



Computer Architecture Overview

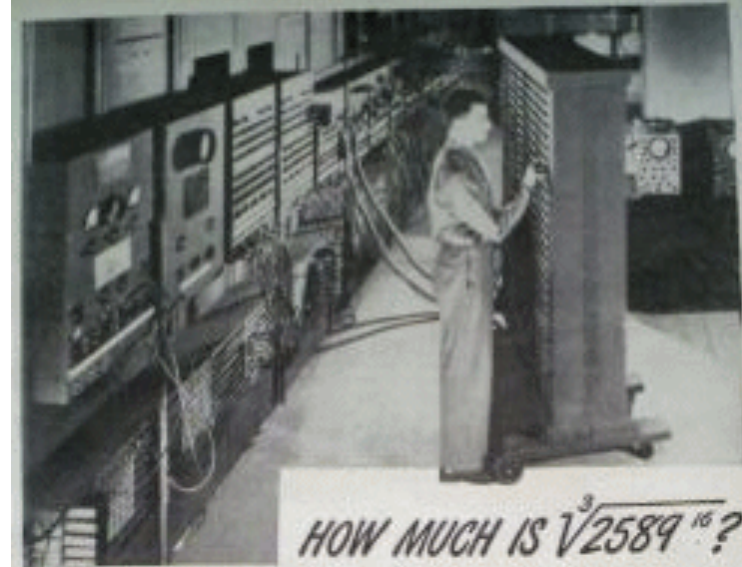
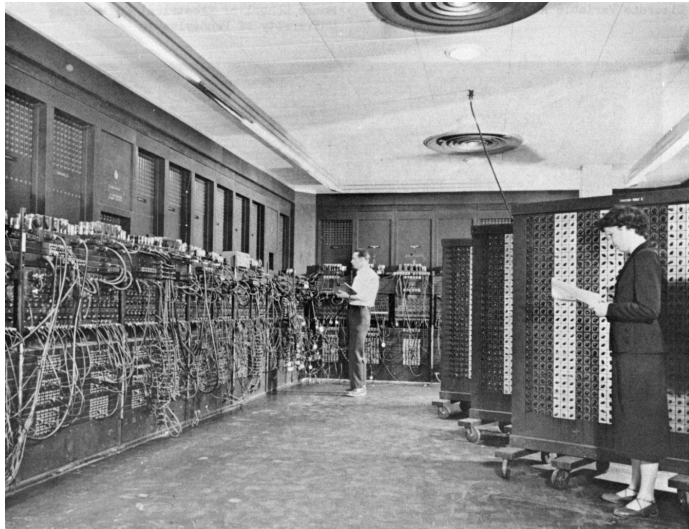
**ICS332
Operating Systems**

Henri Casanova (henric@hawaii.edu)

1946: The ENIAC

- The ENIAC (Electronic Numerical Integrator and Computer) was unveiled in 1946
- The first all-electronic, general-purpose digital computer that could be (re)programmed
- Main sponsor: University of Pennsylvania / Ballistic Research Laboratory
 - Designed by Mauchly and Eckert
- First programmers: a team of 6 women during WWII
- Specs
 - 17,468 vacuum tubes
 - 1,800 sqft
 - 30 tons
 - 174 kilowatt of power
 - 1,000-bit memory

The ENIAC in Pictures



HOW MUCH IS $\sqrt[3]{2589^{16}}$?

The Army's ENIAC can give you the answer in a fraction of a second!

Think that's a stumper? You should see some of the ENIAC's problems! Tests tellers that if put to paper would run off this page and feet beyond . . . addition, subtraction, multiplication, division—square root, cube root, any root. Solved by an incredibly complex system of circuits operating 10,000 electronic tubes and tipping the scales at 30 tons!

The ENIAC is symbolic of many amazing Army devices with a brilliant future for you! The new Regular Army needs men with aptitude for scientific work, and as one of the first trained in the post-war era, you stand to get in on the ground floor of important jobs

YOUR REGULAR ARMY SERVES THE NATION AND MAKING IN WAR AND PEACE

which have never before existed. You'll find that an Army career pays off.

The most attractive fields are filling quickly. Get into the army while the getting's good! 1½, 2 and 3 year enlistments are open in the Regular Army to ambitious young men 18 to 34 (17 with parents' consent) who are otherwise qualified. If you enlist for 3 years, you may choose your own branch of the service, if those still open. Get full details at your nearest Army Recruiting Station.

A GOOD JOB FOR YOU
U. S. Army
CHOOSE THIS
FINE PROFESSION NOW!

OCTOBER 1946

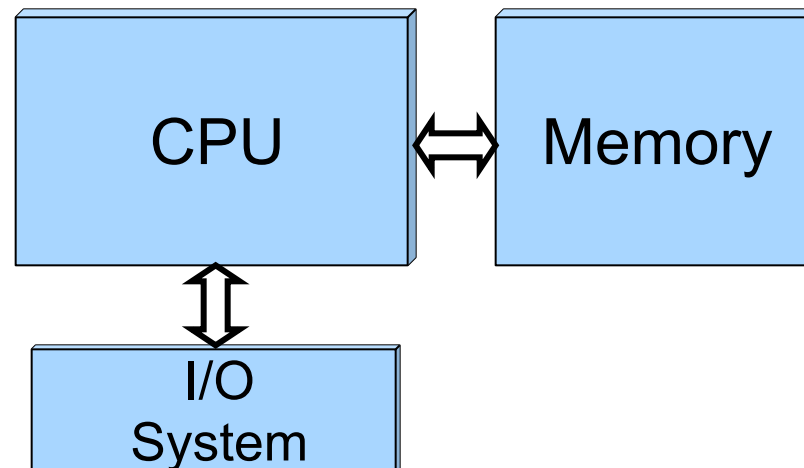


The Von Neumann Architecture

- ENIAC design finalized in 1943
- In 1944, **John von Neumann** learned about ENIAC and joined the group.
- He wrote a memo about computer architecture, formalizing the ideas that came out of ENIAC and transferring them to a wider audience
- This became the Von Neumann architecture, which we still use today...

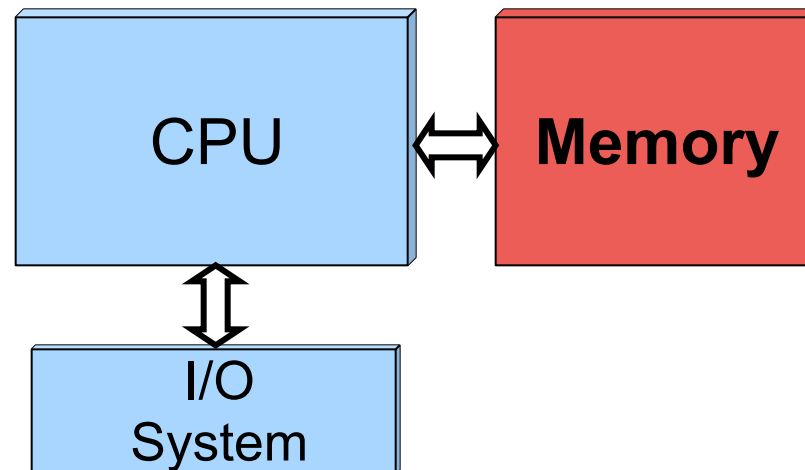
The Von-Neumann Architecture

- Three hardware systems
 - A **Central Processing Unit (CPU)**: performs operations and controls the sequence of operations
 - A **Memory Unit**: stores **both** code and data
 - **Input/Output** devices to interact with the machine
- Computers today are still very close to this basic architecture



The Von-Neumann Architecture

- Three hardware systems
 - A **Central Processing Unit (CPU)**: performs operations and controls the sequence of operations
 - A **Memory Unit**: stores **both** code and data
 - **Input/Output** devices to interact with the machine
- Computers today are still very close to this basic architecture



The Memory Unit

- Called “Memory” or RAM (Random Access Memory)
- All “information” in the computer is in binary form
 - Since Claude Shannon’s M.S. thesis in the 1930’s
 - 0: zero voltage, 1: positive voltage (e.g., 5V)
 - bit (binary digit): the smallest unit of information (0 or 1)
- The basic unit of memory is a **byte**
 - 1 Byte = 8 bits, e.g., “0101 1101”
 - 1 KiB = 2^{10} byte = 1,024 bytes
 - 1 MiB = 2^{10} KiB = 2^{20} bytes (~ 1 Million)
 - 1 GiB = 2^{10} MiB = 2^{30} bytes (~ 1 Billion)
 - 1 TiB = 2^{10} GiB = 2^{40} bytes (~ 1 Trillion)
 - 1 PiB = 2^{10} TiB = 2^{50} bytes (~ 1000 Trillion)
 - 1 EiB = 2^{10} PiB = 2^{60} bytes (~ 1 Million Trillion)
 - ...

