

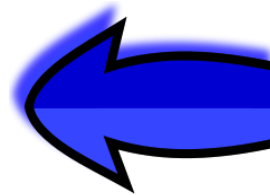


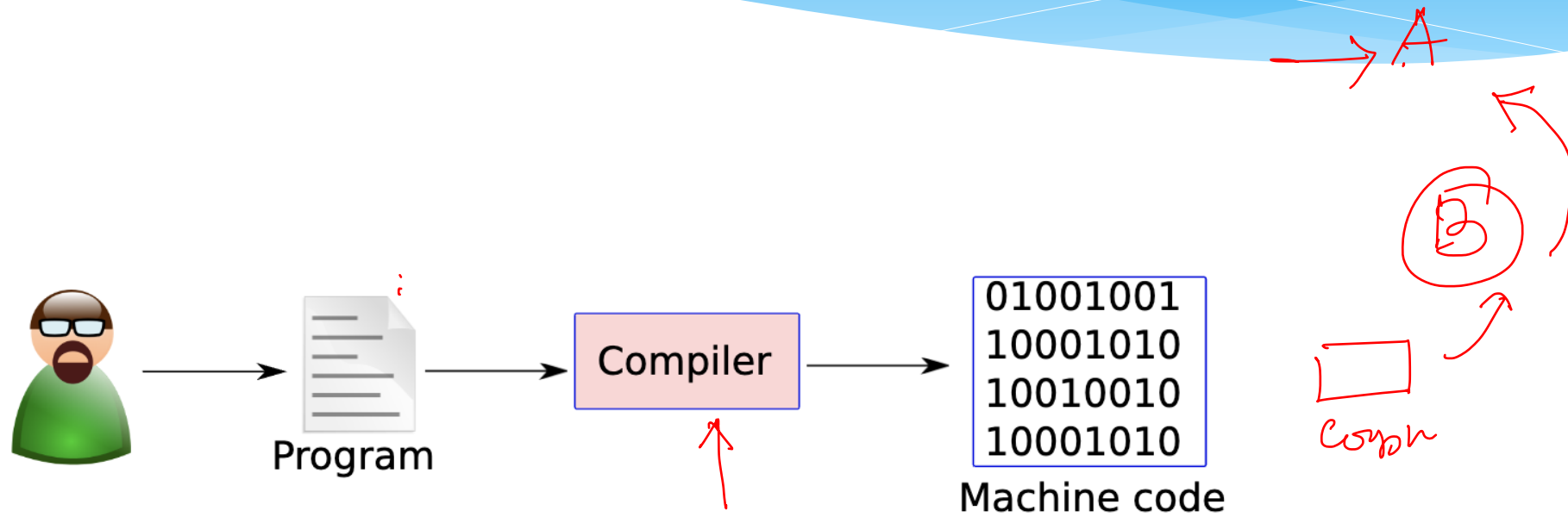
Basic Computer Architecture

Chapter 3: Assembly Language

Outline

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





01-...



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

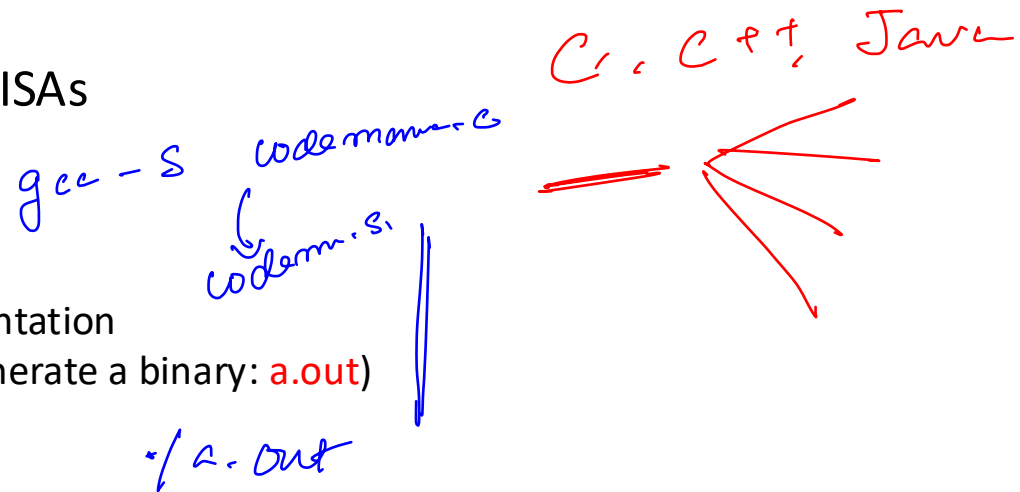
- * **Assemblers are programs** that convert programs written in low level languages to machine code (0s and 1s)

- * **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**)

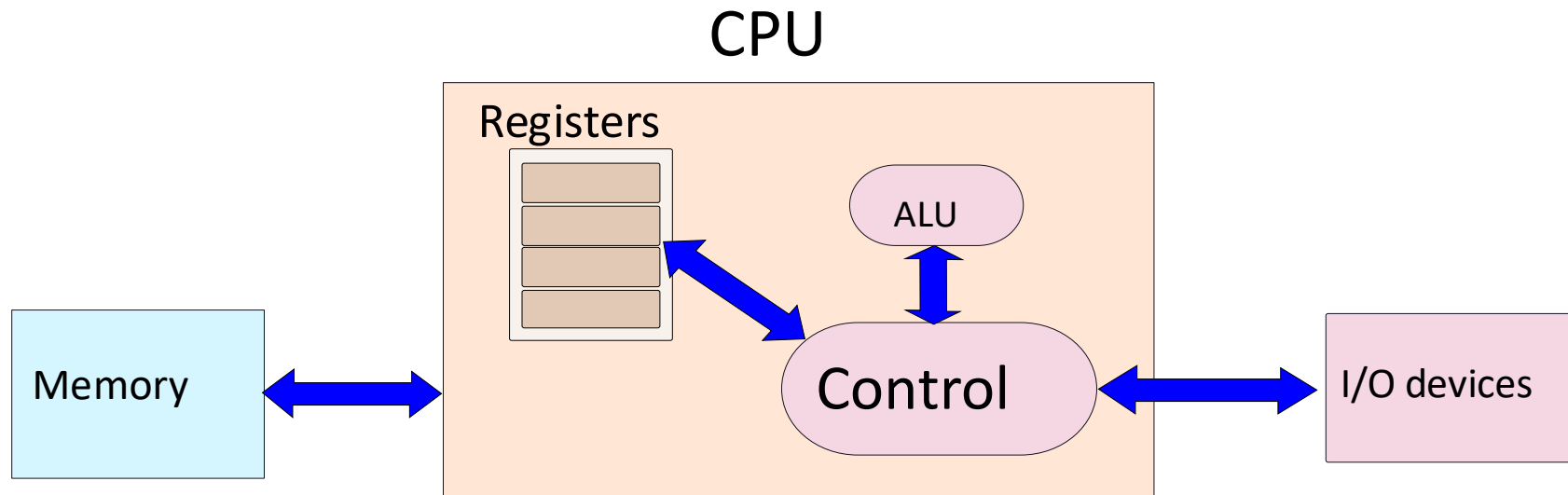


Hardware Designers Perspective

- * 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 ?

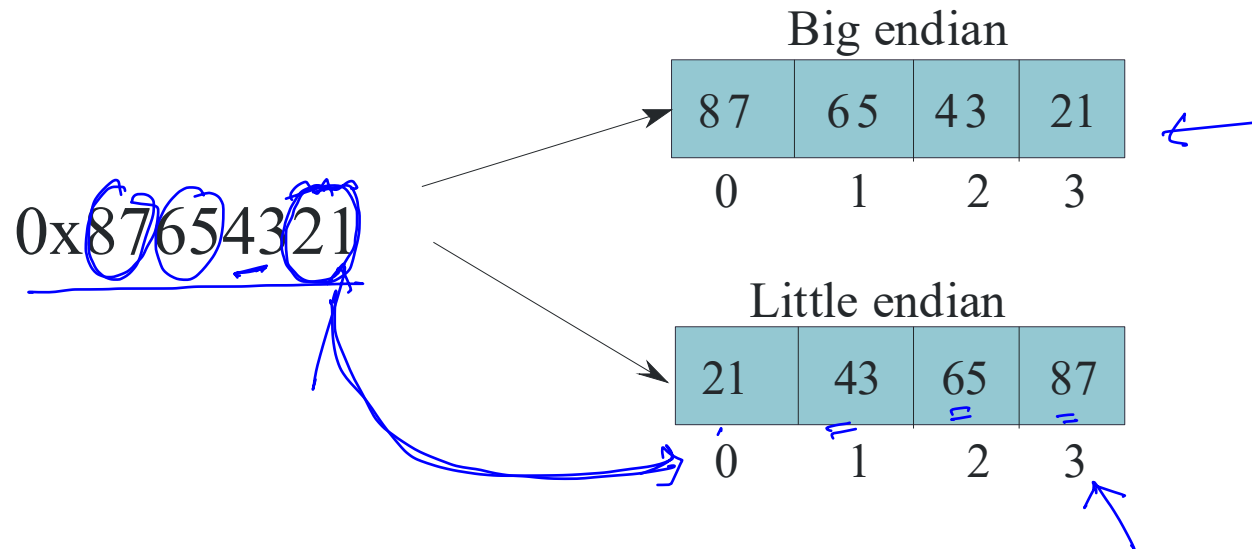
- * Example : How is a 4 byte integer stored ?

- * 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)

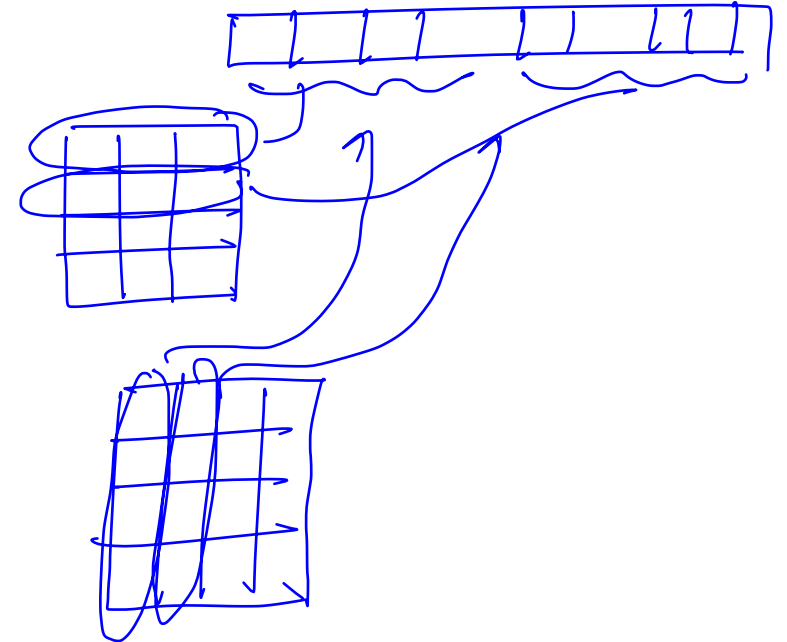
- * Store the first row as an 1D array
- * Then store the second row, and so on...

