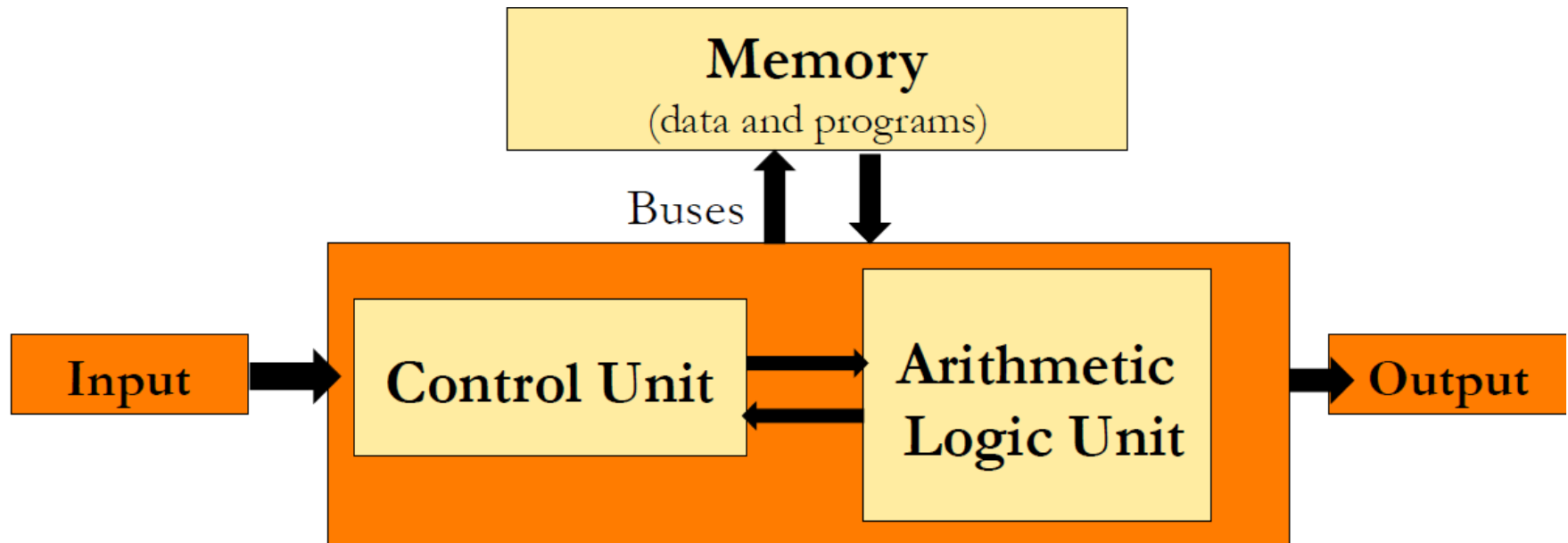

**CSIS 212 : Machine Organization & Assembly
Language
Lecture 2**

Duy Nguyen, Ph.D.
dnguyen@palomar.edu
858.204.5232 (cell)

Von Neuman Architecture



4 Basic Components of a Computer:

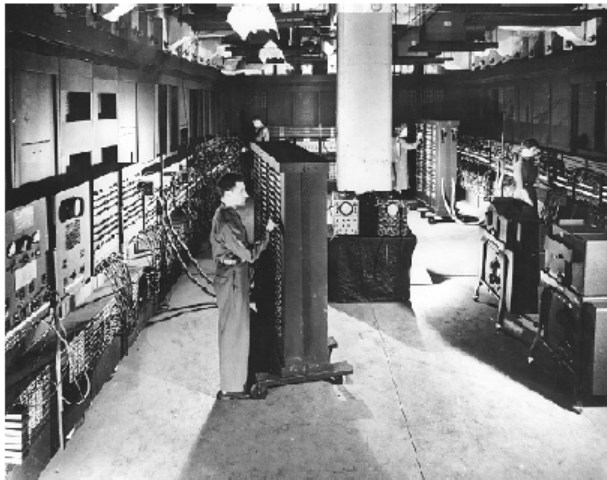
1. **Memory:** a long but finite sequence of cells (1D)
 - Each cell has a distinct address
 - Data in each cell: instruction, data or the address of another cell
2. **Control Unit:** Fetches instructions from memory and decodes them
3. **Arithmetic Logic Unit:** Does simple math operations on data
4. **Input/Output:** The connections with the outside world

The Evolution of Computing

Revolution:

1st Large Scale, General Purpose Electronic Computer

ENIAC



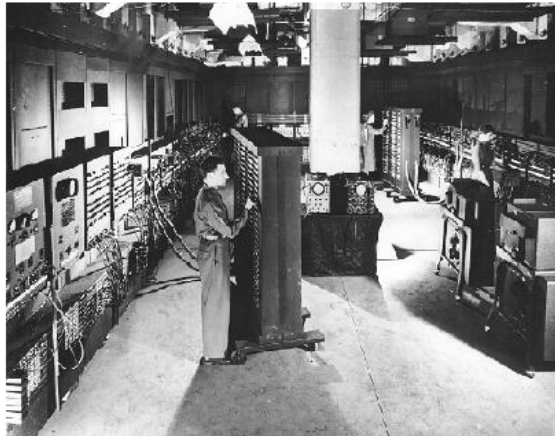
- ❖ More complex electronic circuits
- ❖ Solved more general problems
- ❖ Programming involved configuring external switches or feeding instructions through punched cards