Data Stored in Memory

- Each byte in memory is labeled by a unique **address**
- An address is a number that identifies the memory location of each byte in memory
 - e.g., the byte at address 3 is 00010010
 - e.g., the byte at address 241 is 10110101
- Typically, we write addresses in binary as well
 - e.g., the byte at address 00000011 is 00010010
 - e.g., the byte at address 11110001 is 10110101
- We talk of a **byte-addressable memory**
- All addresses in RAM have the same number of bits
 - e.g., 8-bit addresses
- The processor has instructions that say “Read the byte at address X and give me its value” and “Write some value into the byte at address X”
- The Memory Unit (Bus + RAM) has the hardware to do this

Example Byte-Addressable RAM with 16-bit addresses

address

0000	0000	0000	0000
0000	0000	0000	0001
0000	0000	0000	0010
0000	0000	0000	0011
0000	0000	0000	0100
0000	0000	0000	0101
0000	0000	0000	0110
0000	0000	0000	0111
0000	0000	0000	1000

...

content

0110	1110
1111	0100
0000	0000
0000	0000
0101	1110
1010	1101
0000	0001
0100	0000
1111	0101
...	

Example Byte-Addressable RAM with 16-bit addresses

address	content
0000 0000 0000 0000	0110 1110
0000 0000 0000 0001	1111 0100
0000 0000 0000 0010	0000 0000
0000 0000 0000 0011	0000 0000
0000 0000 0000 0100	0101 1110
0000 0000 0000 0101	
0000 0000 0000 0110	
0000 0000 0000 0111	
0000 0000 0000 1000	1111 0101
...	...

**At address 0000 0000 0000 0010
the content is 0000 0000**

Example Byte-Addressable RAM with 16-bit addresses

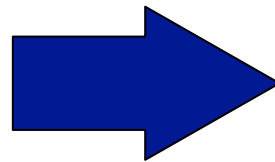
address	content
0000 0000 0000 0000	0110 1110
0000 0000 0000 0001	1111 0100
0000 0000 0000 0010	0000 0000
0000 0000 0000 0011	0000 0000
0000 0000 0000 0100	0101 1110
0000 0000 0000 0101	
0000 0000 0000 0110	
0000 0000 0000 0111	
0000 0000 0000 1000	1111 0101
...	...

**At address 0000 0000 0000 0100
the content is 0101 1110**

Example

- Let's consider machine with 8-bit addresses, and a program that does: "At address 1000 0000, store the address of the first 7 (000 0111) in memory"

Address		Content	
0000	0000	0000	0011
0000	0001	0010	1010
0000	0010	1111	0101
0000	0011	1000	0000
0000	0100	0000	0111
0000	0101	1011	1111
0000	0110	1111	1111
0000	0111	1101	0000
...		...	
1000	0000	1010	1101
1000	0001	1011	0000



Address		Content	
0000	0000	0000	0011
0000	0001	0010	1010
0000	0010	1111	0101
0000	0011	1000	0000
0000	0100	1100	0101
0000	0101	1011	1111
0000	0110	1111	1111
0000	0111	1101	0000
...		...	
1000	0000	0000	0100
1000	0001	1011	0000

Example

- Let's consider machine with 8-bit addresses, and a program that does: "At address 1000 0000, store the address of the first 7 (000 0111) in memory"

Address	Content
0000 0000	0000 0011
0000 0001	0010 1010
0000 0010	111 0101
0000 0011	000 0000
0000 0100	100 0101
0000 0101	011 1111
0000 0110	111 1111
0000 0111	1101 0000
...	...
1000 0000	1010 1101
1000 0001	1011 0000

Address	Content
0000 0000	0000 0011
0000 0001	010 1010
0000 0010	111 0101
0000 0011	000 0000
0000 0100	100 0101
0000 0101	011 1111
0000 0110	111 1111
0000 0111	1101 0000
...	...
1000 0000	0000 0100
1000 0001	1011 0000

Anybody can tell me the name of the concept this program implements?

Indirection

- An address is just information (a number)
- In the previous example, the program implemented **indirection**
 - The memory content at a memory location is the address of another memory location
 - We call this content a pointer / reference
 - It's just an address, that is, just a number
 - At that address there is some content that presumably we care about
 - In the example, the value '7'
 - But if it was another address, then we'd have a double indirection, and so on...

Address vs. Values

- It's the job of the programmer to know what memory content means (to the CPU, it's just a bunch of numbers)
- This is a well-known difficulty when writing assembly (ICS312/ICS331)
- High-level programming languages do all this for you, but in C of course you can do whatever you want
 - e.g., on a 64-bit architecture a C pointer is simply an unsigned long

```
unsigned long x = 42;
```

```
int *ptr = (int *)x; // bogus pointer
```

Hardware Instructions

■ Some high-level pseudo-code

Step 1) Set the content of variable A to the content at address 1000 0000
Step 2) Set the content of variable B to the content at address 1000 0001
Step 3) Add A and B together and store the result in A
Step 4) Set the content at address 1000 0001 to the contents of A
Step 5) Go back to Step 1

■ Assembly translations

```
// MIPS-like (ICS 331)
S1: LOAD A, (1000 0000)
S2: LOAD B, (1000 0001)
S3: ADD A,B
S4: STORE A, (1000 0010)
S5: JMP S1
```

```
// x86-like (ICS 312)
S1: MOV AL, [1000 0000]
S2: MOV BL, [1000 0001]
S3: ADD AL, BL
S4: MOV [1000 0010], AL
S5: JMP S1
```

Instruction Encoding

- Instructions are encoded in binary (the “binary code”), based on the specifications of the microprocessor
- Here are some x86 instruction encodings

Instruction	Encoding (hex)	Size
ADD EAX, 1	83C001	3 bytes
ADD EAX, -1	83C0FF	3 bytes
ADD EAX, -100000	056079FEFF	5 bytes
ADD EAX, EBX	01D8	2 bytes

- More instruction leads to larger executable binaries
 - An **assembler** transforms assembly code into binary code, so assembly programmers typically don't know the binary code for instructions

Address Space

- A program is stored in RAM

code {	Address	Content	Meaning
	0000 0000	83	ADD EAX, 1
	0000 0001	C0	
	0000 0010	01	ADD EAX, EBX
	0000 0011	D8	
	0000 0100	05	ADD EAX, -10000
	0000 0101	60	
	0000 0110	79	
	0000 0111	FE	
	0000 1000	FF	

Address Space

- A program is stored in RAM **along with data**

	Address	Content	Meaning
code	0000 0000	83	ADD EAX, 1
	0000 0001	C0	
	0000 0010	01	ADD EAX, EBX
	0000 0011	D8	
	0000 0100	05	ADD EAX, -10000
	0000 0101	60	
	0000 0110	79	
	0000 0111	FE	
	0000 1000	FF	

data	1000 0010	61	Character 'a'
	1000 0011	00	Character '\0'
	1000 0100	FF	Integer -1
	1000 0101	FF	
	1000 0110	FF	
	1000 0111	FF	

Address Space

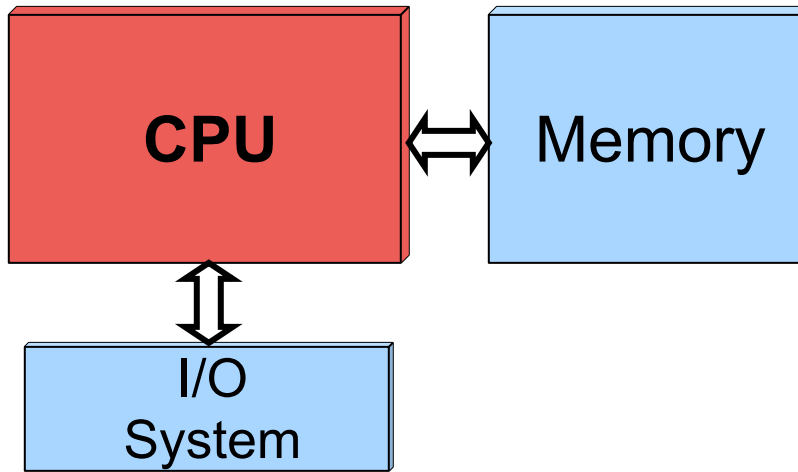
- A program is stored in RAM **along with data**

