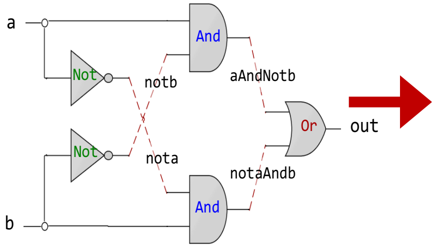
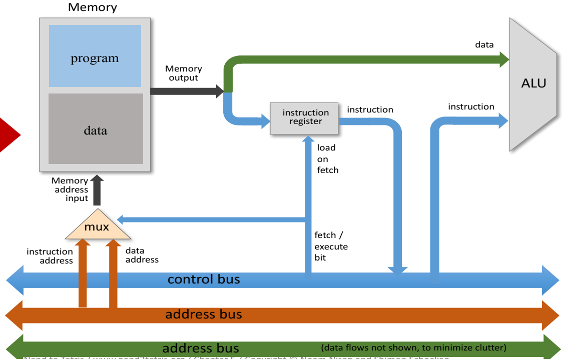
Digital Logic Design



```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```



```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 21

Instruction Set Architecture

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

Slides of first half of the course are adapted from:
https://www.nand2tetris.org
Download s/w tools required for first half of the course from the following link:
https://drive.google.com/file/d/0B9c0BdDJz6XpZUh3X2dPR1o0MUE/view

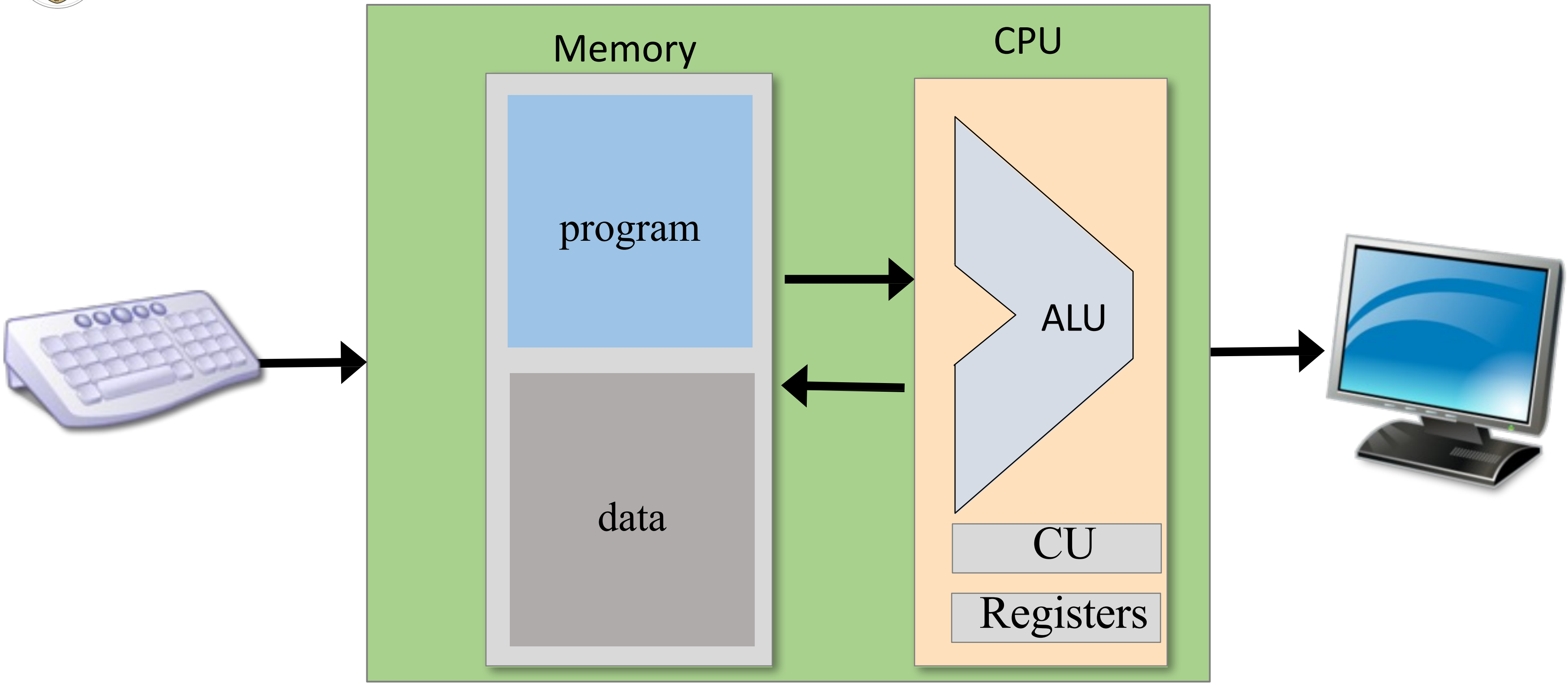
Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

- Overview of Computer System
- Universality of Computer System
- Von Neumann Architecture
- Instruction Set Architecture (ISA)
- Five Dimensions of ISA