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

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

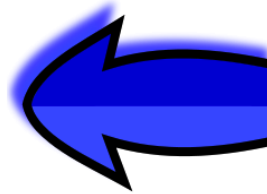
- * **Multidimensional arrays**

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

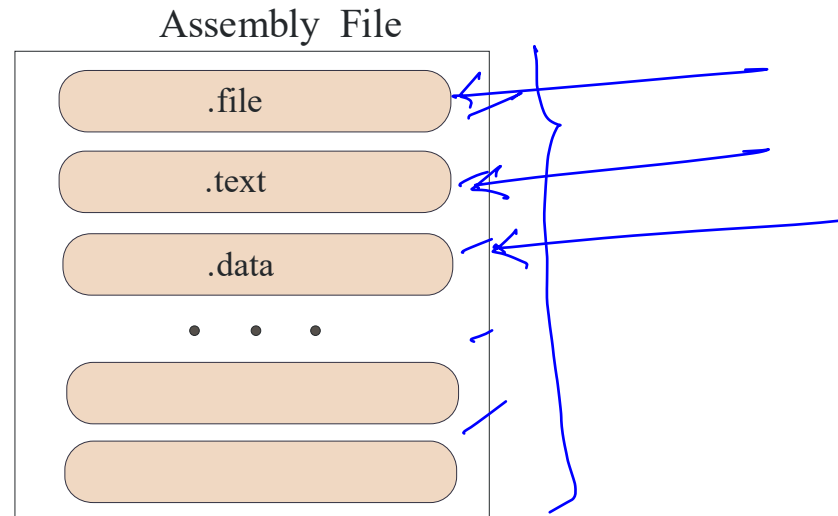


Outline

- * 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

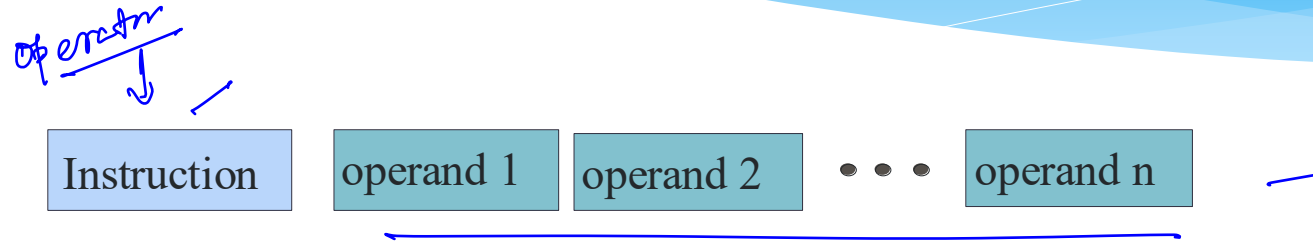
- * .text

- * 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

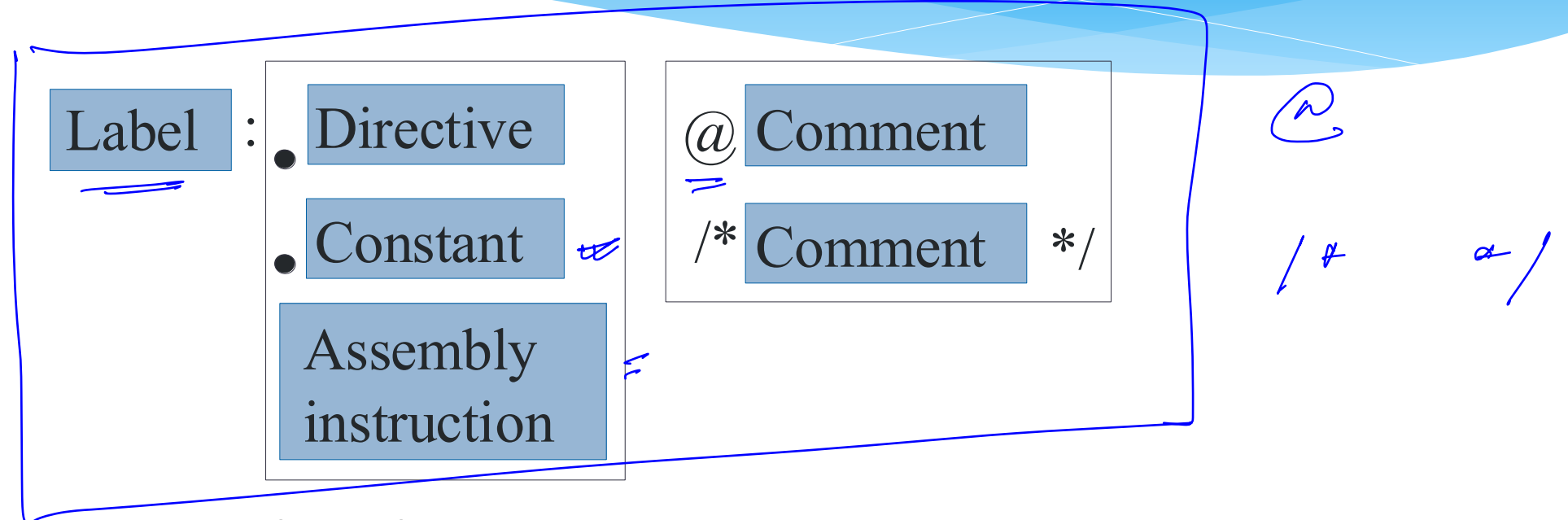
- * **constant** (also known as an immediate)
 - * register
 - * **memory location**

Examples of Instructions

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

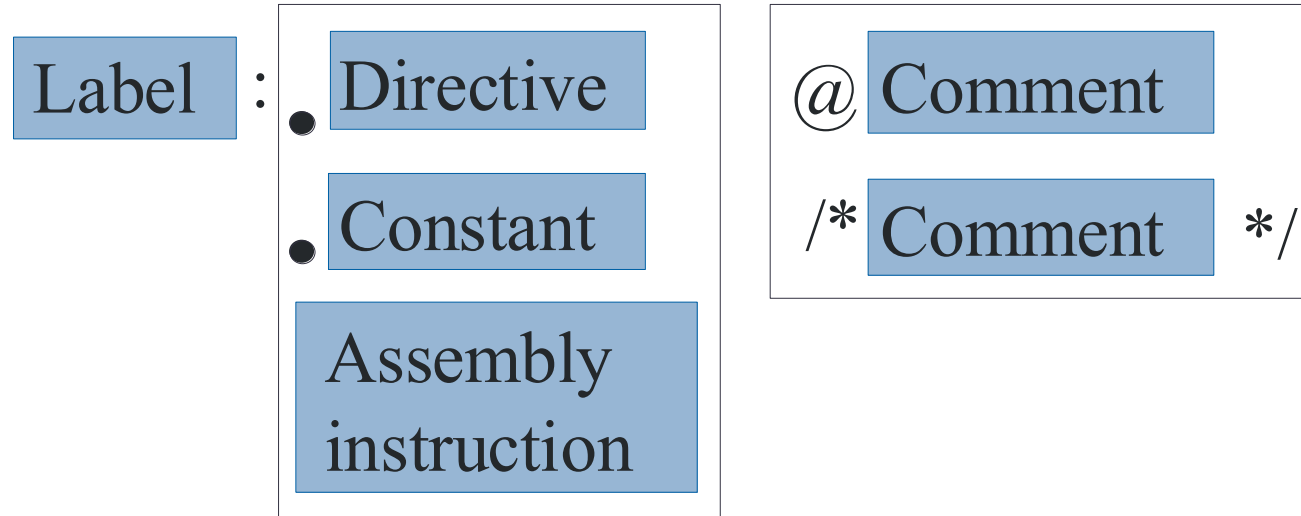
- * **subtract** the contents of $r2$ from the contents of $r1$, and save the result in $r3$
- * **multiply** the contents of $r2$ with the contents of $r1$, and save the results in $r3$

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

- * $r1 \leftarrow r2$

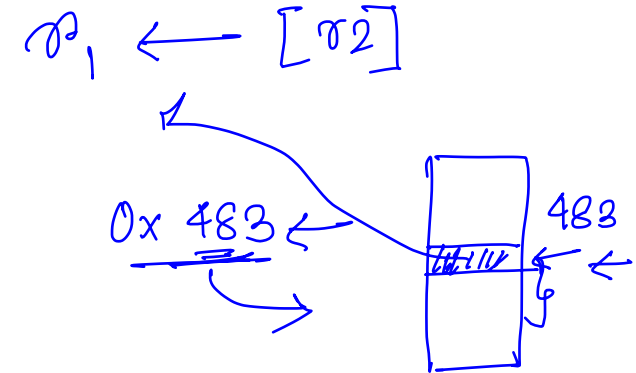
- * **transfer** the contents of register r2 to register r1

$$r_1 \leftarrow r_2$$

- * $r1 \leftarrow r2 + 4$

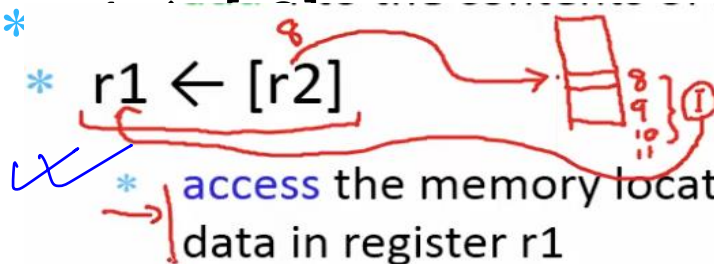
- * **add** 4 to the contents of register r2, and transfer the contents to register r1