	Address	Content	Meaning
code {	0000 0000	83	ADD EAX, 1
	0000 0001	C0	
	0000 0010	01	ADD EAX, EBX
	0000 0011	D8	
	0000 0100	05	ADD EAX, -10000
	0000 0101	60	
	0000 0110	79	
	0000 0111	FE	
	0000 1000	FF	

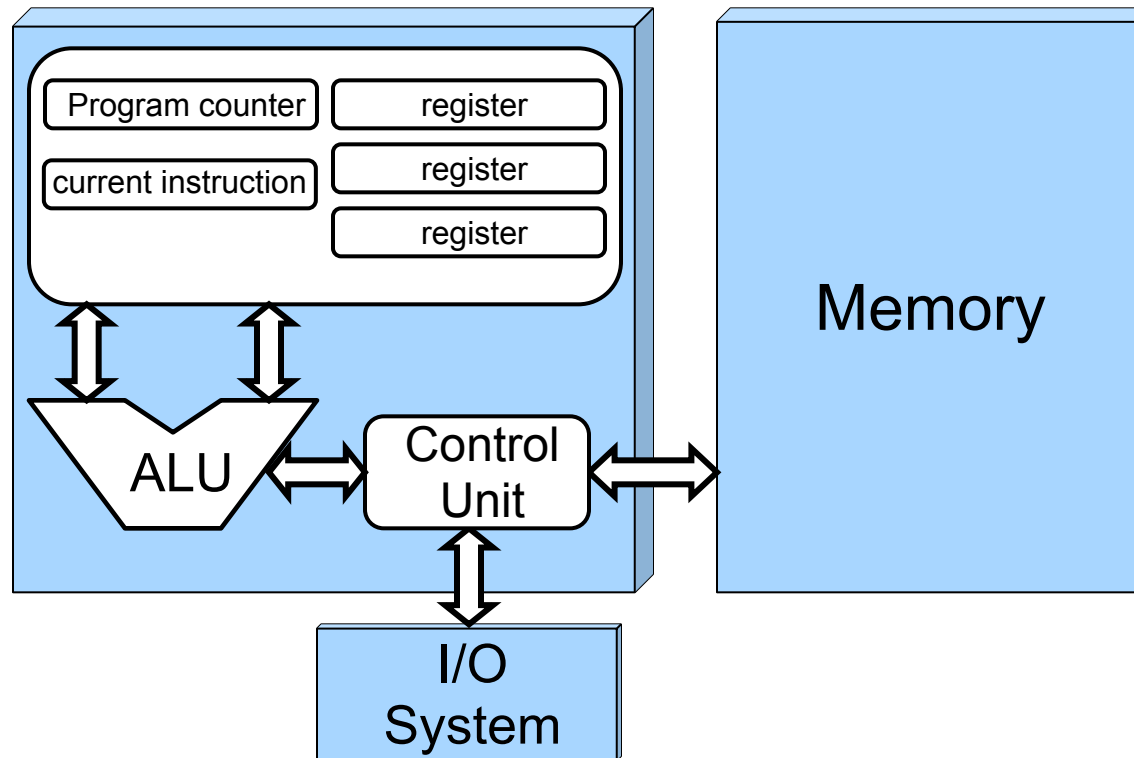
data {	1000 0010	61	Character 'a'
	1000 0011	00	Character '\0'
	1000 0100	FF	Integer -1
	1000 0101	FF	
	1000 0110	FF	
	1000 0111	FF	

- All the bytes in RAM that “belong” to the program are called the program’s **address space**
- This address space contains the code and the data
 - And other things we’ll see later

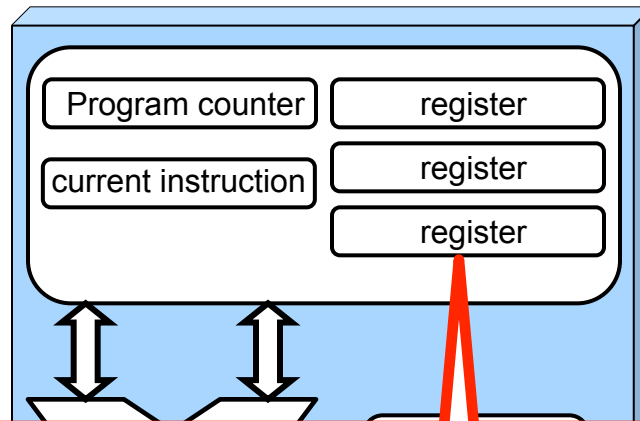
The CPU



What's in the CPU?



What's in the CPU?



Registers: values that hardware instructions work with

Data can be loaded from memory into a register

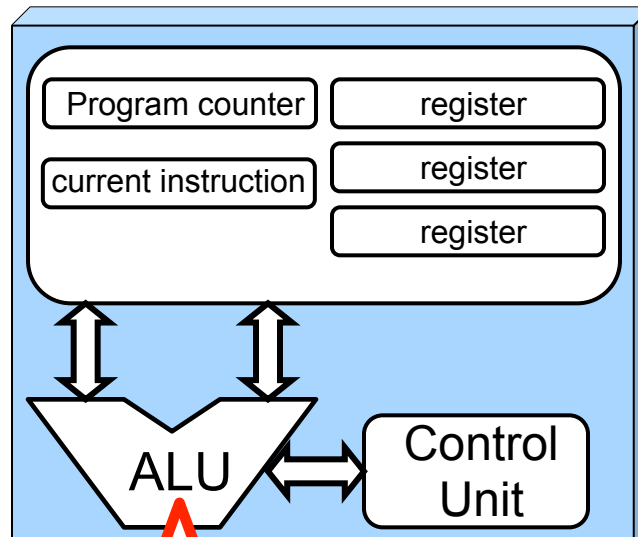
Data can be stored from a register back into memory

Operands and results of computations are ALL in registers

Accessing a register is really fast

There is a limited number of registers (which will make our life a bit difficult)

What's in the CPU?

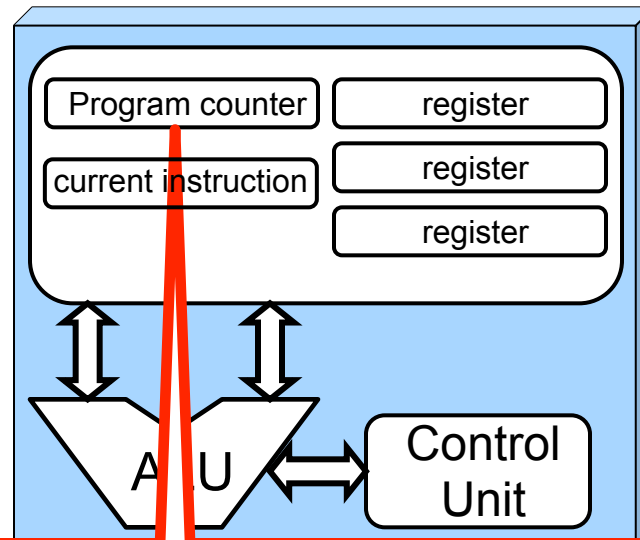


Arithmetic and Logic Unit: what you do computation with

used to compute a value based on current register values and store the result back into a register

+, *, /, -, OR, AND, XOR, etc.

What's in the CPU?