1. Class of ISA
2. Types and Sizes of Operands
3. Operations (including control flow instructions)
4. Memory Addressing Models and Addressing Modes
5. Encoding an ISA

Overview of Computer System



- A **machine language** is an agreed-upon formalism, designed to code low-level programs as a series of machine instructions
- Using these instructions residing inside the ***Memory***, the programmer can command the ***CPU*** to fetch an instruction/data from memory/input device, perform arithmetic/logic operations on that data, and finally store result inside the memory/output device. Moreover, data may need to be moved from one ***Register*** to another and may need to test different Boolean conditions

Universality of Computer Systems

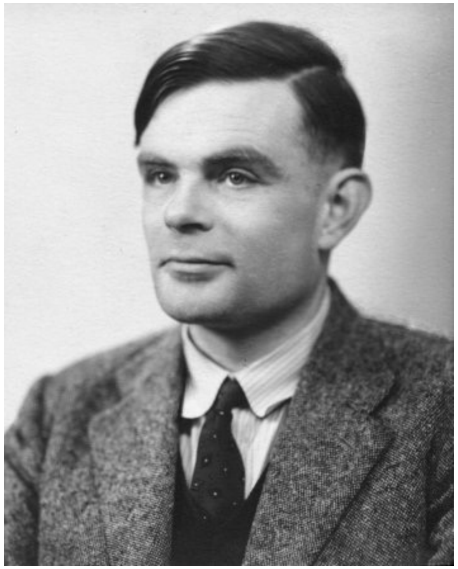
Computers Are Flexible

- Most machines in the world do one thing, e.g., washing machine washes clothes, air conditioners are used to control temperature of a room
- On the other hand a computer e.g., a smart phone can do lots and lots of things, voice communication, word processing, playing games, using internet, watch videos, and so on

Universality

Same **hardware** can run many different **software** programs

Theory



Alan Turing: (1912 – 1954)

Universal Turing Machine (1936)

Practice

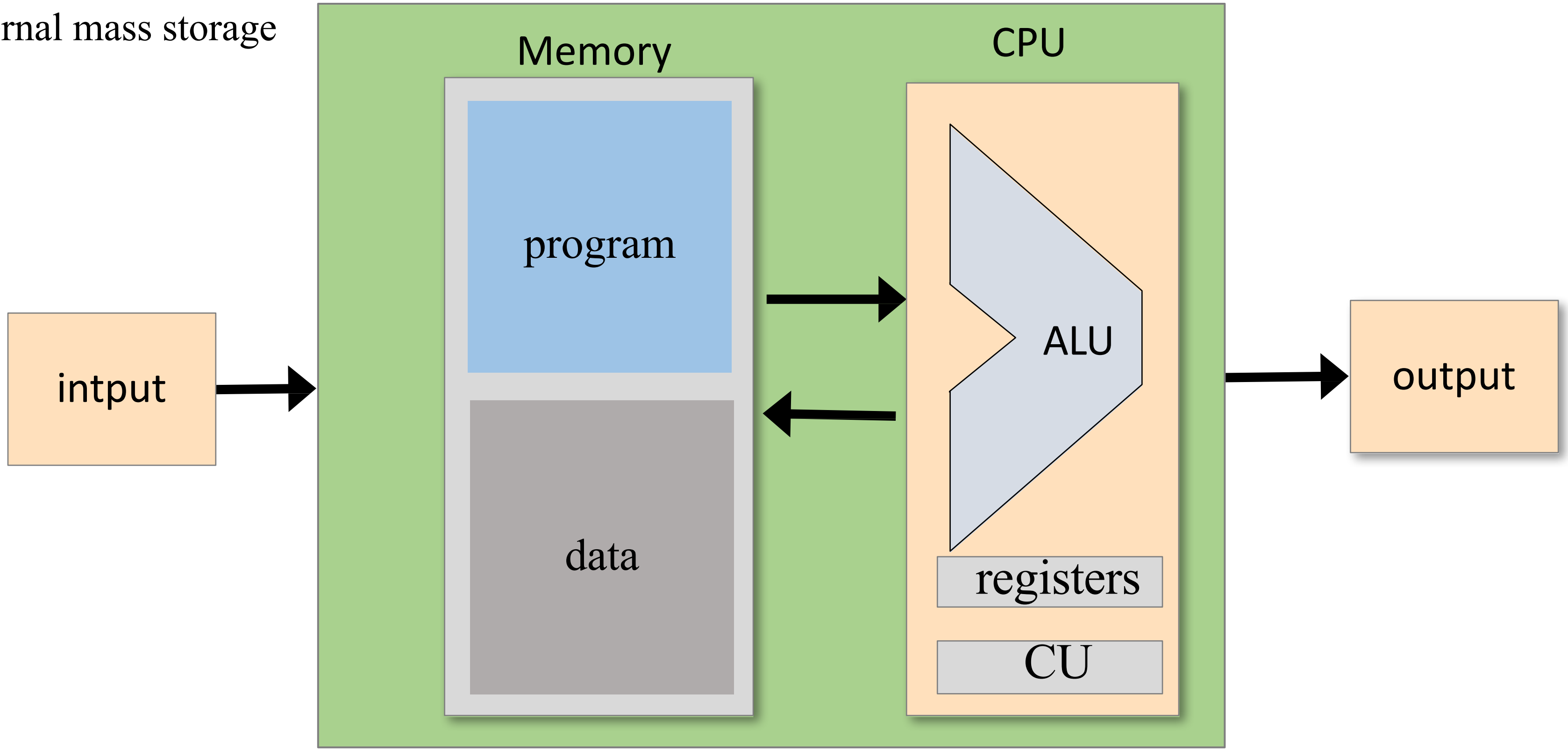
John Von Nuemann: (1903 – 1957)

