

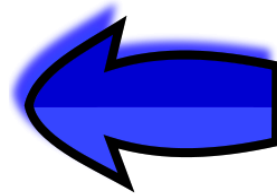


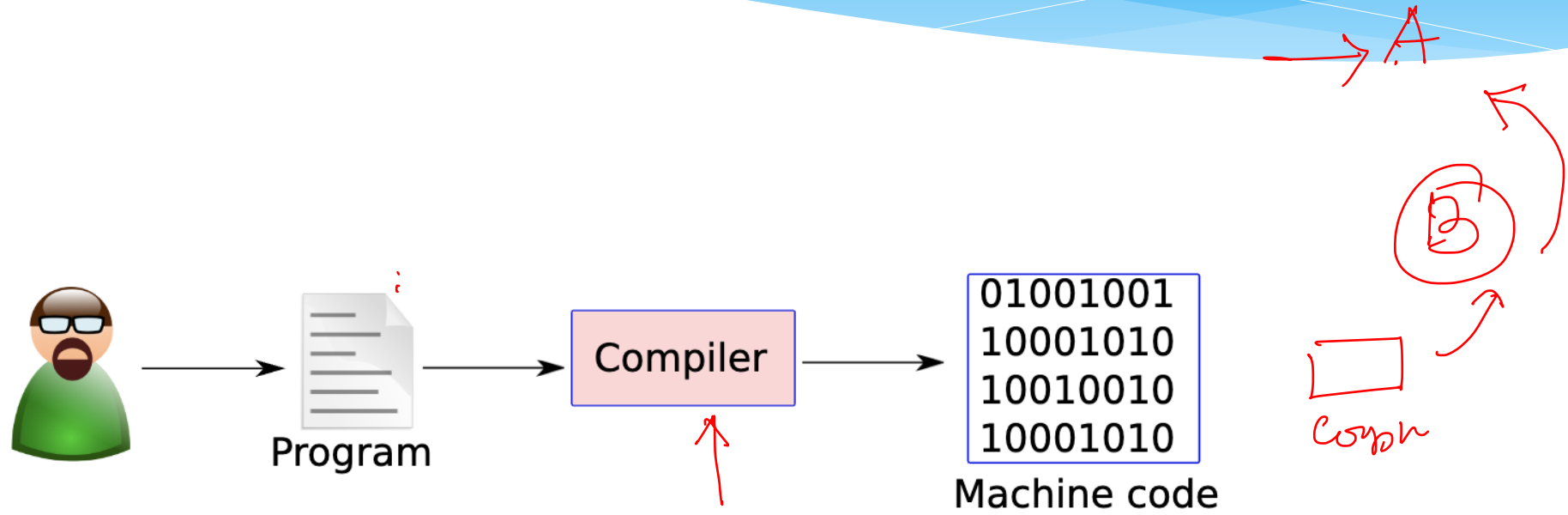
# **Basic Computer Architecture**

## **Chapter 3: Assembly Language**

# Outline

- \* Overview of Assembly Language
- \* Assembly Language Syntax
- \* SimpleRisc ISA
- \* Functions and Stacks
- \* SimpleRisc Encoding





01-...

Assembly → Assembler → MC.

Cross-compiler

# What is Assembly Language

- \* A **low level programming language** uses simple statements that correspond to typically just one machine instruction. These languages are specific to the ISA.
- \* The term “**assembly language**” refers to a family of low-level programming languages that are specific to an ISA. They have a generic structure that consists of a sequence of assembly statements.
- \* Typically, each assembly statement has **two parts**: (1) an instruction code that is a mnemonic for a basic machine instruction, and (2) a list of operands.

# Why learn Assembly Language ?

<https://www.tiobe.com/tiobe-index/>

- \* Software developers' perspective
  - \* Write **highly efficient code**
    - \* Suitable for the core parts of games, and mission critical software
  - \* Write code for operating systems and device drivers
  - \* Use features of the machine that are **not supported** by standard programming languages

# Assemblers

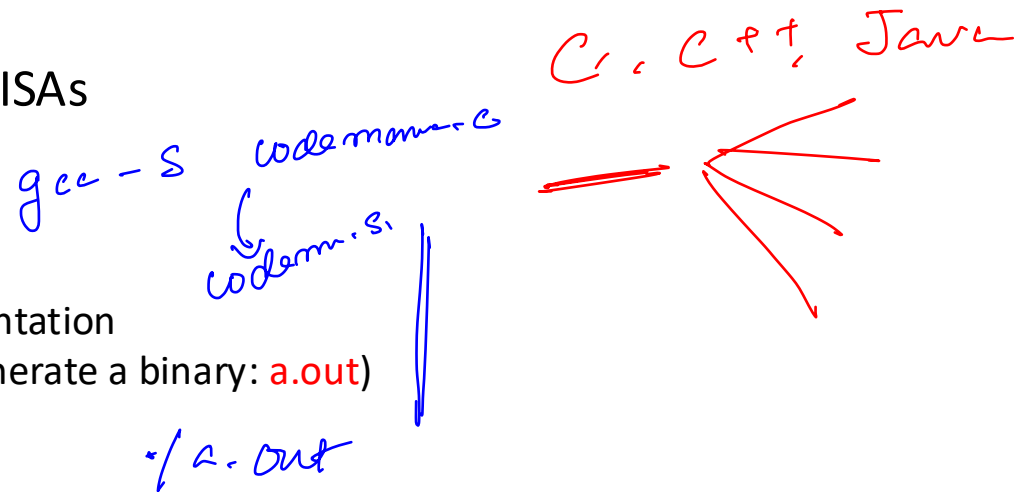
- \* **Assemblers are programs** that convert programs written in low level languages to machine code (0s and 1s)

