

# **Computer Architecture and Programming**

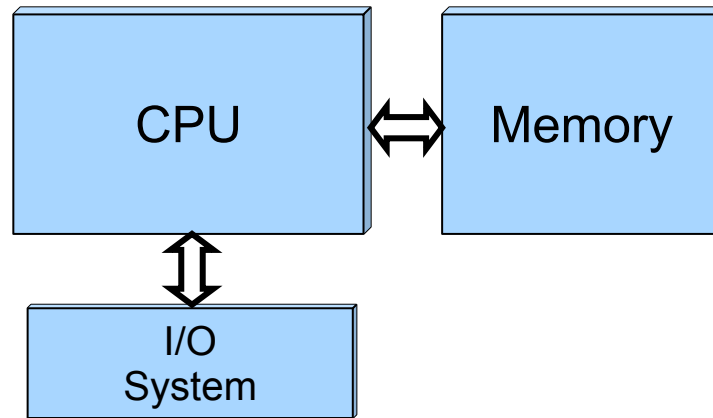
**ICS312  
Machine-level and  
Systems Programming**

Henri Casanova ([henric@hawaii.edu](mailto:henric@hawaii.edu))

# “Computer Architecture”?

- The field of Computer Architecture is about the fundamental structure of computer systems
  - What are the components?
  - How do they interact with each other?
  - How fast does the whole system operate?
  - How much power does it consume?
  - How much does it cost to mass-produce?
  - How to achieve desired speed/power/cost trade-offs?
- The conceptual model for computer architecture, that hasn't fundamentally changed since 1945: the **Von-Neumann architecture**

# Von-Neumann Architecture



- A **processor** that performs operations and controls all that happens
  - A **memory** that contains **code and data**
  - **I/O** of some kind
- 
- Amazingly, it's still possible to think of the computer this way at a conceptual level (model from ~80 years ago!!!)
    - You can just think of the above picture, just with tons of (performance enhancing) bells and whistles

# Data Stored in 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: 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)
  - ...
- Note the “i” in the notations above: means “power of 2”
  - Unlike in a kilometer (km), where k means 1,000 (not 1,024)

# 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
- All addresses on a computer 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”

# 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**



# Address Space

- With  $d$ -bit long addresses we can address  $2^d$  different “things”
  - If I have  $n$  things to address I need  $\lceil \log_2 n \rceil$  bits
- Example:
  - 2-bit addresses
    - 00, 01, 10, 11
    - 4 “things”
  - 3-bit addresses
    - 000, 001, 010, 011, 100, 101, 110, 111
    - 8 “things”
- In our case, these things are “bytes” because we consider byte-addressable RAM
  - One cannot address anything smaller than a byte
- How big is my RAM if my addresses are 13-bit? (poll)

# Address Space

- With  $d$ -bit long addresses we can address  $2^d$  different “things”
  - If I have  $n$  things to address I need  $\lceil \log_2 n \rceil$  bits
- Example:
  - 2-bit addresses
    - 00, 01, 10, 11
    - 4 “things”
  - 3-bit addresses
    - 000, 001, 010, 011, 100, 101, 110, 111
    - 8 “things”
- In our case, these things are “bytes” because we consider byte-addressable RAM
  - One cannot address anything smaller than a byte
- How big is my RAM if my addresses are 13-bit? (poll)  
 $2^{13} \text{ bytes} = 2^3 * 2^{10} = 8 \text{ KiB!}$

# Both Code and Data in Memory

- A program is loaded in memory
- Its **address space** (the bytes that belong to it in memory) contain **both code and data**
- To the CPU, code and data are just bytes, but the programmer (hopefully) knows which bytes are data and which bytes are code
  - Conveniently hidden from you if you've never written assembly
  - But we'll have to keep code/data straight in these lecture notes

Code

Data

## Example Address Space

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

Memory



# The CPU is a Memory Modifier

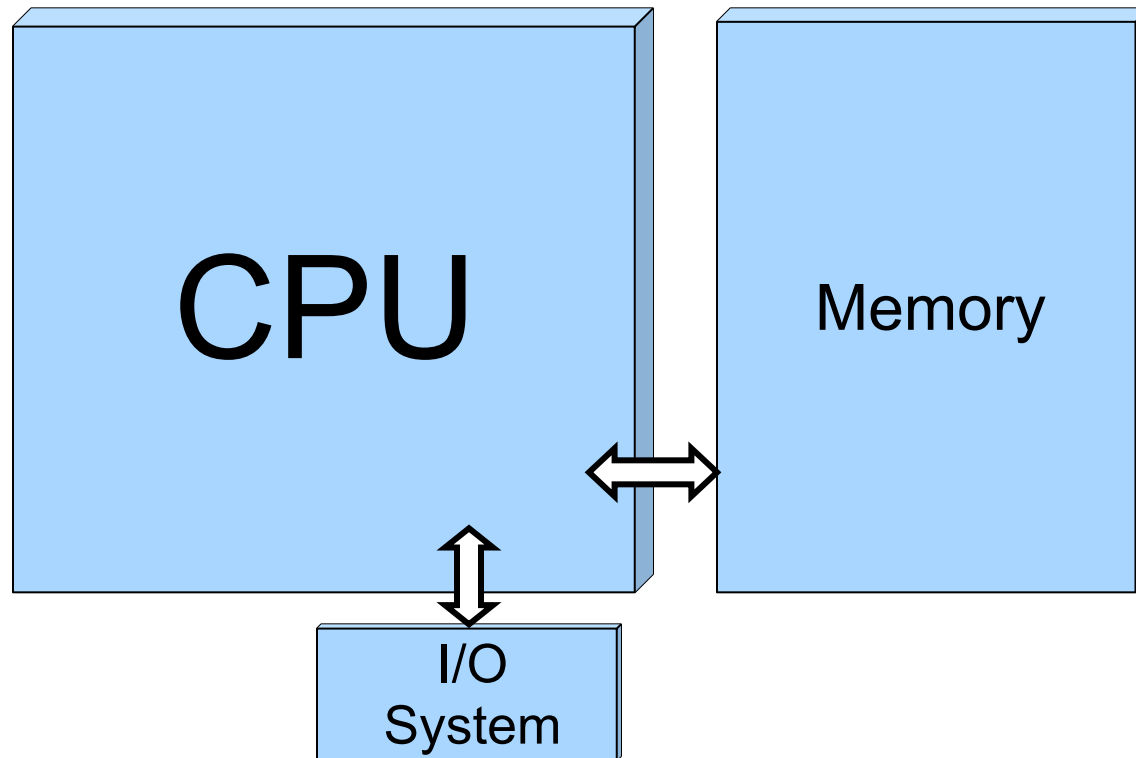
- So now we have a memory in which we can store/retrieve bytes at precise locations
- These bytes presumably have some useful meaning to us
  - e.g., integers, ASCII codes of characters, floating points numbers, RGB values
  - e.g., instructions that specify what to do with the data; when you buy a processor, the vendor defines the instruction set (e.g., instruction “0010 1101” means “increment some useful counter”)
- The CPU is the piece of hardware that modifies the content of memory
  - In fact, one can really think of the CPU as a device that takes us from one memory state (i.e, all the stored content) to another memory state (some new, desired stored content)