Stored Program Computer

Von Neumann Architecture

The Von Neumann architecture is a computer architecture given by a mathematician and physicist John von Neumann describes the design architecture for an electronic digital computer with these components:

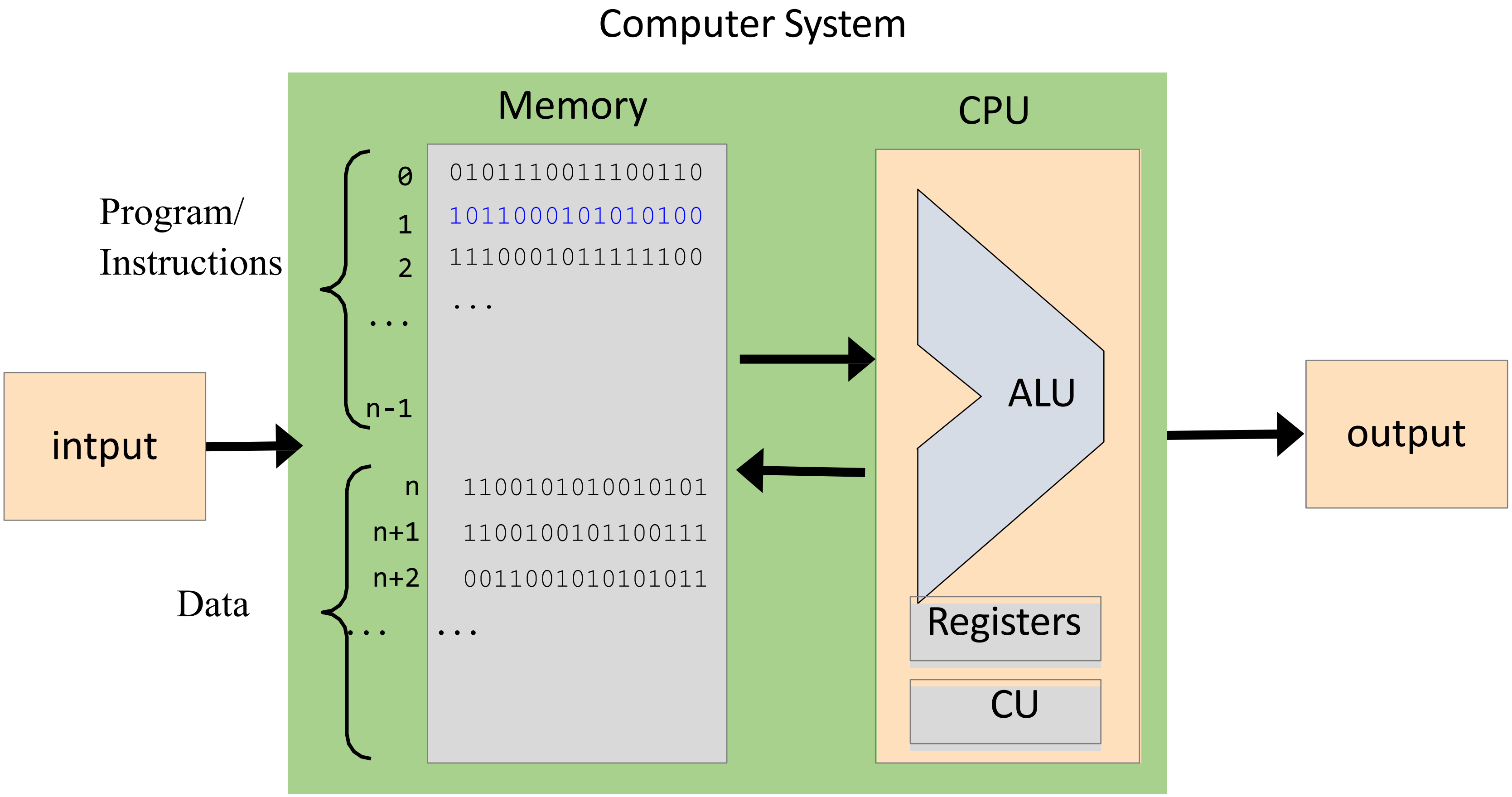
- A Processing Unit that contains an ALU and registers
- A Control Unit that contains an instruction register and program counter
- A Memory unit that stores data and instructions
- An Input and Output mechanism
- An external mass storage



