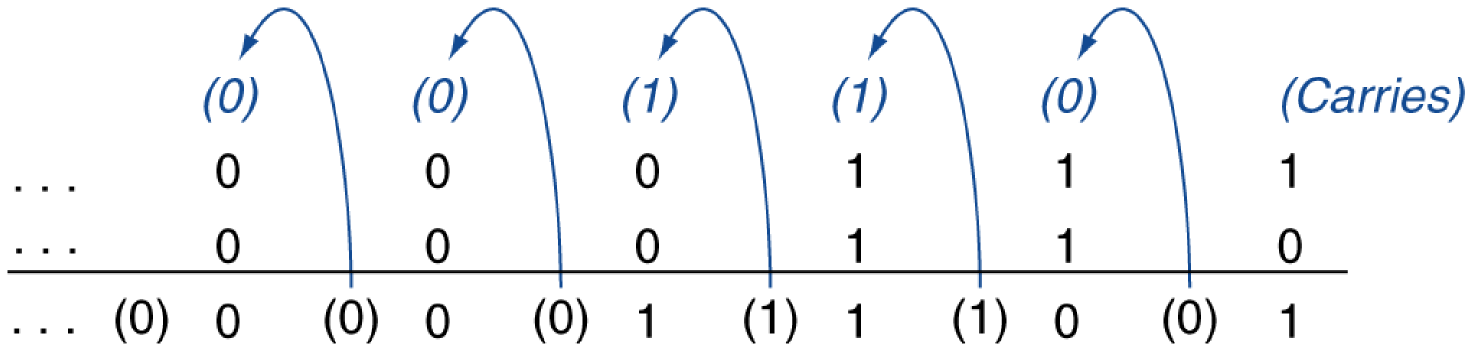
Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

Integer Addition - Overflow

- Example: 7 + 6



- **Overflow** if result is out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

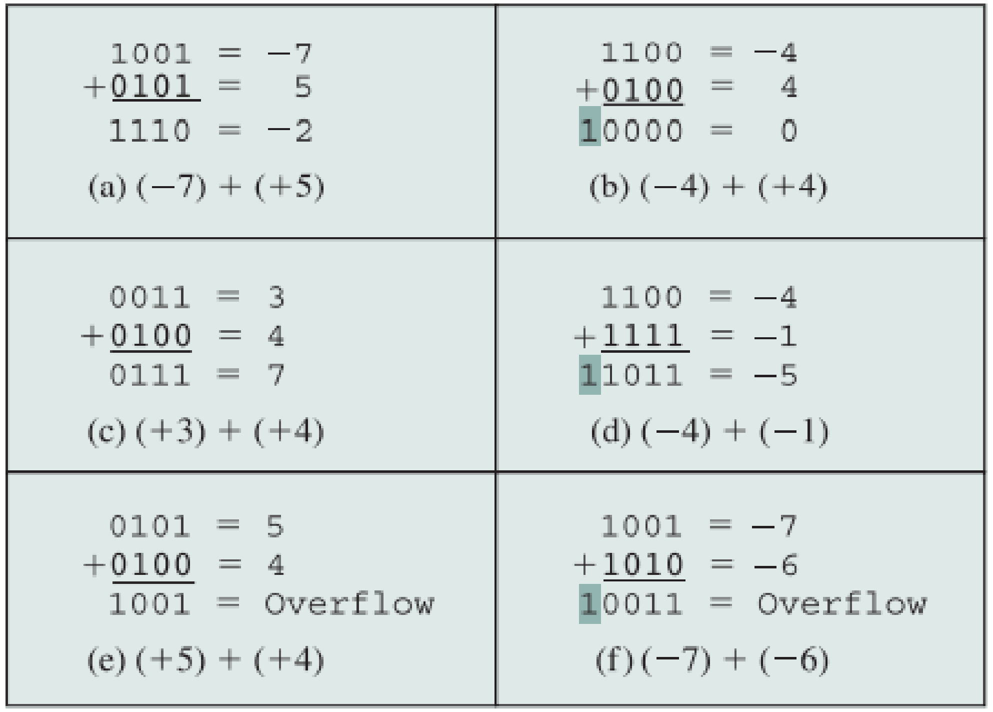
Integer Subtraction - Overflow

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

  -
    ```
       +7:     0000 0000 … 0000 0111
    –6:        1111 1111 … 1111 1010
    +1:        0000 0000 … 0000 0001
    ```
- **Overflow** if result is out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Overflow in Addition / Subtraction

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.



```
  1001  =  −7
 +0101  =   5
  1110  =  −2
(a) (−7) + (+5)
```

```
  1100  =  −4
 +0100  =   4
 10000  =   0
(b) (−4) + (+4)
```

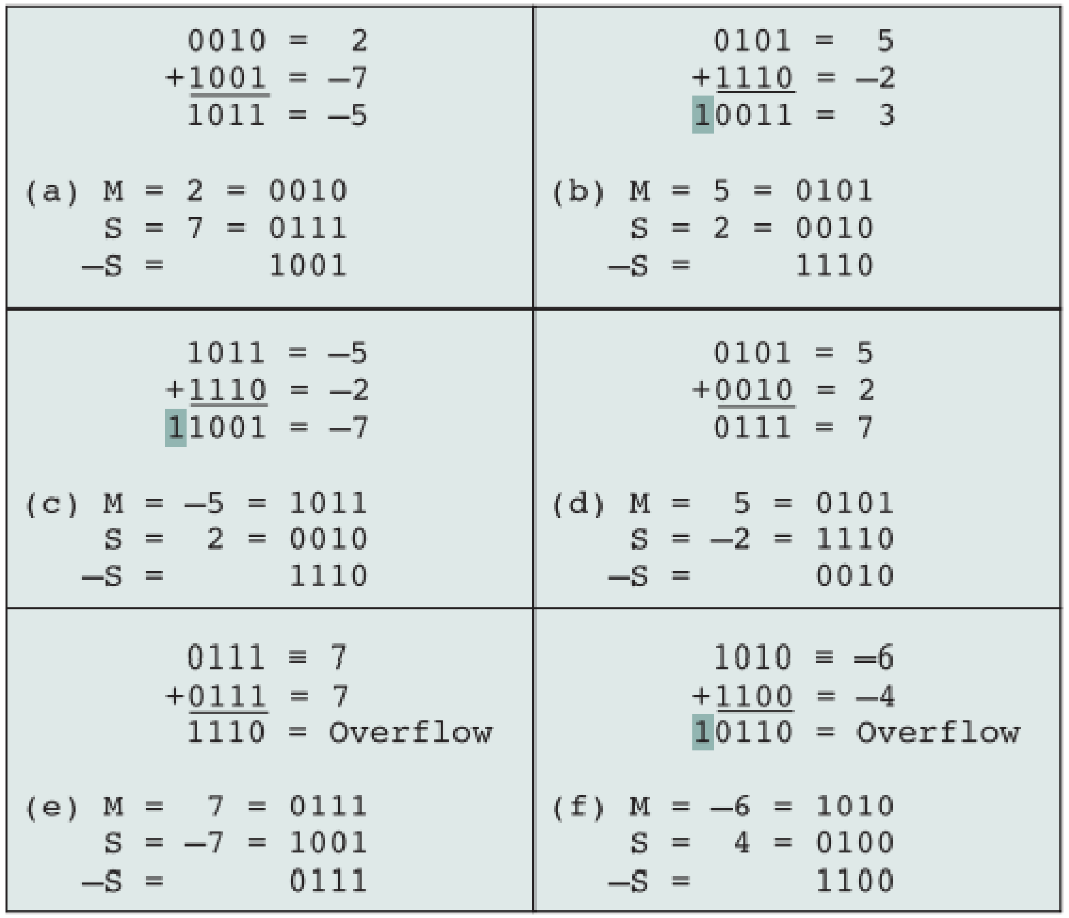
```
  0011  =  3
 +0100  =  4
  0111  =  7
(c) (+3) + (+4)
```

```
  1100  =  −4
 +1111  =  −1
 11011  =  −5
(d) (−4) + (−1)
```

```
  0101  =  5
 +0100  =  4
  1001  =  Overflow
(e) (+5) + (+4)
```

```
  1001  =  −7
 +1010  =  −6
 10011  =  Overflow
(f) (−7) + (−6)
```

Figure 10.3 Addition of Numbers in Twos Complement Representation



```
      0010  =   2
     +1001  =  −7
      1011  =  −5

(a) M = 2 = 0010
    S = 7 = 0111
   −S =     1001
```

```
      0101  =   5
     +1110  =  −2
     10011  =   3

(b) M = 5 = 0101
    S = 2 = 0010
   −S =     1110
```

```
      1011  =  −5
     +1110  =  −2
     11001  =  −7

(c) M = −5 = 1011
    S =  2 = 0010
   −S =      1110
```

```
      0101  =  5
     +0010  =  2
      0111  =  7

(d) M =  5 = 0101
    S = −2 = 1110
   −S =      0010
```

```
      0111  =  7
     +0111  =  7
      1110  =  Overflow

(e) M =  7 = 0111
    S = −7 = 1001
   −S =      0111
```

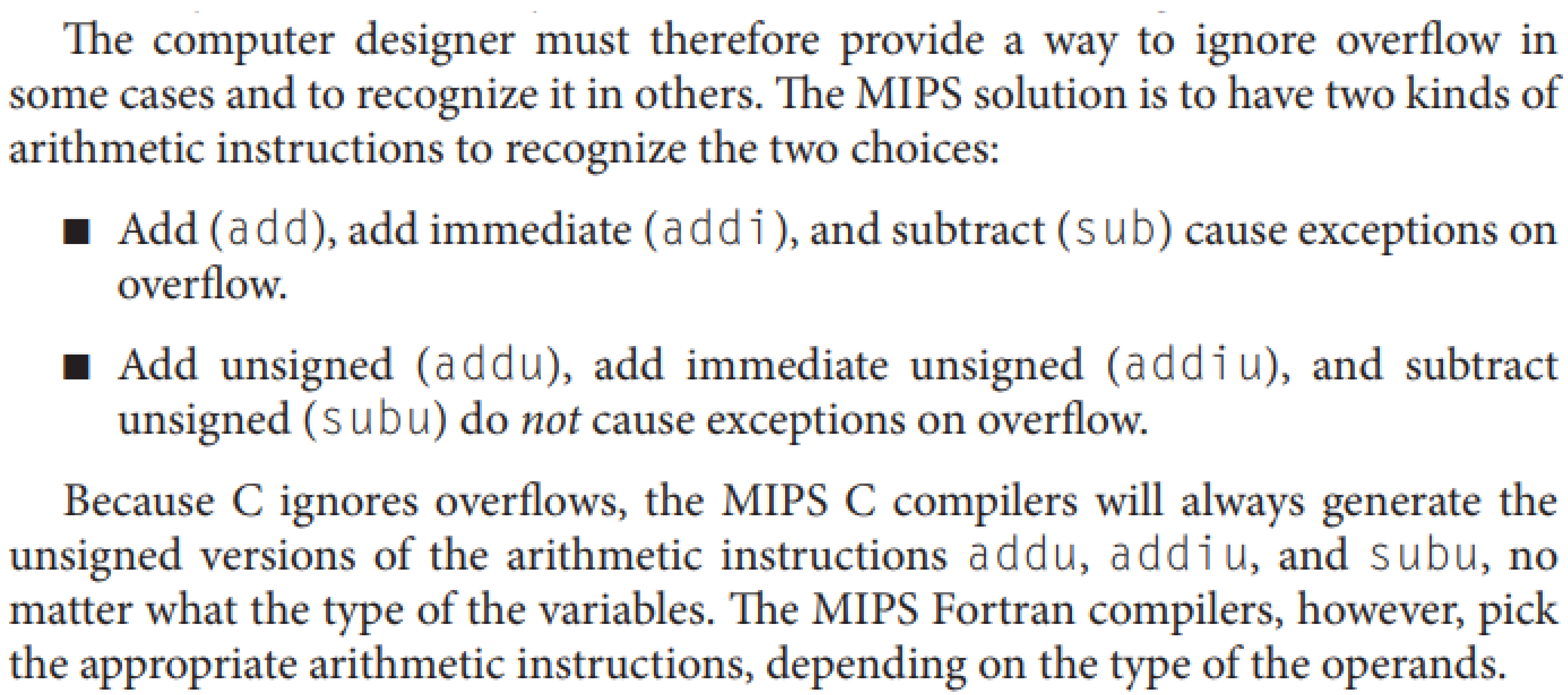
```
      1010  =  −6
     +1100  =  −4
     10110  =  Overflow