WWII The stored program model

The Evolution of Computing

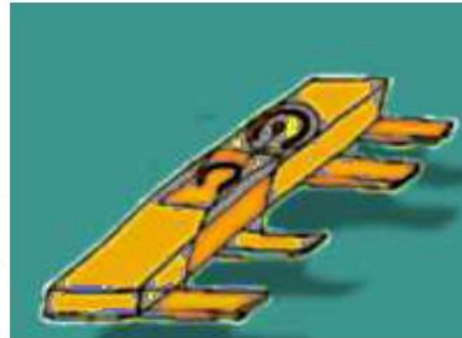
Revolution: Integrated Circuit:

Many digital operations on the same material



ENIAC Stored
Program
Model

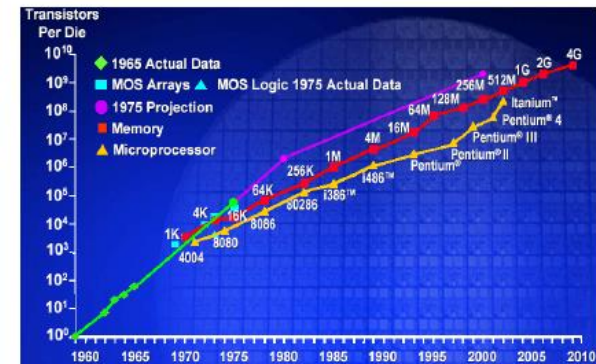
WWII



Integrated Circuit

1949

Exponential Growth of Computation

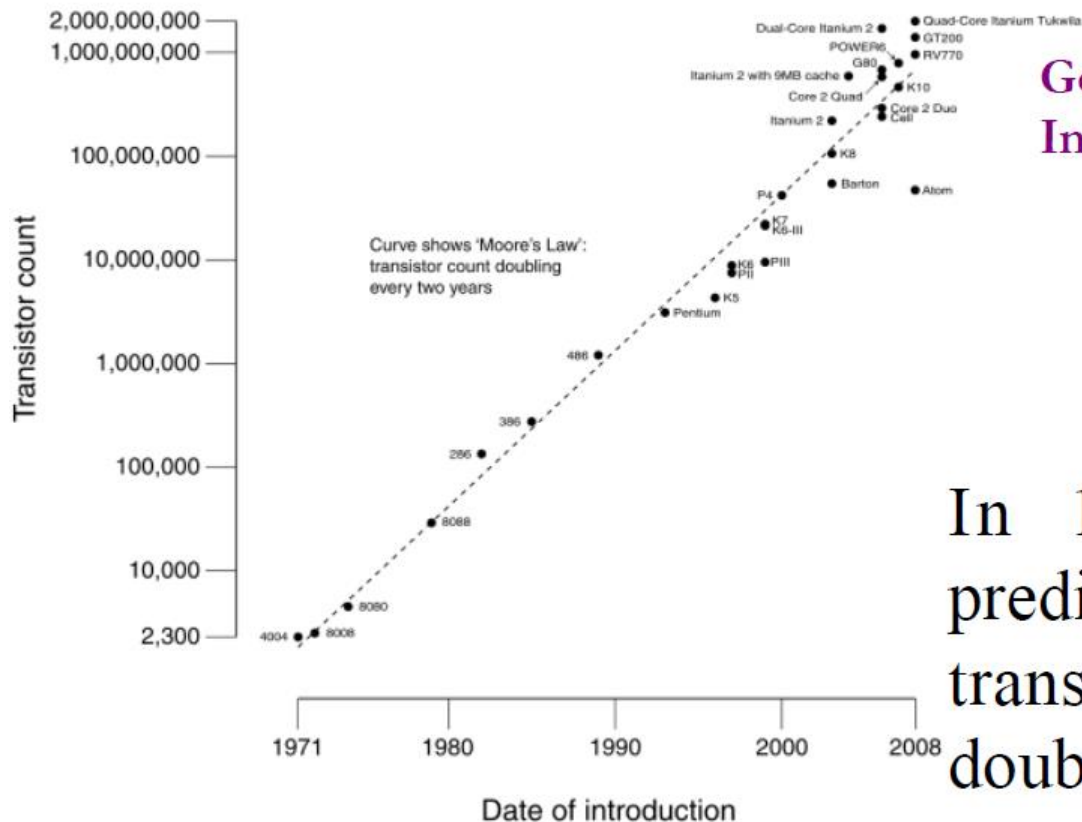


Moore's Law