- \* **Examples :**

- \* nasm, tasm, and masm for x86 ISAs

- \* On a linux system try :

- \* `gcc -S <filename.c>`
    - \* filename.s is its assembly representation
    - \* Then type: `gcc filename.s` (will generate a binary: **a.out**)

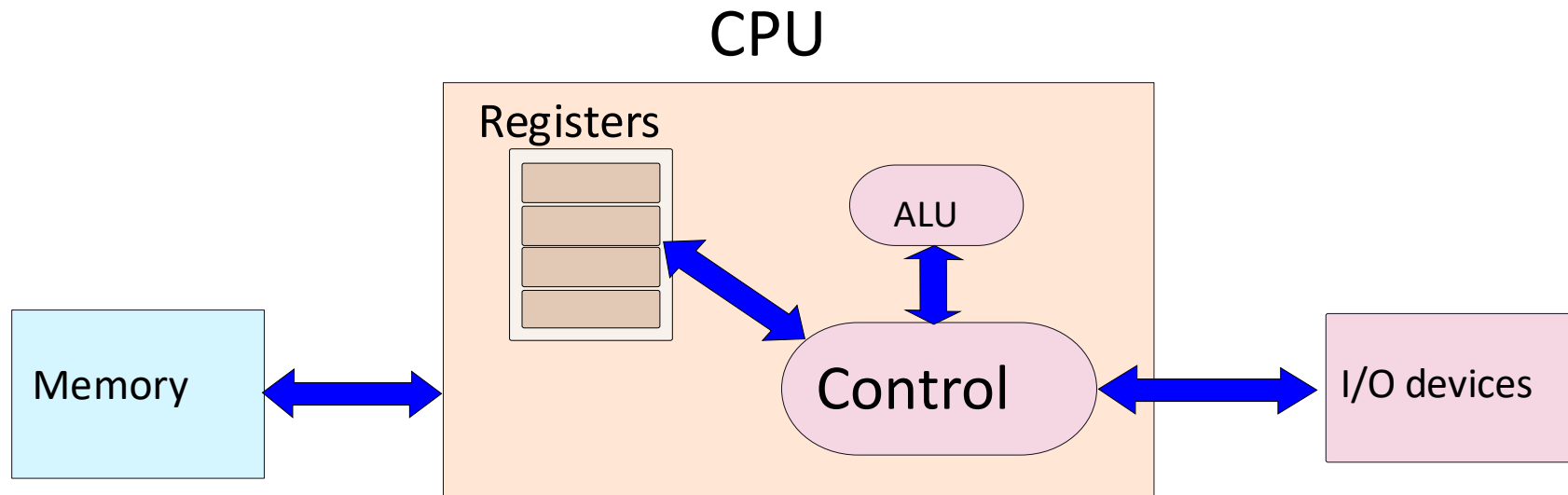


# Hardware Designers Perspective

- \* Learning the assembly language is the same as learning the intricacies of the instruction set
- \* Tells HW designers : what to build ?



# Machine Model – Von Neumann Machine with Registers



# View of Registers

- \* **Registers** → named storage locations

- \* in ARM : r0, r1, ... r15

- \* in x86 : eax, ebx, ecx, edx, esi, edi

- \* Machine specific registers (MSR)

- \* Examples : Control the machine such as the speed of fans, power control settings

- \* Read the on-chip temperature.

- \* Registers with special functions :

- \* stack pointer

- \* program counter

- \* return address

# View of Memory



- \* Memory
  - \* One large array of bytes
  - \* Each location has an **address**
  - \* The address of the first location is 0, and increases by 1 for each subsequent location
- \* The program is stored in a part of the memory
- \* The **program counter** contains the **address** of the current instruction

# Storage of Data in Memory

- \* Data Types

- \* `char` (1 byte), `short` (2 bytes), `int` (4 bytes), `long int` (8 bytes)

- \* How are multibyte variables stored in memory ?