(f) M = −6 = 1010
    S =  4 = 0100
   −S =      1100
```

Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M − S)

CPU Dealing with Overflow

- Some languages (e.g., C) ignore overflow
    - Use MIPS addu, addui, subu instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
    - Use MIPS add, addi, sub instructions
    - On overflow, invoke exception handler
        - Save PC in exception program counter (EPC) register
        - Jump to predefined handler address
        - mfc0 (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

MIPS Instructions and Overflow



The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

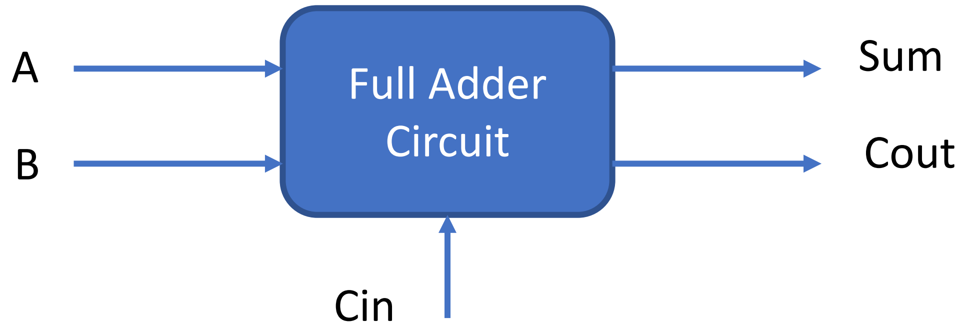
- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu`, no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Logic Circuit of Adder / Subtractor

1 Bit Full Adder Circuit

Truth table represents behaviour of inputs and outputs



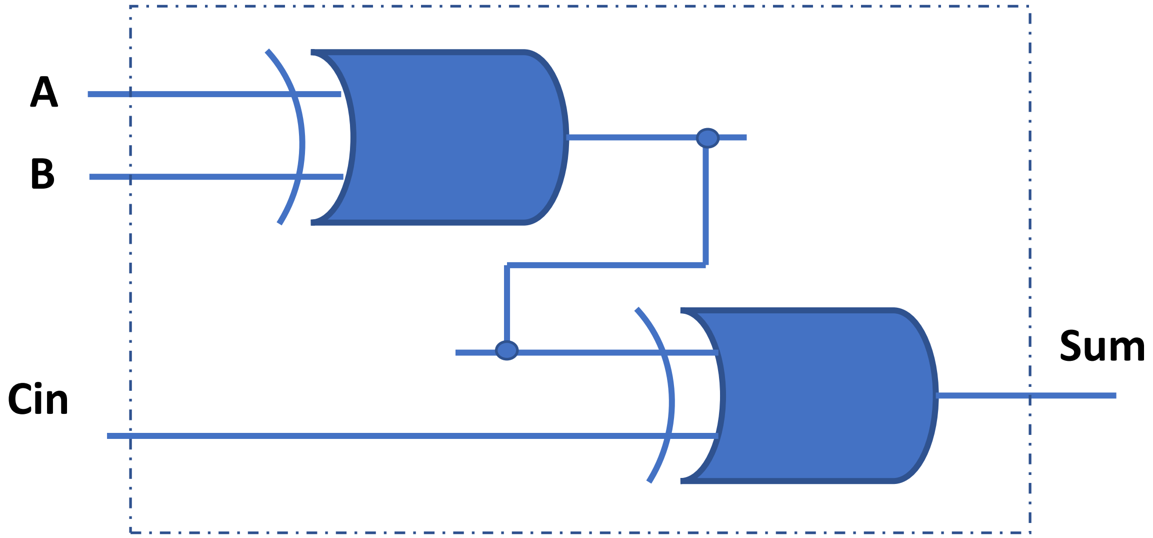
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Boolean Expression:

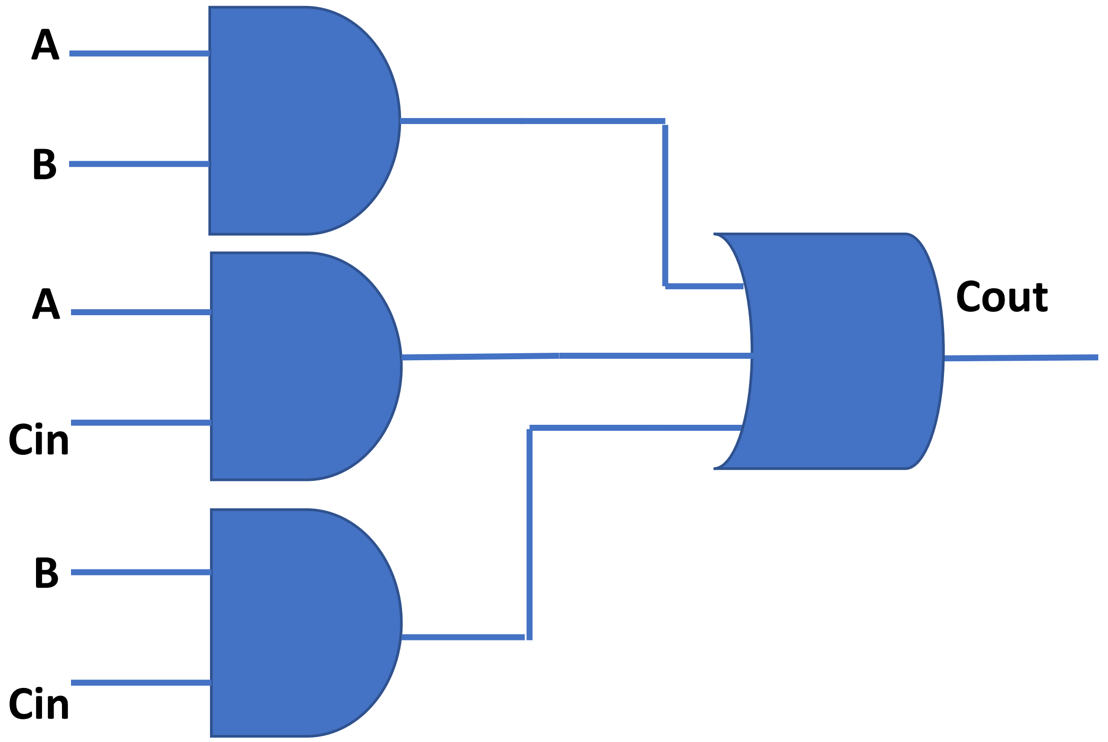
Sum $S_i = X_i \oplus Y_i \oplus C_i$

$C_{i+1} = X_i . Y_i + C_i (X_i \oplus Y_i)$

Full Adder Implementation using Logic Gates



***Sum* = $A \oplus B \oplus Cin$**

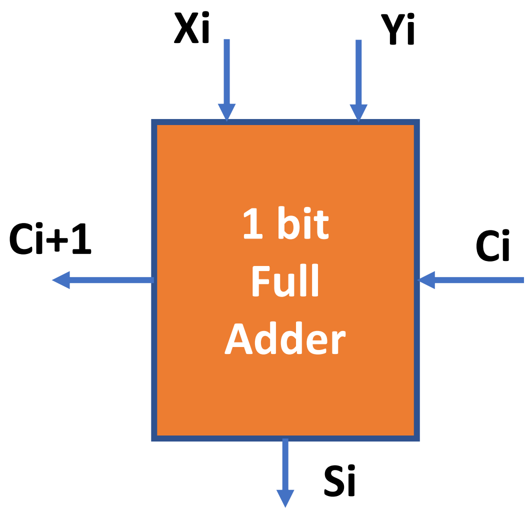


***Cout* = $AB + ACin + BCin$**

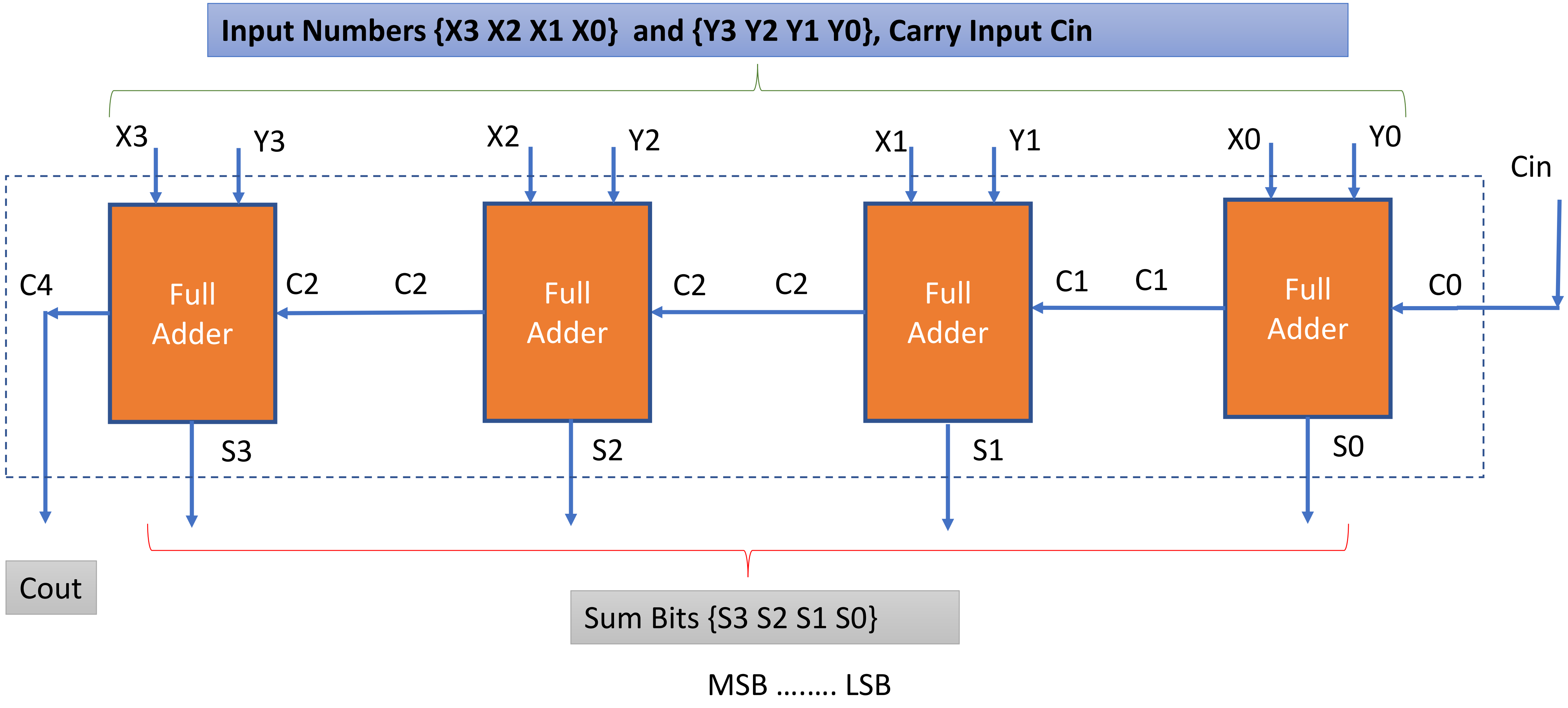
Array of Adders?

Usually, data types defined in any program are 8 bits, 16 bits or 32 bits

We need array of arithmetic and logic circuits to perform ALU operation on CPU registers



Multiple Bits – Ripple Carry Adder



Readings