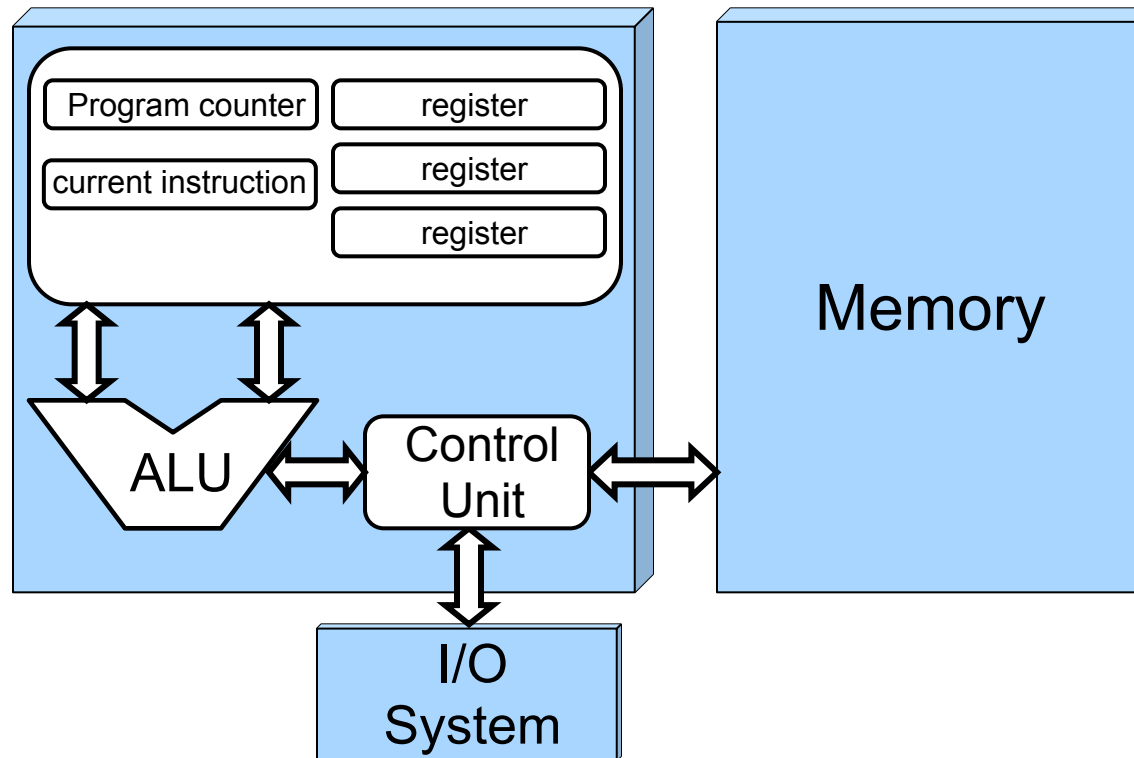
# Disclaimer

- Several of the next few slides may also have been show in ICS332
  - Because ICS312 and ICS332 are not in a prerequisite chain
- So you may have seen them before
- Or you may see them again
  
- Regardless, it's good to be reminded of computer architecture basics!

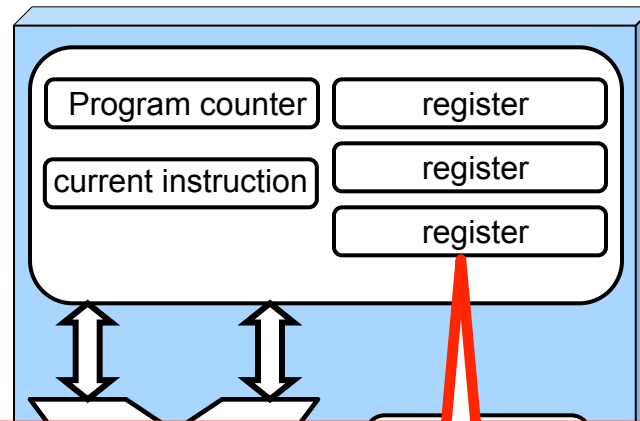
# What's in the CPU?



# 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

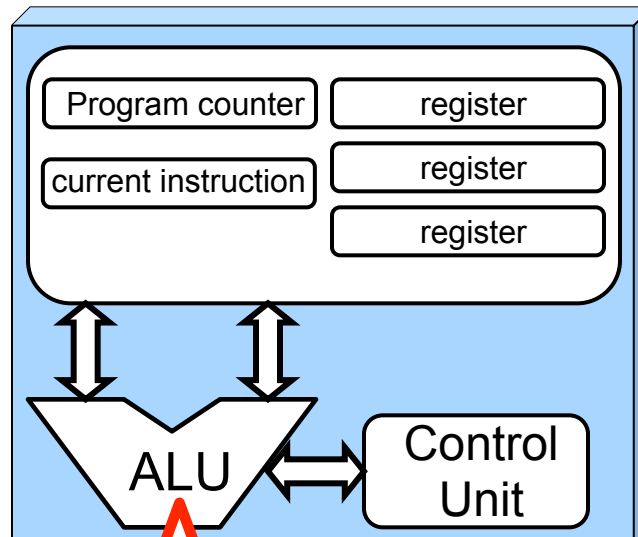
Operands and results of computations are ALL in registers

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?

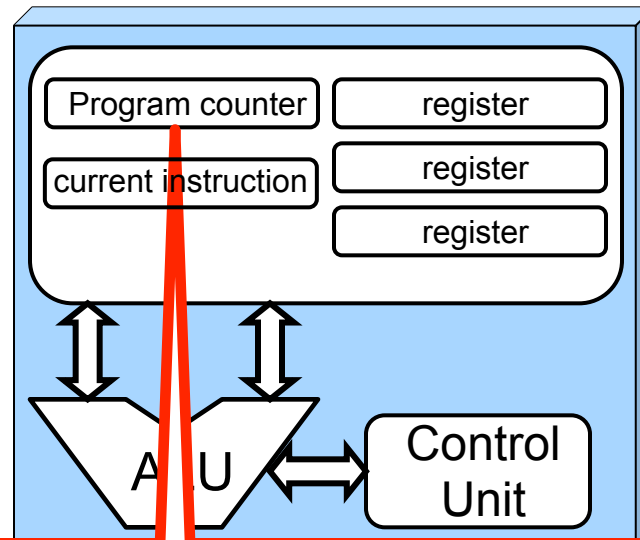


**Arithmetic and Logic Unit:** what you do computation with

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?

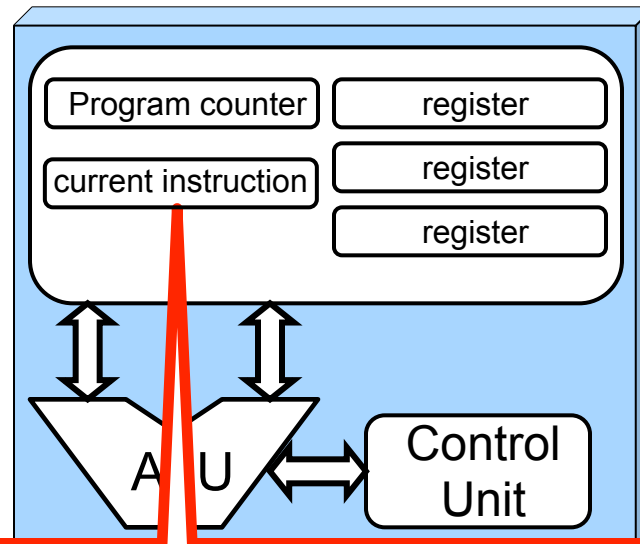


**Program Counter:** Points to the next instruction

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

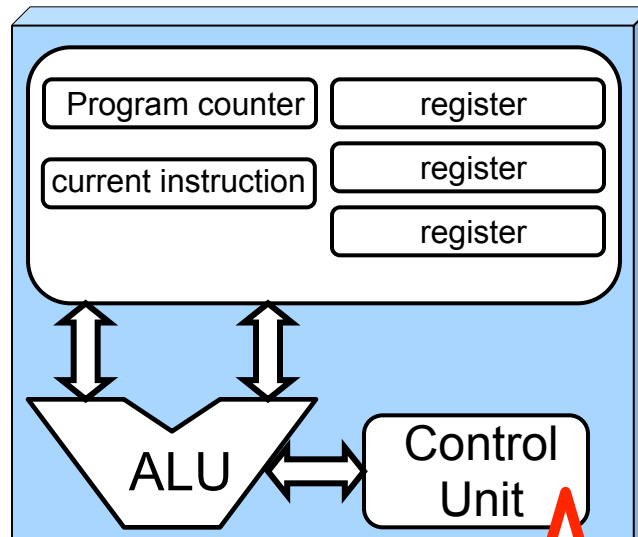
(gets incremented after each instruction, or can be set to whatever value whenever there is a change of control flow)