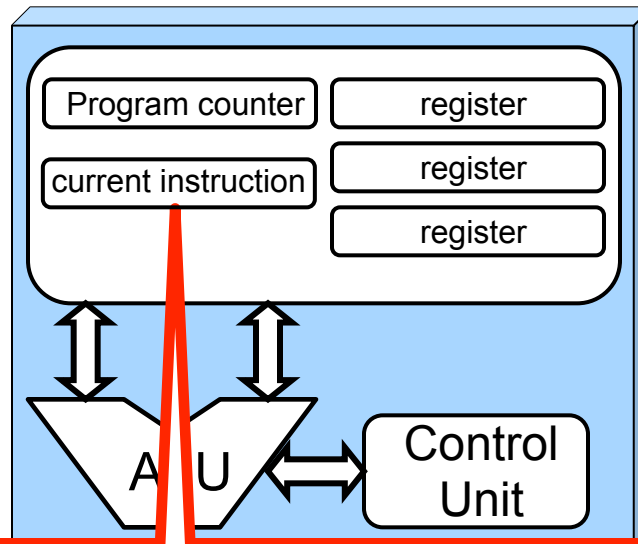


Program Counter: Points to the next instruction

Special register that contains the address in memory of the next instruction that should be executed

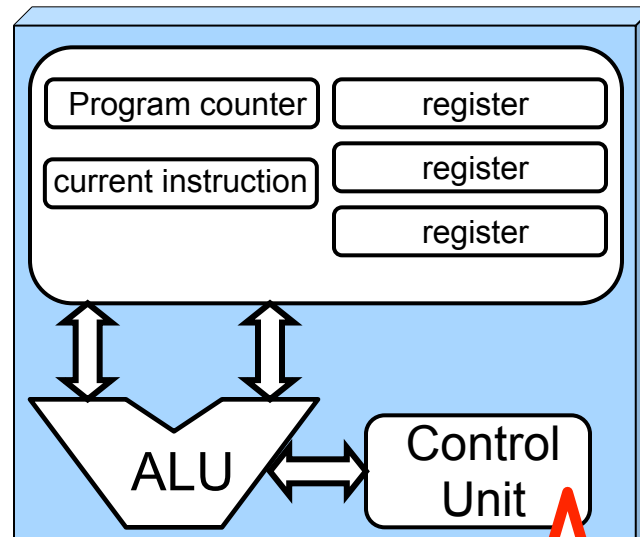
(gets incremented after each instruction, or can be set to whatever value whenever there is a change of control flow)

What's in the CPU?



Current Instruction: Holds the instruction that's currently being executed

What's in the CPU?



Control Unit: Decodes instructions and make them happen

Logic hardware that decodes instructions (i.e., based on their bits) and sends the appropriate (electrical) signals to hardware components in the CPU

Fetch-Decode-Execute Cycle

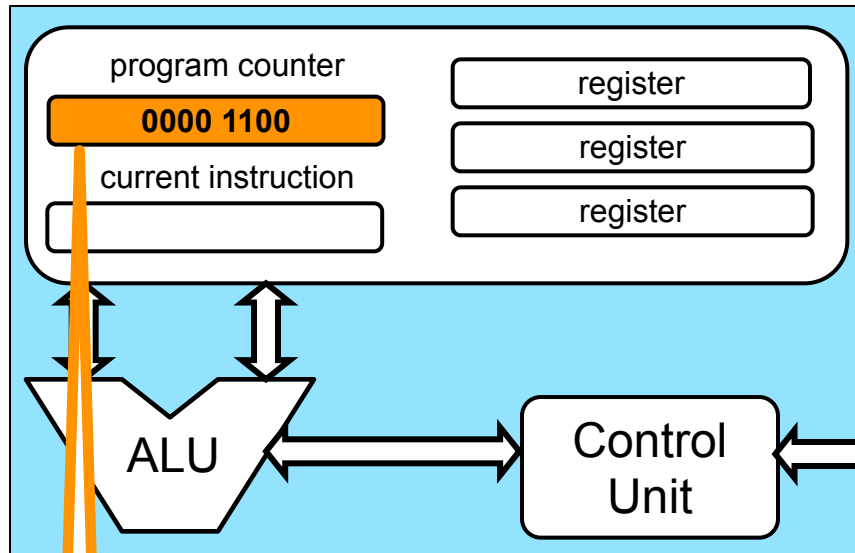
- The **Fetch-Decode-Execute** cycle
 - The control unit **fetches** the next program instruction from memory
 - Using the **program counter** to figure out where that instruction is located in the memory
 - The control unit **decodes** the instruction and signals are sent to hardware components
 - e.g., is the instruction loading something from memory? is it adding two register values together?
 - The instruction is **executed**
 - Operands are fetched from **memory** and put in **registers**, if needed
 - The ALU **executes** computation, if any, and stores the computed results in the **registers**
 - Register values are stored back to **memory**, if needed
 - Repeat
- Computers today implement MANY variations on this model
- But one can still program with the above model in mind
 - But then without understanding performance issues

Fetch-Decode-Execute

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

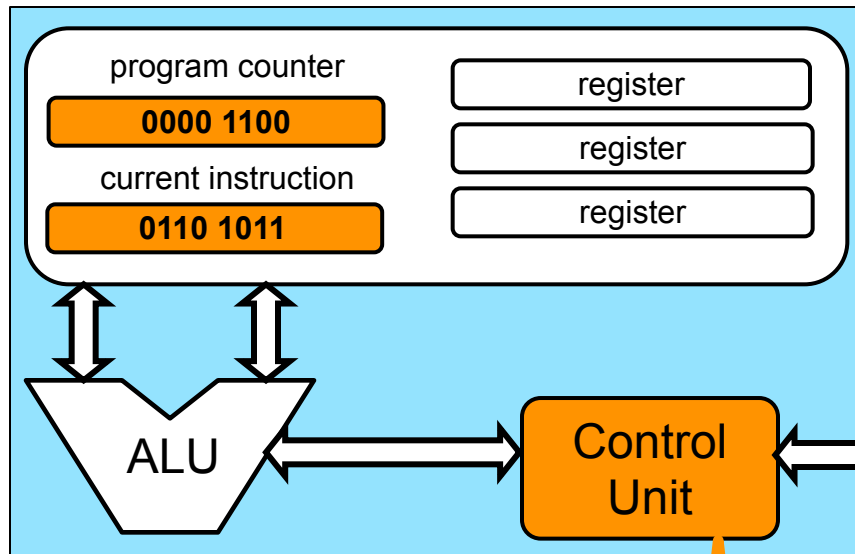


Somehow, the program counter is initialized to some content, which is an address (done by the OS - see ICS332)

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

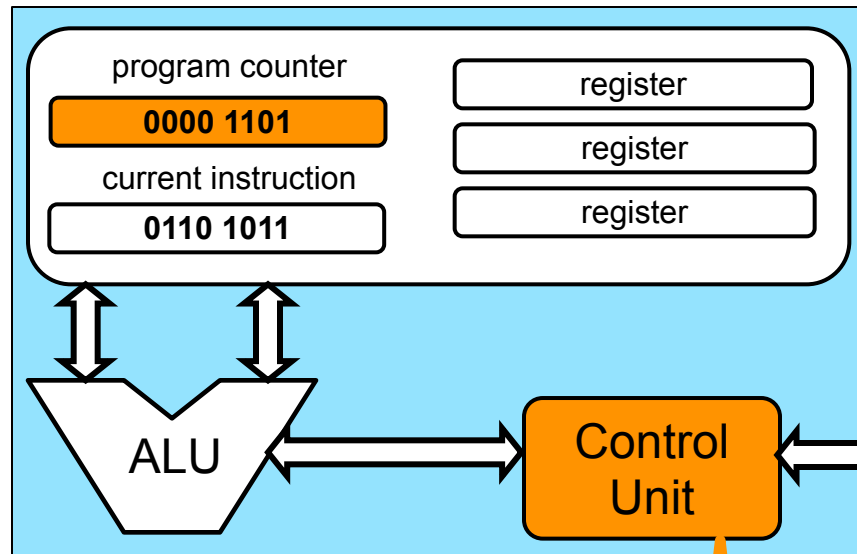


Fetch the content (instruction) at address 0000 1100, which is “0110 1011”, and store it in the “current instruction” register