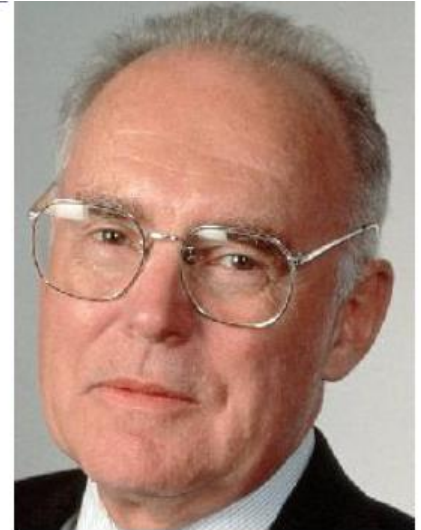
1965

Moore's Law – Technology Trends

CPU Transistor Counts 1971-2008 & Moore's Law

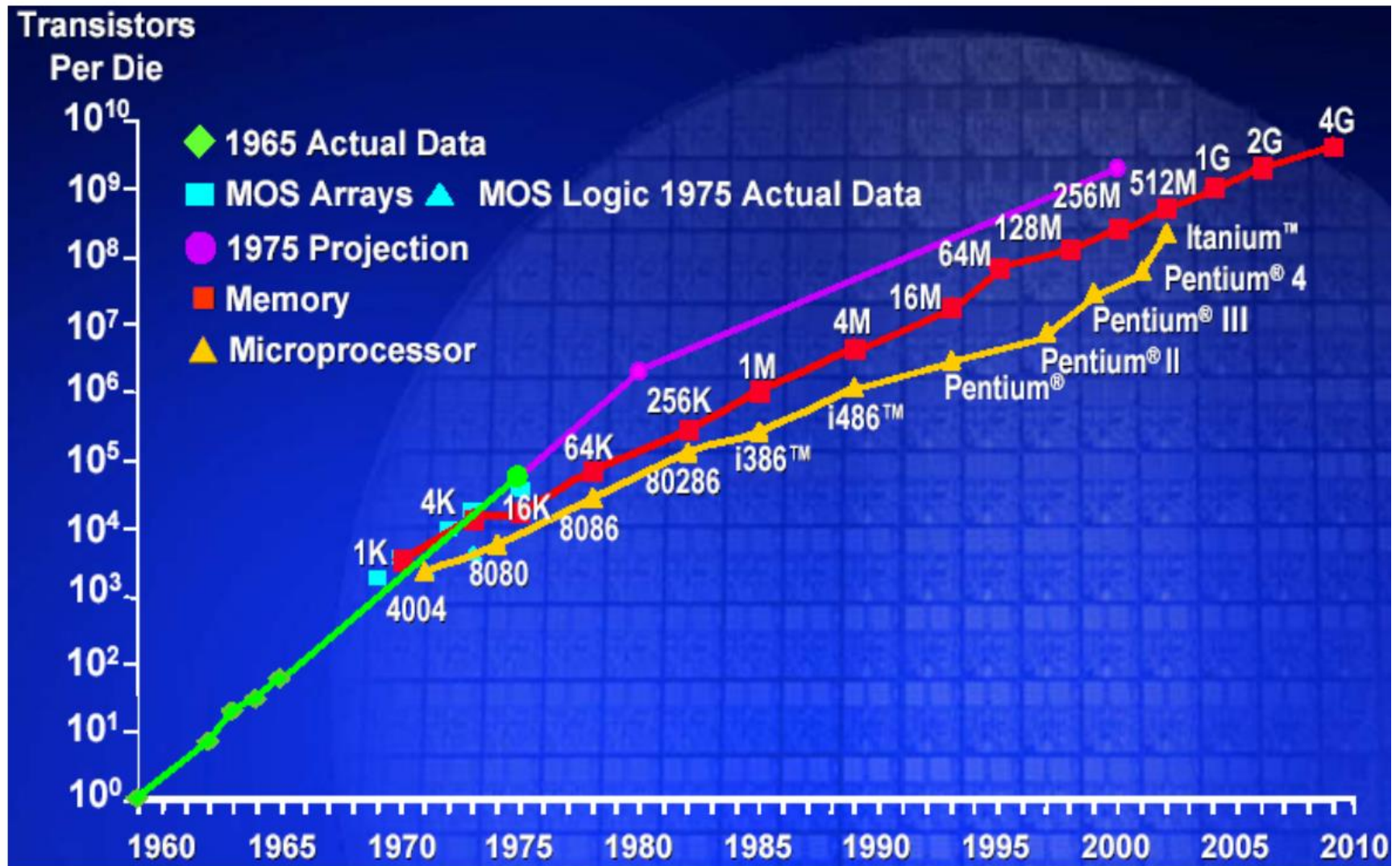


Gordon Moore
Intel Cofounder



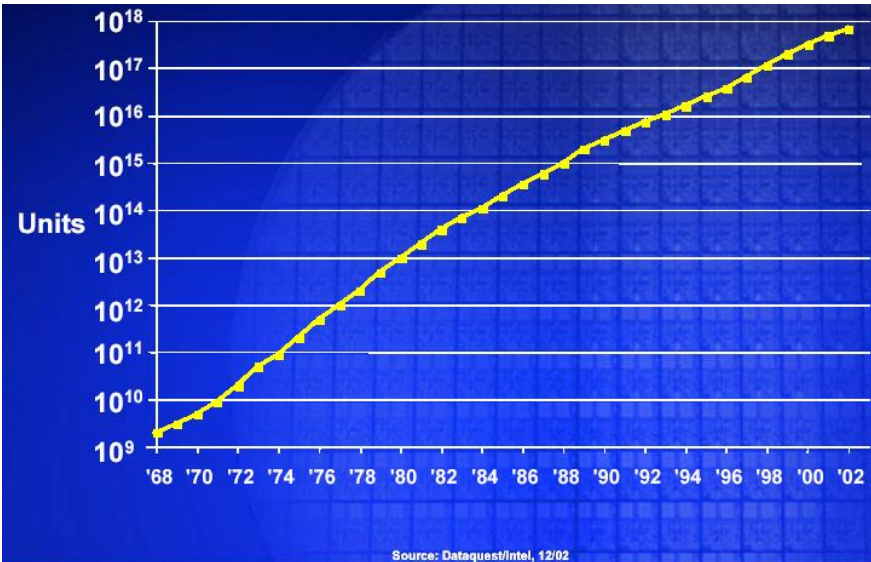
In 1965, Gordon Moore predicted that the number of transistors per chip would double every 18 months (1.5 years)

Exponential Growth in Computing