$$r_1 \leftarrow r_2 + 4$$



- * $r1 \leftarrow [r2]$

- * **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

- * $V \leftarrow [r1]$

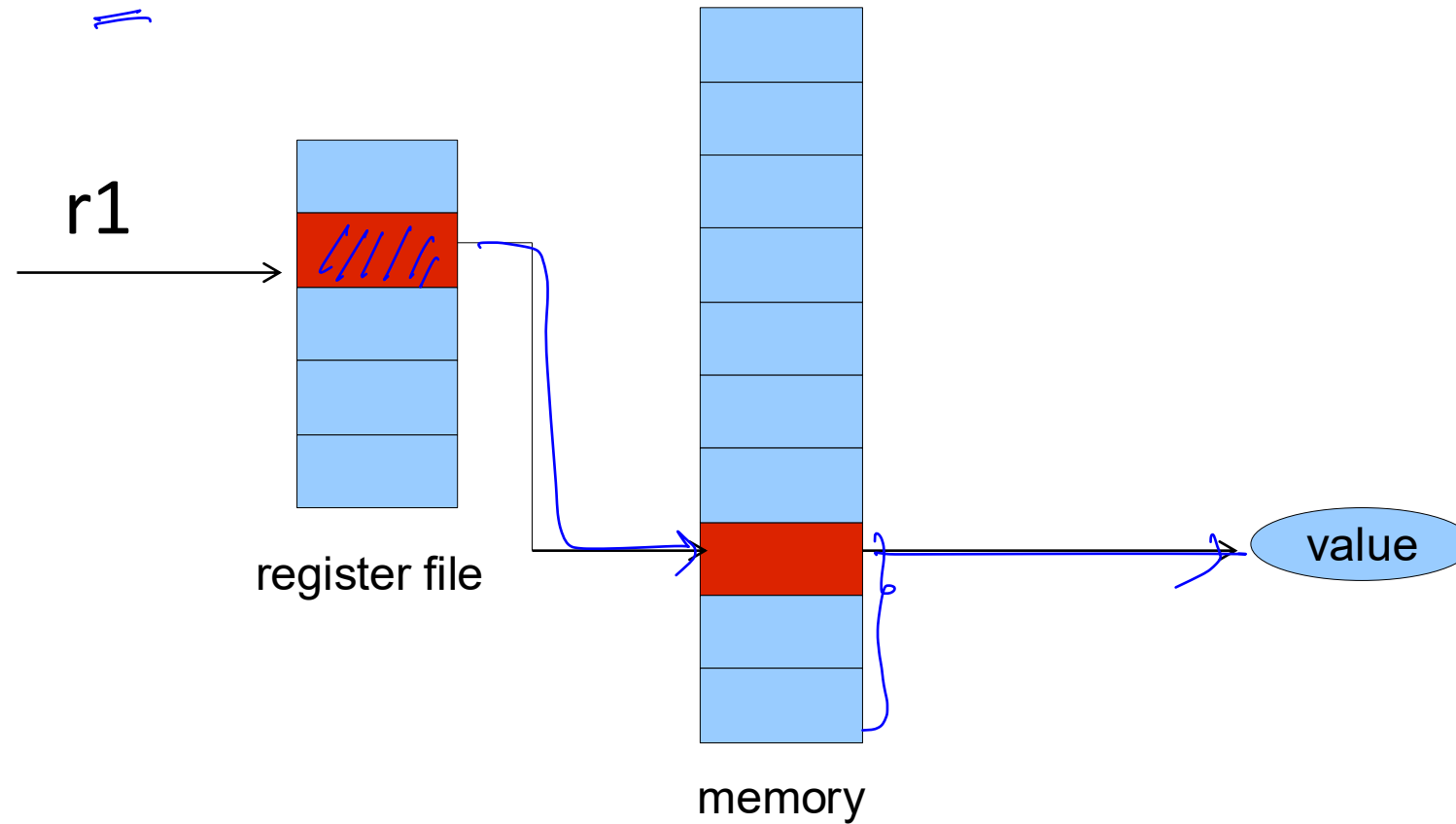
$$r_1 \leftarrow [r_2]$$

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

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

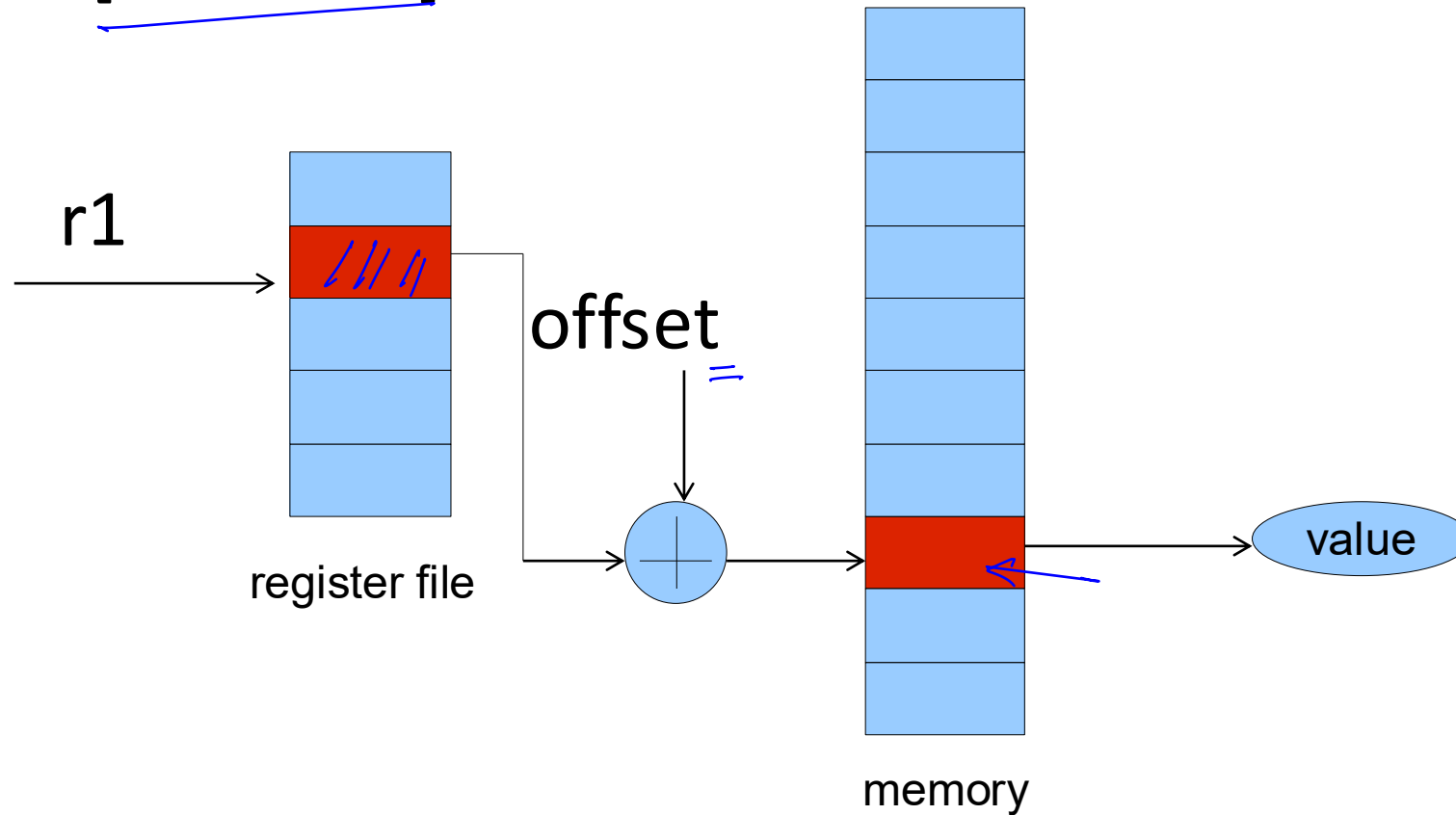
Register Indirect Mode

* $V \leftarrow [r1]$



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] ($V \leftarrow [r1 + r2 + 100]$)

$$r_3 \leftarrow [r_1 + r_2 + 10] + 10 \uparrow$$

* Memory Direct ✓

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

* example : [0x12ABCD03]

* PC Relative ✓

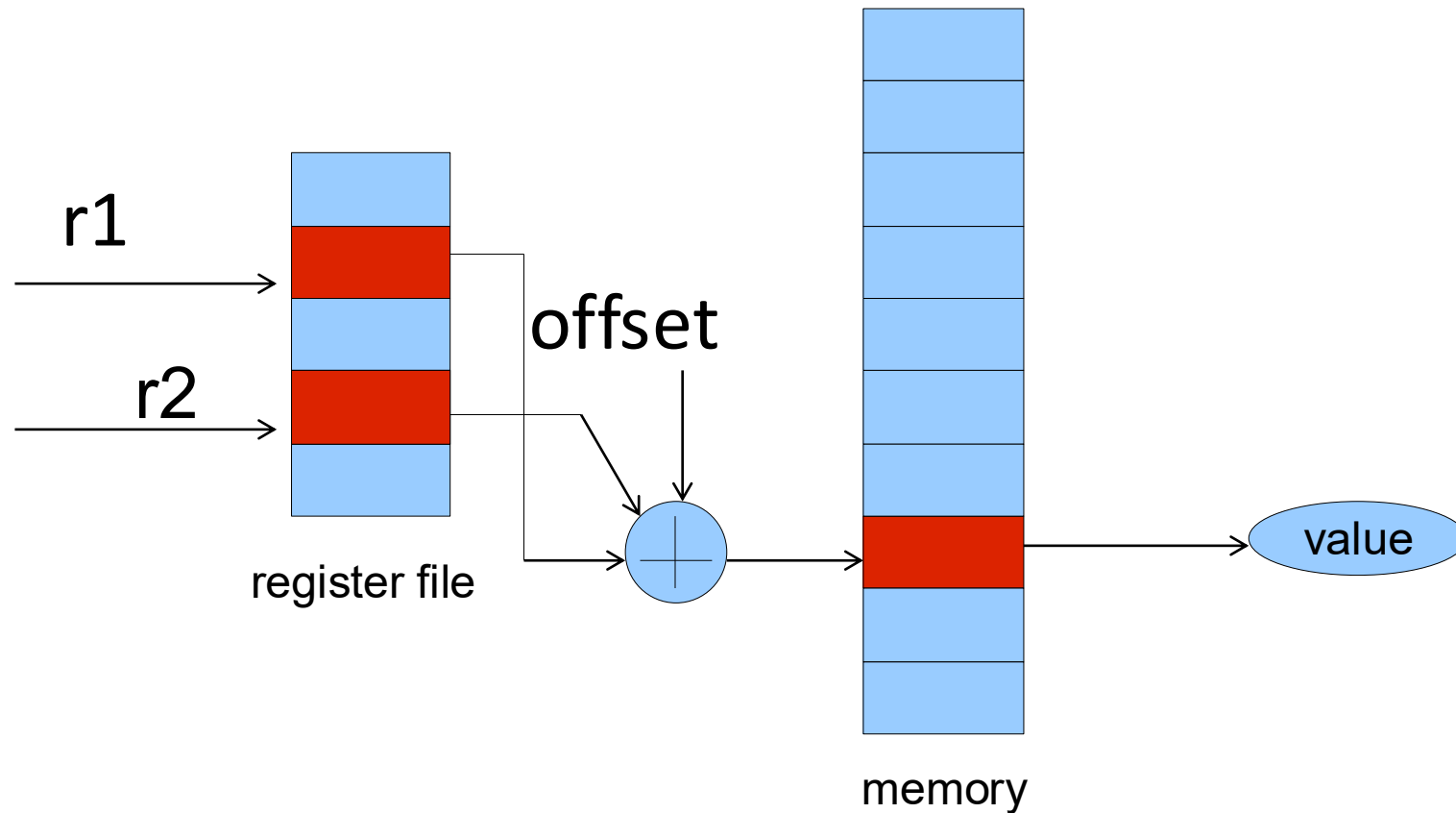
$$* V \leftarrow [\text{pc} + \text{offset}] *$$