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

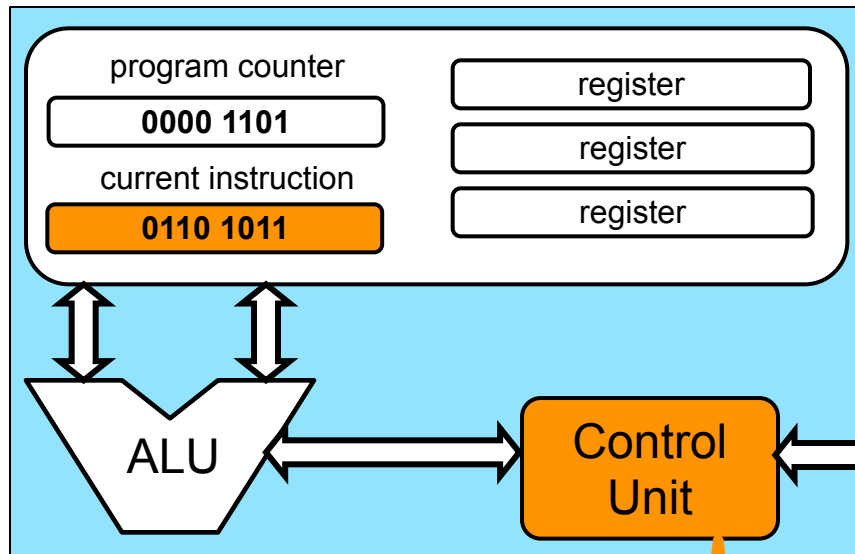


Increment the program counter

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

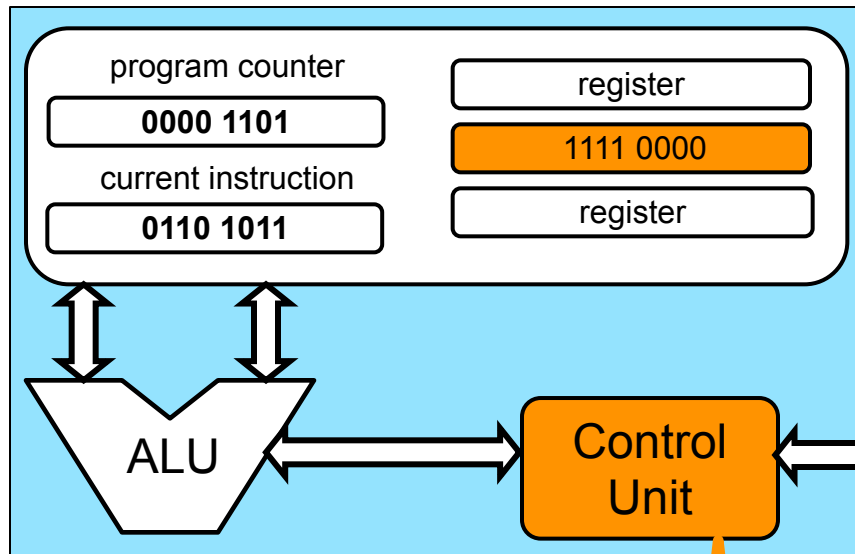


Decode instruction “0110 1011”. Let’s pretend it means: “Load the value at address 1000 0000 and store it in the second register”

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

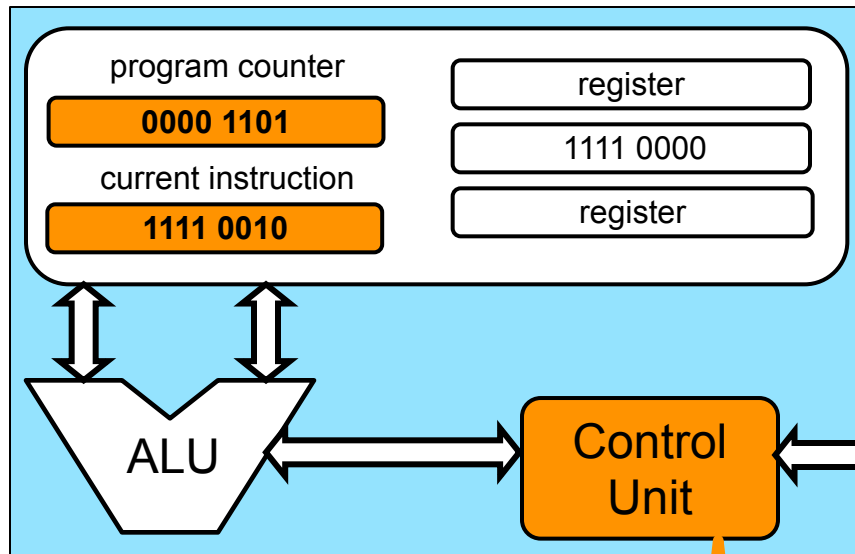


Send signals to all hardware components to **execute** the instruction: load the value at address 1000 0000, which is "1111 0000" and store it in the second register

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

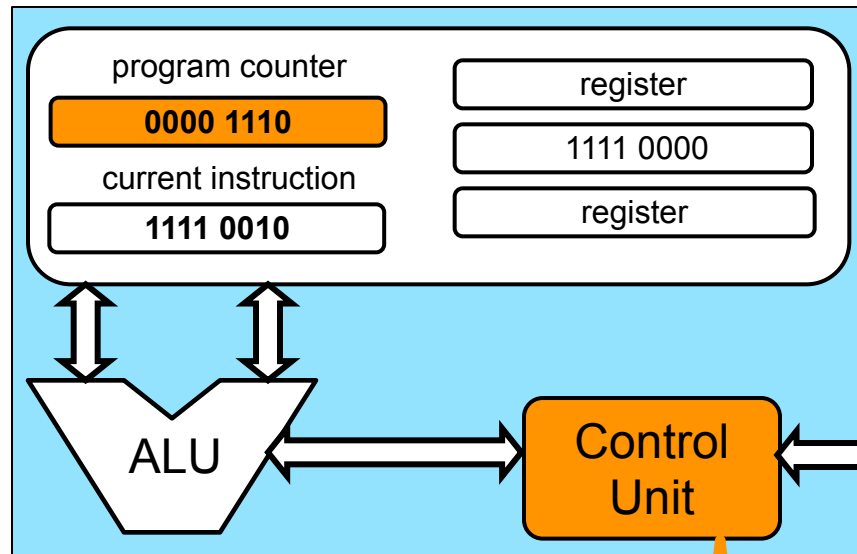


Fetch the content (instruction) at address 0000 1101, which is “1111 0010”, and store it in the “current instruction” register

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

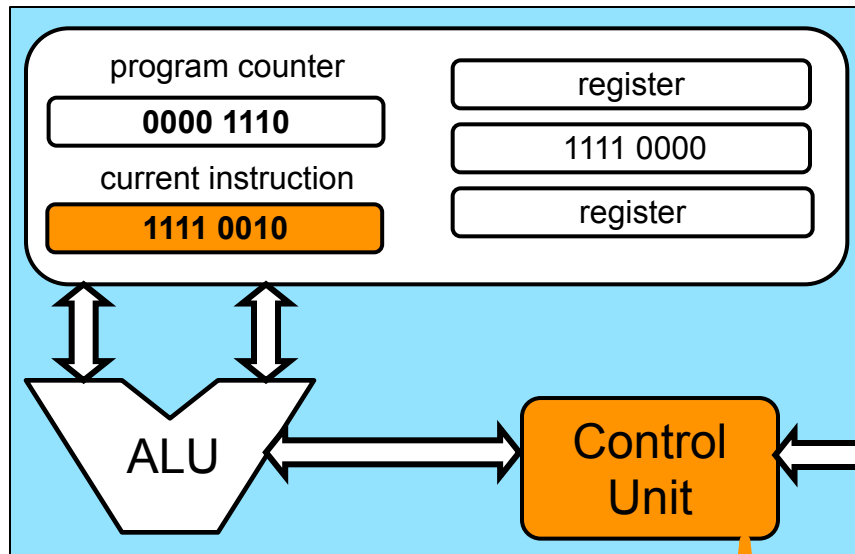


Increment the program counter

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

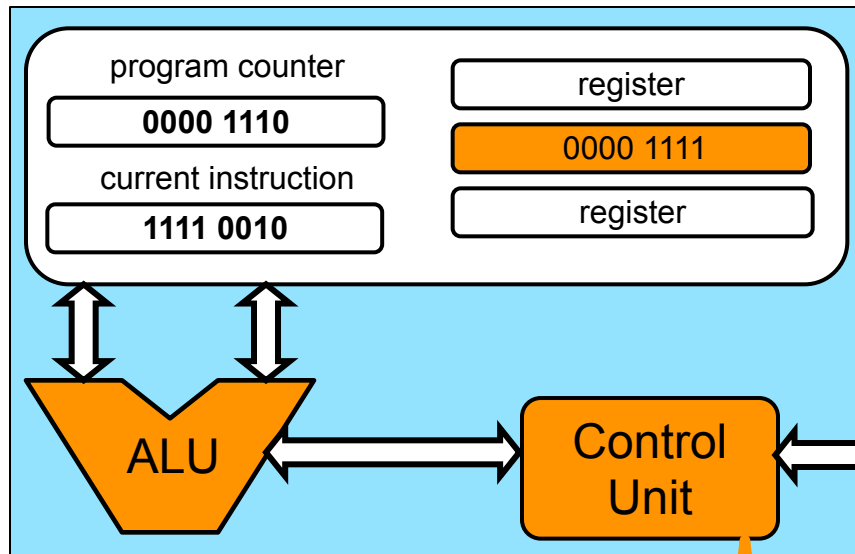


Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Decode instruction “1111 0010”. Let’s pretend it means: “Do a logical NOT on the second register”