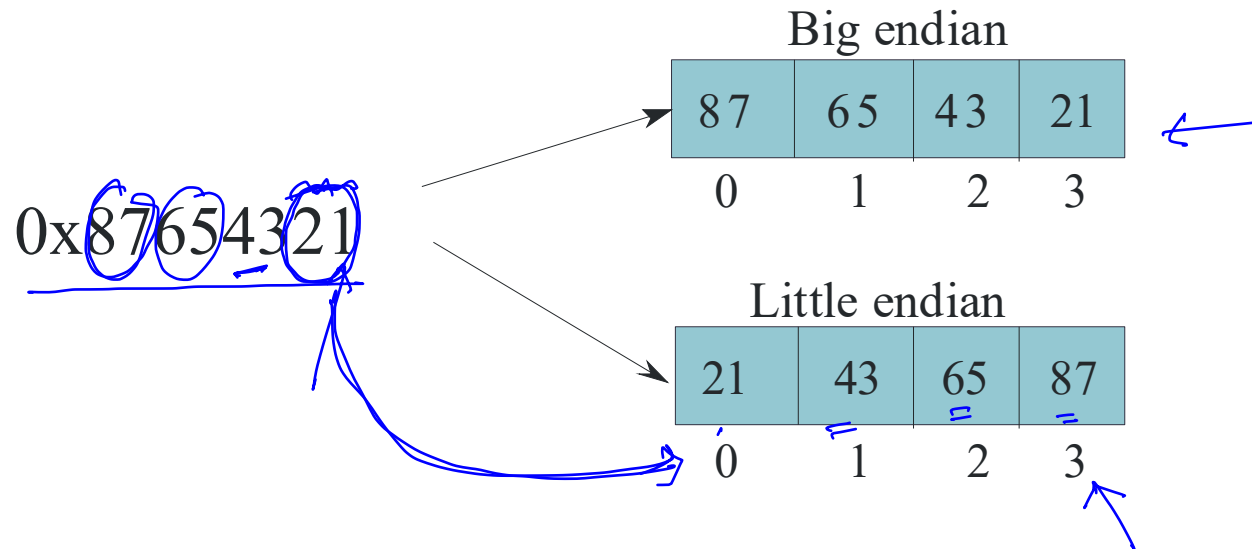
- \* Example : How is a 4 byte integer stored ?

- \* Save the 4 bytes in consecutive locations

- \* Little endian representation (used in ARM and x86) → The LSB is stored in the lowest location

- \* Big endian representation (Sun Sparc, IBM PPC) → The MSB is stored in the lowest location

# Little Endian vs Big Endian



\* Note the order of the storage of bytes

x86 processors use the little endian forma

Early versions of ARM  
processors used to be little endian

# Storage of Arrays in Memory

- \* Single dimensional arrays. Consider an array of integers : `a[100]`



- \* Each integer is stored in either a little endian or big endian format

- \* 2 dimensional arrays :

`int a[100]` ✓

- \* `int a[100][100]` ✓

- \* `float b[100][100]` ✓

- \* Two methods : row major and column major

# Row Major vs Column Major

- \* **Row Major** (C, Python)

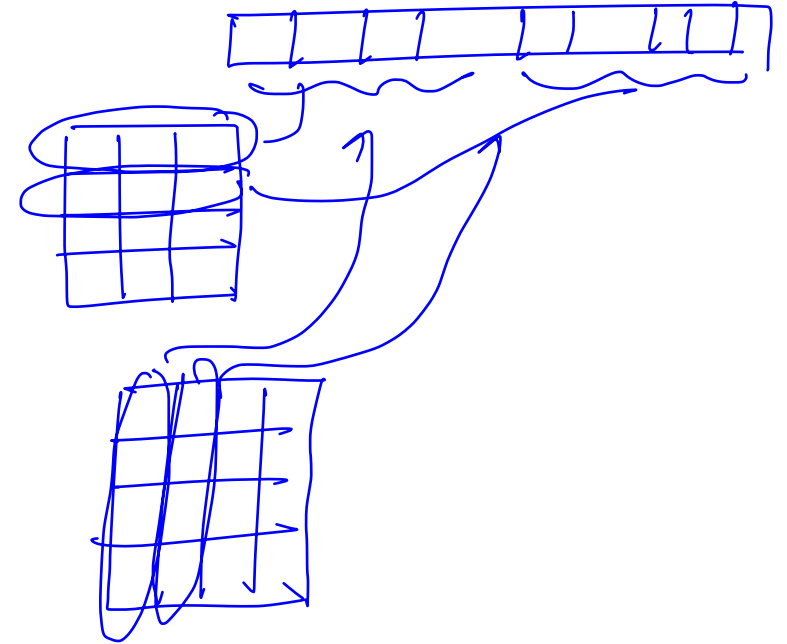
- \* Store the first row as an 1D array
- \* Then store the second row, and so on...

- \* **Column Major** (Fortran, Matlab)

- \* Store the first column as an 1D array
- \* Then store the second column, and so on

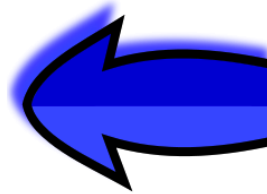
- \* **Multidimensional arrays**

- \* Store the entire array as a sequence of 1D arrays

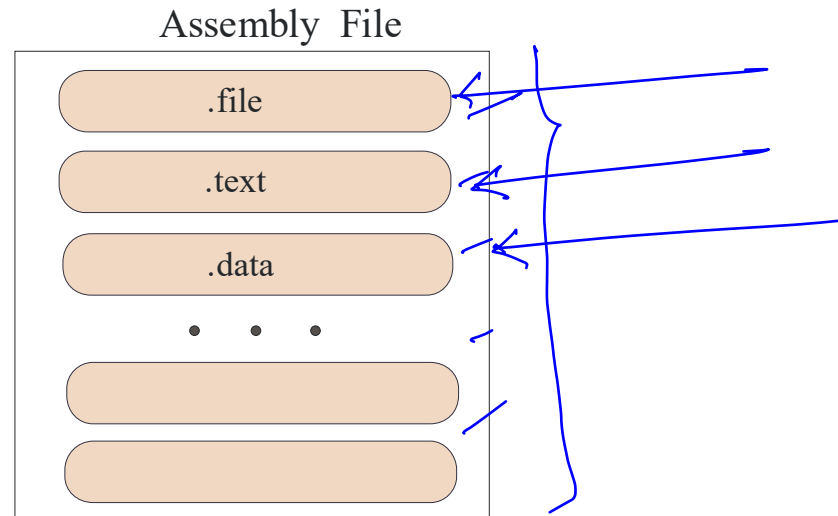


# Outline

- \* Overview of Assembly Language
- \* Assembly Language Syntax
- \* SimpleRisc ISA
- \* Functions and Stacks
- \* SimpleRisc Encoding