# 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

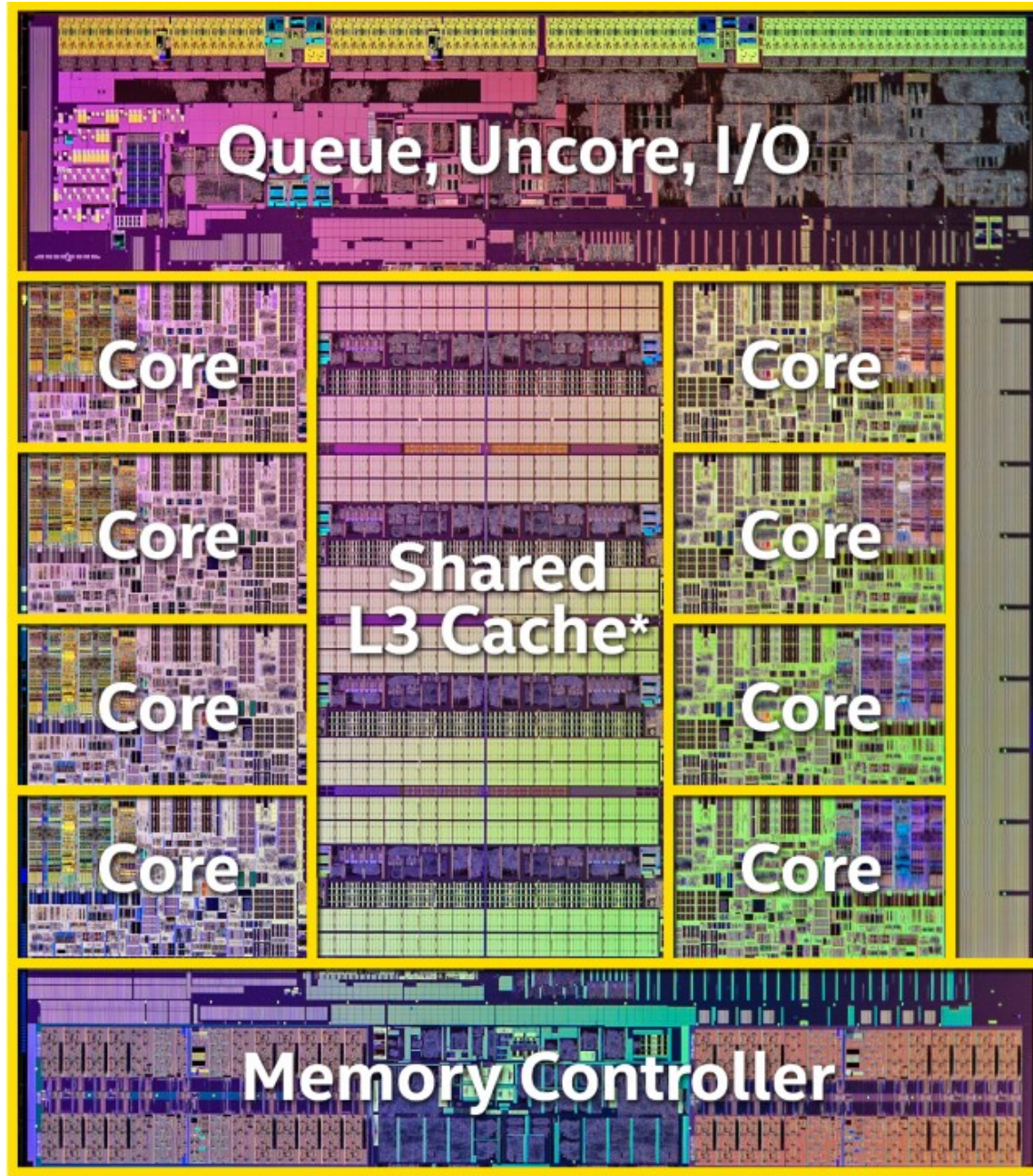
Logic hardware that decodes instructions (i.e., based on their bits) and sends the appropriate (electrical) signals to hardware components in the CPU

# A CPU in its “Glory”: Intel Haswell



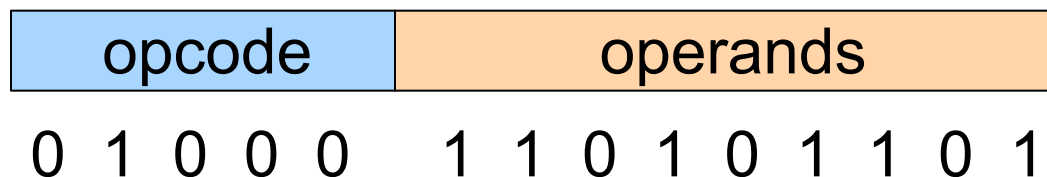
Each Core contains a “CPU” (as in in previous slides) plus more stuff (e.g., L1 and L2 Caches)

In this course we assume a single, very simple core (which we call the CPU)



# Instructions

- Instructions are **encoded** in binary **machine code**
  - e.g.: 01000110101101 may mean “perform an addition of two registers and store the results in another register”
- The CPU is built using **gates** (OR, AND, etc.) which themselves use transistors
- These gates implement **instruction decoding**
  - Based on the bits of the instruction code, signals are sent to different components, which then perform useful tasks
- Typically, an instruction consists of two parts
  - The **opcode**: what the instruction should do
  - The **operands**: the input to the computation



# Instruction Set Architecture (ISA)

- When designing a CPU, one must define the set of all the instructions it understands
  - This is one thing that Intel engineers do
- This is called the ISA: **Instruction Set Architecture**
- Typical ISA include instructions for
  - Performing arithmetic operations on register values
  - Load values from memory into registers
  - Store values from registers into memory
  - Test register values to decide what instruction to execute next
  - ...
- Envision a loooong specification manual that lists all the possible instructions...