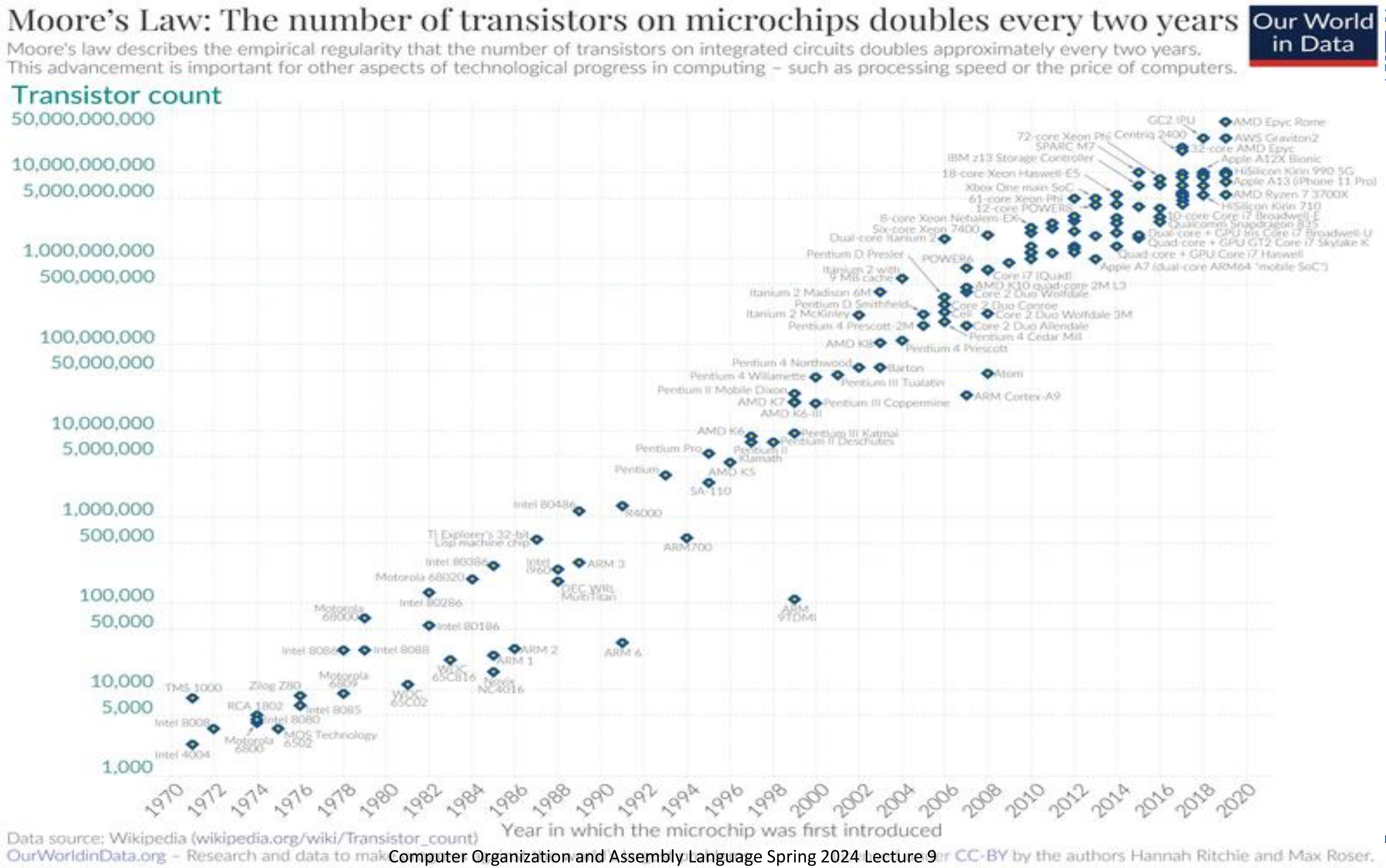
- P&H Textbook Chapter 3

- Search Appendix C P&H Textbook online; this contains useful background material
on digital logic, ALU and related stuff

Topics

- Computer Performance – in light of
  - Power Wall
  - Complexity Wall
  - Moore’s Law
  - Dennard Scaling
- Computing ideas in post-PC era
- What is meant by the Hardware / Software Interface
- Calculate Power dissipation
- Notion of Computer Performance

Power Wall, and Complexity Wall

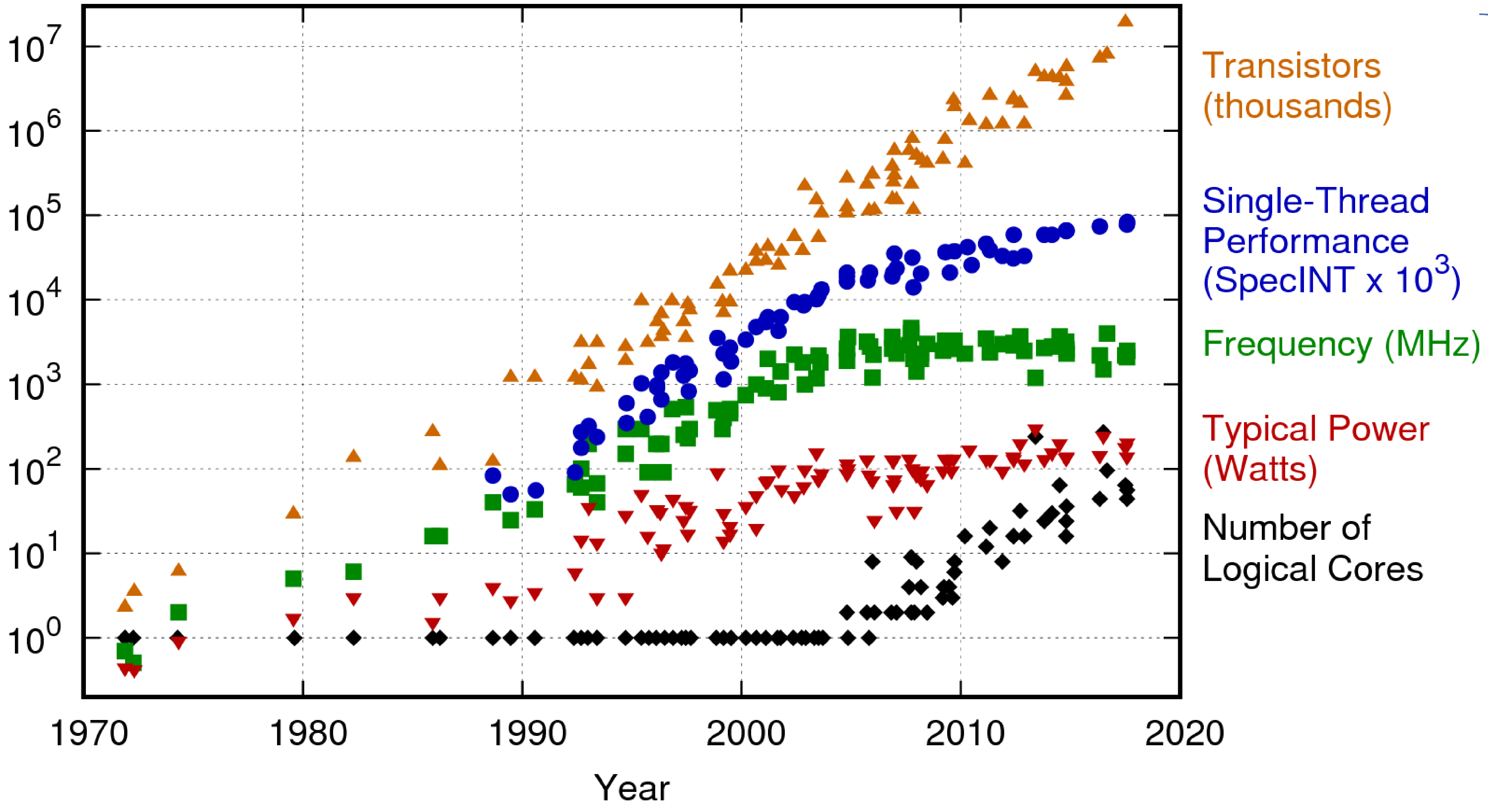


Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

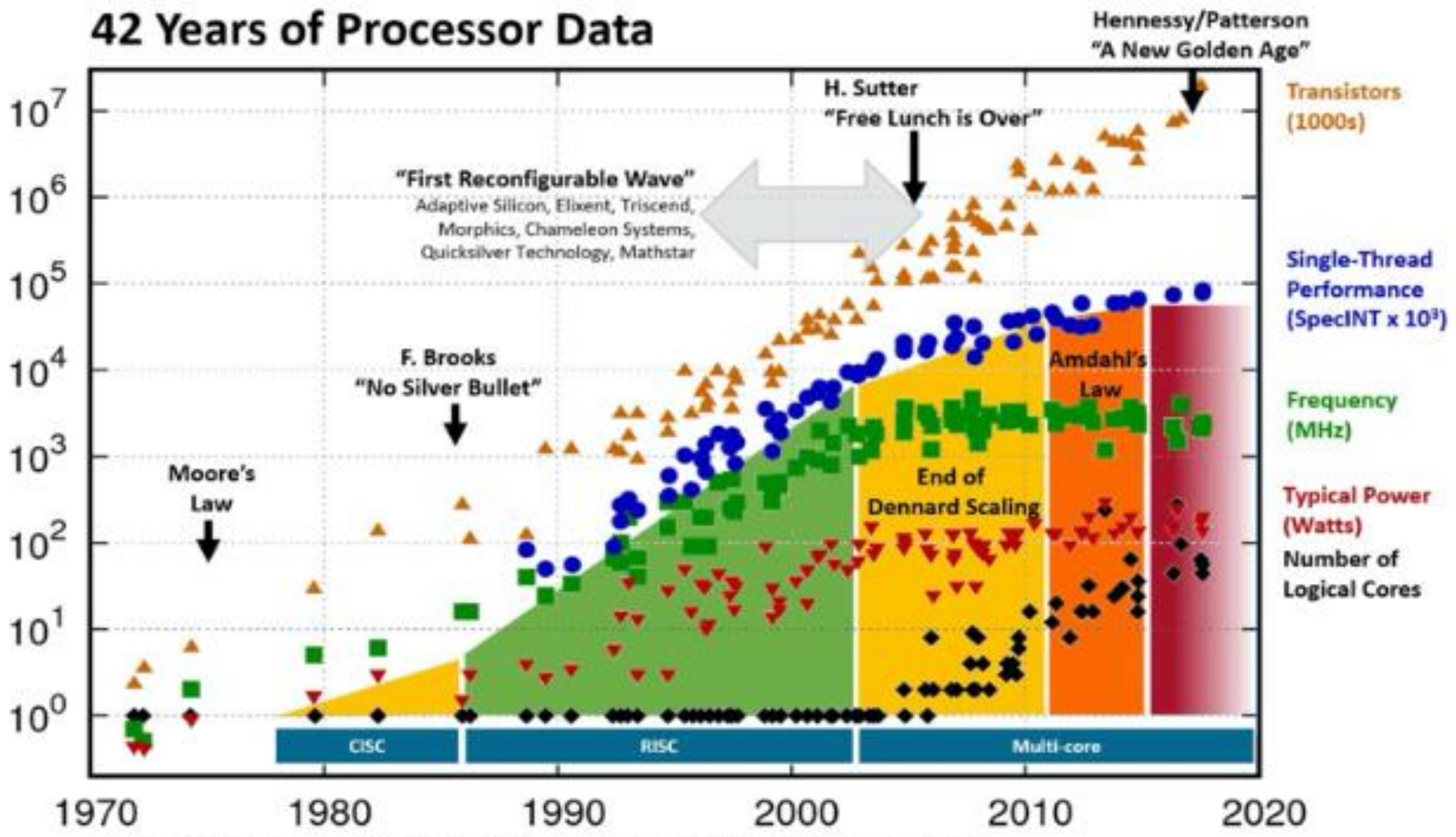
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to mak[e progress against the world's largest problems.] [Licensed und]er CC-BY by the authors Hannah Ritchie and Max Roser.

42 Years of Microprocessor Trend Data

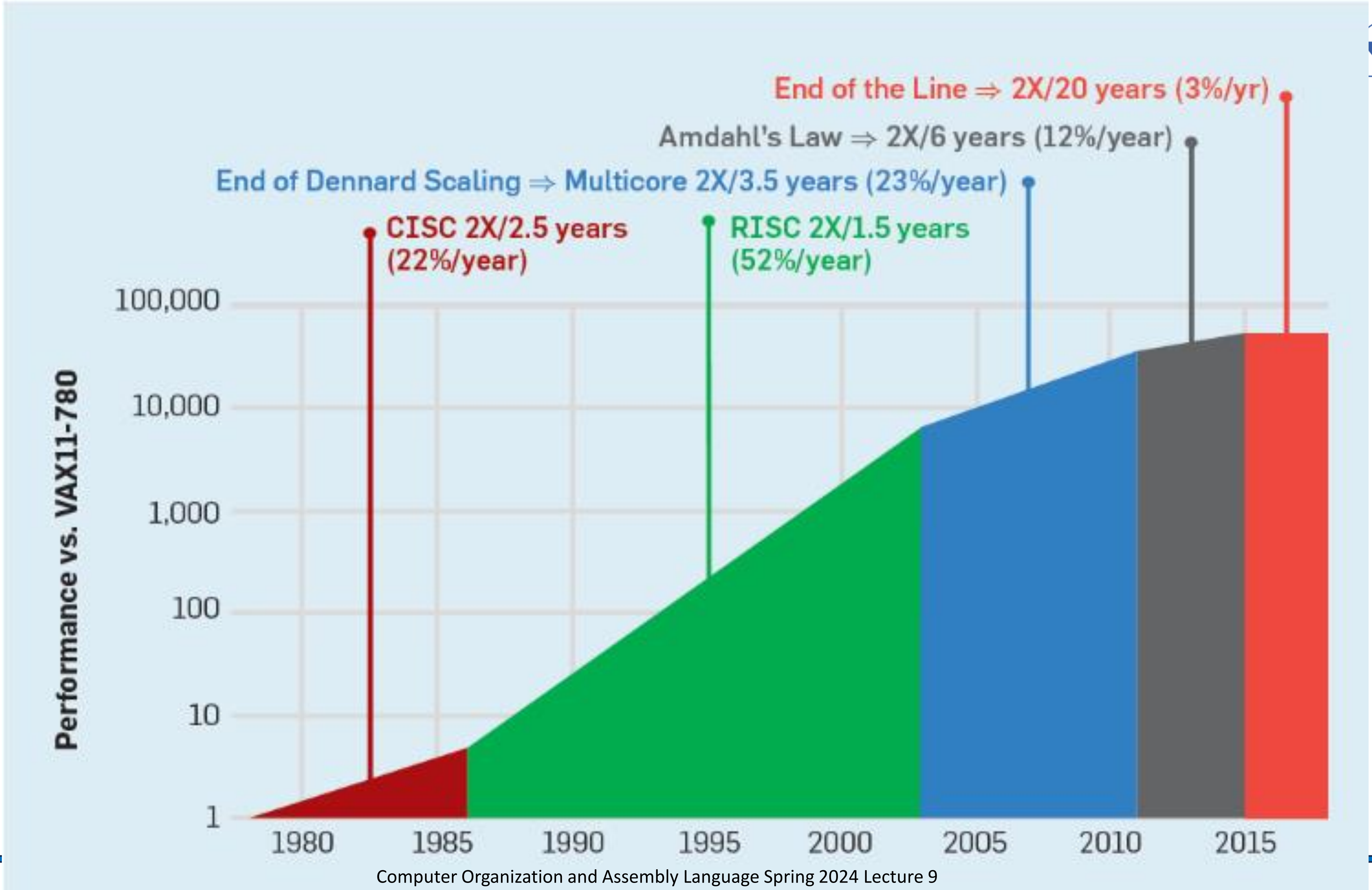


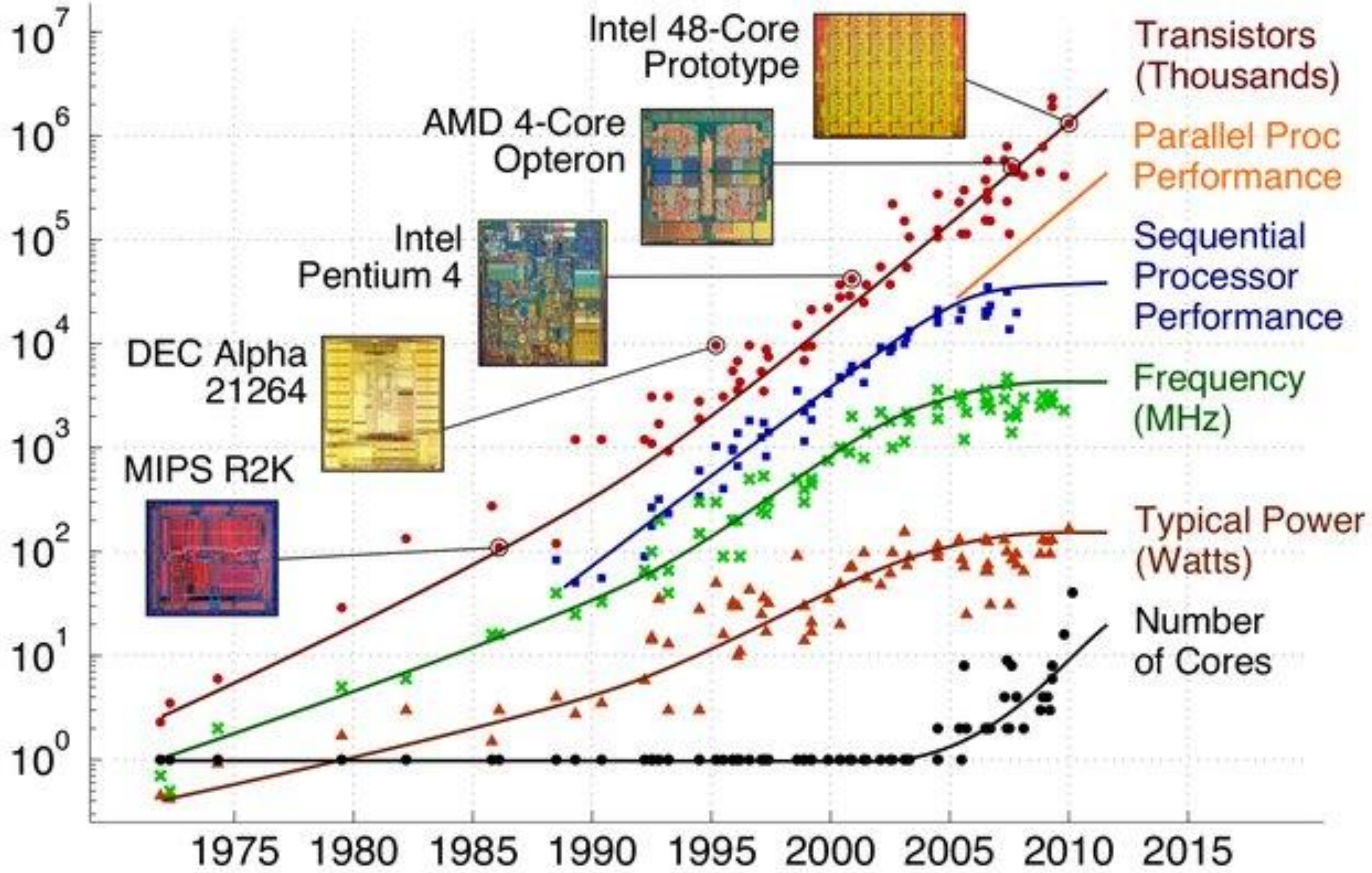
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

42 Years of Processor Data



Hennessy and Patterson, Turing Lecture 2018, overlaid over "42 Years of Processors Data"
https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/;"First Wave" added by Les Wilson, Frank Schirrmeister
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp


End of the Line ⇒ 2X/20 years (3%/yr)
Amdahl's Law ⇒ 2X/6 years (12%/year)
End of Dennard Scaling ⇒ Multicore 2X/3.5 years (23%/year)
CISC 2X/2.5 years (22%/year)
RISC 2X/1.5 years (52%/year)
Performance vs. VAX11-780
100,000
10,000
1,000
100
10
1
1980
1985
1990
1995
2000
2005
2010
2015



Intel 48-Core Prototype