# Assembly File Structure : GNU Assembler



- \* Divided into different **sections**
- \* Each section contains some data, or assembly instructions

# Meaning of Different Sections

- \* .file

- \* name of the source file

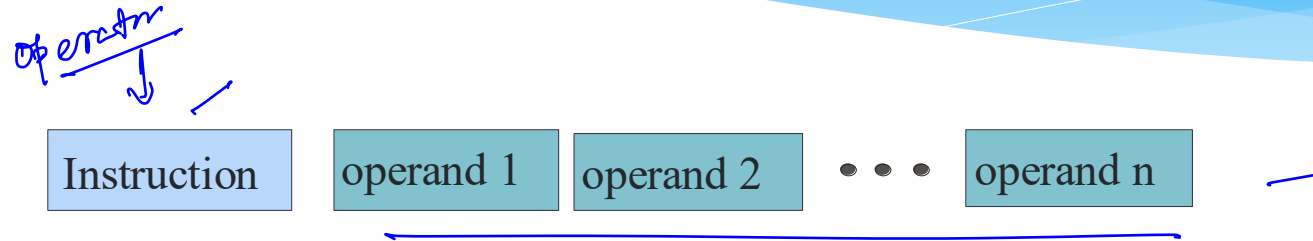
- \* .text

- \* contains the list of instructions

- \* .data

- \* data used by the program in terms of read only variables, and constants

# Structure of a Statement



- \* instruction

- \* textual identifier of a machine instruction

- \* operand

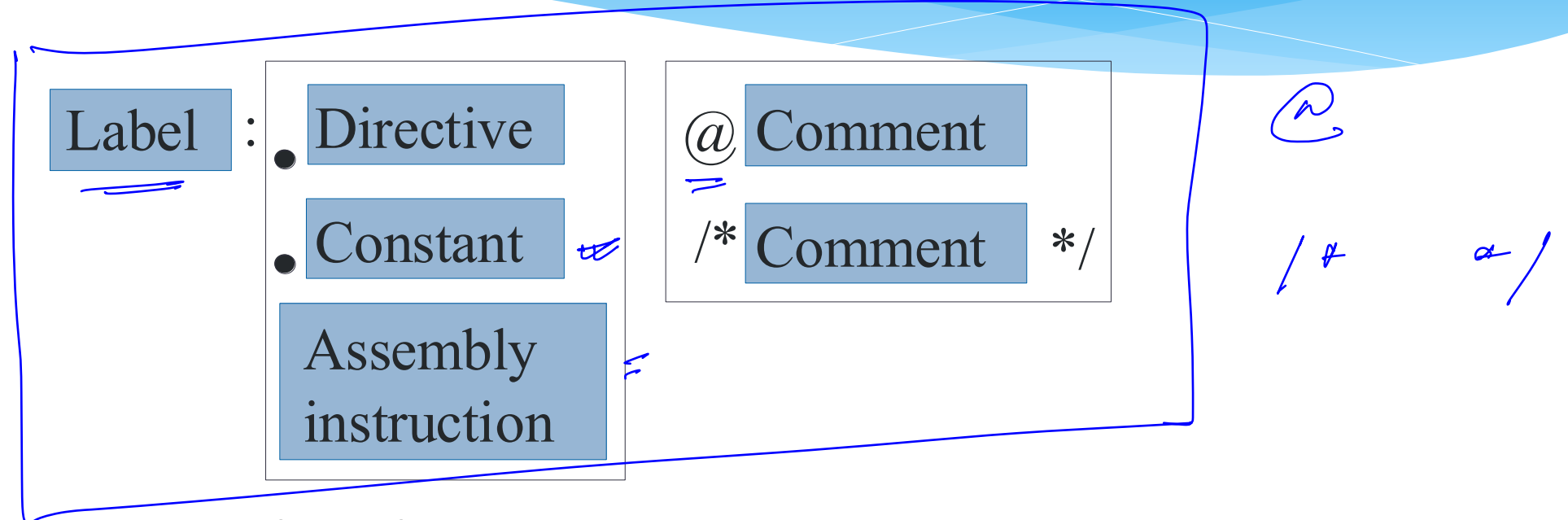
- \* **constant** (also known as an immediate)
  - \* register
  - \* **memory location**

# Examples of Instructions

✓  $\downarrow$   
sub  $r3, r1, r2$       $r3 = r1 - r2$   
mul  $r3, r1, r2$       $r3 = r1 \times r2$