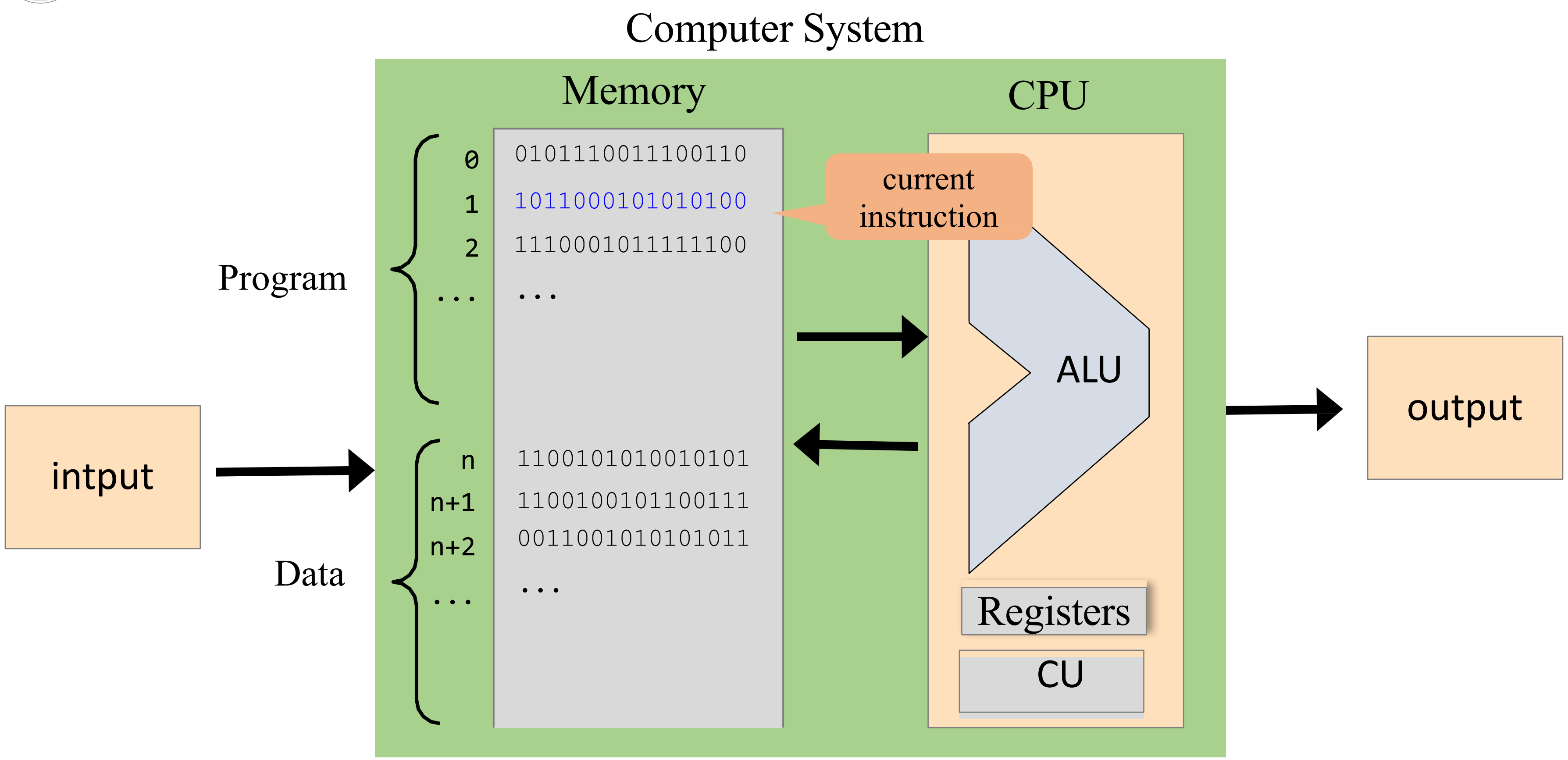
Stored Program Concept

The main idea in the Von Neumann architecture is the stored program concept. We can put the program inside the memory along with the data on which this program is going to operate. This is how

Same hardware can run many different software programs



Machine Language



Handling instructions:

- 1011means "addition" — operation
- 000101010100 means "operate on memory address340" — addressing
- Next we have to execute the instruction at address 2 — control

Assembler and Mnemonics

Instruction: `0100010 0011 0010`

add R3 R2

Sample instruction

Option 1:

- Use **machine language**, in which case a programmer should exactly know the bit positions and their meanings
- Really difficult to write programs in machine language. No one do so these days

Option 2:

- Use **symbolic machine language** instructions, using assembly language of the specific hardware, e.g., **`add R3 R2`**
- A bit easy to write programs in assembly language, but later someone has to transform it to machine language
- A program called the assembler is used to translate the symbolic code into machine code

Assembler and Symbols

- An instruction that instructs to add 1 to the contents of memory location 129, may be encoded in the machine and assembly language like:

Machine Language:

```
1010 0001 10000001
```

add 1 Mem[129]

Assembly Language:

```
add 1, Mem[129]
```

- For a programmer, it is a bit difficult to specify and memorize the memory addresses for different purposes
- A more friendlier syntax can be used if we assume that the symbol **`index`** stands for **`Mem[129]`**
- The assembler will resolve the symbol **`index`** into the specific memory address, i.e., "**`index`**" ➡ **`Mem[129]`**