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

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

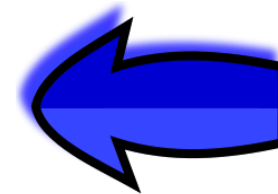
Base-Index-Offset Addressing Mode

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



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 $2^{15} - 1$

Arithmetic/Logical Instructions

- * SimpleRisc has 6 arithmetic instructions
 - * add, sub, mul, div, mod, cmp

| Example | Explanation |
|------------------|---|
| ✓ add r1, r2, r3 | $r1 \leftarrow r2 + r3$ ✓ |
| add r1, r2, 10 | $r1 \leftarrow r2 + 10$ |
| ✓ sub r1, r2, r3 | $r1 \leftarrow r2 - r3$ |
| ✓ mul r1, r2, r3 | $r1 \leftarrow r2 \times r3$ |
| div r1, r2, r3 | $r1 \leftarrow r2 / r3$ (quotient) |
| ✓ mod r1, r2, r3 | $r1 \leftarrow r2 \bmod r3$ (remainder) |
| ✓ cmp r1, r2 | set flags |

↑
 $r_1 - r_2$
"0"
flags. Z = 1
 $r_1 - r_2 > 0$
flags. C_{LT} = 1

Examples of Arithmetic Instructions

- * Convert the following code to assembly

```
a = 3  
b = 5  
c = a + b  
d = c - 5
```

Handwritten assembly:

| | | | |
|------------------|-----------|-----------|-----------|
| <i>r0</i> | <i>r1</i> | <i>r2</i> | <i>r3</i> |
| <i>a</i> | <i>b</i> | <i>c</i> | <i>d</i> |
| <i>mov r0, 3</i> | | | |
| <i>mov r1, 5</i> | | | |

- * Assign the variables to registers

- * $a \leftarrow r0, b \leftarrow r1, c \leftarrow r2, d \leftarrow r3$

```
mov r0, 3  
mov r1, 5  
add r2, r0, r1  
sub r3, r2, 5
```

Examples - II

- * Convert the following code to assembly

```
a = 3  
b = 5  
c = a * b  
d = c mod 5
```

$r0 \rightarrow c$
 $r1 \rightarrow b$
 $r2 \rightarrow c$
 $r3 \rightarrow d$

- * Assign the variables to registers

- * $a \leftarrow r0, b \leftarrow r1, c \leftarrow r2, d \leftarrow r3$

```
mov r0, 3  
mov r1, 5  
mul r2, r0, r1  
mod r3, r2, 5
```

Compare Instruction

- * Compare 3 and 5, and print the value of the flags

```
a = 3 ✓  
b = 5  
compare a and b
```

```
mov r0, 3 ✓  
mov r1, 5 ✓  
cmp r0, r1
```

- * flags.E = 0, flags.GT = 0

Compare Instruction

- * Compare 5 and 3, and print the value of the flags

```
a = 5  
b = 3  
compare a and b
```

```
mov r0, 5  
mov r1, 3  
cmp r0, r1
```

- * flags.E = 0, flags.GT = 1

Compare Instruction

- * Compare 5 and 5, and print the value of the flags

```
a = 5  
b = 5  
compare a and b
```

```
mov r0, 5  
mov r1, 5  
cmp r0, r1
```

- * flags.E = 1, flags.GT = 0

Example with Division

Write assembly code in SimpleRisc to compute: $31 / 29 - 50$, and save the result in r4.

Answer:

SimpleRisc

```
mov r1, 31
mov r2, 29
div r3, r1, r2
sub r4, r3, 50
```

mov r1, 31
mov r2, 29
1 ← div r3, r1, r2
sub r4, r3, 50

Logical Instructions

| | | |
|----------|---|----------------------------|
| $\&$ → | and r1, r2, r3 | $r1 \leftarrow r2 \& r3$ |
| \vee → | or r1, r2, r3 | $r1 \leftarrow r2 \vee r3$ |
| \sim → | not r1, r2 | $r1 \leftarrow \sim r2$ |
| | & bitwise AND, bitwise OR, ~ logical complement | |

$r2 = 0010$
 $r3 = 1101$

 $r1 \leftarrow 0000$

- * The second argument can either be a register or an immediate

Compute $(a \vee b)$. Assume that a is stored in $r0$, and b is stored in $r1$. Store the result in $r2$.

Answer:

or r2, r0, r1

SimpleRisc

Shift Instructions

* Logical shift left (lsl) (<< operator)

* $0010 \ll 2$ is equal to 1000

* ($\ll n$) is the same as multiplying by 2^n

* Arithmetic shift right (asr) (>> operator)

* $0010 \gg 1 = 0001$

* $1000 \gg 2 = 1110$

* same as dividing a signed number by 2^n

$0010 \ll 2 = 1000$
 $1000 = 8$

lsl <<

logical
right
left

Arithmetic
right
left

$-2 = 1110$

$1110 \ll 1$

$1100 = -4$

$x \ll n$

$$x' = \sum_{i=0}^k b_i 2^{i+n} = 2^n x$$

$$x = \sum_{i=0}^k b_i 2^i$$

1 1 1 1 1 1 1 ← b_i
 k bits.
 0 0 1 0 0 1 1 0

$x \ll 1$

$$x' = \sum_{i=0}^k b_i 2^{i+1} = 2 \sum_{i=0}^k b_i 2^i = 2x$$

if $n > k$, then shifting may discard high bits.
 the equality becomes —

$$(\lll \ll 1) \bmod 2^k$$

logical Shift
 \Rightarrow unsigned
 Arithmetic shift
 \Rightarrow Signed (2's Comp)

Arith right
 asr \gg

$$\begin{array}{ccccccc} 0 & 0 & 1 & 0 & \gg & 1 & = \\ \underbrace{}_2 & & & & & & \underbrace{}_1 \end{array}$$

$$\begin{array}{ccccccc} 1 & 0 & 0 & 0 & \gg & 2 & = \\ \underbrace{}_6 & & & & & & \underbrace{}_1 \end{array}$$

Shift Instructions - II

- * logical shift right (lsr) (>>> operator)

- * 1000 >>> 2 = 0010

- * same as dividing the unsigned representation by 2^n

| Example | Explanation |
|-----------------------------------|--|
| ✓ lsl r3, r1, r2 ↖ | $r3 \leftarrow r1 \ll r2$ (shift left) |
| ✓ lsl <u>r3</u> , <u>r1</u> , 4 ↖ | $r3 \leftarrow r1 \ll 4$ (shift left) |
| ✓ lsr r3, r1, r2 ↘ | $r3 \leftarrow r1 \ggg r2$ (shift right logical) |
| lsr r3, r1, 4 ✓ | $r3 \leftarrow r1 \ggg 4$ (shift right logical) |
| ✓ asr r3, r1, r2 ↘ | $r3 \leftarrow r1 \gg r2$ (arithmetic shift right) |
| ✓ asr r3, r1, 4 ↘ | $r3 \leftarrow r1 \gg r2$ (arithmetic shift right) |

Example with Shift Instructions

- * Compute $101 * 6$ with shift operators

| | | |
|----------------|---|----------------------|
| mov r0, 101 | ← | $r_0 = 101$ |
| lsl r1, r0, 1 | ← | $r_1 = r_0 \times 2$ |
| lsl r2, r0, 2 | ← | $r_2 = r_0 \times 4$ |
| add r3, r1, r2 | | $r_3 = r_1 + r_2$ |

Example - II

- * Compute $102 * 7.5$ with shift operators

```
mov r0, 102
```

```
lsl r1, r0, 3
```

```
lsr r2, r0, 1
```

```
sub r3, r1, r2
```

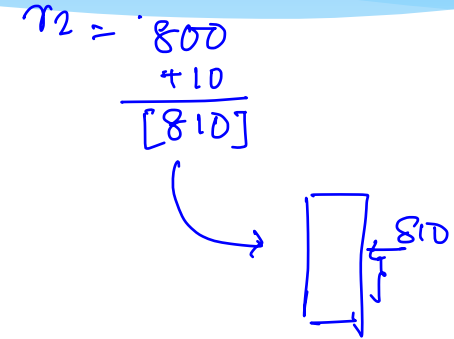
$$r_1 = r_0 \times 2^3$$

$$r_2 = \frac{r_0}{2}$$

$$\begin{aligned} r_3 &= r_1 - r_2 \\ &= r_0 2^3 - \frac{r_0}{2} \\ &= 7.5 \times r_0 \end{aligned}$$

Load-store instructions

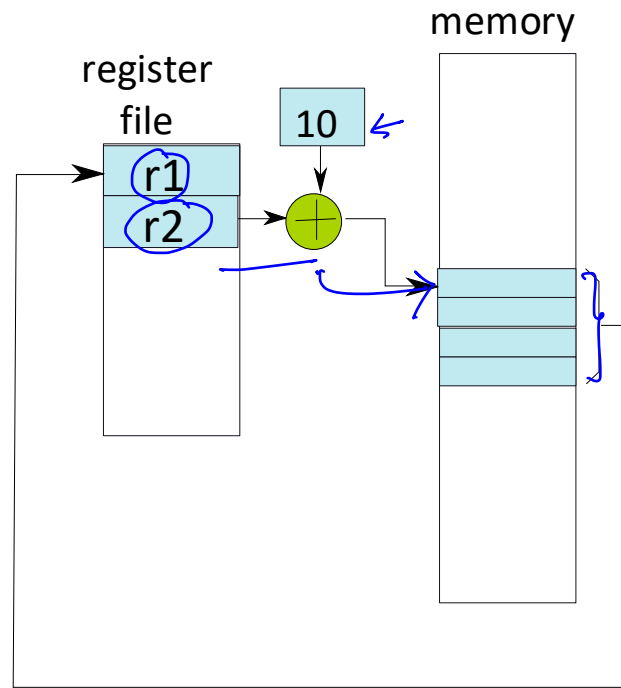
| | |
|---|---------------------------|
| <u>ld</u> <u>r1</u> , <u>10</u> [<u>r2</u>] | $r1 \leftarrow [r2 + 10]$ |
| <u>st</u> <u>r1</u> , <u>10</u> [<u>r2</u>] | $[r2 + 10] \leftarrow r1$ |



- * 2 address format, base-offset addressing
- * Fetch the contents of r2, add the offset (10), and then perform the memory access

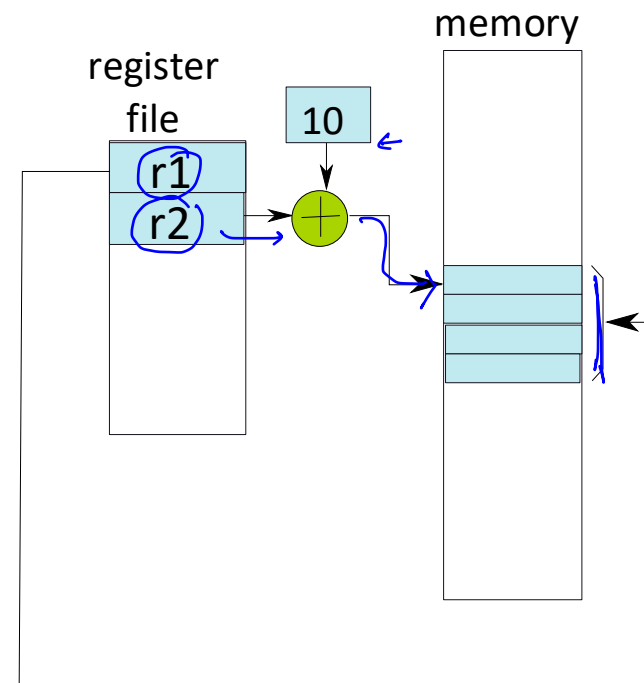
Load-Store

ld r1, 10[r2]