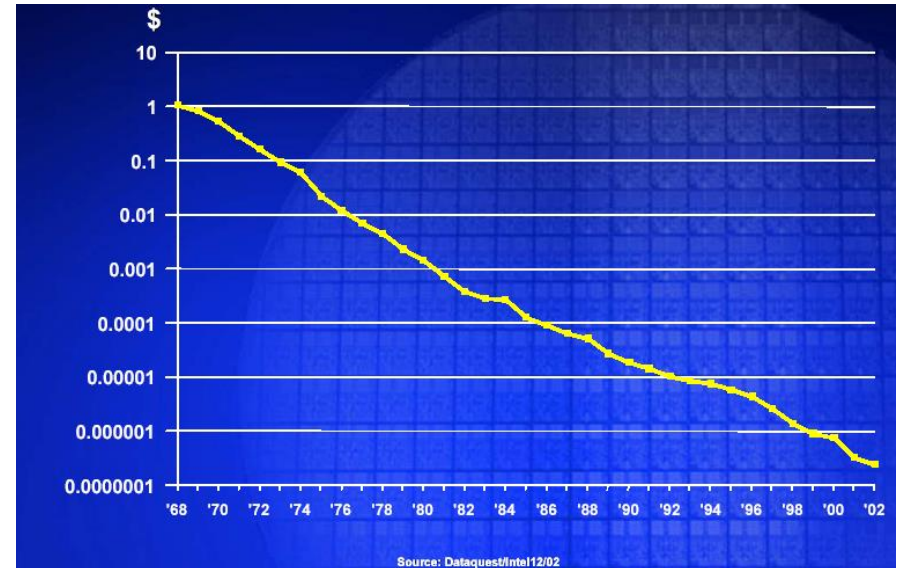


Side Effects of Moore's Law

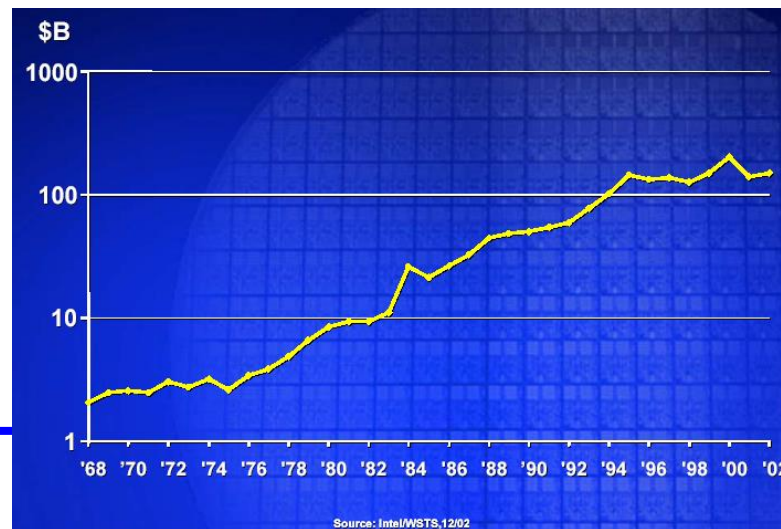
Number of Transistors Shipped/year



Average Transistor Price/year



World-wide Semiconductor Revenues



Current State of Computing

- ❖ Computers are cheap, embedded everywhere
- ❖ Transition from how to we build computers to how to we use computers