Memory

Fetch-Decode-Execute

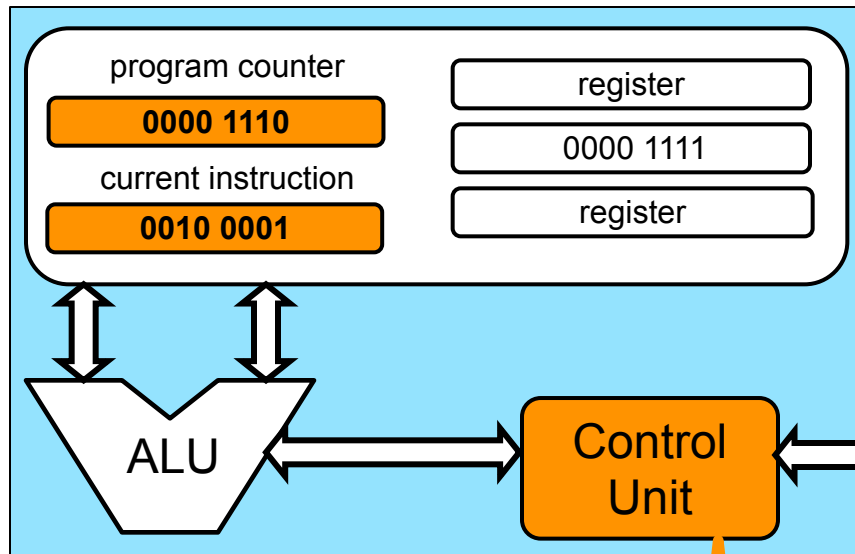


Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Send signals to all hardware components to **execute** the instruction: do a logical NOT on the second register

Memory

Fetch-Decode-Execute

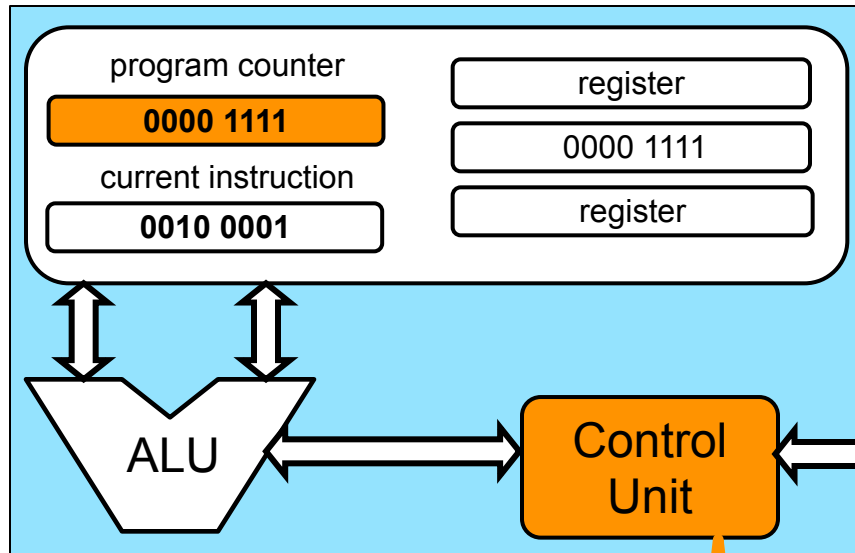


Fetch the content (instruction) at address 0000 1110, which is “0010 0001”, and store it in the “current instruction” register

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

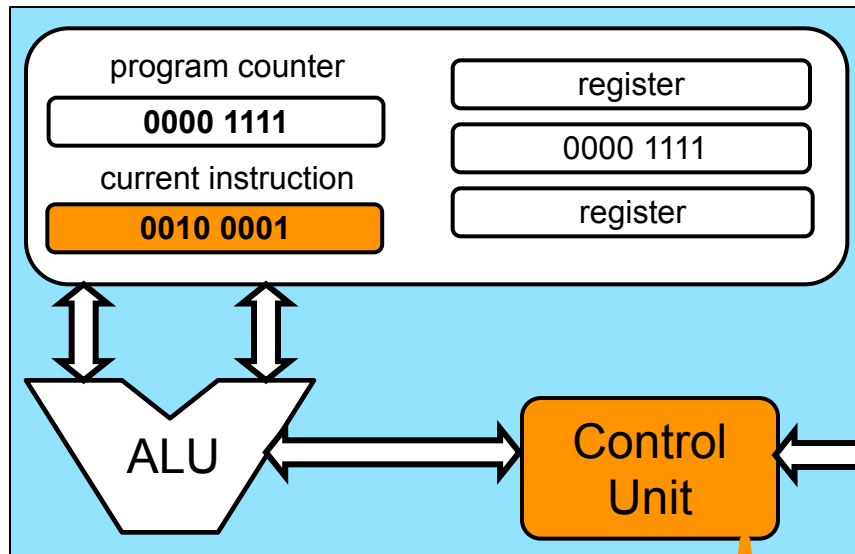


Increment the program counter

Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Memory

Fetch-Decode-Execute

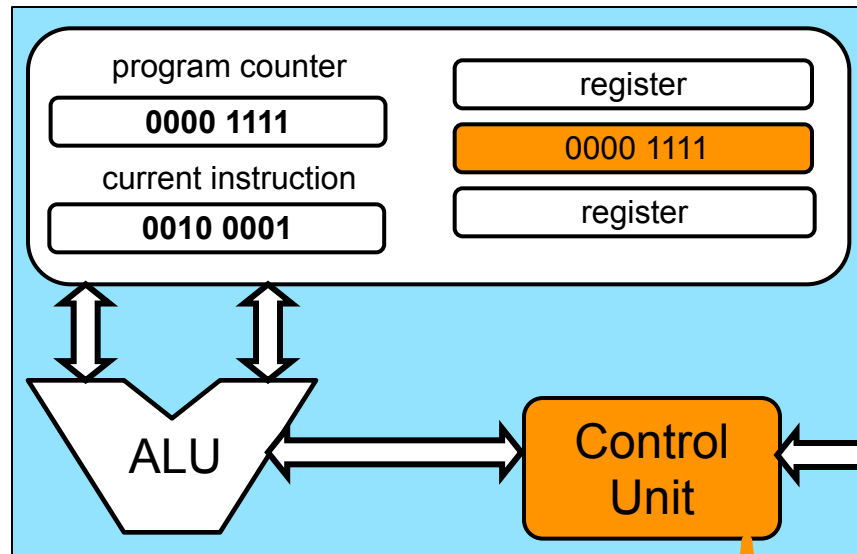


Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

Decode instruction "0010 0001". Let's pretend it means: "Store the value in the second register to memory at address 1111 0010"

Memory

Fetch-Decode-Execute



Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0000 1111

Send signals to all hardware components to **execute** the instruction: store the value in the second register, which is 0000 1111, to memory at address 1111 0010

Memory



Fetch-Decode-Execute

- This is only a simplified view of the way things work
- The “control unit” is not a single thing
 - Control and data paths are implemented by several complex hardware components
- There are multiple ALUs, there are caches, there are multiple CPUs in fact (“cores”)
- Execution is pipelined: e.g., while one instruction is fetched, another one is being executed
- Decades of computer architecture research have gone into improving performance, thus often leading to staggering hardware complexity
 - Doing smart things in hardware requires more logic gates and wires, thus increasing processor cost
- But conceptually, fetch-decode-execute is it