(a)

`st r1, 10[r2]`



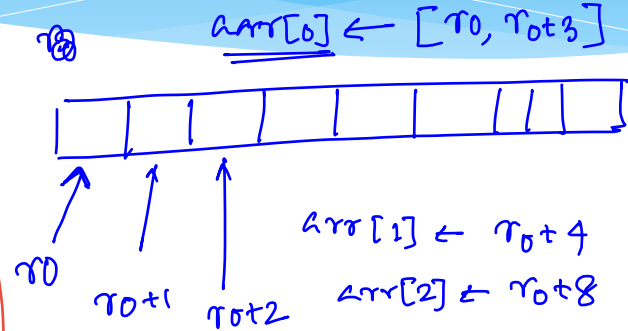
(b)

Example – Load/Store

* Translate :

base reg = r0

```
int arr[10];  
arr[3] = 5; —  
arr[4] = 8; —  
arr[5] = arr[4] + arr[3];
```



```
/* assume base of array saved in r0 */  
mov r1, (5) r1=5  
st r1, 12[r0] —  
mov r2, 8  
st r2, 16[r0]  
add r3, r1, r2 =  
st r3, 20[r0] ✓
```

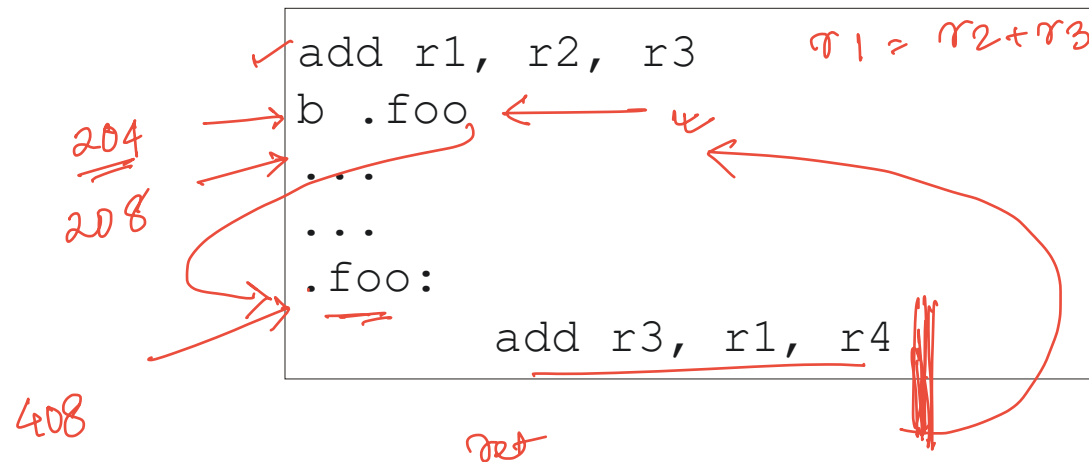
Branch Instructions

* Unconditional branch instruction

~~Feb 6~~

| | |
|---------------|----------------|
| <u>b</u> .foo | branch to .foo |
|---------------|----------------|

b < label



PC = 204

PC = 408

return address = 208
ra

PC = ra

Conditional Branch Instructions

| | |
|------------|---------------------------------------|
| ✓ beq .foo | branch to .foo if <u>flags.E</u> = 1 |
| bgt .foo | branch to .foo if <u>flags.GT</u> = 1 |

- * The flags are only set by cmp instructions
- * beq (branch if equal)
 - * If flags.E = 1, jump to .foo
- * bgt (branch if greater than)
 - * If flags.GT = 1, jump to .foo

Examples

- * If $r1 > r2$, then save 4 in $r3$, else save 5 in $r3$

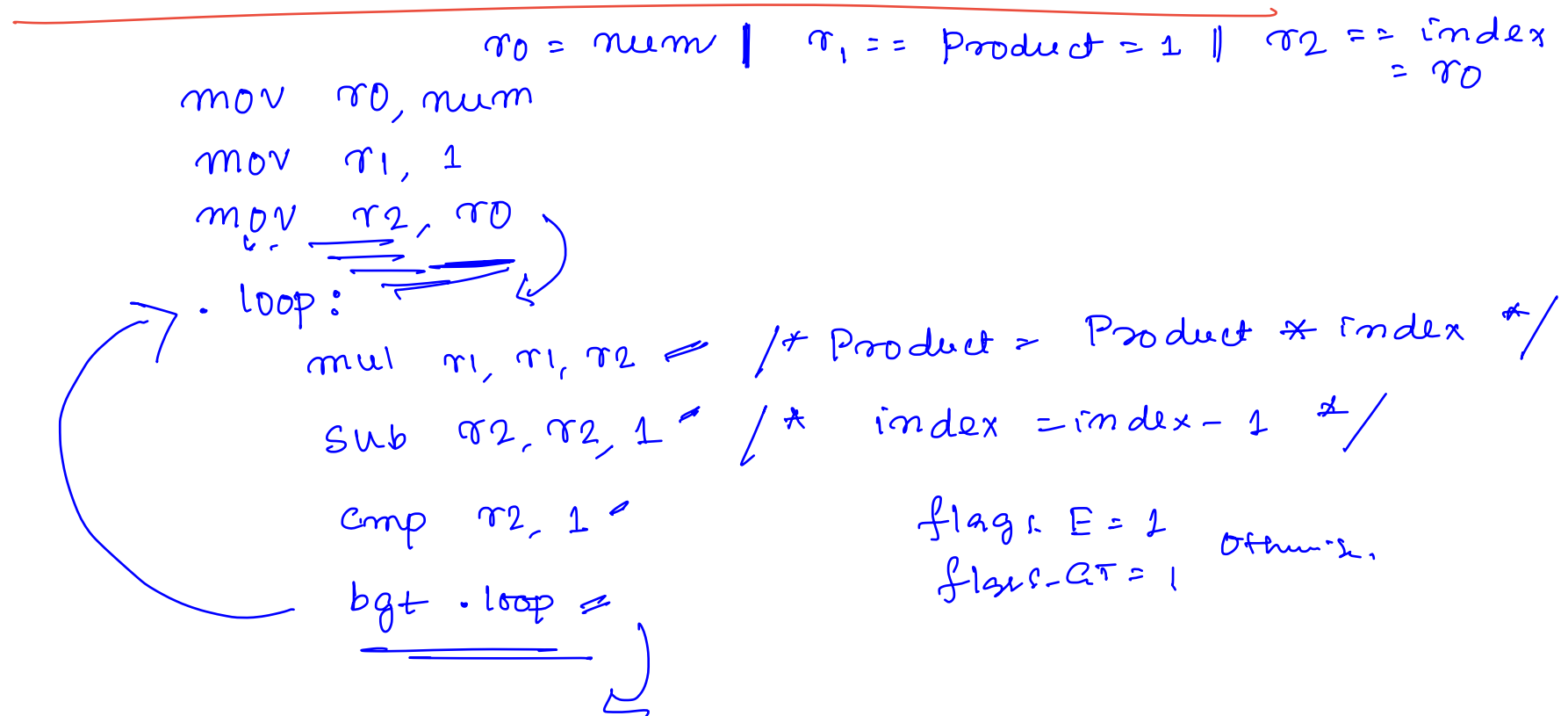
```
cmp r1, r2 ✓  
bgt .gtlabel ✓  
mov r3, 5  
...  
...  
✓.gtlabel:  
    mov r3, 4
```

```
int product = 1
```

```
int index
```

```
for( index = num; index > 1; index -- )
```

```
    product = product * index
```



Example - II

Answer: Compute the factorial of the variable num.

r0 = num

num! = 1 x 2 x 3 x ... x (num-1) x (num)

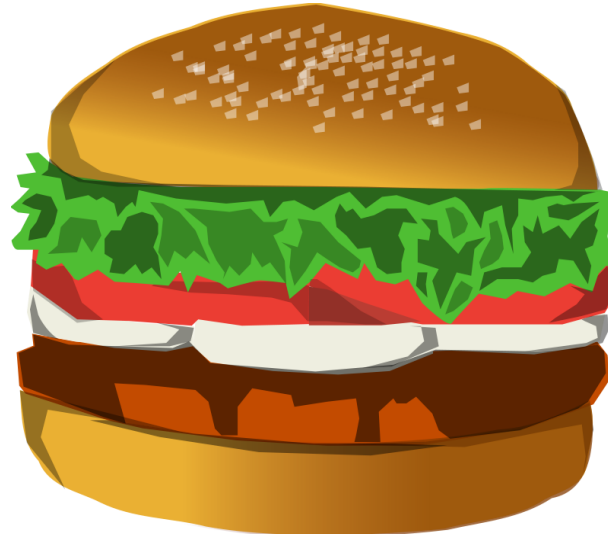
C

```
int prod = 1;
int idx;
for(idx = num; idx > 1; idx --) {
    prod = prod * idx
}
```

Let us now try to convert this program to SimpleRisc .