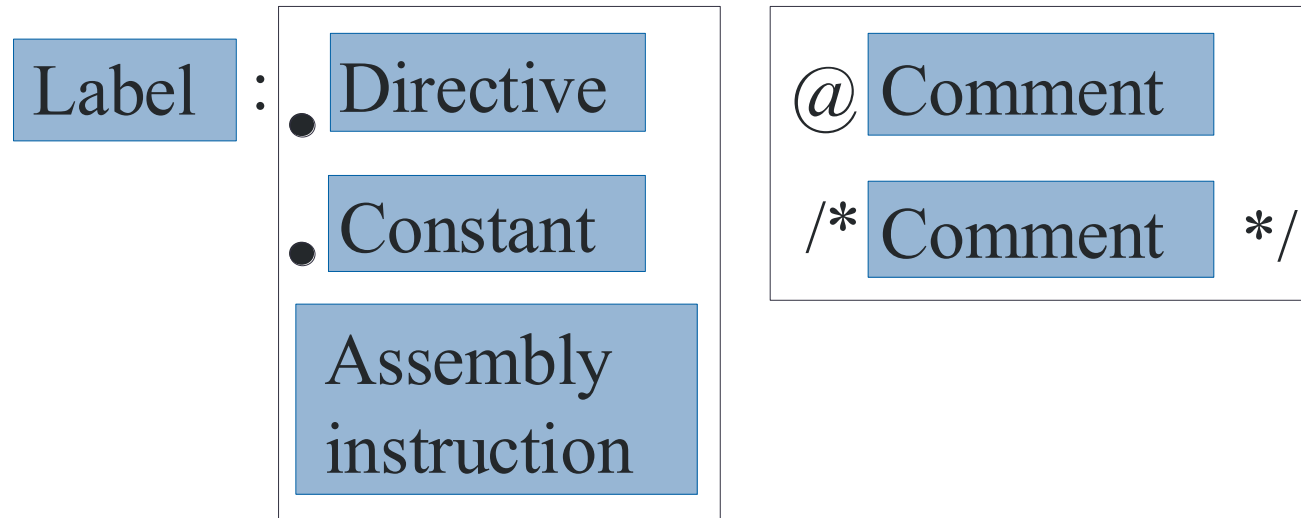
- \* **subtract** the contents of  $r2$  from the contents of  $r1$ , and save the result in  $r3$
- \* **multiply** the contents of  $r2$  with the contents of  $r1$ , and save the results in  $r3$

# Generic Statement Structure



- \* **label** → identifier of a statement
- \* **directive** → tells the assembler to do something like declare a function
- \* **constant** → declares a constant

# Generic Statement Structure - II



- \* **assembly statement** → contains the assembly instruction, and operands
- \* **comment** → textual annotations ignored by the assembler

# Types of Instructions

- \* **Data Processing** Instructions

- \* add, subtract, multiply, divide, compare, logical or, logical and

- \* **Data Transfer** Instructions

- \* transfer values between registers, and memory locations

- \* **Branch** instructions

- \* branch to a given label

- \* **Special** instructions

- \* interact with peripheral devices, and other programs, set machine specific parameters

# Nature of Operands

- \* Classification of instructions

- \* If an instruction takes **n** operands, then it is said to be in the **n-address** format

- \* Example : add r1, r2, r3 (3 address format)

Sub r1, r2, r3  
          └──────────┘  
          Operands

- \* Addressing Mode

- \* The method of specifying and accessing an operand in an assembly statement is known as the **addressing mode**.

# Register Transfer Notation

- \* This notation allows us to specify the semantics of instructions

- \*  $r1 \leftarrow r2$

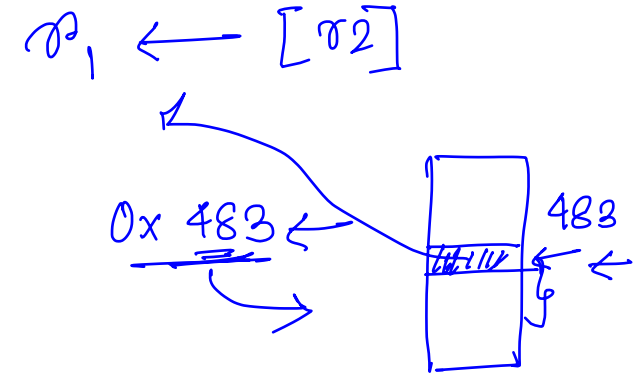
- \* **transfer** the contents of register r2 to register r1

$$r_1 \leftarrow r_2$$

- \*  $r1 \leftarrow r2 + 4$

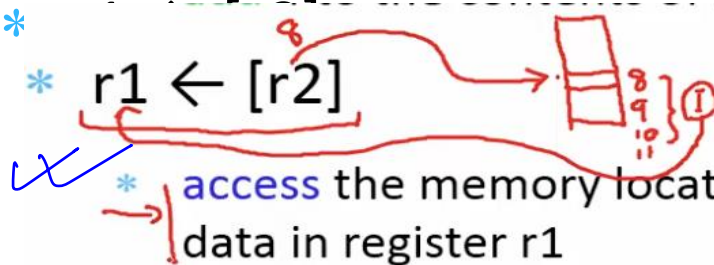
- \* **add** 4 to the contents of register r2, and transfer the contents to register r1

$$r_1 \leftarrow r_2 + 4$$



- \*  $r1 \leftarrow [r2]$

- \* **access** the memory location that matches the contents of r2, and store the data in register r1