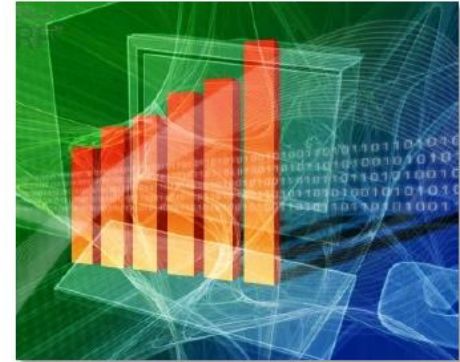


The Next Revolution

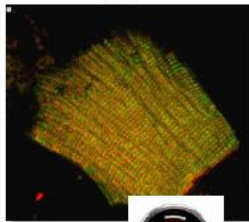
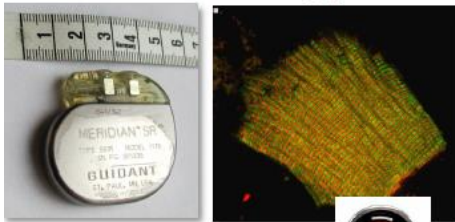
Ecological Monitoring



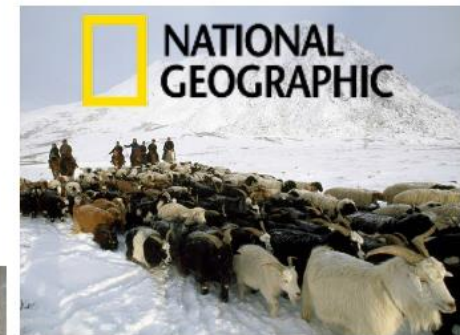
Financial Computation



Biotechnology

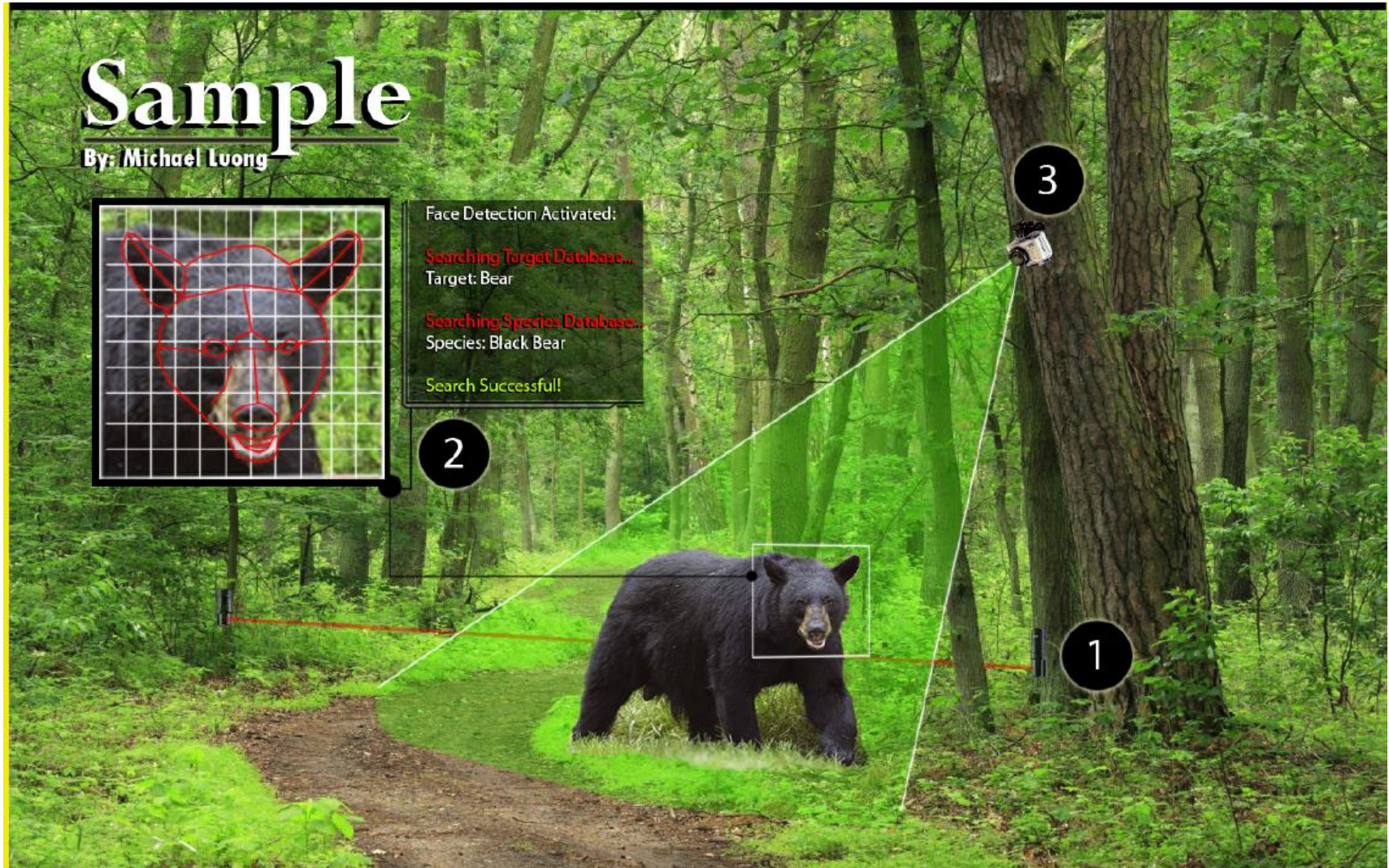


Robotics



“The use of [these embedded computers] throughout society could well dwarf previous milestones in the information revolution.”