SimpleRisc

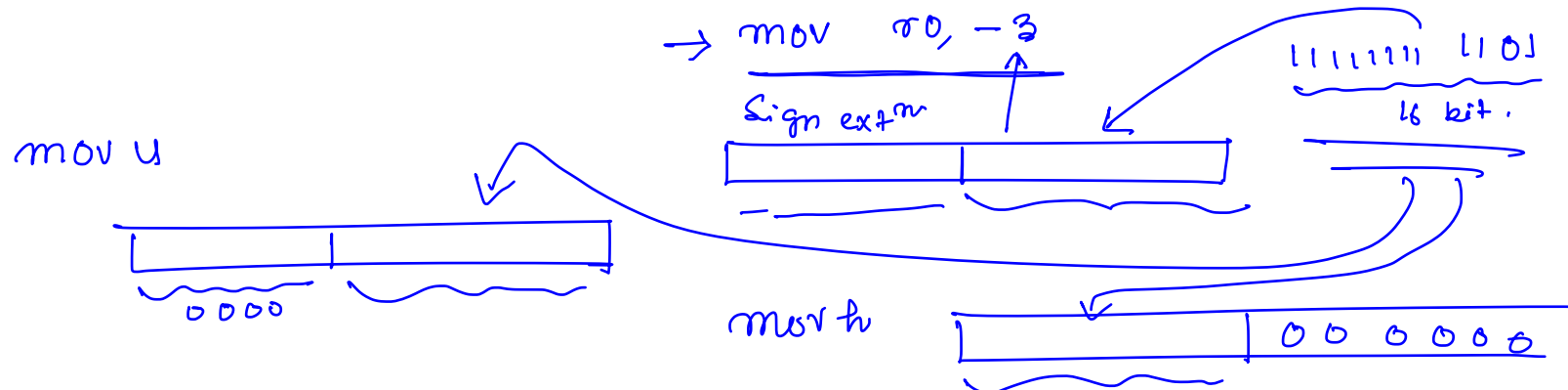
```
mov r1, 1          /* prod = 1 */
mov r2, r0         /* idx = num */
.loop:
    mul r1, r1, r2  /* prod = prod * idx */
    sub r2, r2, 1   /* idx = idx - 1 */
    cmp r2, 1       /* compare (idx, 1) */
    bgt .loop       /* if (idx > 1) goto .loop*/
```



- * Write a SimpleRisc assembly program to find the smallest number that is a **sum of two cubes in two different ways** \rightarrow 1729 $= 10^3 + 9^3$ $2 = \underline{1^3 + 1^3}$
 $= 12^3 + 1^3$

Modifiers

- * We can add the following modifiers to an instruction that has an immediate operand
- * Modifier :
 - * default : mov → treat the 16 bit immediate as a signed number (automatic sign extension)
 - * (u) : movu → treat the 16 bit immediate as an unsigned number
 - * (h) : movh → left shift the 16 bit immediate by 16 positions

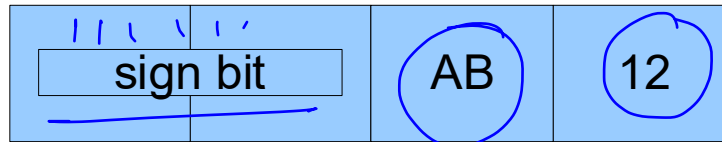


Mechanism

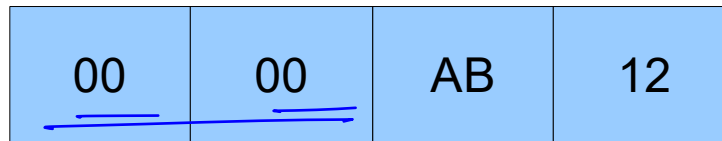
- * The processor **internally converts** a 16 bit immediate to a 32 bit number
- * It uses **this 32 bit number** for all the computations
- * Valid only for arithmetic/logical insts
- * We can control the generation of this 32 bit number
 - * sign extension (**default**)
 - * treat the 16 bit number as unsigned (**u suffix**)
 - * load the 16 bit number in the upper bytes (**h suffix**)

More about Modifiers

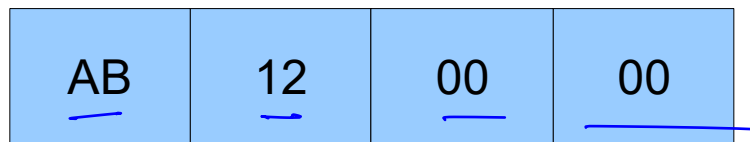
- * default : mov r1, 0xAB 12



- * unsigned : movu r1, 0xAB 12



- * high: movh r1, 0xAB 12



Examples

- * Move : 0x ^{11 11 11 11} FF FF A3 2B in r0

```
mov r0, 0xA32B
```

- * Move : 0x 00 00 A3 2B in r0

```
movu r0, 0xA32B
```

- * Move : 0x A3 2B 00 00 in r0

```
movh r0, 0xA32B
```

0x FFFF 3

Example

* Set $r0 \leftarrow 0x12\ AB\ A9\ 2D$

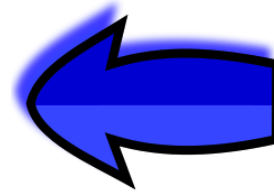
→ movh r0, 0x 12 AB
addu r0, 0x A9 2D

12 AB 00 00
+ 00 00 A9 2D

0000 A9 2D
0000

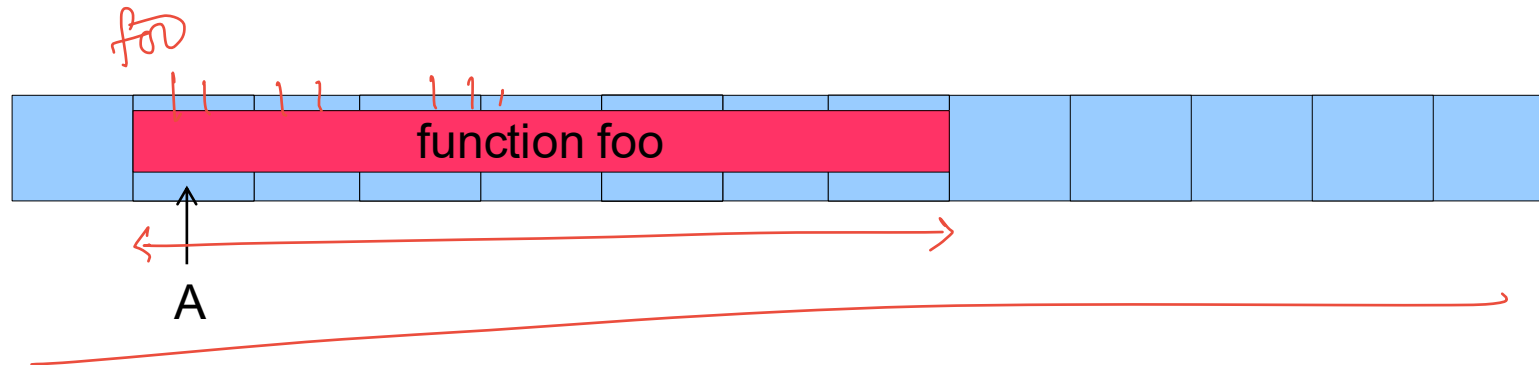
Outline

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



Implementing Functions

- * Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
- * Every function has a starting address in memory (e.g. foo has a starting address A)

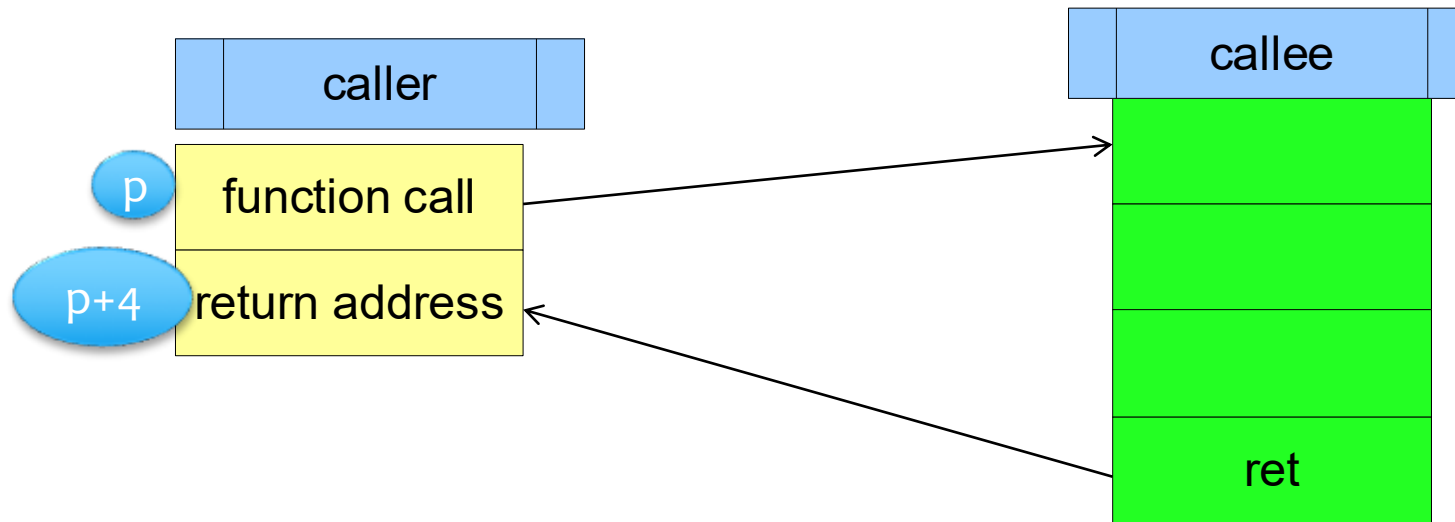


Implementing Functions - II

- * To call a function, we need to set :
 - * $pc \leftarrow A$
- * We also need to store the location of the pc that we need to come to after the function returns
- * This is known as the **return address**
- * We can thus call any function, execute its instructions, and then return to the saved **return address**

*b beq <pc>
pc ←
b .foo
pc ← .foo
call <label>*

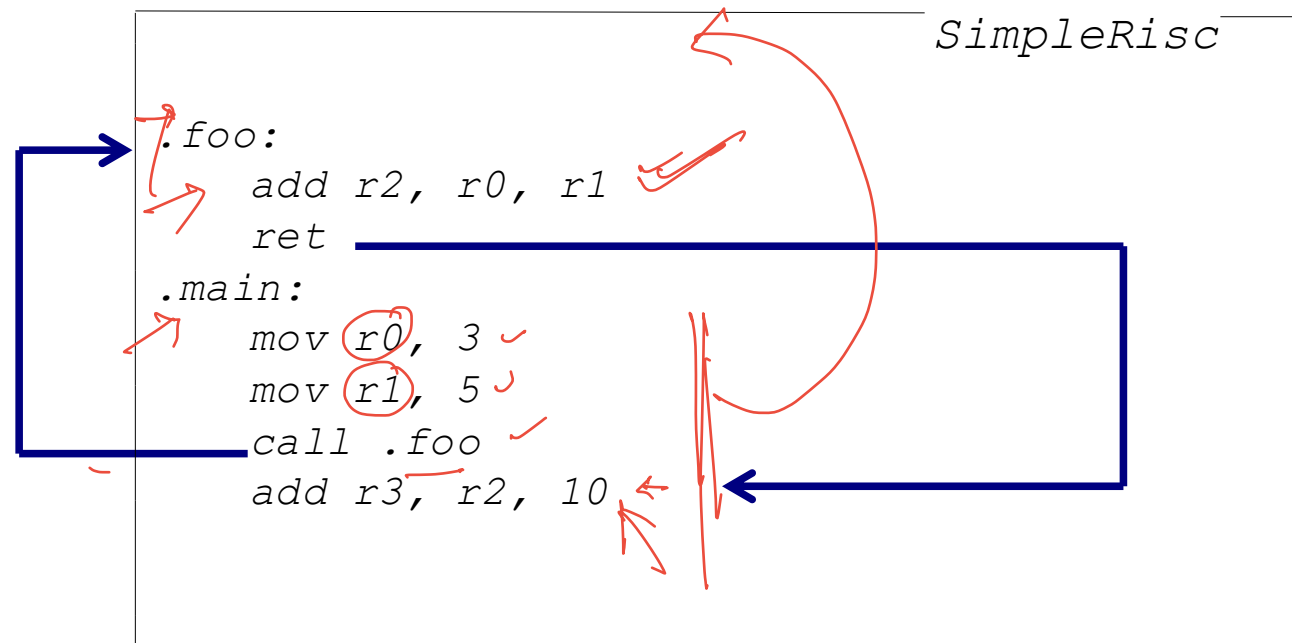
Notion of the Return Address



- * PC of the call instruction $\rightarrow p$
 - * PC of the return address $\rightarrow p + 4$
- because, every instruction takes 4 bytes