AMD 4-Core Opteron
Intel Pentium 4
DEC Alpha 21264
MIPS R2K
Transistors (Thousands)
Parallel Proc Performance
Sequential Processor Performance
Frequency (MHz)
Typical Power (Watts)
Number of Cores
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1975
1980
1985
1990
1995
2000
2005
2010
2015

CPU Architecture Today
Heat becoming un unmanageable problem

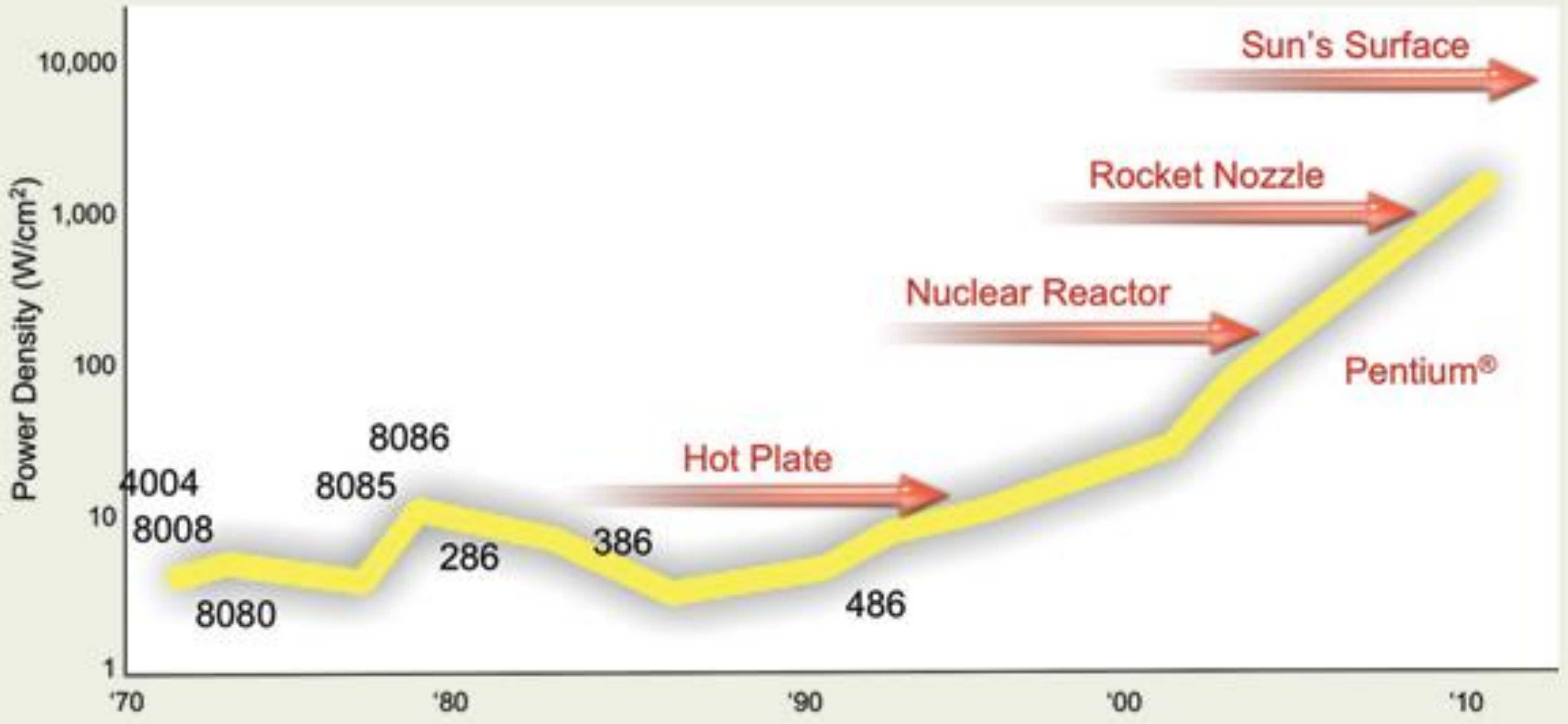


Figure 1. In CPU architecture today, heat is becoming an unmanageable problem. (Courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004)

CPU: From GHz to multi-core

Moore's Law:

- *~ the number of transistors on an IC doubles every two years.*
  - Less space, more complexity.
  - Shorter gates, higher clock rate.

Strategy of the 80s and 90's:

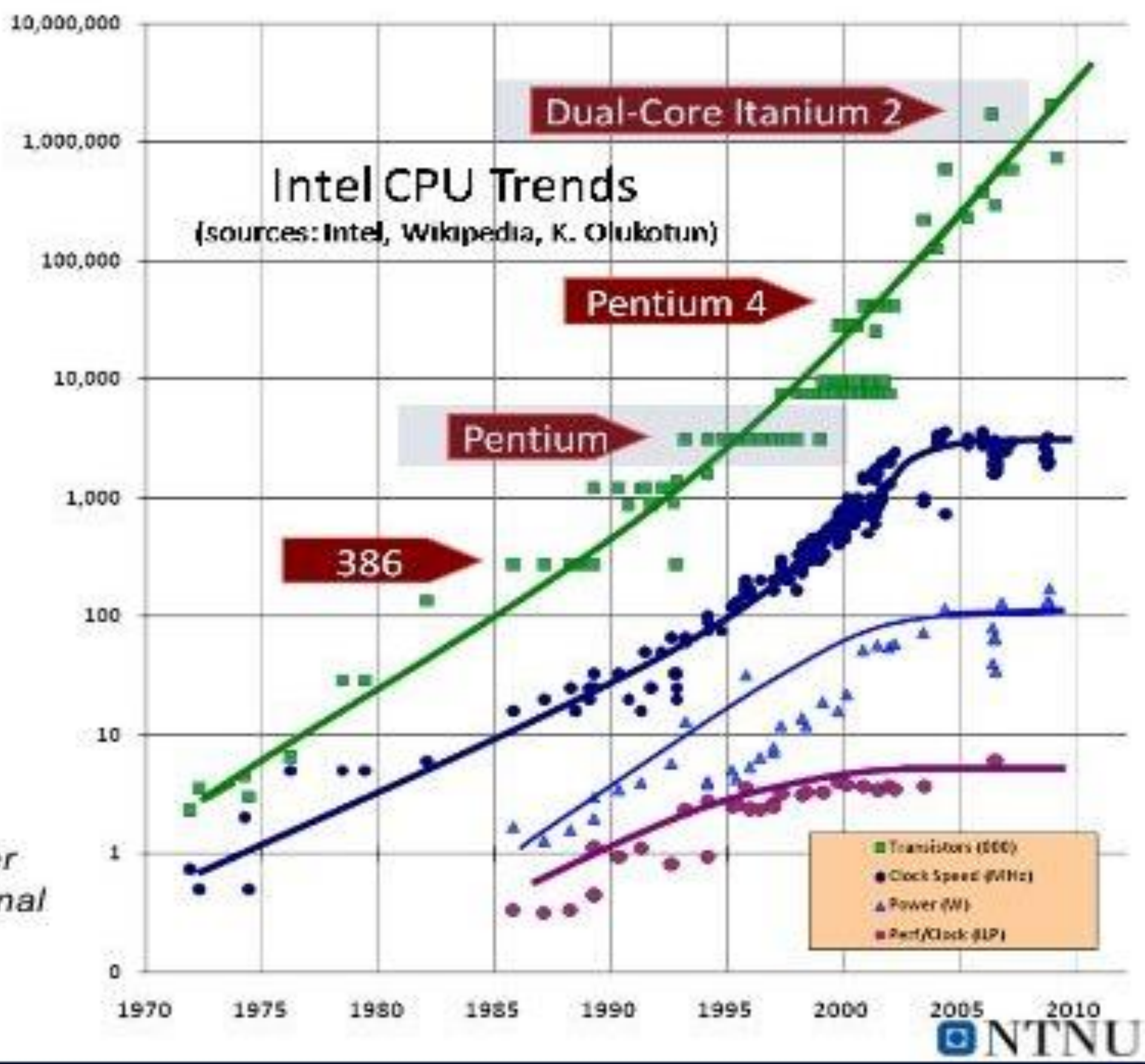
- Add more complexity!
- Increase the clock rate!

Pollack's Rule:

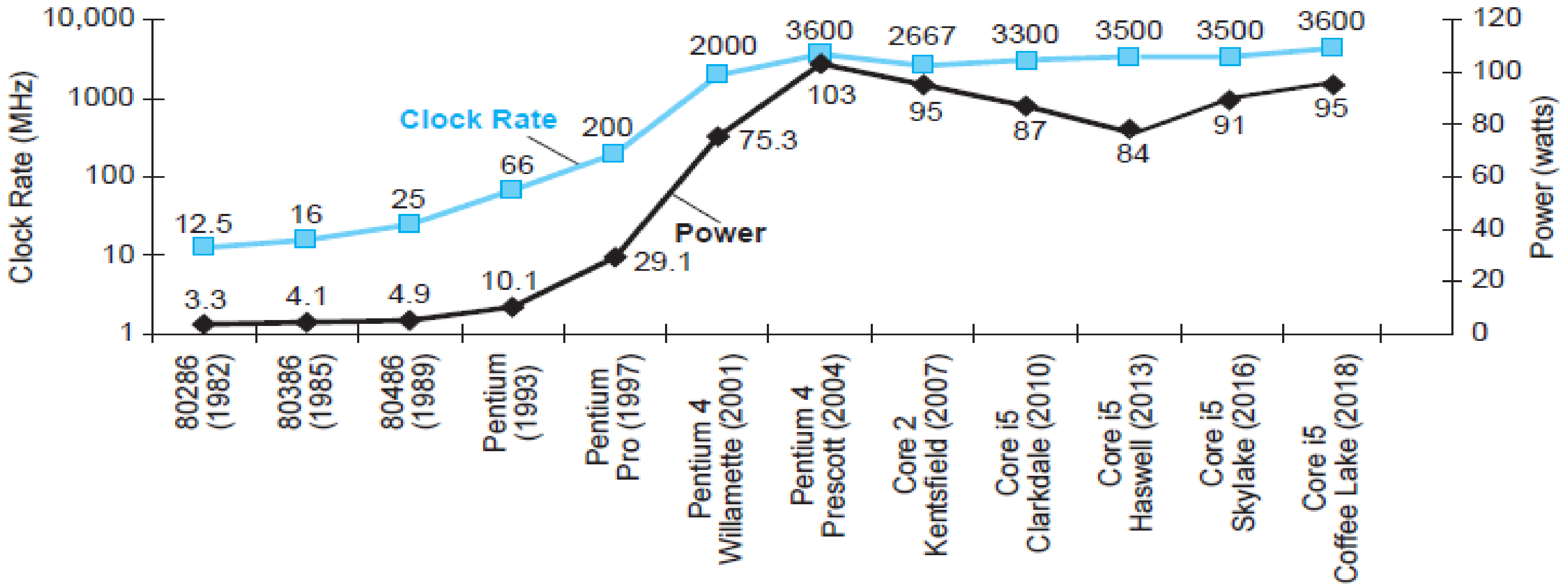
- *The performance increase is ~ square root of the increased complexity.* [Borkar 2007]

The Power Wall:

- *Increasing clock rate and transistor current leakage lead to excess power consumption, while RC delays in signal transmission grow as feature sizes shrink.* [Borkar et al. 2005]



CPU Power Trends

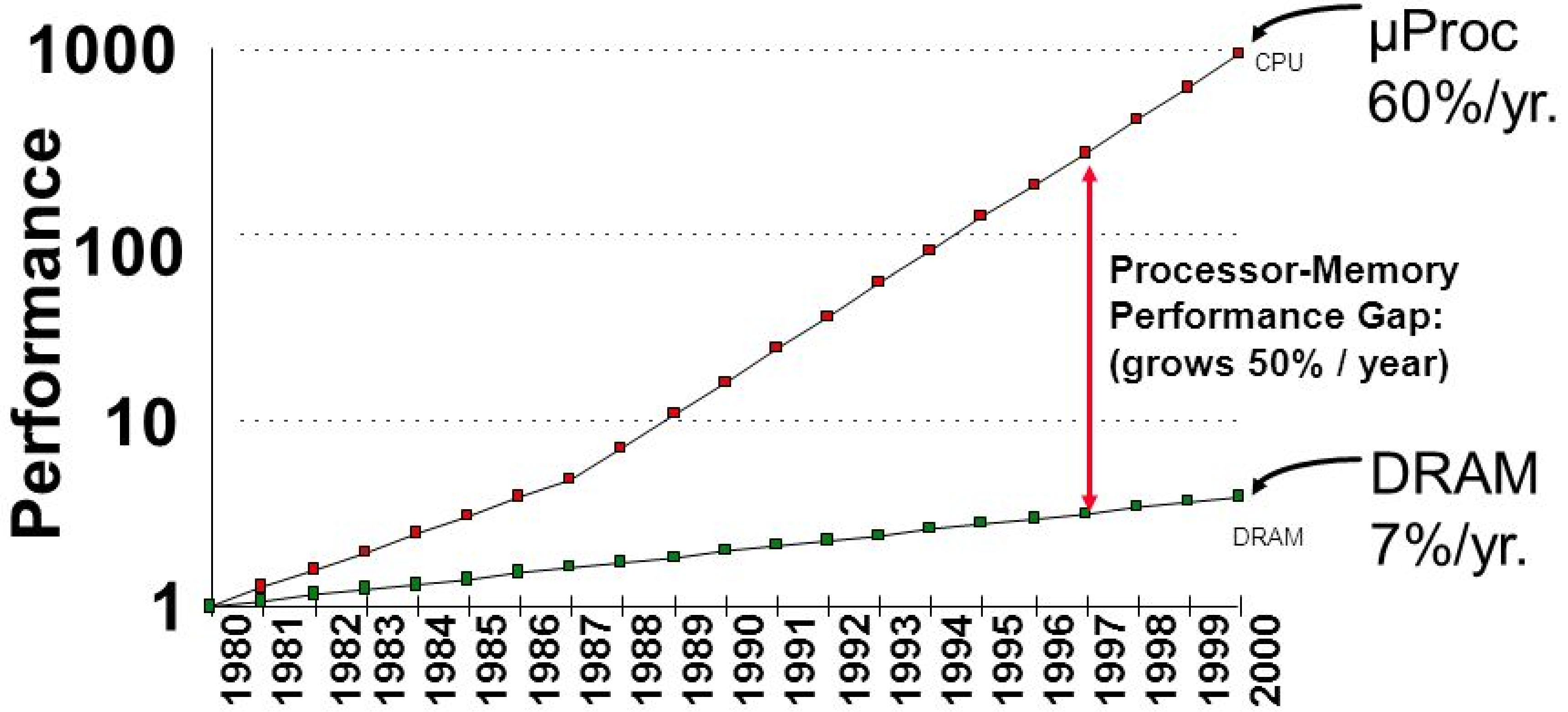


- In CMOS IC technology

$$\underbrace{\text{Power}}_{\times 30} = \text{Capacitive load} \times \underbrace{\text{Voltage}^2}_{5\text{V} \to 1\text{V}} \times \underbrace{\text{Frequency}}_{\times 1000}$$

Memory Hierarchy: Motivation Processor-Memory (DRAM) Performance Gap

(i.e. Gap between memory access time and CPU cycle time