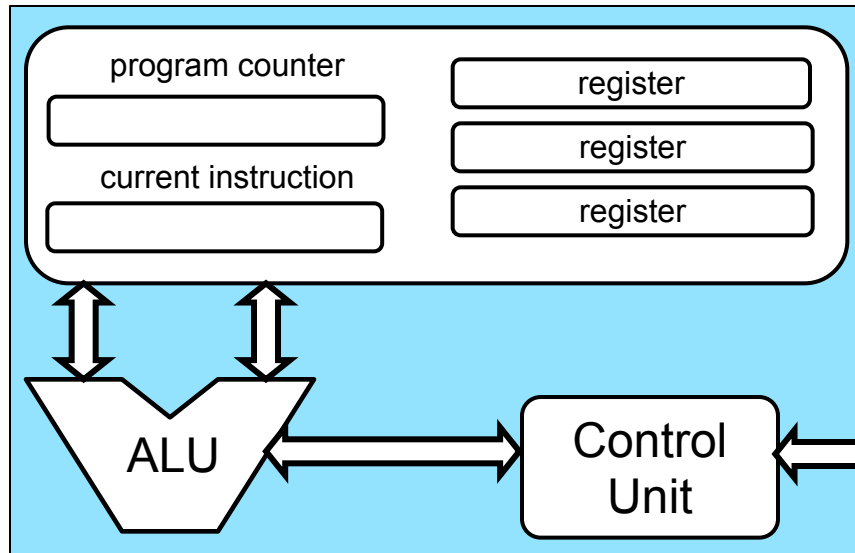


# Fetch-Decode-Execute Cycle

- 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



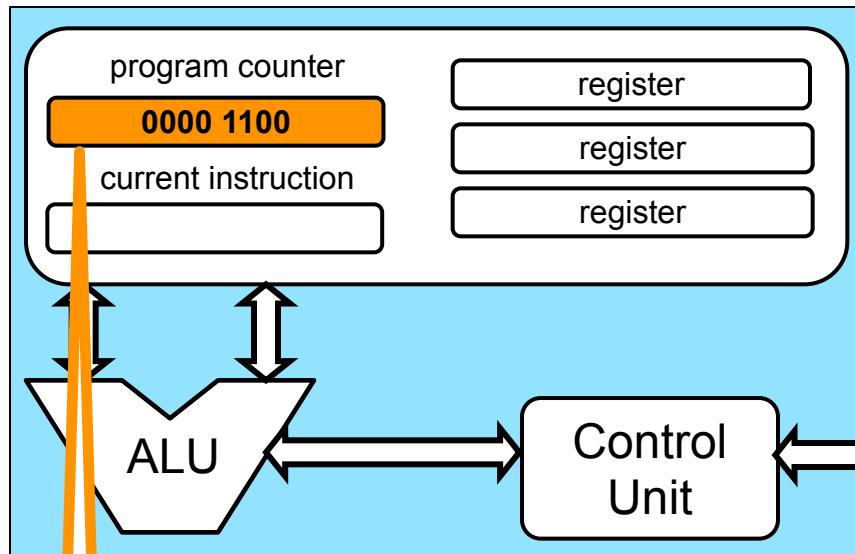
# Fetch-Decode-Execute



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

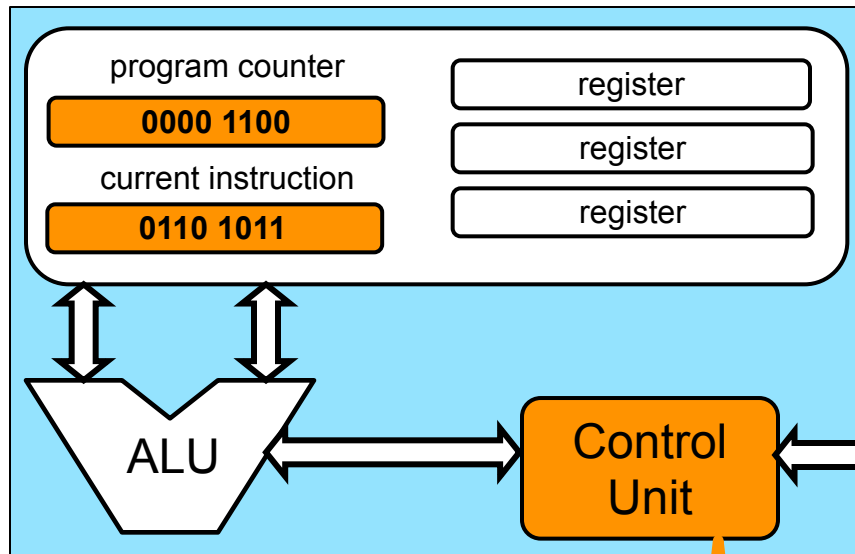


Somehow, the program counter is initialized to some content, which is an address (done by the OS - see ICS332)

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

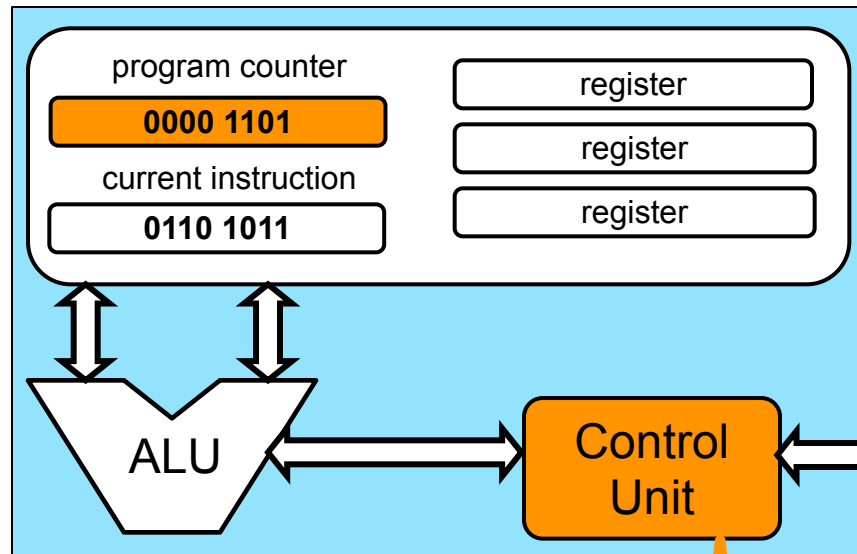


**Fetch** the content (instruction) at address 0000 1100, which is “0110 1011”, and store it in the “current instruction” register