How do we pass arguments/ return values

- * Solution : use registers



Problems with this Mechanism

* Space Problem

- * We have a limited number of registers
- * We cannot pass more than 16 arguments
- * **Solution** : Use memory also



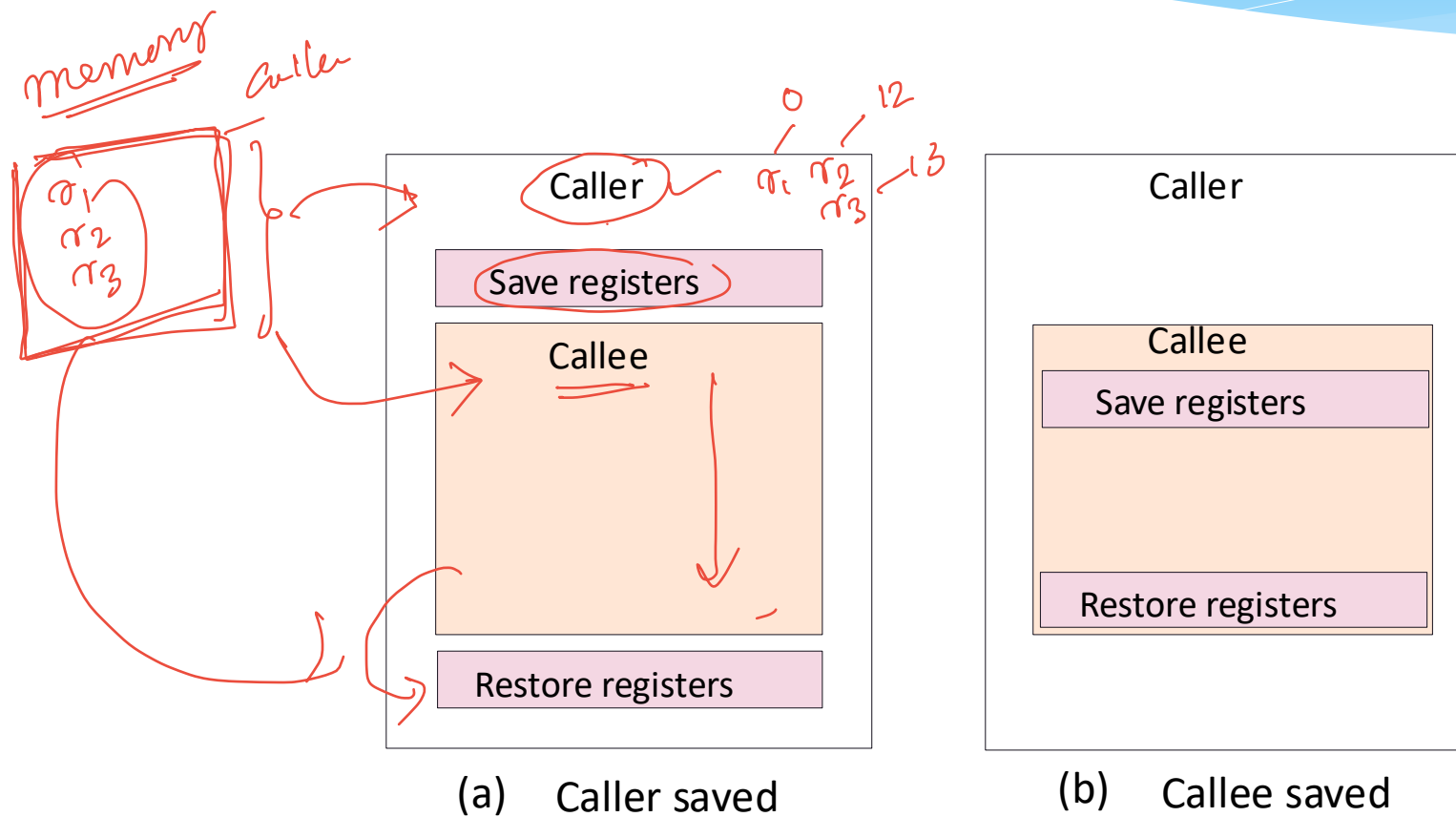
* Overwrite Problem

- * What if a function calls itself ? (recursive call)
- * The callee can **overwrite** the registers of the caller
- * **Solution** : Spilling

Register Spilling

- * The notion of **spilling**
 - * The caller can **save** the set of registers it needs
 - * **Call** the function
 - * And then **restore** the set of registers after the function returns
 - * Known as the **caller saved scheme**
- * **callee saved scheme**
 - * The callee **saves** the registers, and later **restores** them

Spilling

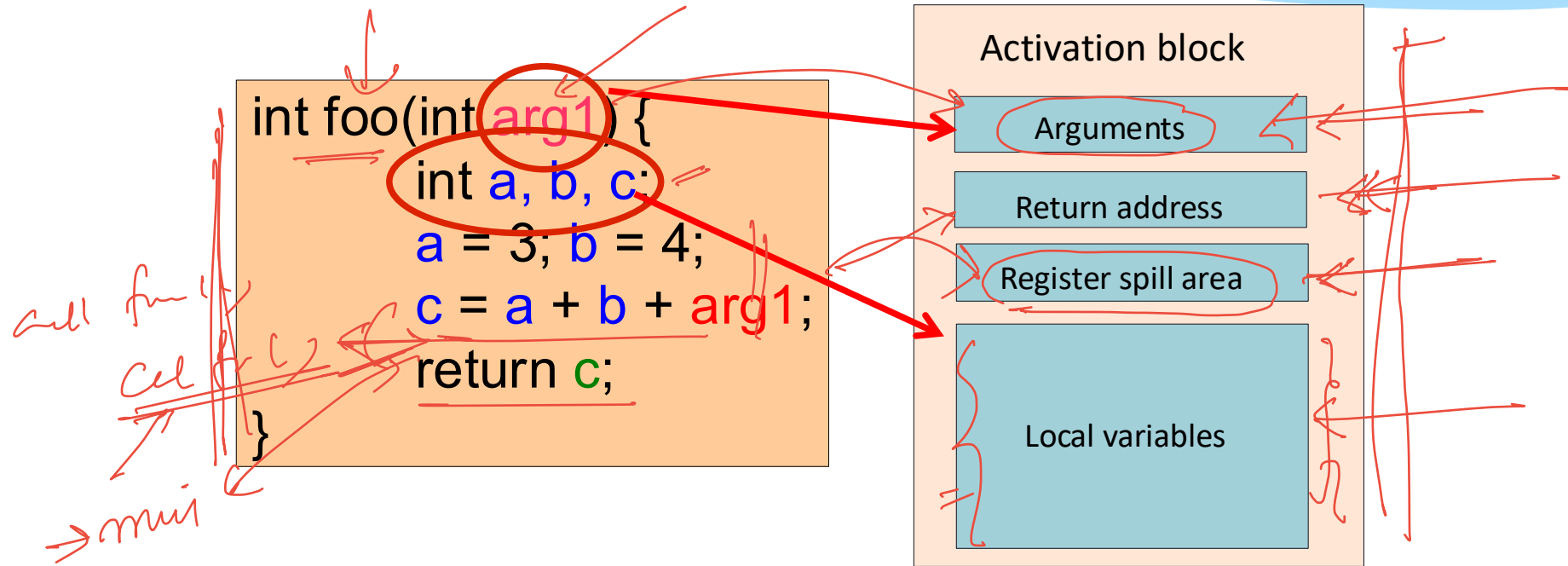


Problems with our Approach



- * Using memory, and spilling solves both the **space problem** and **overwrite problem**
- * However, there needs to be :
 - * a strict agreement between the caller and the callee regarding the set of **memory locations that need to be used**
 - * Secondly, after a function has finished execution, all **the space that it uses needs to be reclaimed**

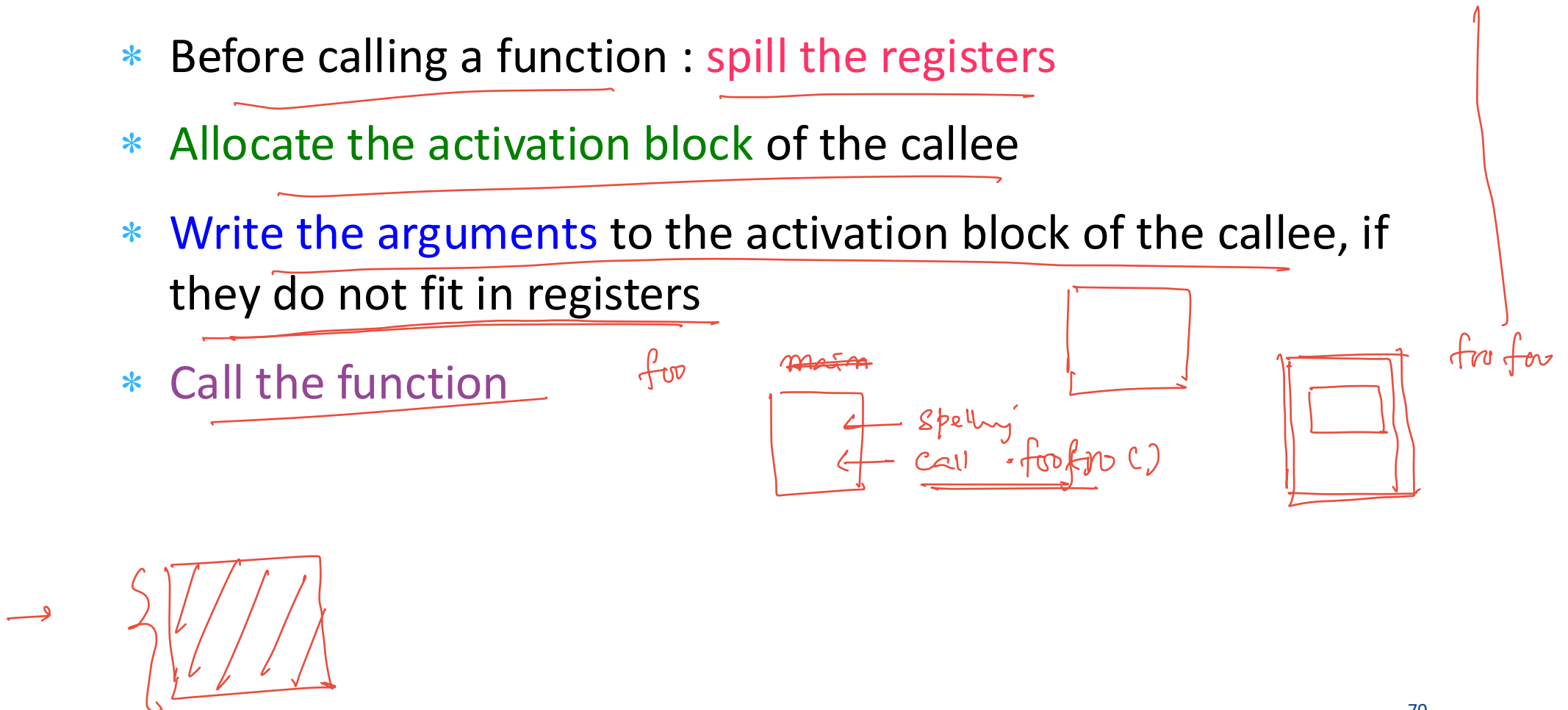
Activation Block



- * **Activation block** → memory map of a function
arguments, register spill area, local vars

How to use activation blocks ?