# Addressing Modes

- \* Let  $V$  be the value of an operand, and let  $r1, r2$  specify registers

- \* Immediate addressing mode

- \*  $V \leftarrow \text{imm}$ , e.g. 4, 8, 0x13, -3

$$r_1 \leftarrow 4$$

$r_2$

- \* Register direct addressing mode

- \*  $V \leftarrow r1$

- \* e.g.  $r1, r2, r3 \dots$

$$r_1 \leftarrow r_2$$

- \* Register indirect

- \*  $V \leftarrow [r1]$

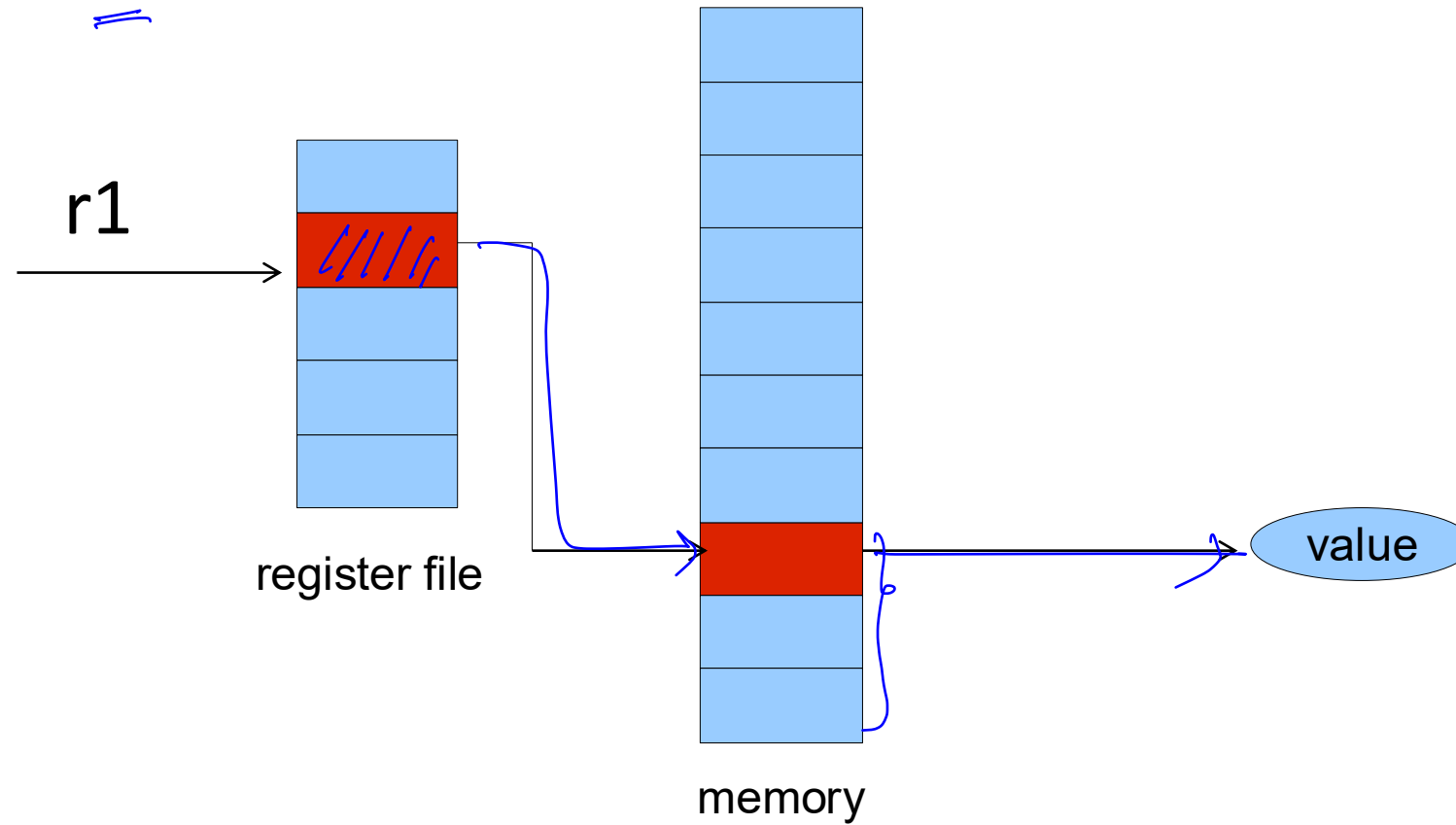
$$r_1 \leftarrow [r_2]$$

$$r_1 \leftarrow [r_2 + 10] = 10[r_2]$$

- \* Base-offset :  $V \leftarrow [r1 + \text{offset}]$ , e.g.  $20[r1]$  ( $V \leftarrow [20+r1]$ )