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

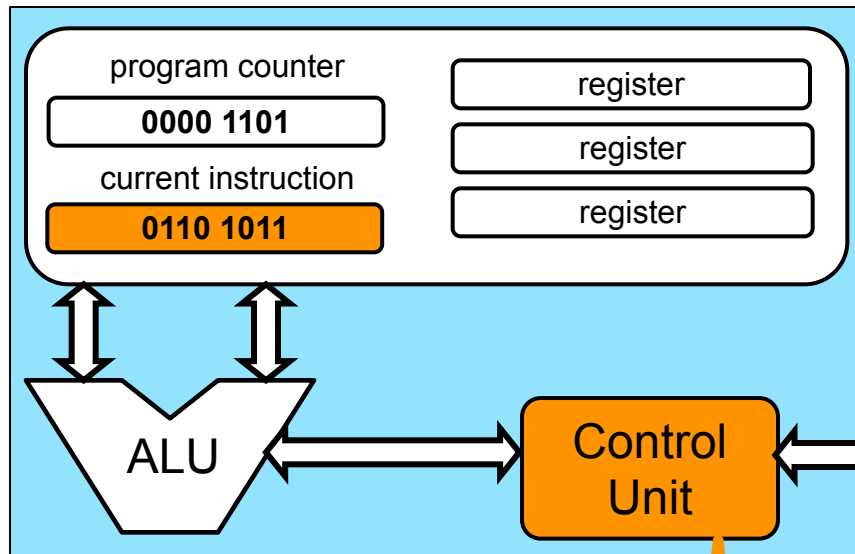


Increment the program counter

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

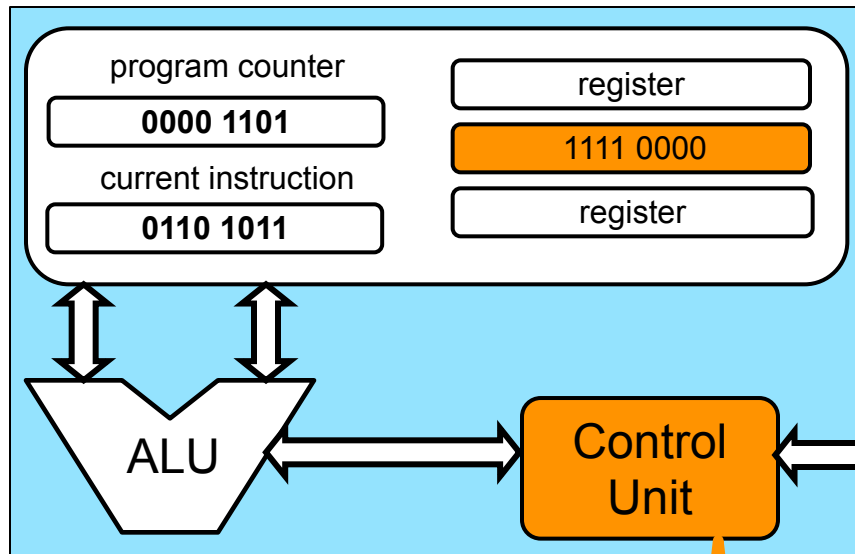


**Decode** instruction "0110 1011". Let's pretend it means: "Load the value at address 1000 0000 and store it in the second register"

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

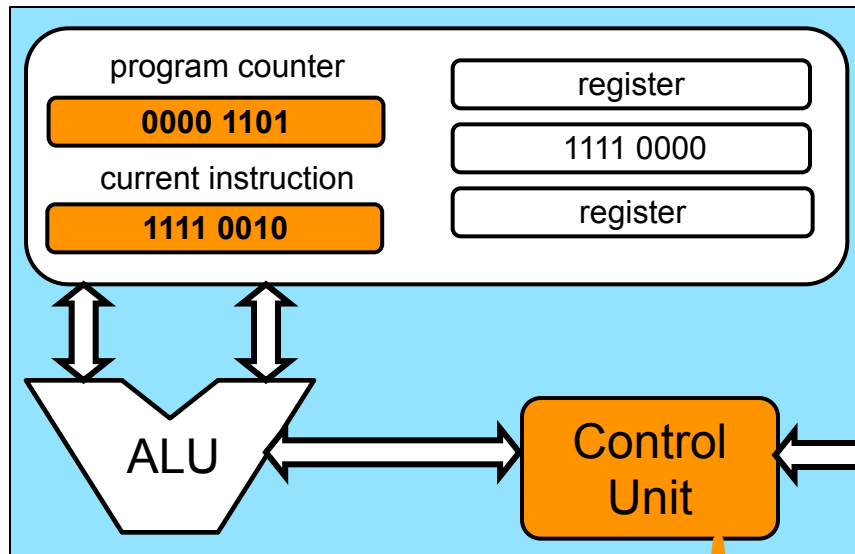


Send signals to all hardware components to **execute** the instruction: load the value at address 1000 0000, which is "1111 0000" and store it in the second register

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

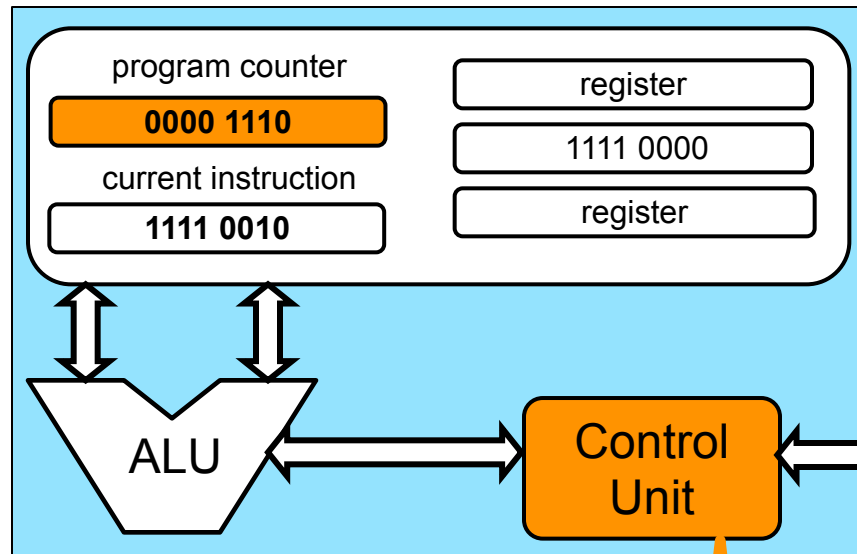


**Fetch** the content (instruction) at address 0000 1101, which is “1111 0010”, and store it in the “current instruction” register