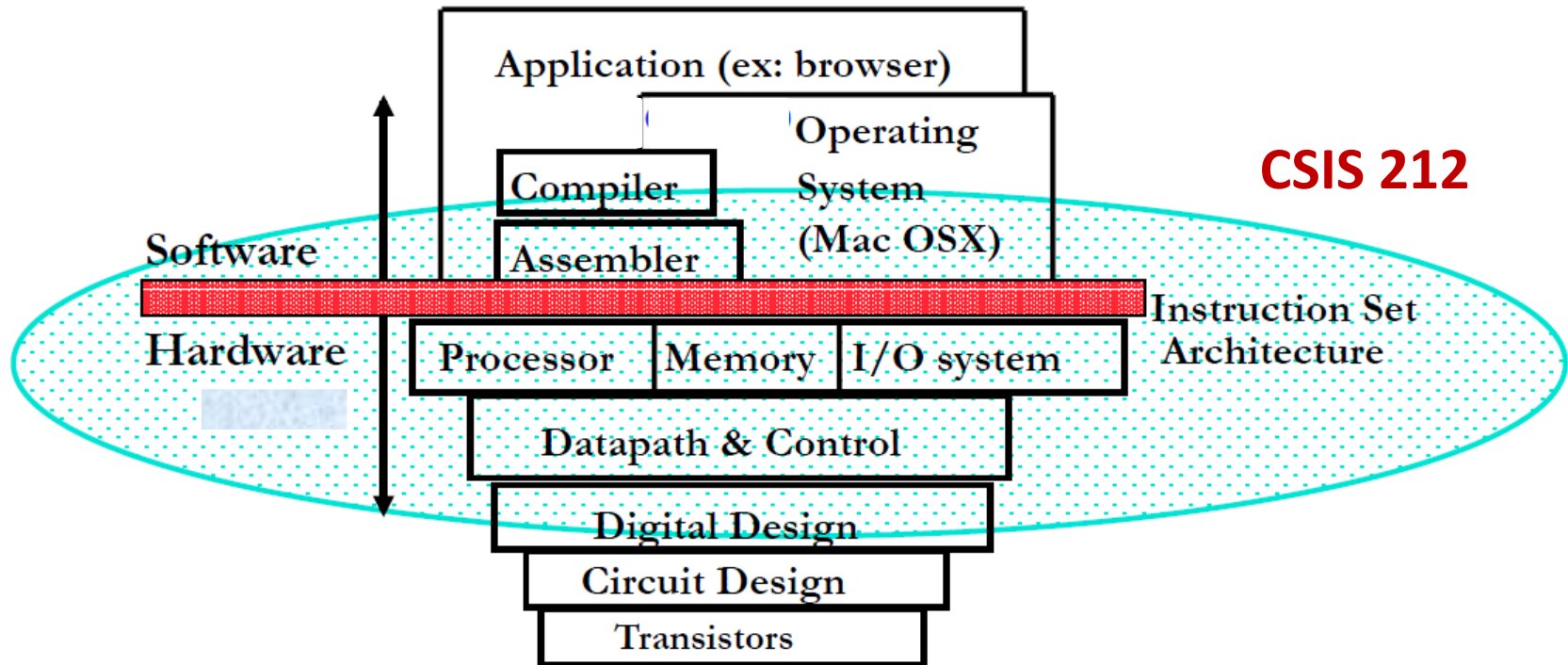
National Geographic – Engineers for Exploration



Computing Systems Moving Towards Low SWaPs

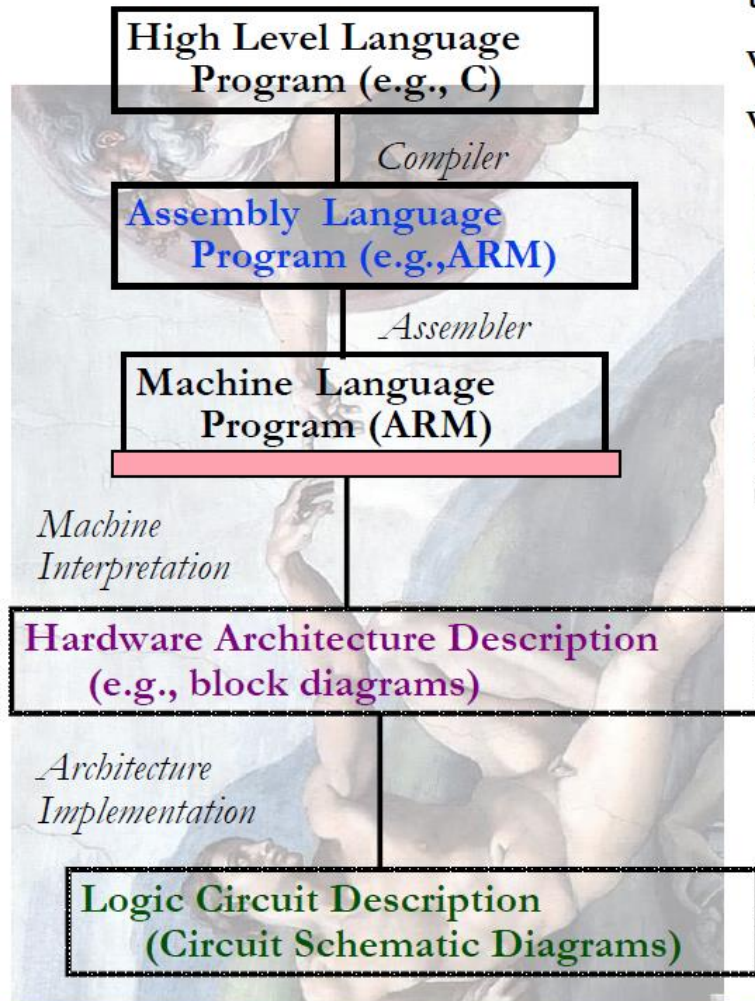
- **Increasingly smaller**
- **Higher performance**
- **More memory**
- **Lower power**
- **Embedded**
- **Everywhere**
- **... but extremely complex**