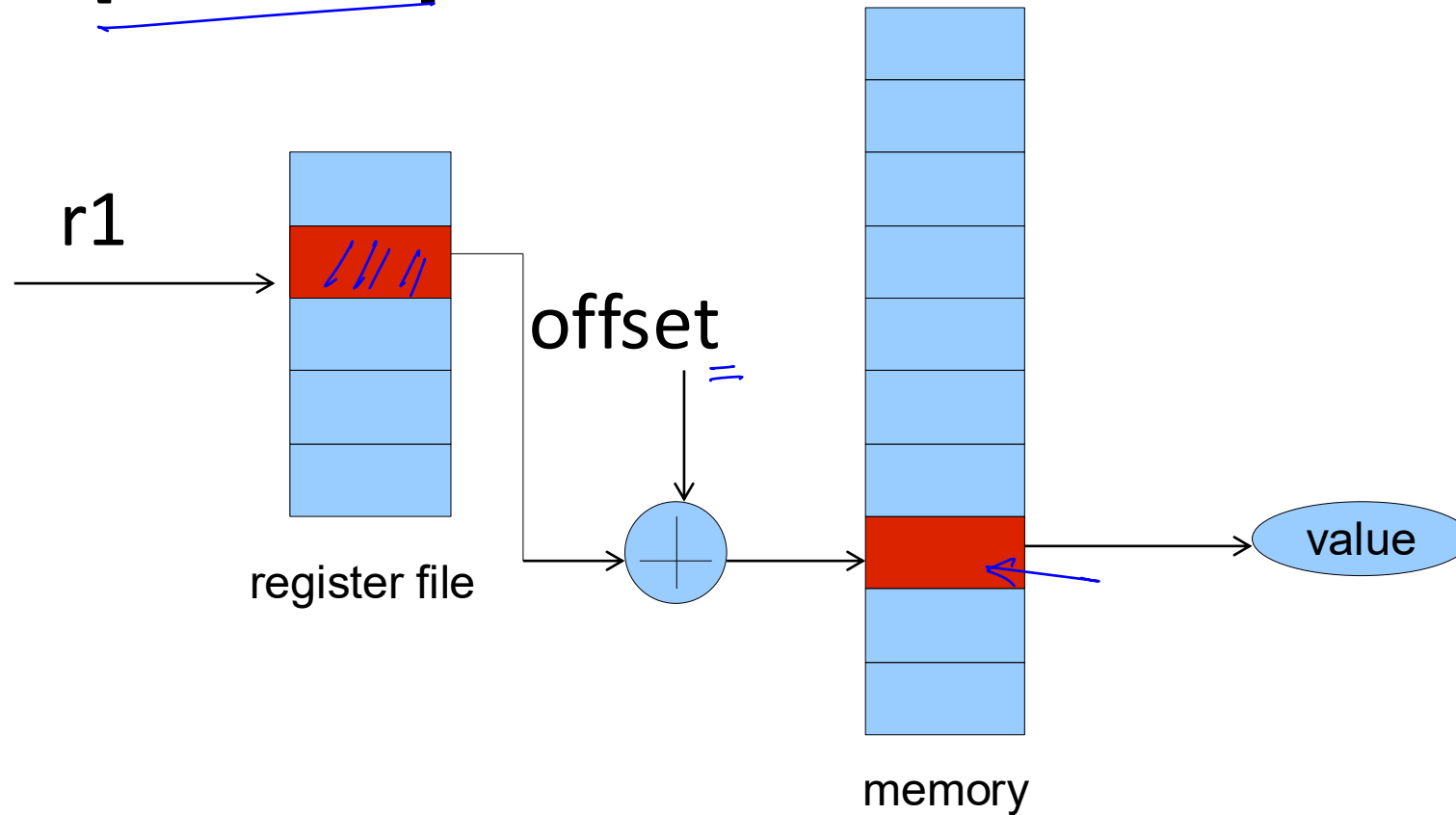
# Register Indirect Mode

\*  $V \leftarrow [r1]$



# Base-offset Addressing Mode

\*  $V \leftarrow [r1 + \text{offset}]$



# Addressing Modes - II

## \* Base-index-offset ✓

$$* V \leftarrow [r1 + r2 + \text{offset}]$$

\* example: 100[r1,r2] ( $V \leftarrow [r1 + r2 + 100]$ )

$$r_2 \leftarrow [r_1 + r_2 + 10] + 10 \uparrow$$

## \* Memory Direct ✓

$$* V \leftarrow [\text{addr}]$$

\* example : [0x12ABCD03]