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



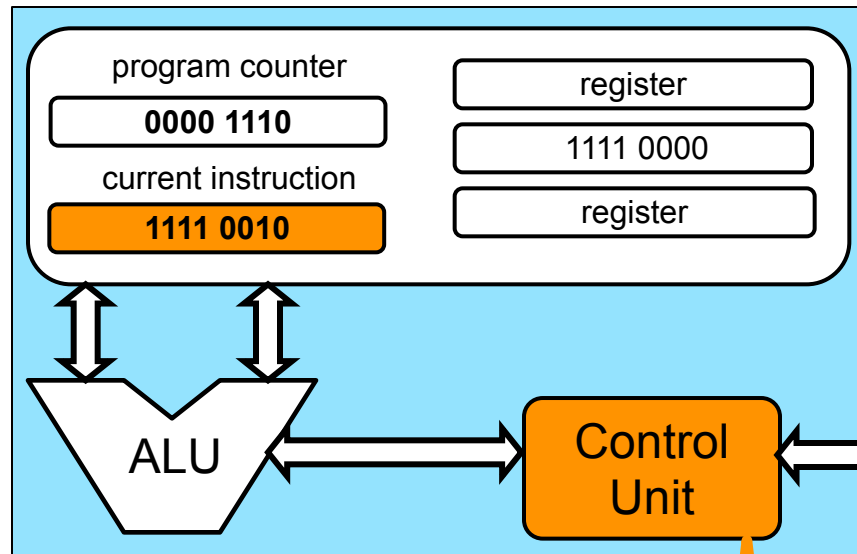
Increment the program counter

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

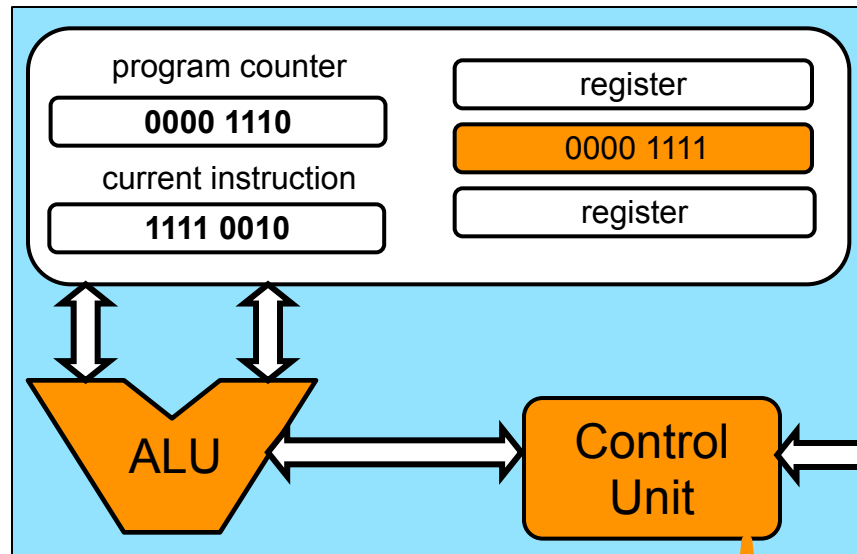


Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

**Decode** instruction "1111 0010". Let's pretend it means: "Do a logical NOT on the second register"

Memory

# Fetch-Decode-Execute

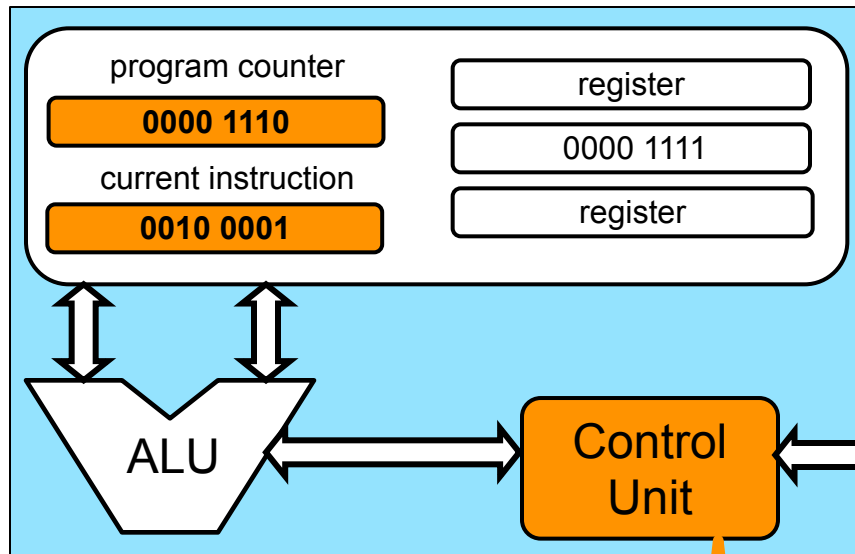


Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

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

Memory

# Fetch-Decode-Execute

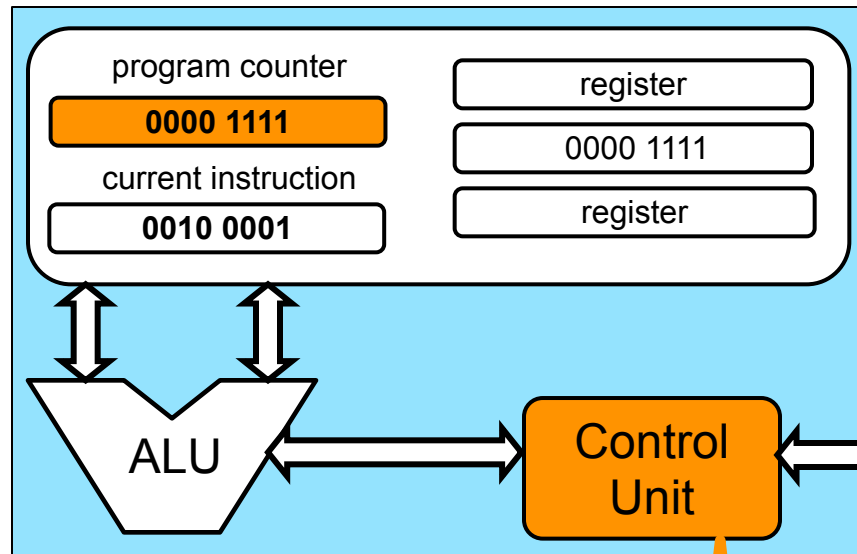


**Fetch** the content (instruction) at address 0000 1110, which is “0010 0001”, and store it in the “current instruction” register

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

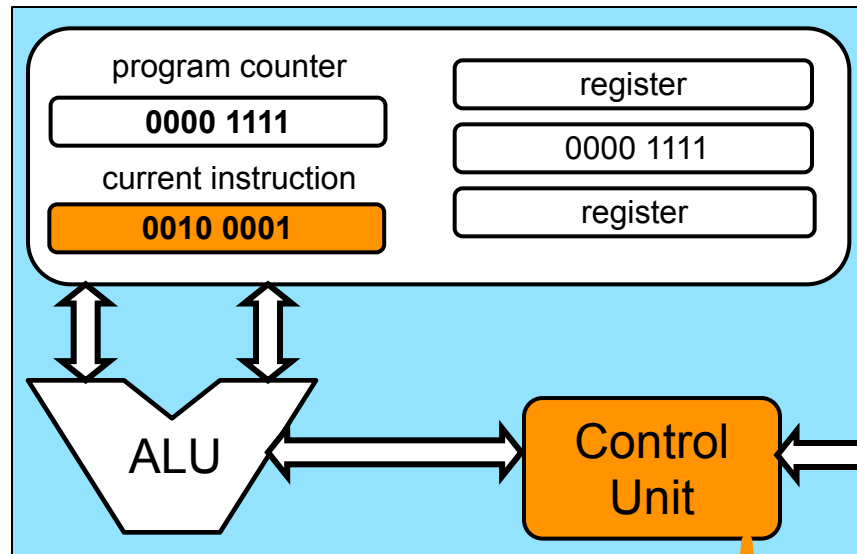


Increment the program counter

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

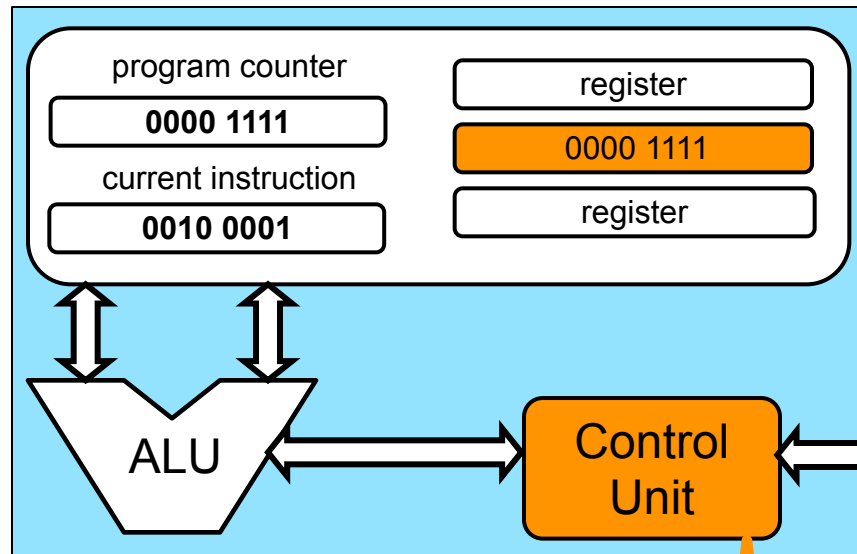


Address	Value
0000 1100	0110 1011
0000 1101	1111 0010
0000 1110	0010 0001
...	...
1000 0000	1111 0000
...	...
1111 0010	0101 1111

**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

# Building a CPU in Minecraft

- Let's look at an amazing Minecraft CPU
  - Some of the features we can already understand
  - Some of the features we'll understand later
  - Some of the features we won't see in this course

<https://www.youtube.com/watch?v=FDiapbD0Xfg&t=95s>



# Assembly language

- It's really difficult for humans to read/remember binary instruction encodings
  - But people used to do it!
  - One would typically use hexadecimal encoding, but still it seems impossible to memorize all this in today's world
- Therefore, it is typical to use a set of *mnemonics*
- We call these mnemonics the **assembly language**
  - It is often said that the CPU understands assembly language
  - This is not technically true: the CPU understands machine code, which we, as humans, choose to represent using assembly language
- An **assembler** is a computer program that transforms assembly code into machine code (i.e., from a human-readable format into a binary CPU-readable format)

# Machine vs. Assembly code

- Say that on an architecture the opcode F2 means “add”, and that we have an add instruction that takes as operands three registers, each identified by an index
- Then the instruction “add the 1st register to the 7th register and store the result in the 4th register” could be (inefficiently) encoded as F2010704
  - Some (older) programmers can read machine code pretty well
- Instead, we defined the assembly equivalent to be, for instance:  
**add R1 ,R7 ,R4**
  - **add** is called the mnemonic
  - **R1 ,R7 ,** and **R4** are the operands
- There is a one-to-one correspondance between an assembly instruction and a machine instruction
  - Not true of high-level languages!