Assembly Language:

```
add 1, index
```

Dimensions of ISA

Instruction Set Architecture (ISA)

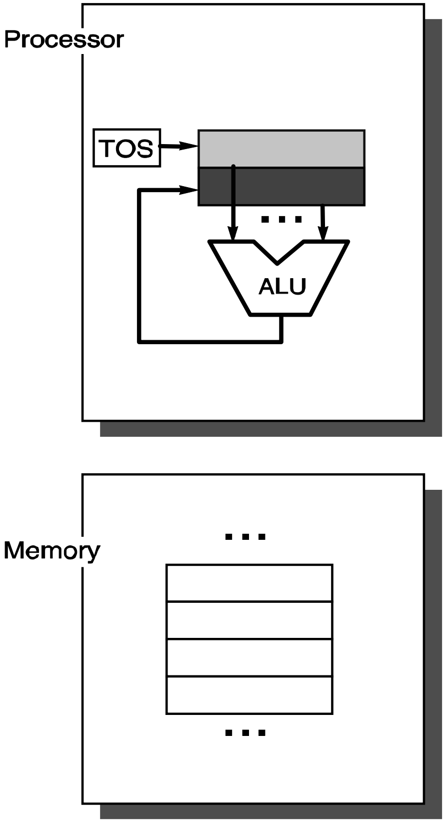
- Every computer has an ***Instruction Set Architecture (ISA)***, which is the set of instructions, registers, memory space and other features visible to the assembly language programmer
- It is an Interface between hardware and low-level software and sometimes referred to as a machine language, although it is not entirely accurate.
- Example ISAs: x86, ARM, MIPS, PowerPC, SPARC, RISC-V
- **Five dimensions of ISA:**
  1. Class of ISA
  2. Types and Sizes of Operands
  3. Operations (including control flow instructions)
  4. Memory Addressing Models and Addressing Modes
  5. Encoding an ISA

1. Classes of ISA

Stack Based Machine

- Operands are implicit at the top of the stack for ALU operation. One operand for push/pop. The result is also stored at top of stack. (Maximum number of operands allowed is one)
- Sample Code : **`a = (b+c)*d-e`**

```
push b
push c
add
push d
mul
push e
sub
pop a
```

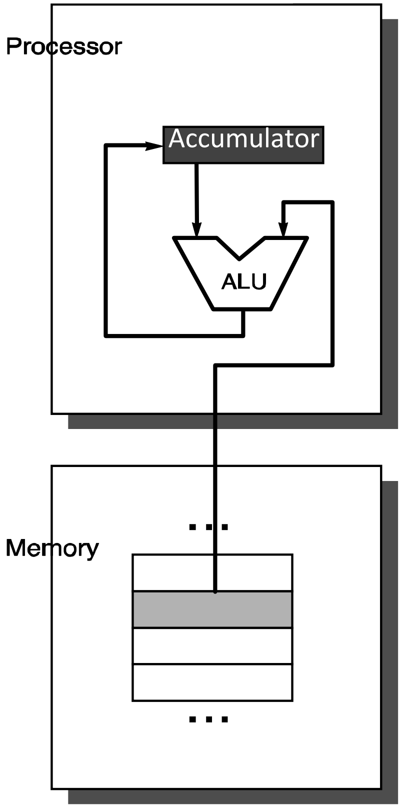


- Attributes
  - Short instructions
  - Compiler is easy to write
  - Inefficient code
- Example: Early machines are HP 3000/70. Today Java VM

Accumulator Based Machine

- One operand is in the Accumulator register (implicit) and the other is in the memory (explicit). The result is stored in the Accumulator. (Only one operand allowed)
- Sample Code : **`a = (b+c)*d-e`**

```
load b
add c
mul d
sub e
store a
```

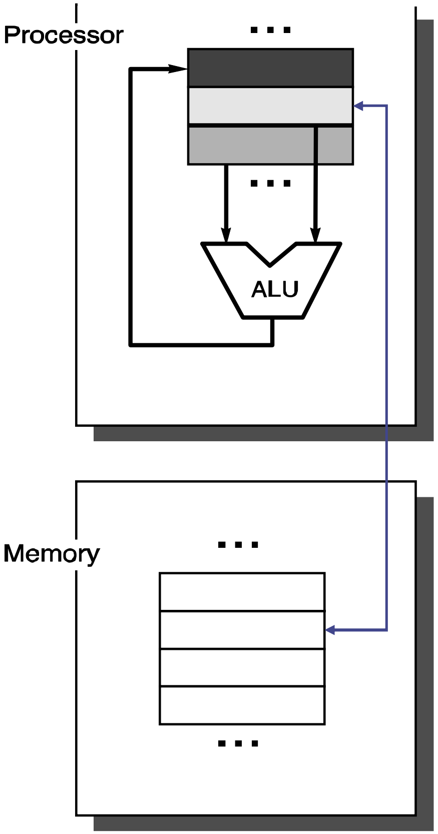


- Attributes
  - Simple design
  - Short instructions
  - Many load, store instructions
- Example: Early machines are DEC PDP-8, IBM 7090. Today used in DSP Processors