EECC551 - Shaaban

(Review from 550)

Reducing Power

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

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

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

Computing Ideas in Post-PC Era

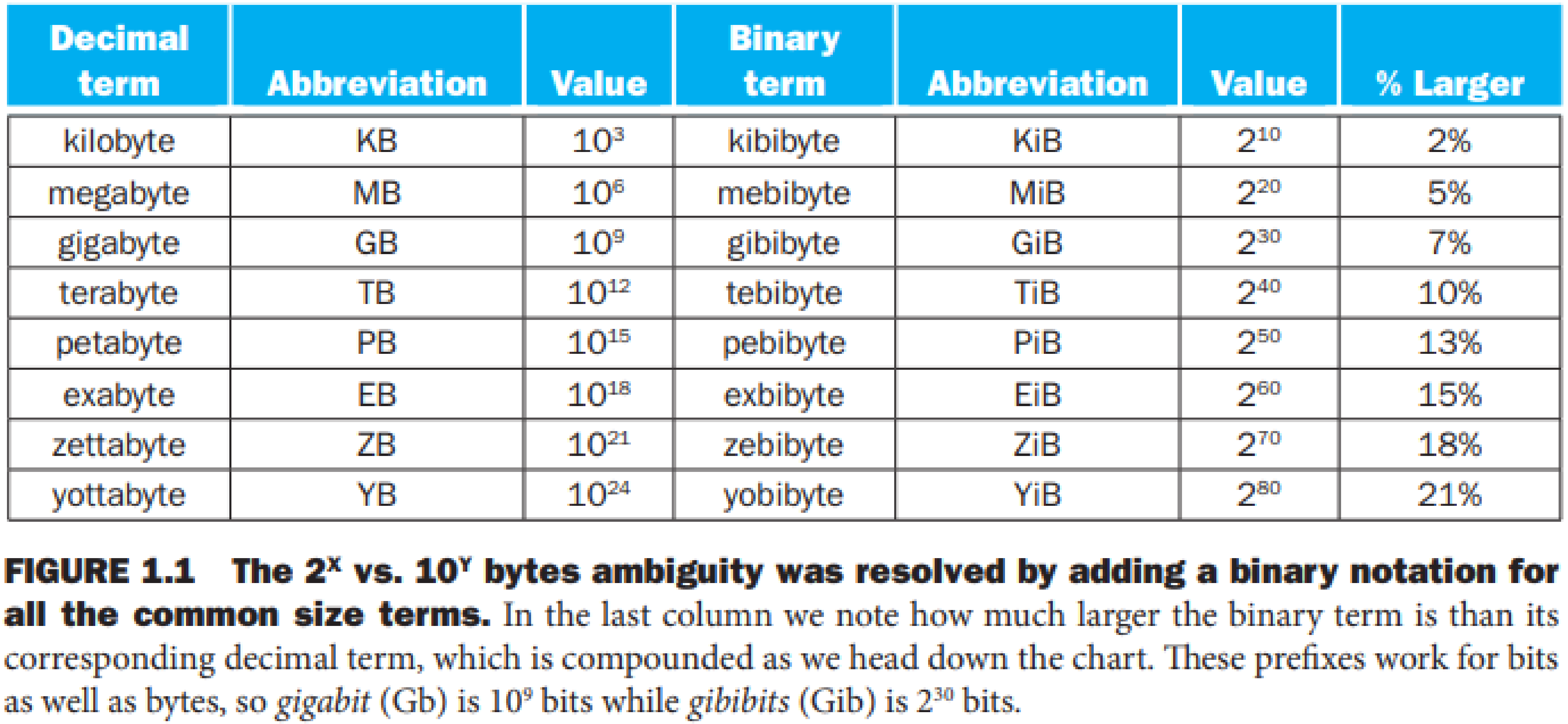
Current High-Performance Computing Architectures Post-Moore's-Law Era

- Model Based Abstract Software Development for Complex Systems
- Parallel Processing through Multicores
- Instruction Level Parallelism through Pipelining
- Speculation and Prediction in RISC Architecture
- Performance Evaluation to make the 'Common Case Fast'
- Use variety of memory hierarchies to improve performance
- Cloud based Computing and Storage
- Role of Network in Computer Performance

Upcoming Computing Architectures
Post-PC Era

- Mobile and Embedded Computing
- Domain Specific Architectures – E.g. Google Tensor Processor
- Domain Specific Languages and Compilers – E.g. Tensorflow
- Open Instruction Set Architectures – RISC – V
- Agile Computing through Reconfigurable Hardware and Customized Instructions
- Security in CPU Hardware

The Scale in Computing

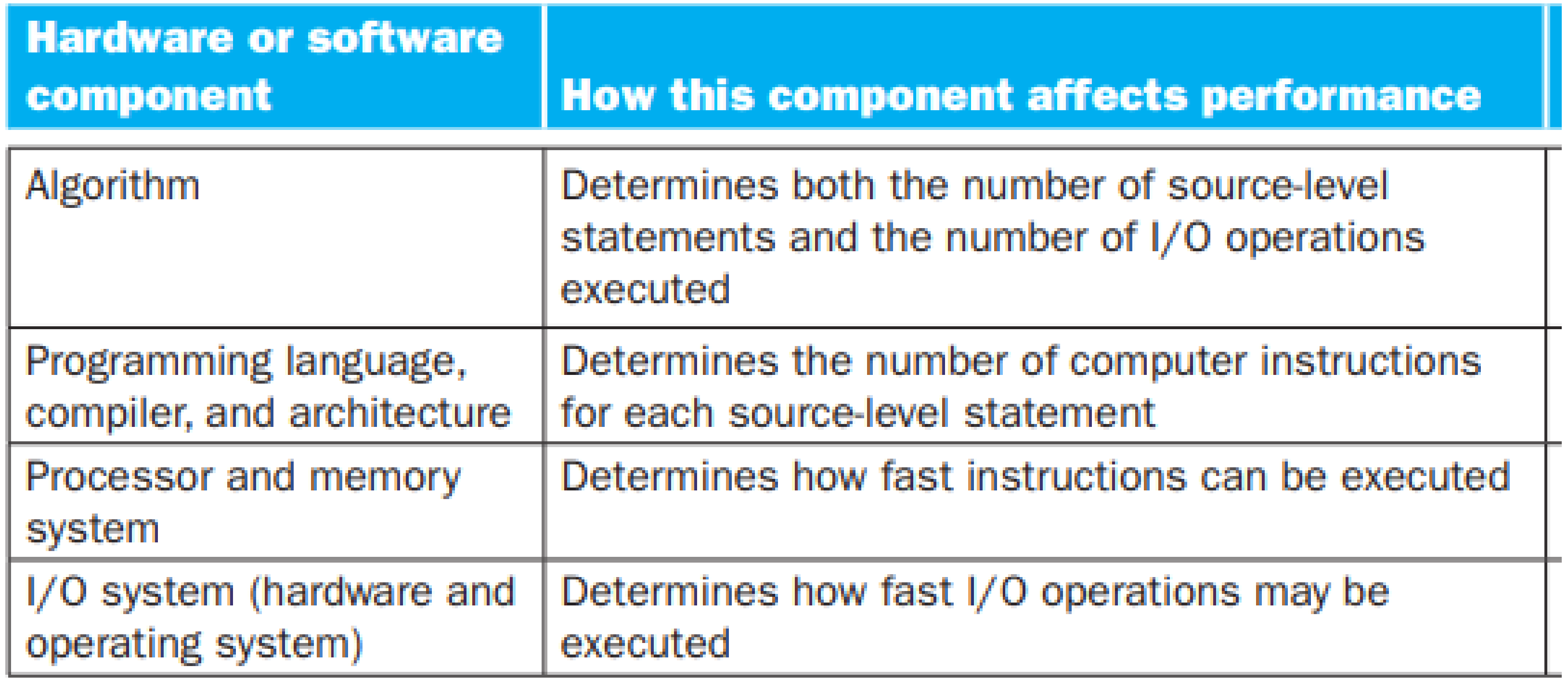


Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	2%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	5%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	7%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	10%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	13%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	15%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	18%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	21%