# Assembler

- An **assembler** is a computer program that transforms assembly code into machine code (i.e., from a human-readable format into a binary CPU-readable format)
- It's a pretty simple program, since there is a one-to-one correspondance between assembly instructions and machine instructions
- Assembly code is NOT portable across architectures
  - Different ISAs, different assembly languages

# The 80x86 Architecture

- For this course we need to pick a processor family with a given ISA (Instruction Set Architecture)
- We will use the Intel 80x86 ISA (x86 for short)
  - The most common today in existing personal computers
  - Although now all Apple machines have an ARM processor
- We could have picked other ISAs
  - ARM, MIPS
    - In ICS331/ICS431/EE460 you'd (likely) be exposed to those
- Some courses in some curricula subject students to two or even more ISAs in a single semester, but in this course we'll just focused on one
  - If you know one kind of assembly, it's easy to pick up another
  - The point of this course is to gain deep understanding of concepts, not become assembly programming prodigies

# x86 History (partial)

- In the late 70s Intel creates the 8088 and 8086 processors
  - 16-bit registers, 1 MiB of memory



- In 1982: the 80286
  - New instructions, 16 MiB of memory
- In 1985: the 80386
  - 32-bit registers, 5 GiB of memory



- 1989: 486; 1992: Pentium; 1995: P6
  - Only incremental changes to the architecture



# x86 History

- 1997 - now: improvements, new features galore
  - MMX and 3DNow! extensions
  - New instructions to speed up graphics (integer and float)
  - New cache instructions, new floating point operations
  - Virtualization extensions
  - etc..
- 2021: the “Golden Cove” code name (12th generation)
  - “All models support: AES-NI, CLMUL, RDRAND, SHA, TXT, MMX, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2, AVX, AVX2, FMA3, AVX-512, AVX-VNNI, TSX, VT-x, VT-d”
- Several manufacturers build x86-compliant processors
  - And have been for a long time



# x86 History

- Amazingly, this architecture has witnessed few fundamental changes since the 8086
  - All in the name of backward compatibility
  - Imposed early as “the one ISA” (Intel was the first company to produce a 16-bit architecture, which secured its success)
- Many argue that it’s an unsightly ISA
  - Due to it being a set of add-ons rather than a modern re-design
  - Famous quote by Mike Johnson (AMD): “The x86 isn’t all that complex... it just doesn’t make a lot of sense” (1994)
- But it’s relatively easy to implement in hardware, and constructors have been successfully making faster and faster x86 processors for decades, explaining its wide adoption
- Still in use today in 64-bit processors (dubbed x86-64)
  - In this course we do 32-bit x86

# ISA specification Example: x86

Let's look at the Web site <http://ref.x86asm.net/>

pf	OF	po	so	o	proc	st	m	rl	x	mnemonic	op1	op2	op3	op4	iext	tested f	modif f	def f	undef f	f values	description, notes
	00		r						L	ADD	<b>r/m8</b>	r8					o..szapc	o..szapc			Add
	01		r						L	ADD	<b>r/m16/32</b>	r16/32					o..szapc	o..szapc			Add
	02		r							ADD	<b>r8</b>	r/m8					o..szapc	o..szapc			Add
	03		r							ADD	<b>r16/32</b>	r/m16/32					o..szapc	o..szapc			Add
	04									ADD	<b>AL</b>	imm8					o..szapc	o..szapc			Add
	05									ADD	<b>eAX</b>	imm16/32					o..szapc	o..szapc			Add
	06									PUSH	<b>ES</b>										Push Word, Doubleword or Quadword Onto the Stack
	07									POP	<b>ES</b>										Pop a Value from the Stack
	08		r						L	OR	<b>r/m8</b>	r8					o..szapc	o..sz.pc	.....a..	o.....c	Logical Inclusive OR
	09		r						L	OR	<b>r/m16/32</b>	r16/32					o..szapc	o..sz.pc	.....a..	o.....c	Logical Inclusive OR
	0A		r							OR	<b>r8</b>	r/m8					o..szapc	o..sz.pc	.....a..	o.....c	Logical Inclusive OR
	0B		r							OR	<b>r16/32</b>	r/m16/32					o..szapc	o..sz.pc	.....a..	o.....c	Logical Inclusive OR
	0C									OR	<b>AL</b>	imm8					o..szapc	o..sz.pc	.....a..	o.....c	Logical Inclusive OR
	0D									OR	<b>eAX</b>	imm16/32					o..szapc	o..sz.pc	.....a..	o.....c	Logical Inclusive OR
	0E									PUSH	<b>CS</b>										Push Word, Doubleword or Quadword Onto the Stack

operands

opcode  
in HEX

Human-readable  
Mnemonic (assembly)

what it  
does

- **pf** Prefix
- **OF** OF Prefix
- **po** Primary Opcode
- **so** Secondary Opcode

- **x** Lock Prefix/FPU Push/FPU Pop
- **mnemonic** Instruction Mnemonic
- **op1, op2, ...** Instruction Operands
- **iext** Instruction Extension Group
- **grp1, grp2, grp3** Main Group, Sub-group, Sub-sub-group
- **tested f, modif f, def f, undef f** Tested, Modified, Defined, and Undefined Flags
- **f values** Flags Values
- **description, notes**



# High-Level Languages

- It used to be that *all* computer programmers did all day was write assembly code
- This was difficult for many reasons
  - Difficult to read and maintain (in spite of using the mnemonics)
  - Difficult to debug
  - Different from one computer to another!
- The exclusive use of assembly language for all programming prevented the (sustainable) development of large software projects with more than a few (very good) programmers
- This was the main motivation for developing **high-level programming languages**
  - FORTRAN, Cobol, C, etc.

# High-level Languages

- The first successful high-level language was FORTRAN
  - Developed by IBM in 1954 to run on they 704 series
  - Used for scientific computing
- The introduction of FORTRAN led people to believe that there would never be bugs again because it made programming so easy!
  - But high-level languages led to larger and more complex software systems, hence leading to bugs
- Another early programming language was COBOL
  - Developed in 1960, strongly supported by DoD
  - Used for business applications
- In the early 60s IBM had a simple marketing strategy
  - On the IBM 7090 you used FORTRAN to do science
  - On the IBM 7080 you used COBOL to do business
- Many high-level languages have been developed since then, and they are what most programmers use
  - Fascinating history (see ICS 313)

# High-level Languages

- Having high-level programming languages is good, but CPUs do not understand them
  - As we saw, they only understand very basic instructions to manipulate registers, etc.
- Therefore, there needs to be a **translation** from a high-level language to machine code
- The translation is done in two steps: by a **compiler** and then by an **assembler**
- Let's see this on a picture....