Register-Register/Load-Store Machine

- Both operands are registers. Values in memory must be loaded into a register and stored back (Maximum number of operands allowed are three)
- Sample Code : `a = (b+c)*d-e`

```
load r1, b
load r2, c
add r3, r1, r2
load r1, d
mul r4, r1, r3
load r1, e
sub r5, r4, r1
store r5, a
```

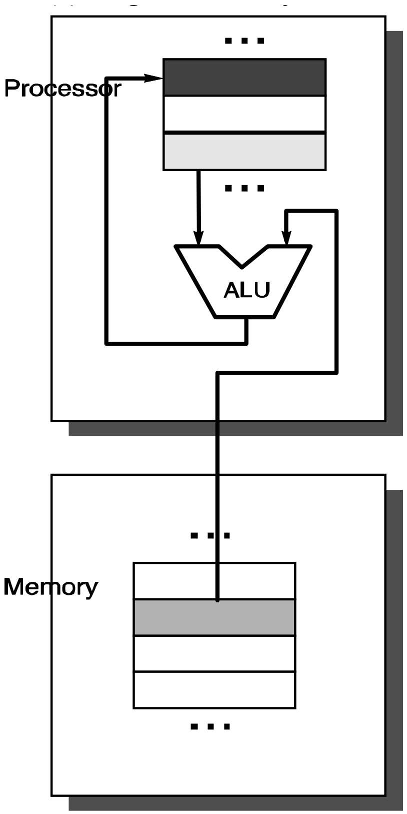


- Attributes
  - Allows fast access to temporary values
  - Reduced traffic to memory
  - Simple fixed length instructions encoding
  - Higher instruction count, and many load, store instructions
- Example: PDP-11, CRAY-1, MIPS, PowerPC, SPARC (RISC Arch)

Register-Memory Machine

- There is no implicit operand, one input operand is in register and other is in memory. (Maximum number of operands allowed are three)
- Sample Code : `a = (b+c)*d-e`

```
    load r1, b
    add r3 r1, c
    mul r4, r3, d
    sub r5, r4, e
    store r5, a
```



- Attributes
  - ➢ Small instruction count
  - ➢ Instruction length varies
  - ➢ Clock per instruction varies
  - ➢ Harder to pipeline
- Example: IBM 360/370, Motorola 68000, VAX, 8086

2. Types & Sizes of Operands

Types and Sizes of Operands

How is the type of the operand designated?

- The type of the operand is usually encoded in the opcode – e.g.,
  - LDB–load byte
  - LDW–load word
- Common operand types: (imply their sizes)
  - Character (8 bits or 1 byte)
  - Half word (16 bits or 2 bytes)
  - Word (32 bits or 4 bytes)
  - Double word (64 bits or 8 bytes)
  - Single precision floating point (4 bytes or 1 word)
  - Double precision floating point (8 bytes or 2 words)

Registers

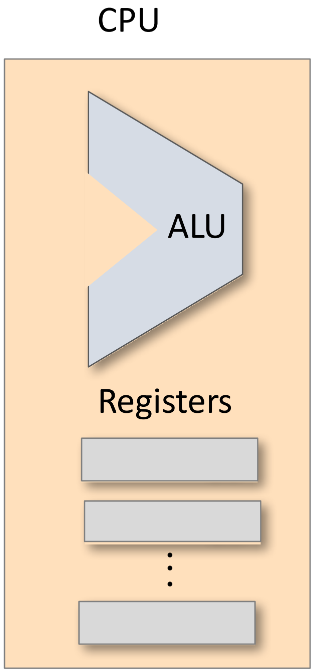
- The smallest amount of memory that actually resides inside the CPU is called Registers. Every CPU typically contains a few, easily accessed registers built from the fastest technology available. Their number and functions are a central part of the machine language
- Two most important registers that every architecture have are Instruction Pointer and Instruction Register

➢ **Program Counter / Instruction Pointer:** This register stores the address of the next instruction to be executed by the CPU

0000000000001001

➢ **Instruction Register:** This register is used to contain and later decode the instruction to be executed by the CPU

00100010 0011 0010



Registers (cont…)

- Other than PC and IP, there are many other registers that are used to store data and memory addresses. They are accessible to assembly language programmers and their number vary from architecture to architecture

Data Registers: There are various data registers, which are used to store temporary data during any ongoing operations. Their contents can be accessed by the assembly programmer