FIGURE 1.1 The 2^X vs. 10^Y bytes ambiguity was resolved by adding a binary notation for all the common size terms. In the last column we note how much larger the binary term is than its corresponding decimal term, which is compounded as we head down the chart. These prefixes work for bits as well as bytes, so *gigabit* (Gb) is 10^9 bits while *gibibits* (Gib) is 2^{30} bits.

The Hardware / Software Interface

Interface of Hardware and Software



Hardware or software component	How this component affects performance
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed
Programming language, compiler, and architecture	Determines the number of computer instructions for each source-level statement
Processor and memory system	Determines how fast instructions can be executed
I/O system (hardware and operating system)	Determines how fast I/O operations may be executed

Hardware / Software Interface

Software

Instruction Set Architecture

Hardware

C/C++/Python
↓
Assembly / Machine Code
↓
Registers, Datapath
↓
Gates and Logic
↓
Technology

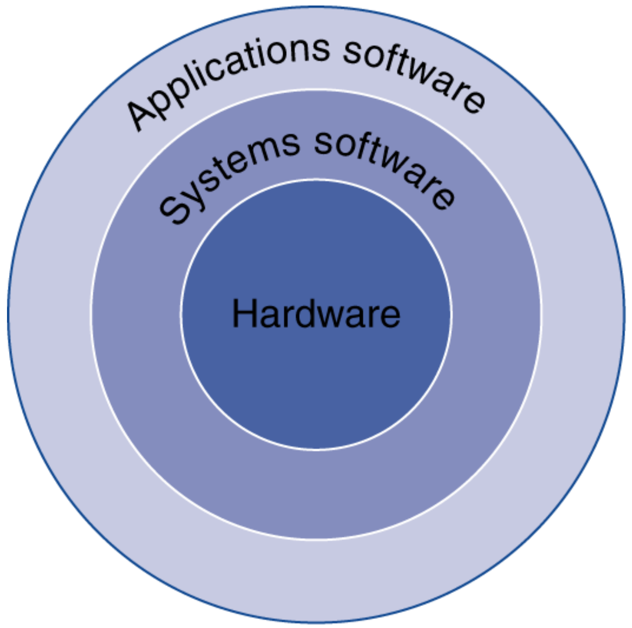
Programmer View

Why Abstraction?

- Delving into the depths reveals more information

- An abstraction omits unneeded detail, helps us cope with complexity

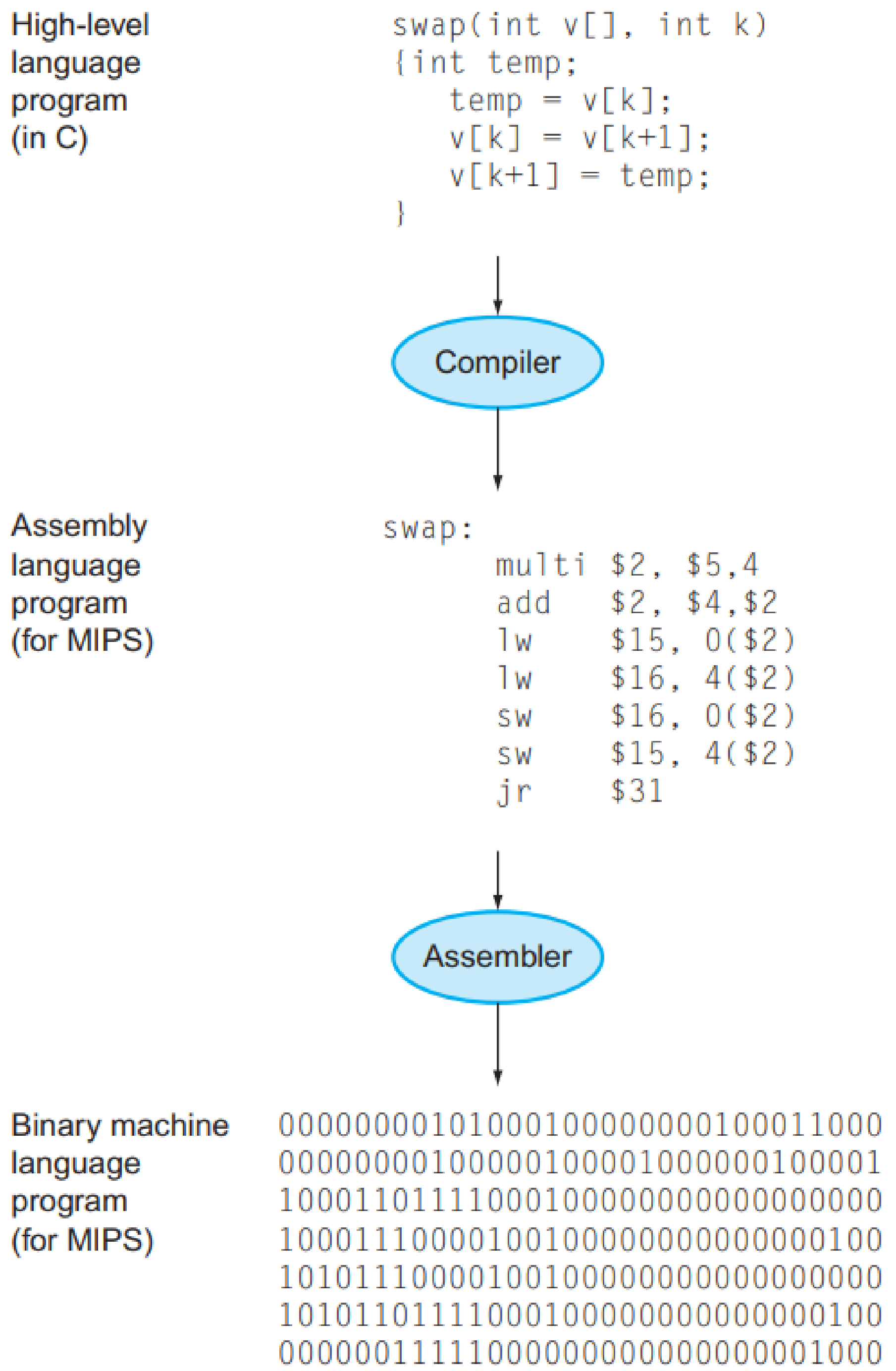
Below a Computer Program



- Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

Compiling a Program

Example of Software Abstraction


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

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