# The Big (Simplified) Picture

## High-level code

```
char *tmpfilename;
int num_schedulers=0;
int num_request_submitters=0;
int i,j;

if (!(f = fopen(filename,"r"))) {
    xbt_assert(0,"Cannot open file %s",filename);
}
while(fgets(buffer,256,f)) {
    if (strcmp(buffer,"SCHEDULER",9))
        num_schedulers++;
    if (strcmp(buffer,"REQUESTSUBMITTER",16))
        num_request_submitters++;
}
fclose(f);
tmpfilename = strdup("/tmp/jobsimulator_
```

## COMPILER

## ASSEMBLER

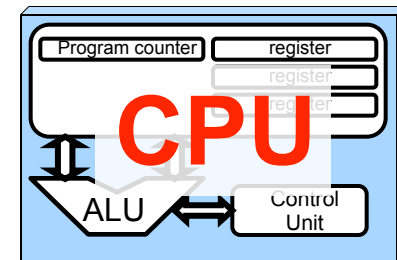
## Machine code

```
010000101010110110
101010101111010101
101001010101010001
101010101010100101
111100001010101001
000101010111101011
010000000010000100
000010001000100011
101001010010101011
000101010010010101
010101010101010101
101010101111010101
101010101010100101
111100001010101001
```

## Assembly code

```
mov     eax, list_msg
call    print_string
push    dword 10
push    Array
call    printArray
add     esp, 8
push    plus_one
push    dword 10
push    Array
call    map
add     esp, 12
call    print_nl
mov     eax, mapped1_msg
call    print_string
push    dword 10
push    Array
```

## CPU



# The Big (Simplified) Picture

## High-level code

```
char *tmpfilename;  
int num_schedulers=0;  
int num_request_submitters=0;  
int i,j;  
  
if (!(f = fopen(filename,"r"))) {  
    xbt_assert(0,"Cannot open file %s",filename);  
}  
while(fgets(buffer,256,f)) {  
    if (strcmp(buffer,"SCHEDULER",9))  
        num_schedulers++;  
    if (strcmp(buffer,"REQUESTSUBMITTER",16))  
        num_request_submitters++;  
}  
fclose(f);  
tmpfilename = strdup("/tmp/jobsimulator_
```

## Hand-written Assembly code

```
push    ebp  
mov     ebp, esp  
push    ebx  
mov     eax, 0  
mov     ebx, [ebp+8]  
shr     ebx, 1  
adc     eax, 0  
neg     eax  
inc     eax  
pop     ebx  
pop     ebp
```

**ASSEMBLER**

## Machine code

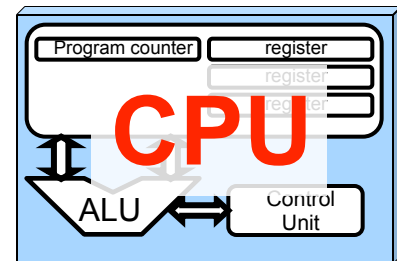
```
010000101010110110  
101010101111010101  
101001010101010001  
101010101010100101  
111100001010101001  
000101010111101011  
010000000010000100  
000010001000100011  
101001010010101011  
000101010010010101  
010101010101010101  
101010101111010101  
101010101010100101  
111100001010101001
```

## Assembly code

```
mov     eax, list_msg  
call    print_string  
push    dword 10  
push    Array  
call    printArray  
add     esp, 8  
push    plus_one  
push    dword 10  
push    Array  
call    map  
add     esp, 12  
call    print_nl  
mov     eax, mapped1_msg  
call    print_string  
push    dword 10  
push    Array
```

**COMPILER**

**CPU**



# This course's topics:

## High-level code

```
char *tmpfilename;  
int num_schedulers=0;  
int num_request_submitters=0;  
int i,j;  
  
if (!(f = fopen(filename,"r"))) {  
    xbt_assert(0,"Cannot open file %s",filename);  
}  
while(fgets(buffer,256,f)) {  
    if (strcmp(buffer,"SCHEDULER",9))  
        num_schedulers++;  
    if (strcmp(buffer,"REQUESTSUBMITTER",16))  
        num_request_submitters++;  
}  
fclose(f);  
tmpfilename = strdup("/tmp/jobsimulator_
```

## Hand-written Assembly code

```
push    ebp  
mov     ebp, esp  
push    ebx  
mov     eax, 0  
mov     ebx, [ebp+8]  
shr     ebx, 1  
adc     eax, 0  
neg     eax  
inc     eax  
pop     ebx  
pop     ebp
```

# ASSEMBLER

## Machine code

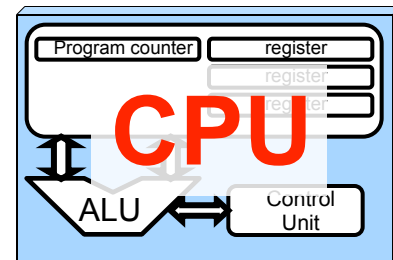
```
010000101010110110  
101010101111010101  
101001010101010001  
101010101010100101  
111100001010101001  
000101010111101011  
010000000010000100  
000010001000100011  
101001010010101011  
000101010010010101  
010101010101010101  
101010101111010101  
101010101010100101  
111100001010101001
```

## Assembly code

```
mov     eax, list_msg  
call    print_string  
push    dword 10  
push    Array  
call    printArray  
add     esp, 8  
push    plus_one  
push    dword 10  
push    Array  
call    map  
add     esp, 12  
call    print_nl  
mov     eax, mapped1_msg  
call    print_string  
push    dword 10  
push    Array
```

# COMPILER

# CPU





# What we do in this course

- First part of the semester (bulk of the course)
  - Learn how to write assembly code
    - For the x86 architecture
  - Use an assembler to generate binary code from our assembly code and then run it
- Second part of the semester (shorter, but absolutely fundamental)
  - Learn about important tools tools
    - loader, linker, **compiler**, debugger, etc.

# Why should we learn all this?

## ■ There are many “small” reasons

- Write assembly code for embedded devices
- Read generated assembly to understand malware
- Be able to develop “one-shot compilers” for non-programming languages
- Truly understand high-level concepts (indirection, data structures)
- Understand the limitations of high-level languages or why some things can be slow while others can be fast

- **Big meta-reason:** this course should go a long way in giving you a **holistic understanding** of how a program goes from just a text file to a running code
  - You can describe in details how you go from “I wrote a piece of C that calls a function that adds 2 and 2 together and prints the result” to “the computer prints 4”
    - In its full glory only after you’ve taken ICS332
  - The complexity is actually quite stunning, and there should be something satisfying in knowing how things work from top to bottom!
- 99% of students come into ICS312 thinking “why do we have to do this???”, and ~75% leave thinking “now I feel like a computer scientist!!” (and I ask ChatGPT fewer uninformed basic questions!)

- This is based on my own discussions with students/alumni, not an official study





# Important Takeaways

- Von Neumann: CPU, RAM, I/O
- Memory: stores both code and data
  - At each address a one-byte value is stored
  - A program's address space: all the bytes it uses / cares about
- CPU: a memory modifier
  - Program counter, current instruction, general-purpose registers
  - Fetch-decode-execute cycle based on a clock
  - The Instruction Set Architecture defines the machine code a CPU understands (binary-encoded instructions)
  - Assembly code is conceptually the same (but human-readable mnemonics)
- ISA: we use 32-bit x86 in this course (created by Intel)
- Compiler: high-level code to assembly code
- Assembler: assembly code to machine code

# Conclusion

- 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)
- Next week we'll have an in-class quiz on this module