How Do We Handle Complexity?



CSIS 212 focuses on the interaction between software and hardware. It is less about writing assembler code and more about writing efficient real-time programs. This can all be done using the C language, which is much simpler to write. We will learn to write simple assembler codes in this class, primarily to get you to understand the hardware interaction so that you can write more efficient C code.

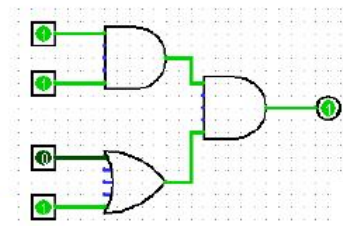
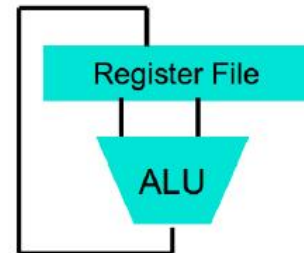
Levels of Representation



```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
ldr r0, [r2]  
ldr r1, [r2, #4]  
str r1, [r2]  
str r0, [r2, #4]
```

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



Enough of History Lesson ...

- We still need to understand the system
 - As a programmer, you will be manipulating data
 - Data can be anything: numbers (integers, floating points, characters), text, pictures, videos
 - Writing efficient code involves understanding how “data” and programs are actually represented in memory
-
- Take a break and let's talk bits and bytes: Number representation
 - And... getting around the Pi

The Raspberry Pi

- Raspbian operating system – Linux
- Pre-installed compilers that you will need: **gcc**, **g++**, and **as** (assembler)
- Text editor (do not use Word)
 - Vi, emacs, nano, IDE (Integrated Development Environment)
- IDE
 - Text editor, compiler, linker, loader, debugger
- Debugger
 - gdb

External vs. Internal Representation

- **External representation**
 - Convenient for programmer – usually decimal or hex
- **Internal representation**
 - Actual representation of data in the computer's memory and registers: always binary (1's and 0's)
- **Important to be able to convert between the decimal, binary, and hex**

Numbers: Positional Notation

- The meaning of a digit depends on its position in a number.
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ in base b represents the value

$$d_n * b^n + d_{n-1} * b^{n-1} + \dots + d_2 * b^2 + d_1 * b^1 + d_0 * b^0$$

Representation in Different Bases

- **We can represent numbers in any base**
- **Most widely used are:**
 - **Base 2 (binary)**
 - **Base 8 (octal)**
 - **Base 10 (decimal)**
 - **Base 16 (hexadecimal)**

Which Base Do We Use?

- **Decimal (base 10)**: great for humans, especially when doing arithmetic
- **Hex(base 16)**: if a human is looking at long strings of binary numbers, its much easier to convert to hex and look 4 bits/symbol
 - Terrible for arithmetic on paper
- **Binary (base 2)**: what computers use; you will learn how computers do $+$, $-$, $*$, $/$
 - To a computer, numbers always binary
 - Regardless of how number is written:
 - $32_{\text{ten}} == 32_{10} == 0x20 == 100000_2 == 0b100000$
 - Use subscripts “ten” , “hex” , “two”

Consider 101

- In base 10, it represents the number 101 (one hundred one) = $1 * 10^2 + 0 * 10^1 + 1 * 10^0$
= 101
- In base 2, $101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
= $4 + 1 = 5$
- In base 16, $101_{16} = 1 * 16^2 + 0 * 16^1 + 1 * 16^0$
= $256 + 1 = 257$

Binary: Base 2

- Used by computers
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0,1\}$, represents the value

$$d_n * 2^n + d_{n-1} * 2^{n-1} + \dots + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0$$

Decimal: Base 10

- Used by humans
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, represents the value

$$d_n * 10^n + d_{n-1} * 10^{n-1} + \dots + d_2 * 10^2 + d_1 * 10^1 + d_0 * 10^0$$