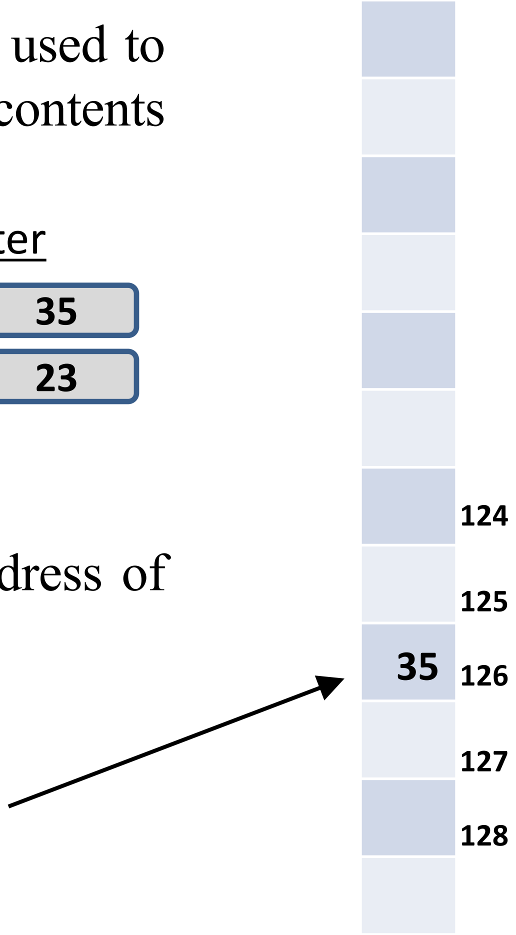
```
add R1, R2
```

	Before	After
R1:	12	35
R2:	23	23

Address Registers: These registers are used to hold the address of the location to be accessed from memory

```
store @A, R2
```

A:	126
R2:	35



3. Operations

Machine Operations

- Usually correspond to the operations that the hardware is designed to support
- Most computers generally provide full set of operations for the first three categories, i.e., arithmetic/logical, data transfer and control

Operator Type	
Arithmetic and Logical	Integer arithmetic and logical operations: add, subtract, multiply, divide, and, or, not.
Data Transfer	Move instructions with memory addressing
Control	Branch, jump, procedure call, return, trap
System	Synchronization, memory management instructions
Floating Point	Add, subtract, multiply, divide, compare
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression and decompression operation

Machine Operations (cont…)

- The basic operations that we are interested right now are:
  - **Arithmetic Operations:** add, subtract, ….

    `ADD R2, R1, R3.` // R2 <-- R1 + R3 where R1, R2, R3 are registers

    `ADD R2,R1,foo` // R2 <-- R1 + foo where foo stands for the value of the memory location pointed at by the user-defined label foo

  - **Logical Operations:** and, not, or, ...

    `AND R1,R1,R2` // R1 <-- Bitwise AND of R1 and R2

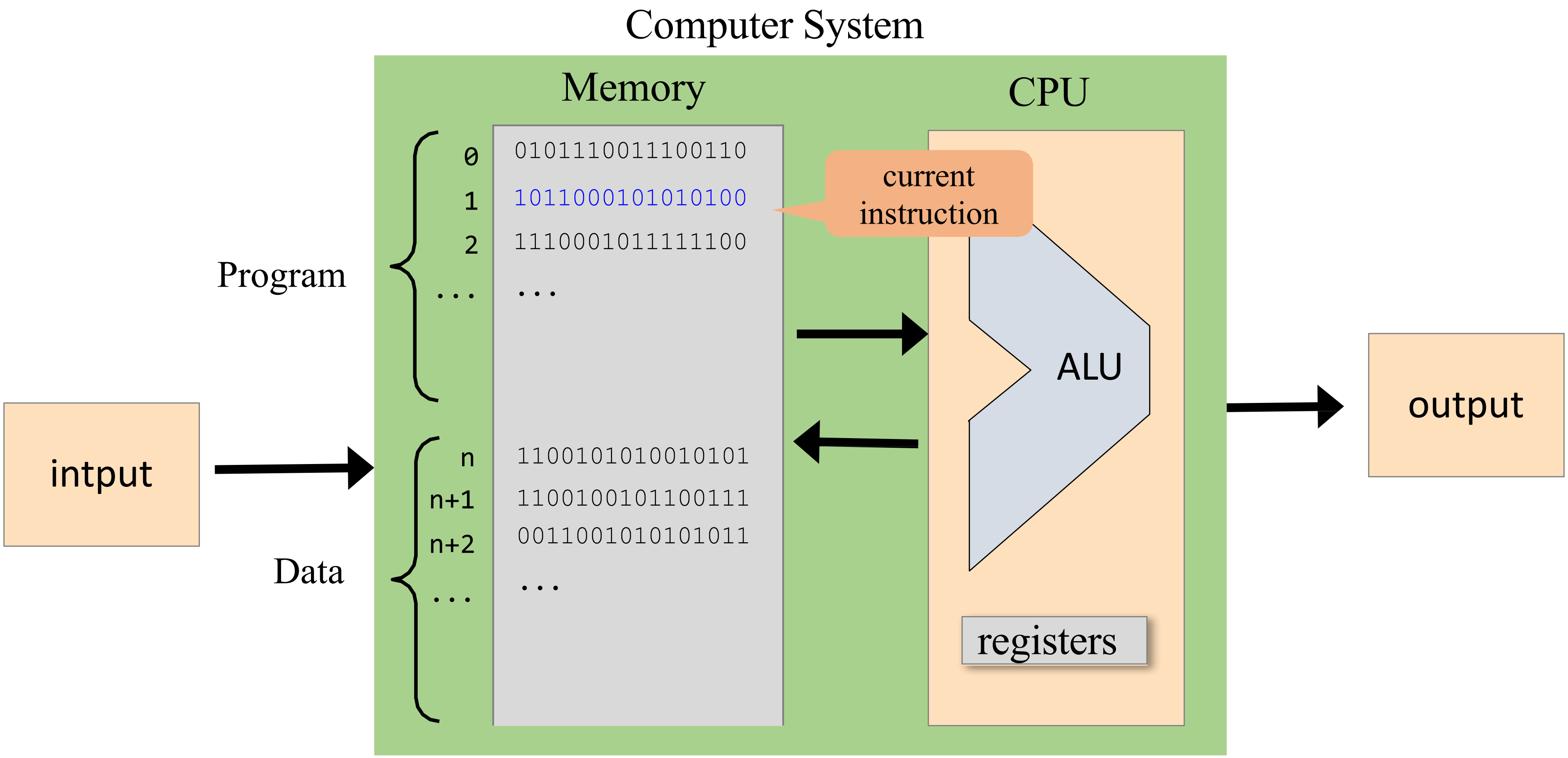
  - **Flow Control:** Flow control instructions change the flow of control, i.e., instead of executing the next instruction, the program branches to the address specified in the branching instructions. Four types of control instructions are conditional branches, unconditional branches, procedure calls and procedure returns