- * Assume caller saved spilling
- * Before calling a function : spill the registers
- * Allocate the activation block of the callee
- * Write the arguments to the activation block of the callee, if they do not fit in registers
- * Call the function



Using Activation Blocks - II

- * In the called function
 - * Read the arguments and transfer to registers (if required)
 - * Save the return address if the called function can call other functions
 - * Allocate space for local variables
 - * Execute the function
- * Once the function ends
 - * Restore the value of the return address register (if required)
 - * Write the return values to registers, or the activation block of the caller
 - * Destroy the activation block of the callee

Using Activation Blocks - III

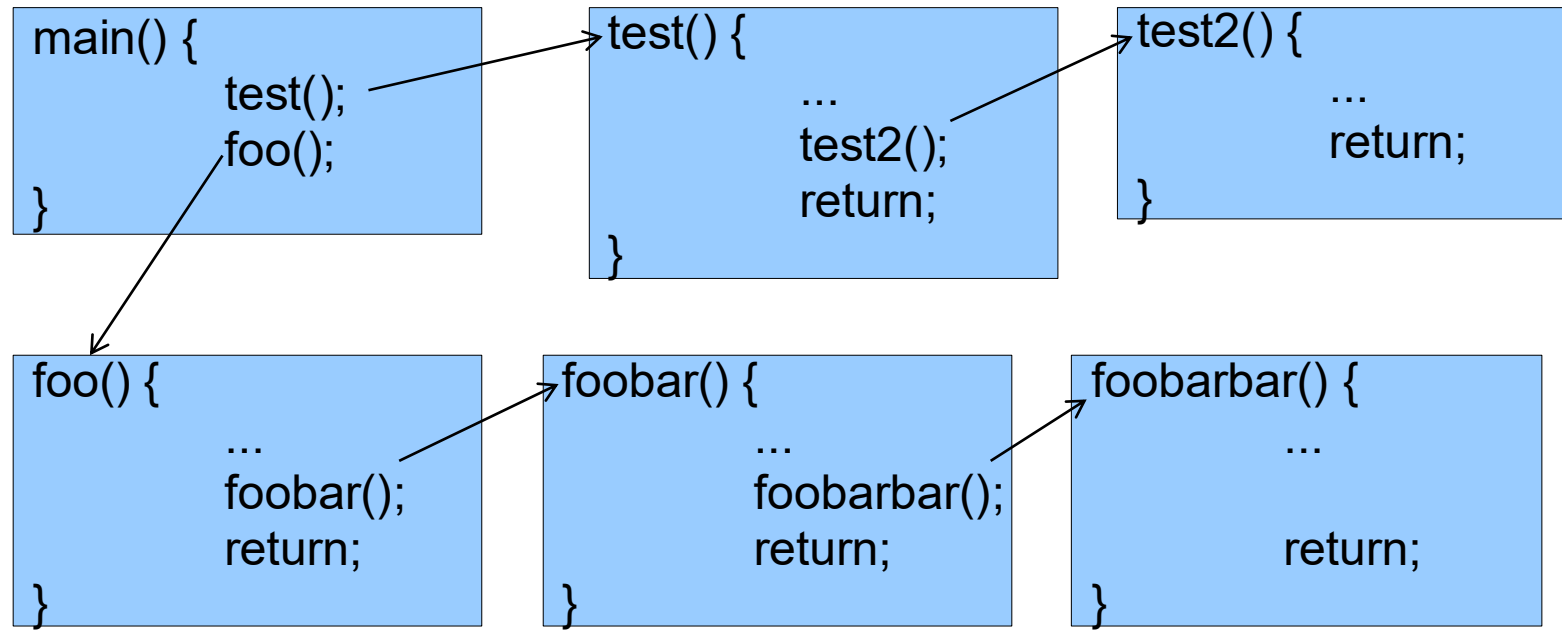
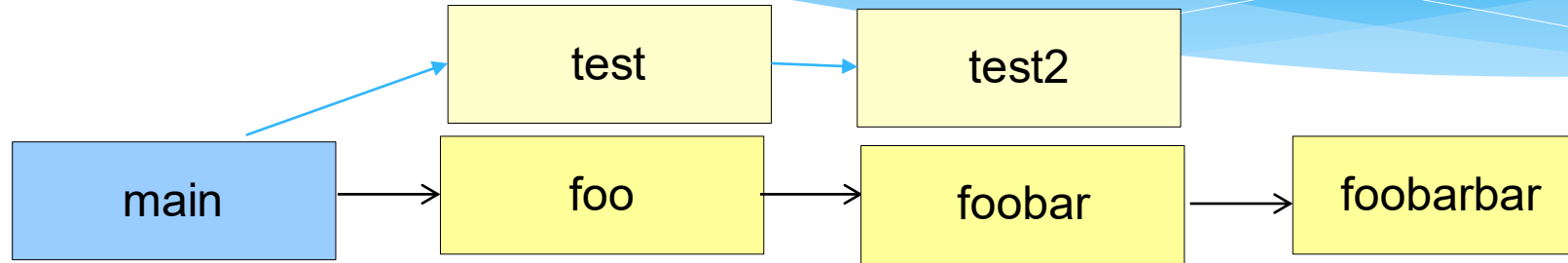
- * Once the function ends (contd ...)
 - * Call the **ret** instruction
 - * and return to the caller
- * **The caller :**
 - * **Retrieve the return values** from the registers of from its activation block
 - * **Restore** the spilled registers
 - * **continue ...**



Organising Activation Blocks

- * All the information of an executing function is stored in its **activation block**
- * These blocks need to be dynamically **created and destroyed** – millions of times
- * What is the correct way of managing them, and ensuring **their fast creation and deletion** ?
- * Is there a pattern ?

Pattern of Function Calls

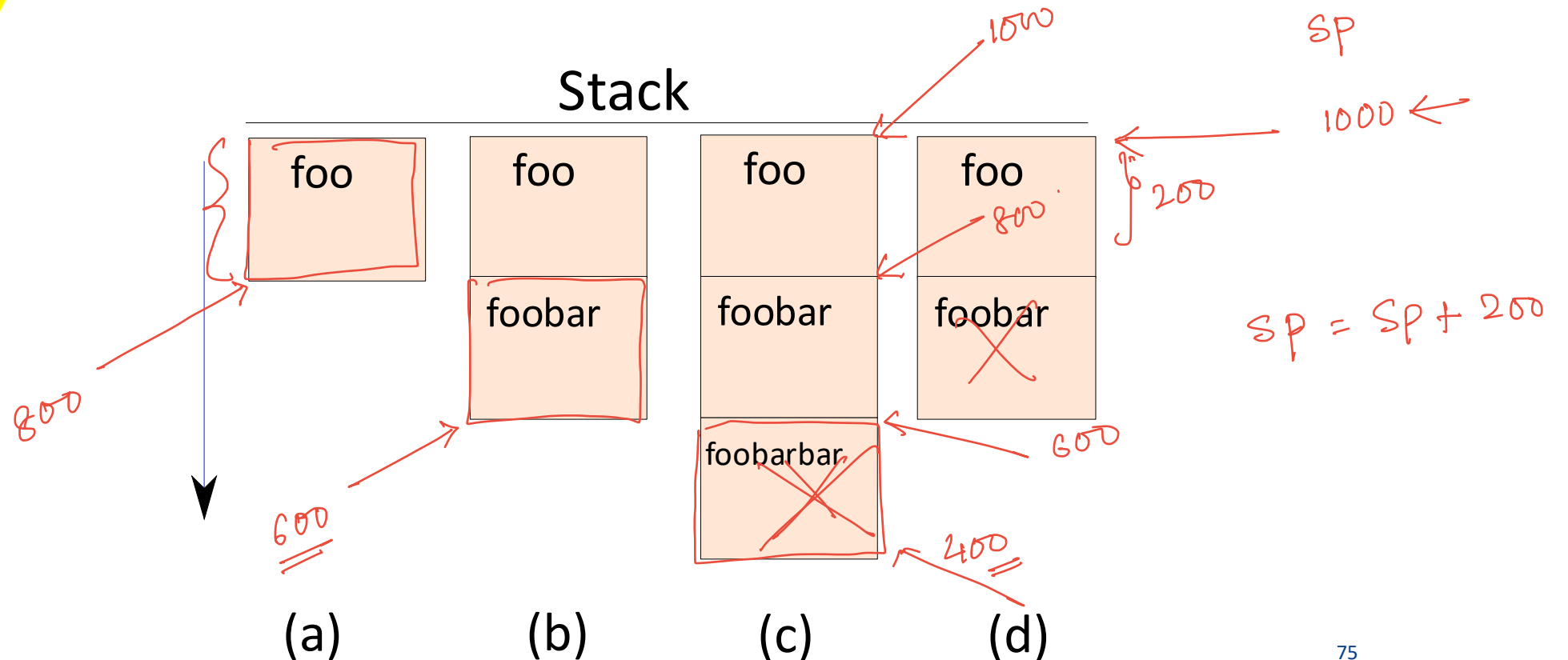


Pattern of Function Calls

* Last in First Out



Use a **stack** to store activation blocks



Working with the Stack

- * Allocate a part of the memory to **save the stack**
- * Traditionally **stacks** are downward growing.
 - * The first activation block starts at the **highest address**
 - * Subsequent activation blocks are **allocated lower addresses**
- * The **stack pointer register (sp (14))** points to the beginning of an activation block
- * Allocating an activation block :
 - * $sp \leftarrow sp - \text{<constant>}$
- * De-allocating an activation block :
 - * $sp \leftarrow sp + \text{<constant>}$

What has the Stack Solved ?

- * Space problem
 - * Pass as many parameters as required in the activation block
- * Overwrite problem
 - * Solved by activation blocks
- * Management of activation blocks
 - * Solved by the notion of the stack
- * The stack needs to primarily be managed in software

call and ret instructions

| | |
|-----------|---|
| call .foo | $ra \leftarrow PC + 4 ; PC \leftarrow address(.foo);$ |
| ret | $PC \leftarrow ra$ |

- * **ra** (or r15) \leftarrow return address register

- * **call** instruction

- * Puts $pc + 4$ in **ra**, and jumps to the function

- * **ret** instruction

- * Puts **ra** in **pc**