```
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
```

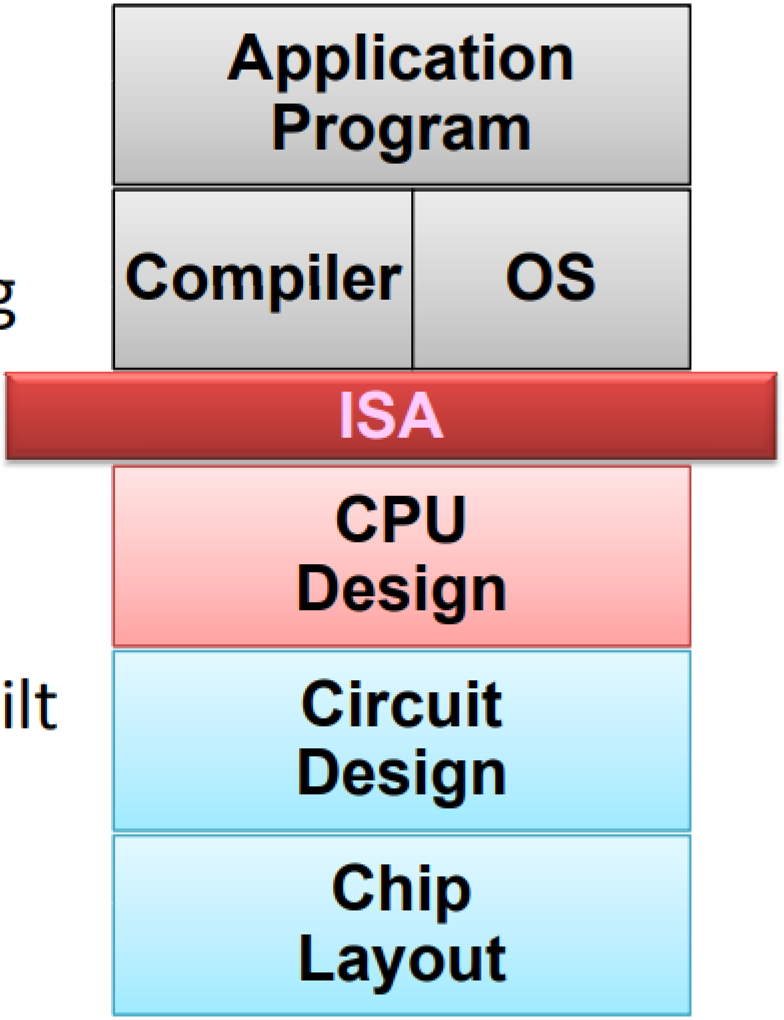
```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111000100000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111000100000000000000100
00000011111000000000000000001000
```

What is an Instruction Set Architecture ISA?

- Lowest level of Computer Architecture that is visible to a programmer
- ISA comprises instructions in Assembly Language and Corresponding Binary Machine Code
- It is the language of the CPU machine
- Primitive syntax compared to a high-level language
- It is easily interpreted and understood by the CPU
- Designed to maximize performance
- Designed to minimize cost and design time

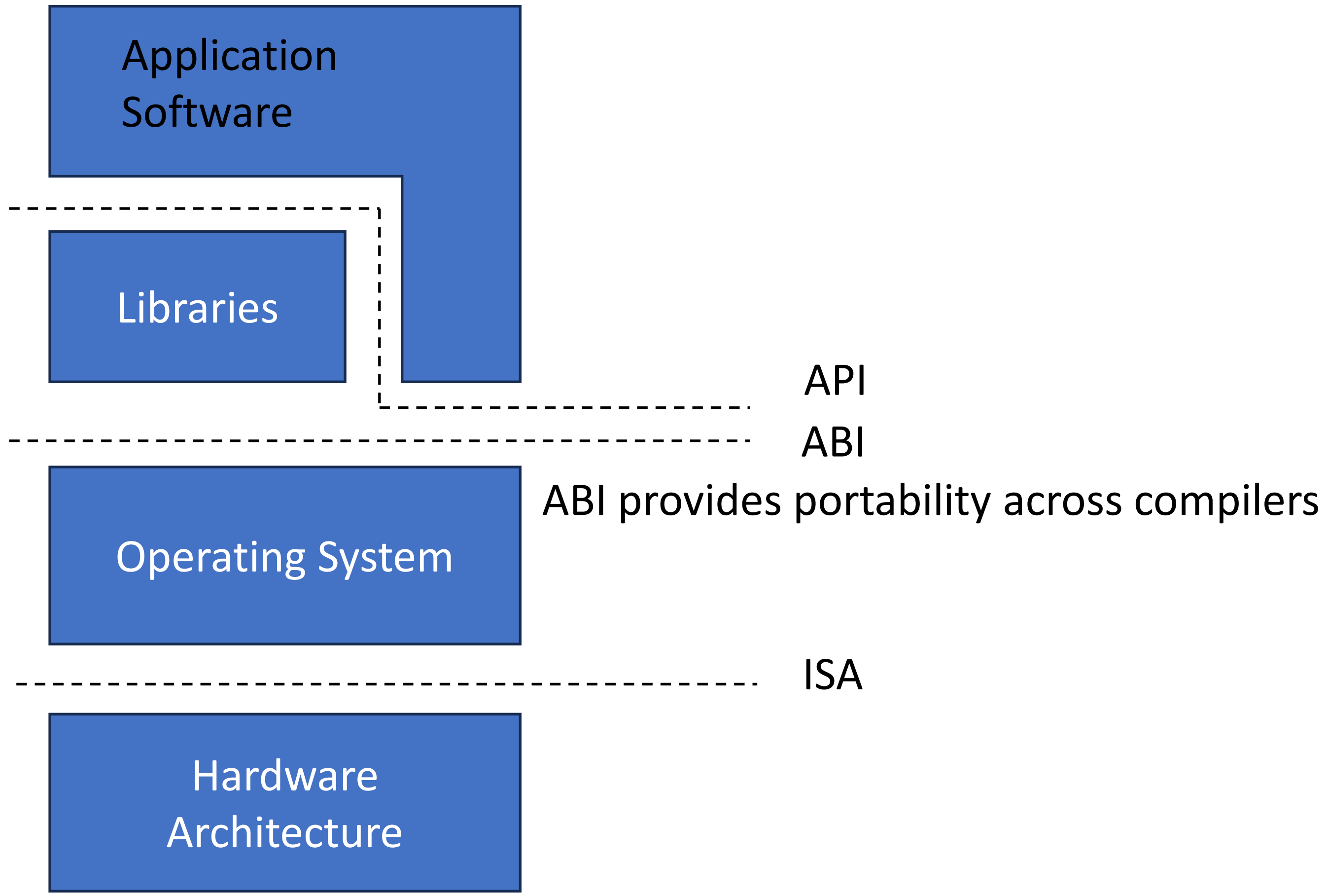
Role of ISA within a Computer

- Assembly Language View
  - Processor state (RF, mem)
  - Instruction set and encoding
- Layer of Abstraction
  - Above: how to program machine - HLL, OS
  - Below: what needs to be built - tricks to make it run fast

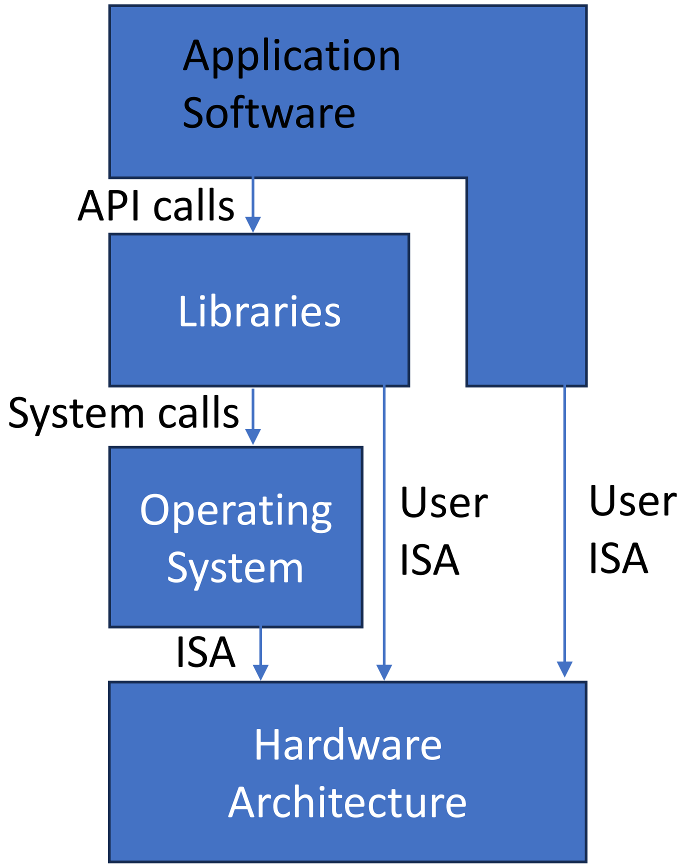


Key:
RF = Register File
Mem = Memory
HLL = High Level Language
OS = Operating System

Operating System Interface Software / Hardware

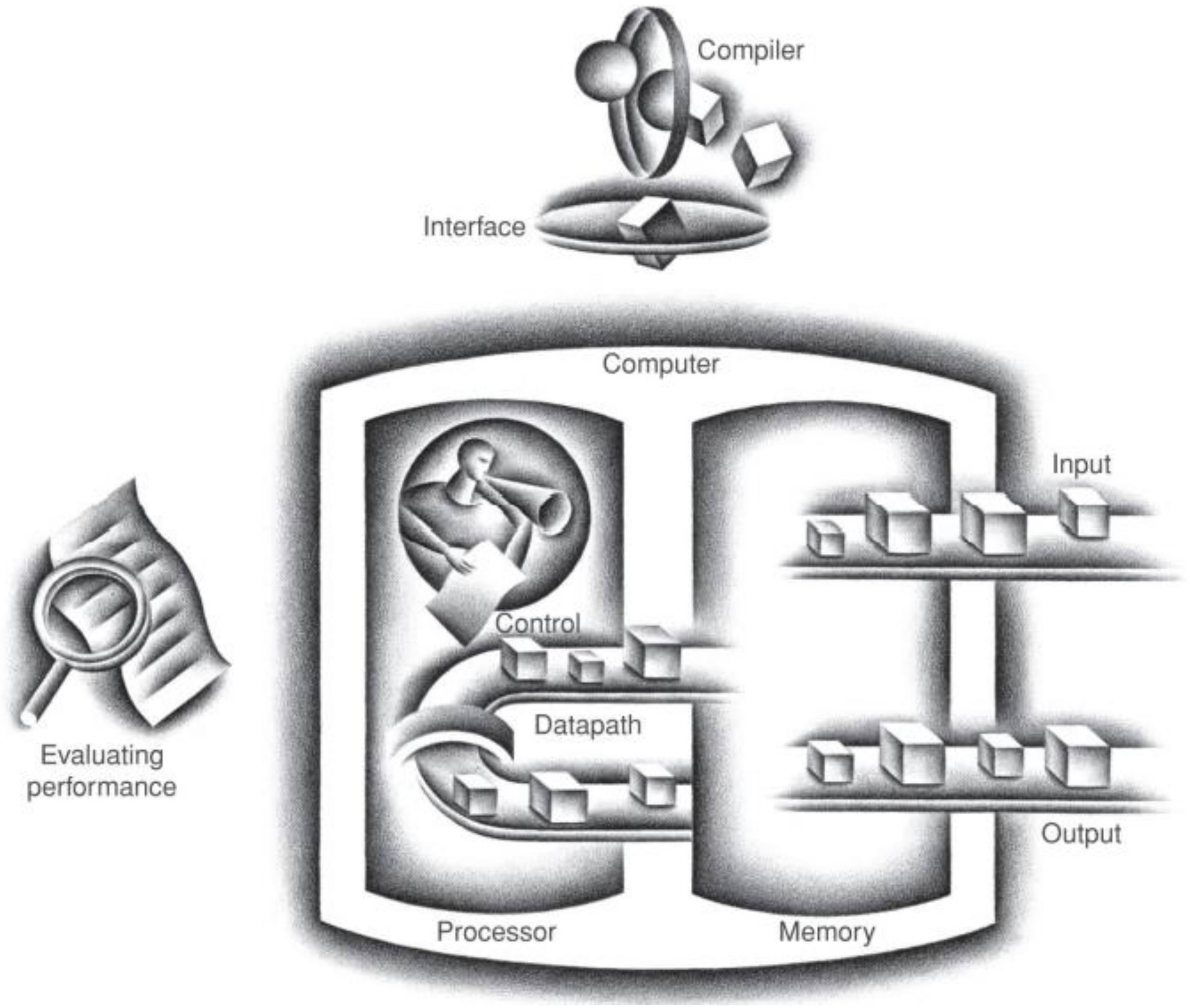


ABI provides portability across compilers



Exploring Computer Organization

Typical Computer Organization



Computer
-
Main Components

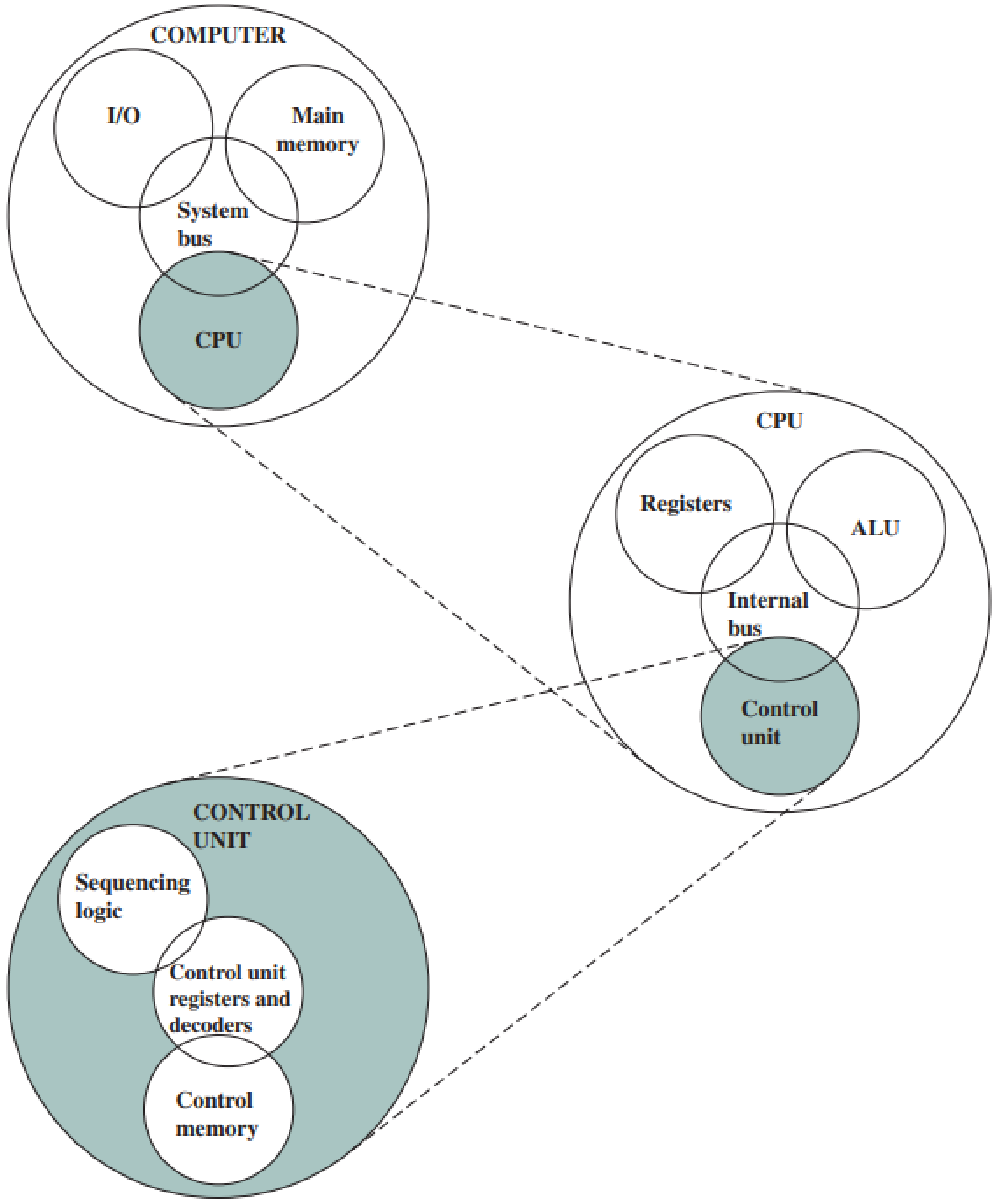


Figure 1.1 The Computer: Top-Level Structure

A Multi-core Computer

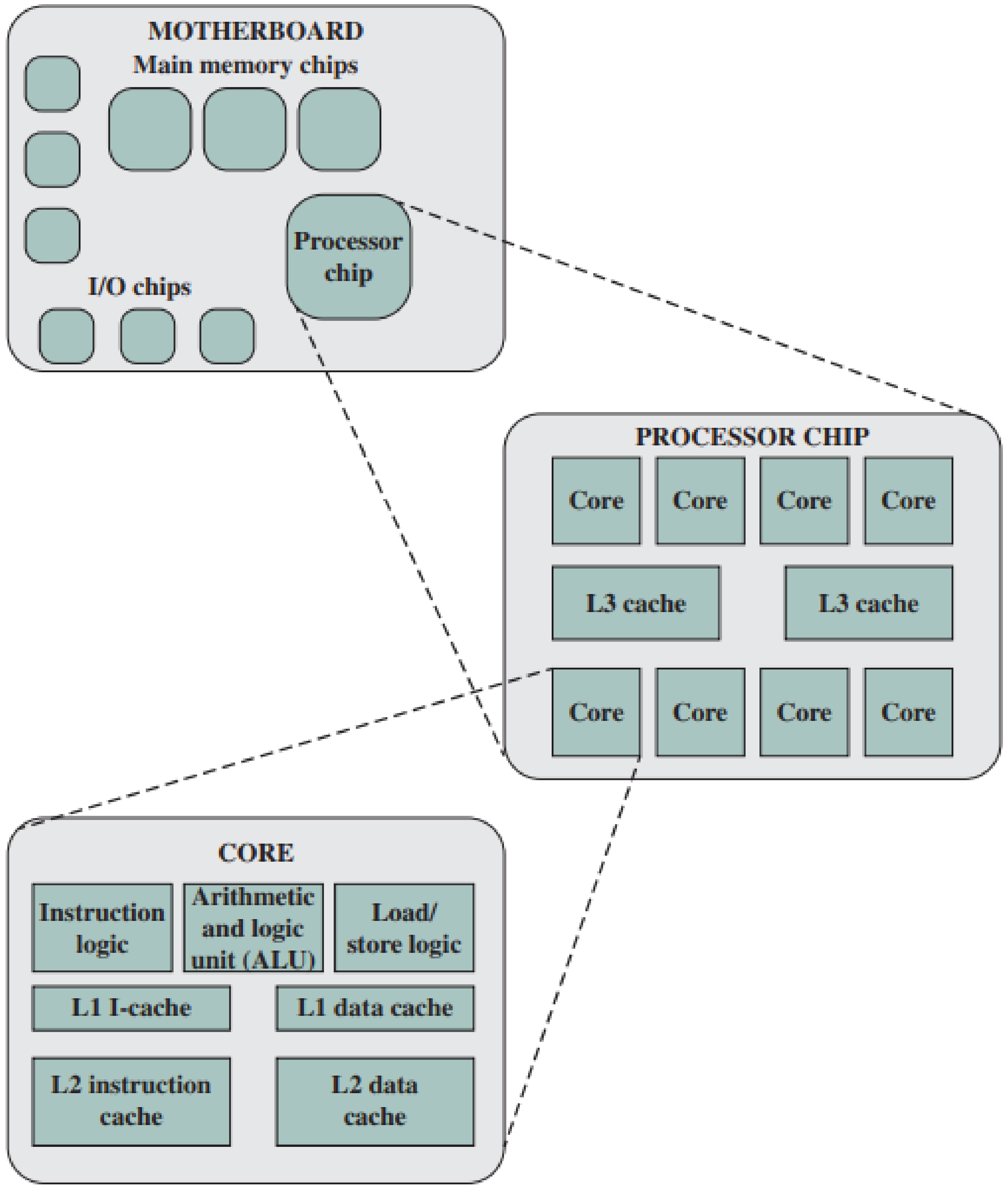
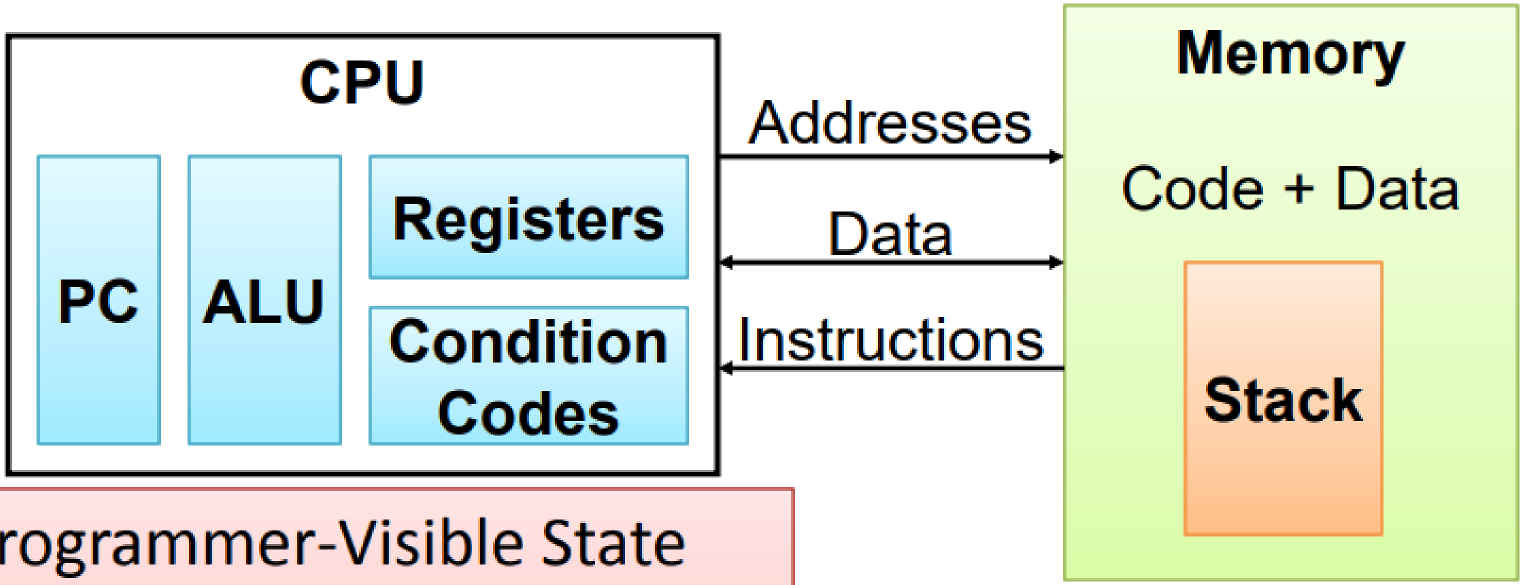


Figure 1.2 Simplified View of Major Elements of a Multicore Computer

Computer Abstraction