## \* PC Relative ✓

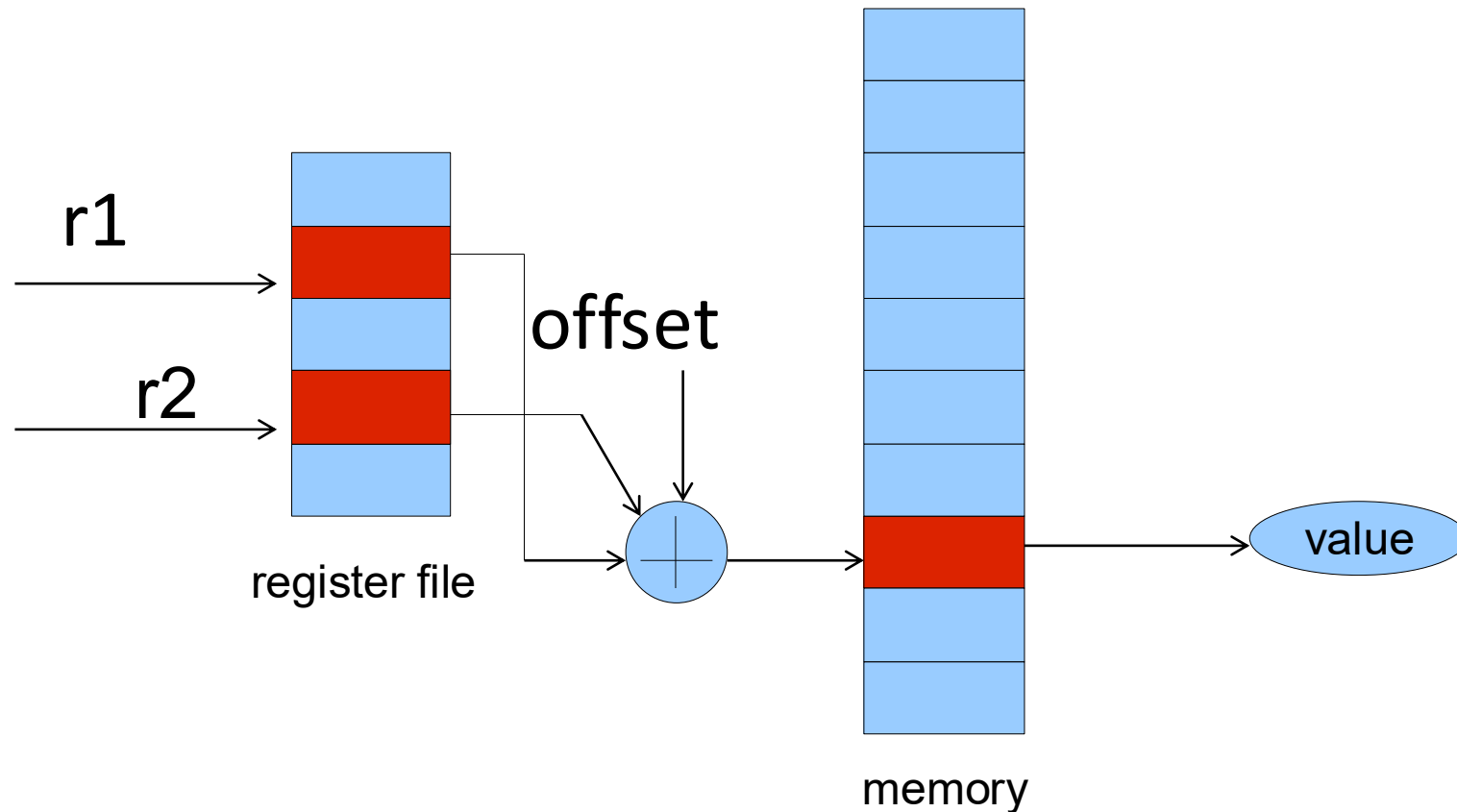
$$* V \leftarrow [\text{pc} + \text{offset}] *$$

\* example: 100[pc] ( $V \leftarrow [\text{pc} + 100]$ )

$$r_1 \leftarrow [\text{PC} + \text{offset}]$$

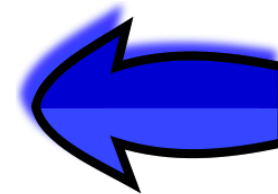
# Base-Index-Offset Addressing Mode

\*  $V \leftarrow [r1+r2 + \text{offset}]$



# Outline

- \* Overview of Assembly Language
- \* Assembly Language Syntax
- \* SimpleRisc ISA
- \* Functions and Stacks
- \* SimpleRisc Encoding



# SimpleRisc

- \* Simple RISC ISA
- \* Contains only 21 instructions
- \* We will design an assembly language for SimpleRisc
- \* Design a simple binary encoding,
- \* and then implement it ...



# Survey of Instruction Sets

ISA	Type	Year	Vendor	Bits	Endianness	Registers
VAX	CISC	1977	DEC	32	little	16
SPARC	RISC	1986	Sun	32	big	32
	RISC	1993	Sun	64	bi	32
PowerPC	RISC	1992	Apple,IBM,Motorola	32	bi	32
	RISC	2002	Apple,IBM	64	bi	32
PA-RISC	RISC	1986	HP	32	big	32
	RISC	1996	HP	64	big	32
m68000	CISC	1979	Motorola	16	big	16
	CISC	1979	Motorola	32	big	16
MIPS	RISC	1981	MIPS	32	bi	32
	RISC	1999	MIPS	64	bi	32
Alpha	RISC	1992	DEC	64	bi	32
x86	CISC	1978	Intel,AMD	16	little	8
	CISC	1985	Intel,AMD	32	little	8
	CISC	2003	Intel,AMD	64	little	16
ARM	RISC	1985	ARM	32	bi(little default)	16
	RISC	2011	ARM	64	bi(little default)	31

# Registers

- \* SimpleRisc has 16 registers
  - \* Numbered : r0 ... r15
  - \* r14 is also referred to as the stack pointer (sp)
  - \* r15 is also referred to as the return address register (ra)
- \* View of Memory
  - \* Von Neumann model ✓
  - \* One large array of bytes ✓
- \* Special flags register → contains the result of the last comparison
  - \* flags.E = 1 (equality), flags.GT = 1 (greater than)

Comp  $r_1, r_2$

$r_1 == r_2$

$r_1 > r_2$

# mov instruction

mov r1,r2	$r1 \leftarrow r2$ ✓
mov r1,3	$r1 \leftarrow 3$ ✓

mov r1,r2

- \* Transfer the contents of one register to another
- \* Or, transfer the contents of an immediate to a register
- \* The value of the immediate is embedded in the instruction
  - \* SimpleRisc has 16 bit immediates (2's comp)
  - \* Range  $-2^{15}$  to  $2^{15} - 1$