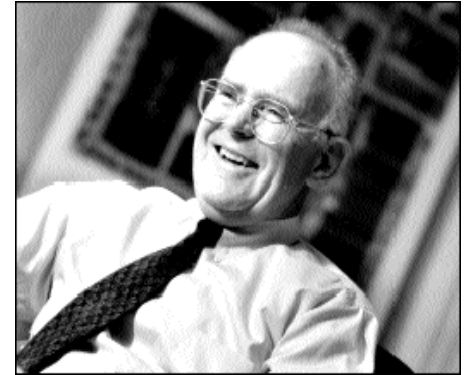


Multi-Core

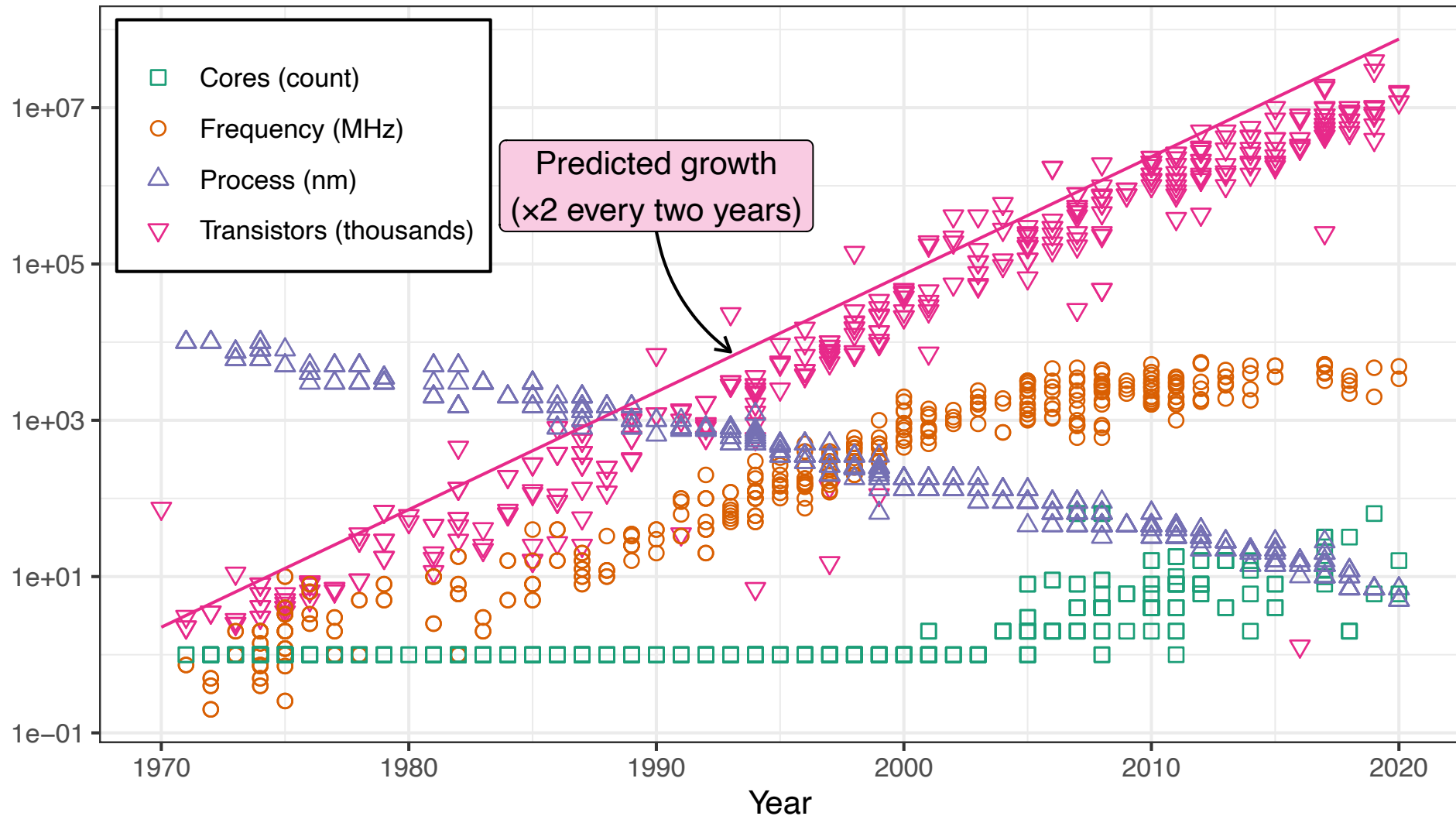
- What we have described is what happens in a single core
- But nowadays all our machines are multi-core (e.g., my laptop has 10 cores)
- Let's see why that is...

Moore's Law

- In 1965, Gordon Moore (co-founder of Intel) predicted that **transistor density** of semiconductor chips would **double** roughly **every 24 months** (often “misquoted” as 18 months)
- He was right
- But, the law was often wrongly interpreted as: “Computers get twice as fast every 2 years”
- This wrong interpretation was true for a while, but no longer...