    `goto 200` // shift the flow of control to instruction at address 200

    `if cond goto 200` //if true shift the flow of control to instruction at addr 200

4. Memory Addressing Models & Addressing Modes

Computer System



Addressing Modes

- In assembly language programming, the term addressing modes refers to the way in which the operand of an instruction is specified. Different architectures support different addressing modes
- When an instruction requires two operands, the first operand is generally the destination, which contains data in a register or memory location and the second operand is the source. Generally, the source data remains unaltered after the operation
- The four basic modes of addressing are:
  - ➢ Register addressing mode
  - ➢ Immediate Addressing mode
  - ➢ Direct / Absolute addressing mode
  - ➢ Register Indirect addressing mode

Addressing Modes Example

R1 = 400, R2 = 50, R3 = 27, R4 = 60, R5 = 500

Register: In register addressing mode, both the operands are placed in general purpose registers, and the register codes are specified in the instruction

```
add R1, R2 // R1 ← R1+ R2
```

Immediate: In this mode, one operand is in register and other is part of the instruction as a constant. Its limitation is that the range of constant/operand is restricted by available bits in the instruction

```
add R3, 54 // R3 ← R3+ 54
```

Direct/Absolute: In this addressing mode, one operand is in register and the other is in memory, whose effective address is part of the instruction

```
add R4, M[750] // R4 ← M[750] + R4
```

Register InDirect: In this addressing mode, one operand is in register and other is in the memory, whose address is placed in a register, which is specified in the instr.

```
add R2, @R5 // R2 ← M[R5] + R2
```

348	200
349	250
350	60
398	350
399	450
400	700
499	500
500	800
750	50
800	300
1000	65
1001	99

Addressing the I/O Devices

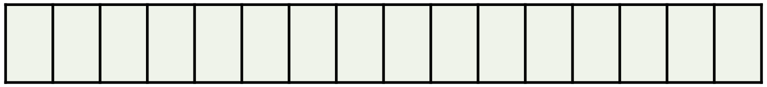
- The way a microprocessor need to read/write different memory locations, similarly the microprocessor also need to read/write different I/O devices like the keyboard, mouse, monitor, printer, etc. This linking is also be called I/O Interfacing. An I/O interface acts as a communication channel between the processor and the externally interfaced device. The interfacing of the I/O devices can be done in two ways
  - **Memory Mapped I/O Interfacing:** Both memory and I/O devices have same address space. So addressing capability of memory become less because some part is occupied by the I/O. In memory mapped I/O, there are same read-write instructions for memory and I/O devices, so CPUs are cheaper, faster and easier to build
  - **Isolated I/O Interfacing:** The I/O devices are given a separate addressing region (separate from the memory). These separate address spaces are known as ‘Ports’. In isolated I/O, there are different read-write instructions for memory and I/O devices. x86-64 use Isolated I/O

Note: Data can be transferred between CPU and I/O devices in three modes, namely Program controlled I/O, Interrupt initiated I/O, and Direct Memory Access

5. Encoding of ISA

Encoding of ISA

- What is the size of an instructions
  - Fixed length: MIPS
  - Variable length: X86_64 (1-15 Bytes)
- How to encode the operations?
- How to encode the operands?
- How to encode the addressing modes?
- How to manage the flow control mechanism?



Real World ISAs

Architecture	Type	# Opr	Data Size	Registers	Addr Size	Use
x86	Reg-Mem	2	8/16/32/64	4/8/24	16/32/64	Personal Computers
ARM	Reg-Reg	3	32/64	16	32/64	Cell phones, embedded
MIPS	Reg-Reg	3	32/64	32	32/64	Work station, embedded
Alpha	Reg-Reg	3	64	32	64	Work station
SPARC	Reg-Reg	3	32/64	24-32	32/64	Work station, embedded
IBM360	Reg-Mem	2	32	16	24/31/64	Main frame
VAX	Mem-Mem	3	32	16	32	Mini Computer

Things To Do

Coming to office hours does NOT mean you are academically weak!