- Programmer-Visible State
  - PC Program Counter
  - Register File
    - Heavily used data
  - Condition Codes

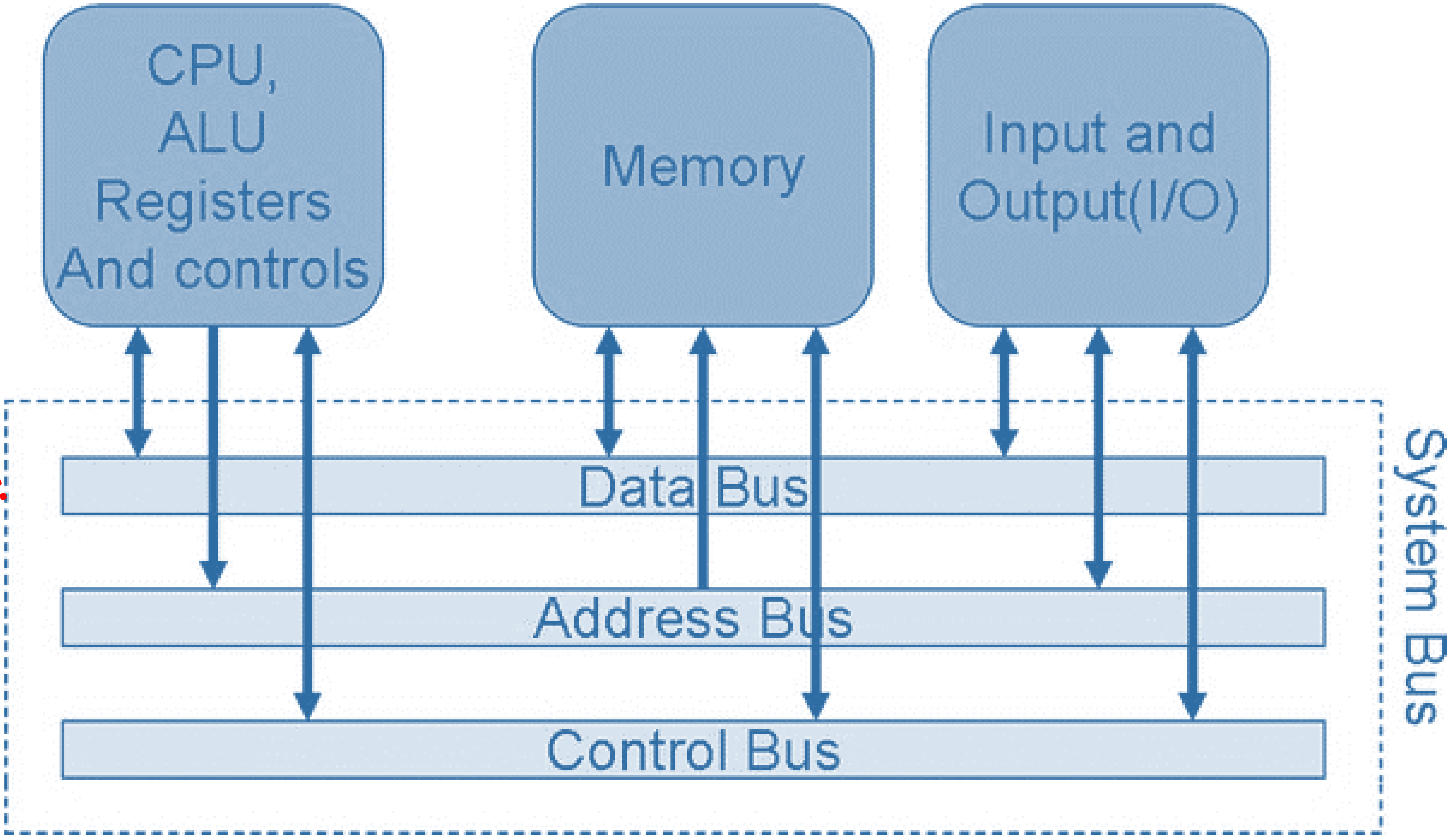
- ☐ Memory
  - Byte array
  - Code + data
  - stack

23

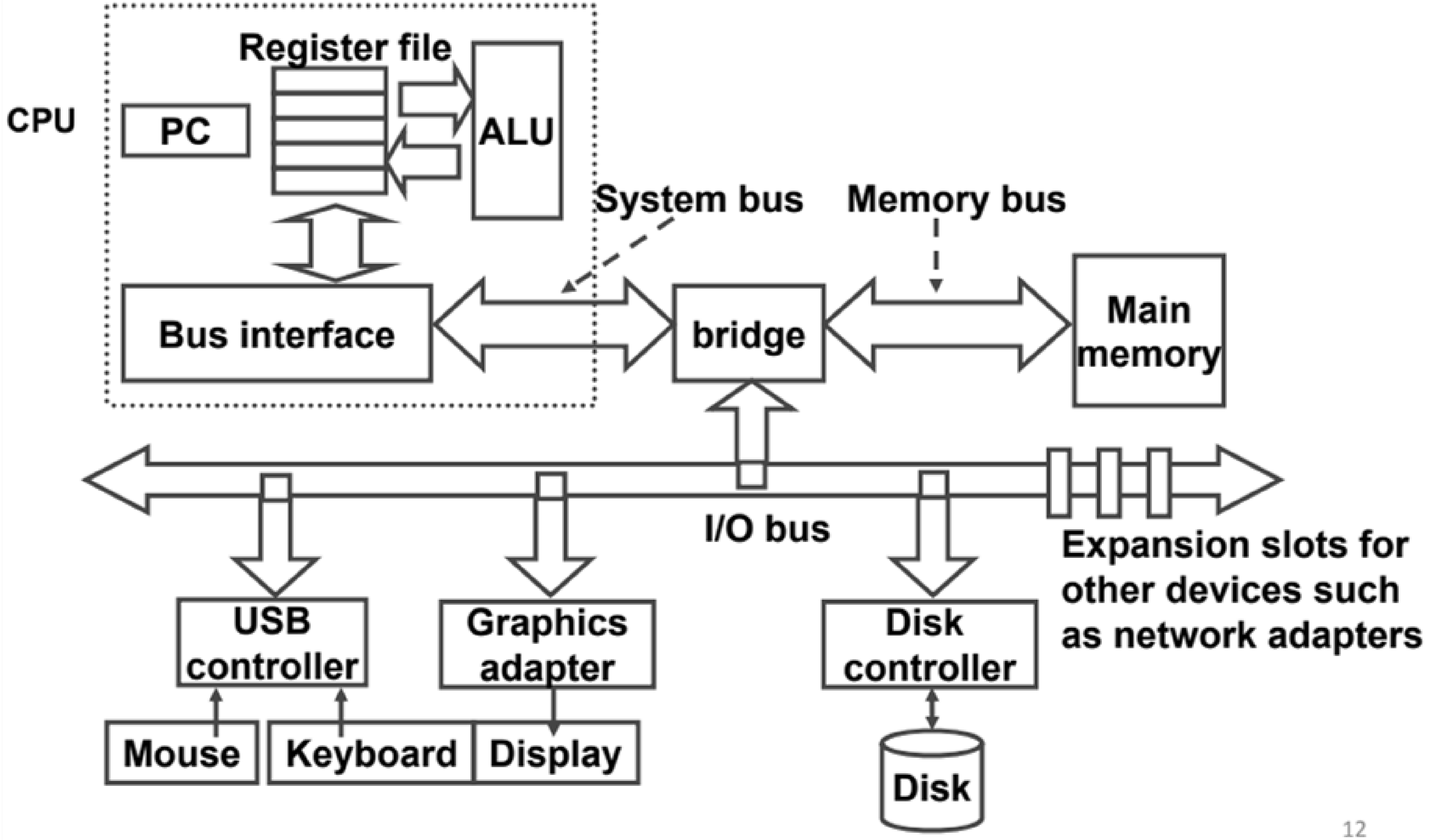
The Von-Neumann Architecture

Three Buses in Von-Neumann:

1. Data Bus
2. Address Bus
3. Control Bus



Hardware Abstraction Example



Computer Performance

Simple Computer Performance

- Time to execute a program – in seconds
- Clock Speed of Computer – MHz or GHz
- In terms of the width of Data Bus (16 bit, 32 bit, 64 bit)
- In terms of the size of different memory (8GB / 256GB)
  - Cache size
  - RAM
  - Hard disk
  - SSD, etc.
- In terms of multiple cores and multiple I/O available
- Software
  - Algorithm
  - Language / Compiler
  - Optimization – Parallel, Vector, etc.

What is MIPS?

- MIPS is a leading example of RISC Architecture
- Used in many household products such as Nintendo, Sony, Mobile phones, Routers, etc.
- Simple and Small Instruction Set

MIPS = Microprocessor without Interlock Pipeline Stages ✓

Vs

MIPS = Million Instructions Per Second

What is MFLOPS

FLOPS = Floating Point Linear Operations Per Second

$$\mathrm{MFLOPS}\ rate = \frac{Number\ of\ executed\ floating\ point\ operations\ in\ a\ program}{Execution\ time\ \times 10^6}$$

Modern notion of computer performance

user

<u>Software Performance</u>
Programming Languages
Operating System
Compilers
Algorithms
Parallelism / Threads

Software optimization

<u>Hardware Performance</u>
Processor Architecture
Vector Processing
Memory Hierarchy
Multicores
Network support

Hardware optimization

Make Common Case FAST