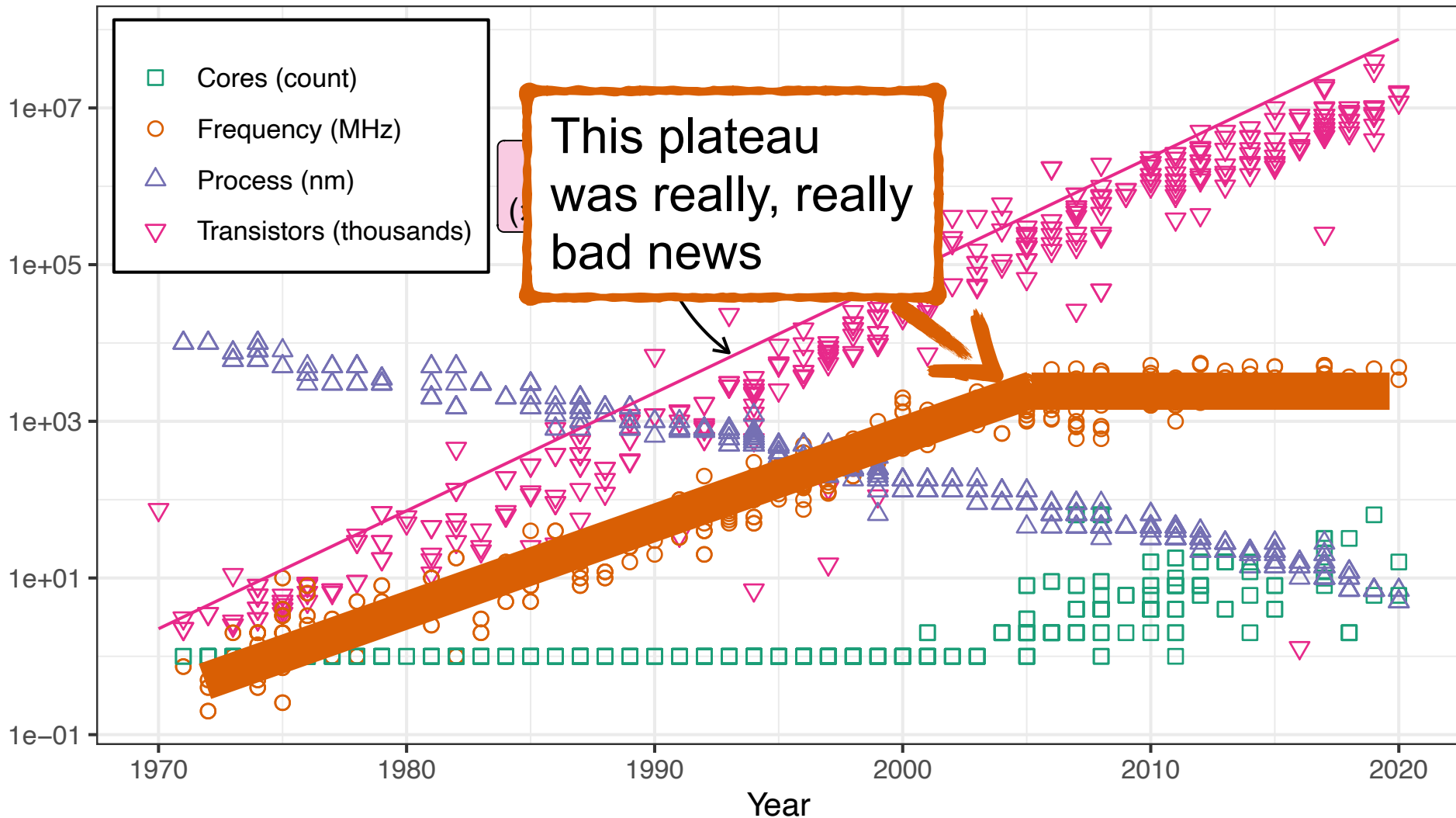


50-year Trend



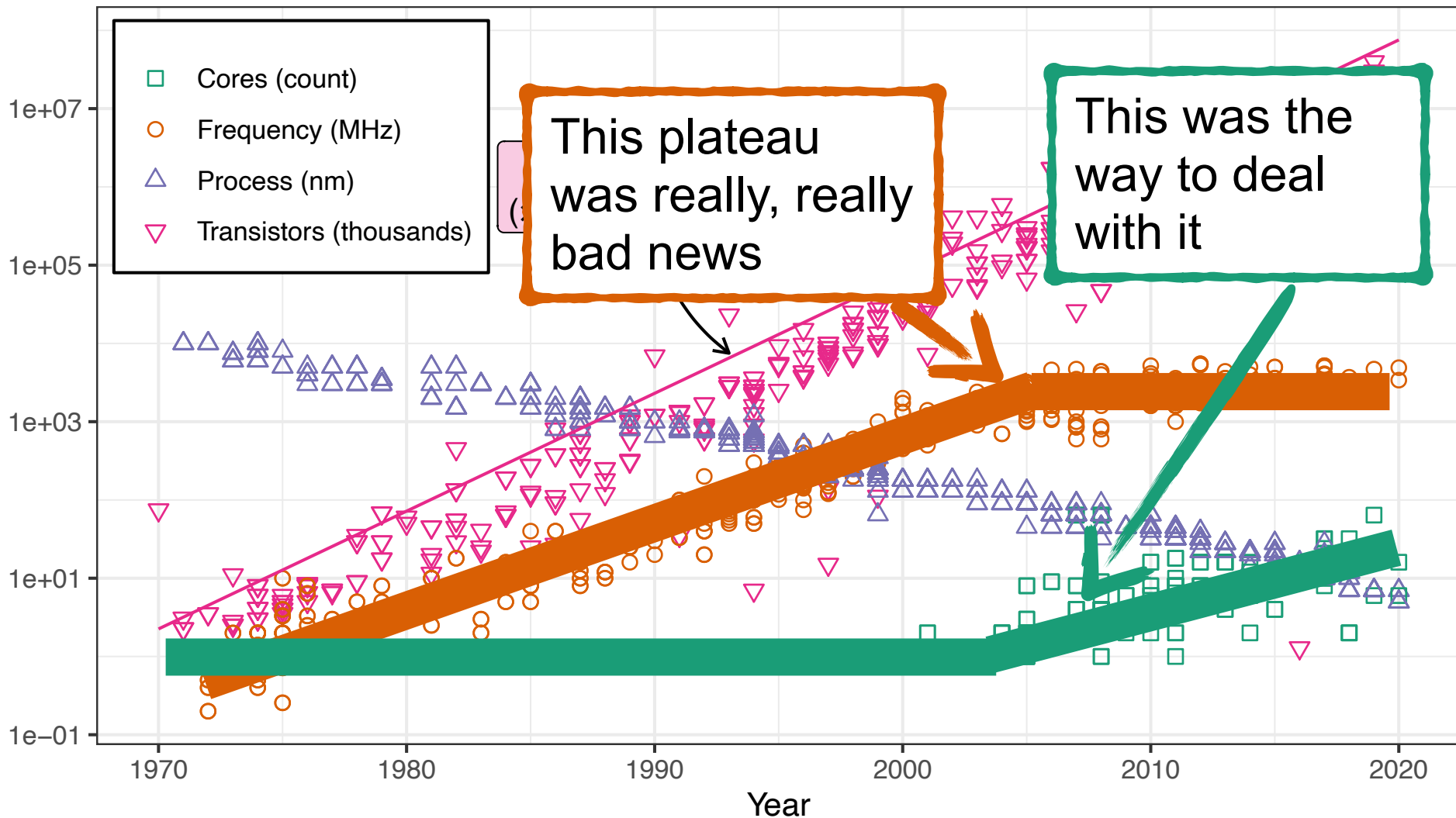
Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

50-year Trend



Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

50-year Trend

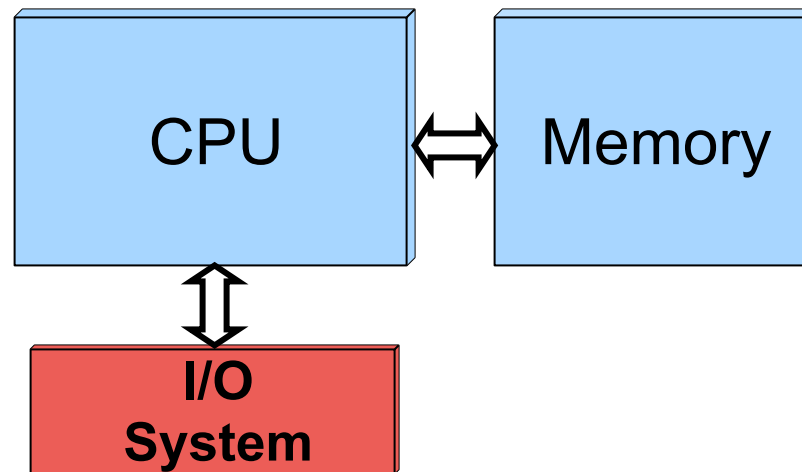


Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

Multi-core Chips

- Constructors cannot increase clock rate further
 - Power/heat issues
- They bring you **multi-core processors**
 - Multiple “low” clock rate processors on a chip
- It’s really a solution to a problem, not a cool new advance
- Most developers would rather have a 100GHz core than 50 2GHz cores
 - In which case we would not need to write concurrent programs
- But we don’t have 100GHz cores, which is why you should take ICS 432 :)

I/O



I/O

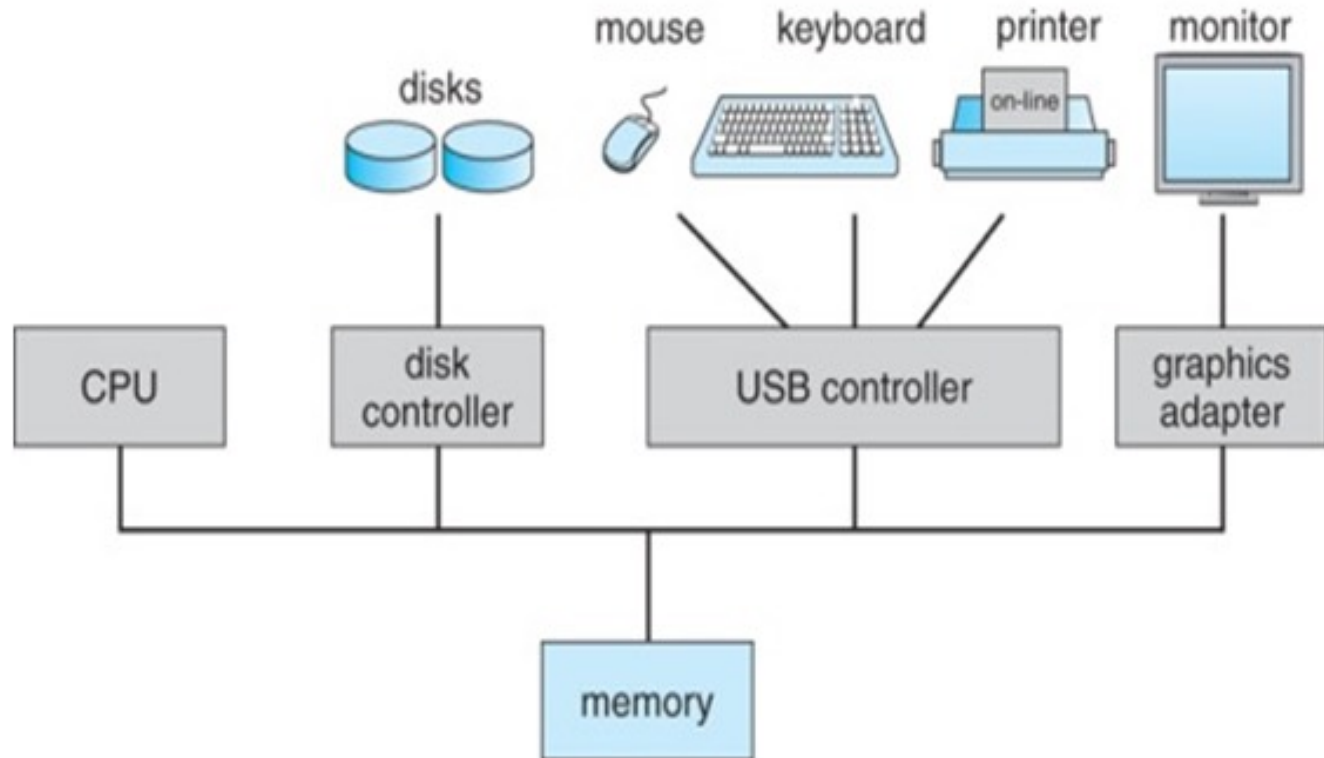


Figure 1.2 A modern computer system.

[reproduced from Operating Systems Concepts (Silberschatz, Galvin, Gagne)]

- We've all used many I/O devices (screens, keyboard, disks, ..)
- These all have their specific hardware controllers
- That's all I am going to say for now

Conclusion

- Computer Architecture is obviously a very large topic
- If you want to know more
 - Take a computer architecture course
 - Classic Textbook: Computer Organization and Design, Fourth Edition: The Hardware/Software Interface (Patterson and Hennessy, Morgan Kaufmann)
- Let's now talk more about memory...