Octal: Base 8

- Allows use of 7 segment displays, nixie tubes, etc
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0,1,2,3,4,5,6,7\}$, represents the value

$$d_n * 8^n + d_{n-1} * 8^{n-1} + \dots + d_2 * 8^2 + d_1 * 8^1 + d_0 * 8^0$$




Hexadecimal – Base 16

- Like binary, but shorter!
- Each digit is a “nibble”, or half a byte
- Indicated by prefacing number with 0x
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$, represents the value
$$d_n * 16^n + d_{n-1} * 16^{n-1} + \dots + d_2 * 16^2 + d_1 * 16^1 + d_0 * 16^0$$
- Table 1.1, page 13 (ARM) shows the decimal, hex and binary equivalents of all hex digits

Base Conversions

- You should be able to convert between any two bases (see chapter 2 in textbook)
 - Unsigned decimal to Binary
 - Unsigned decimal to Hex
 - Hex to Binary
- Practice some conversions in class...

Bits Can Represent Anything

- Numbers:
 - integers (signed, unsigned)
 - floats
- Characters:
 - 26 letters using 5 bits ($2^5 = 32$)
 - upper/lower case + punctuation: 128 characters using 7 bits (“ASCII”)
 - standard code to cover all the world’s languages 8,16,32 bits (“Unicode”) www.unicode.com
- Logical values:
 - 0: False, 1: True
- Colors: Example  *Red (00)*  *Green (01)*  *Blue (11)*
- locations / addresses, instructions
- **MEMORIZE:** N bits can be used to represent at most 2^N things

How We Store Numbers

- Binary numbers in memory are stored using a finite, fixed number of bits typically:
 - 8 bits (byte)
 - 16 bits (half word)
 - 32 bits (word)
 - 64 bits (double word or quad)
- If positive pad extra digits with leading 0s
- A byte representing $4_{10} = 00000100$

Integer Representation

We will look at representation for:

- Unsigned numbers
 - Only positive values
 - All the bits determine the magnitude of the positive number
- Signed numbers
 - These can be positive or negative
 - We cannot use all the bits to just represent the magnitude

Unsigned Numbers

- Some types of numbers, such as memory addresses, will never be negative
- Some programming languages reflect this with types such as “unsigned int”, which only hold positive numbers
- In an unsigned byte, values will range from 0 to 255

How To Represent Signed Numbers?

- So far, unsigned numbers
- Obvious solution: define leftmost bit to be sign!
- Representation called sign and magnitude
 - Have a separate bit for sign (Most significant Bit MSB)
 - Set it to 0 for positive, and 1 for negative

Number Representation

- Just what we do with numbers!

- Add them

- Subtract them

- Multiply them

- Divide them

- Compare them

- Example: $10 + 7 = 17$

- ...so simple to add in binary that we can build circuits to do it!

- subtraction just as you would in decimal

- Comparison: How do you tell if $X > Y$?

$$\begin{array}{r} \overset{\textcolor{red}{1}}{1} \overset{\textcolor{red}{1}}{1} \\ \\ + \\ \hline \end{array}$$

$$\begin{array}{r} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{array}$$

Shortcomings of Sign and Magnitude Representation

- Arithmetic circuit complicated
 - Special steps depending whether signs are the same or not
- Also, two zeros
 - $0x00000000 = +0_{\text{ten}}$
 - $0x80000000 = -0_{\text{ten}}$
- Therefore sign and magnitude abandoned !

Two's Complement

- Flip all the bits and add 1
- For n bits, in unsigned version $-x = 2^n - x$
- Can represent -128 to 127 in a byte
- Only one zero (00000000)
- Used in modern computers

Negating in Hexadecimal

- Obtain the hex representation
- Replace each digit by $(15 - \text{digit})$
- Add one

Addition and Subtraction

- **Positive and negative numbers are handled in the same way**
- **The carry out from the most significant bit is ignored**
- **To perform the subtraction $A - B$, compute $A + (\text{two's complement of } B)$**

Overflow

- **Overflow occurs when an addition or subtraction results in a value which cannot be represented using the number of bits available**
- **In that case, the algorithms we have been using produce incorrect results**

Handling Overflow

- **Hardware can detect when overflow occurs**
- **Software may or may not check for it**
 - C and Java don't!
- **We will write assembler codes in this class to detect overflow as an exercise**

Detecting Overflow in Hardware

- In the ARM architecture, there is a special register: the Current Program Status Register that has 4 bits (flags) that are set after an arithmetic operation is executed based on what 'happened'.



- N flag: 1 if result is negative
- Z flag: 1 if result is zero
- V flag: 1 if oVerflow (more next slide)
- C flag: 1 if there is a Carry out of MSB

How To Detect Overflow

- Overflow occurs if adding two negative numbers produces a positive result or if adding two positive numbers produces a negative result.
- On an addition, an overflow occurs if and only if the carry into the sign bit differs from the carry out from the sign bit.
- Overflow detected using V flag which is set as:
 $C_{in} \text{ XOR } C_{out}$

In Lab Exercises

- **Practice writing, compiling, and executing codes on the Pi**

Reading Assignment

- **Read chapters 1-3 in the Plantz online textbook**