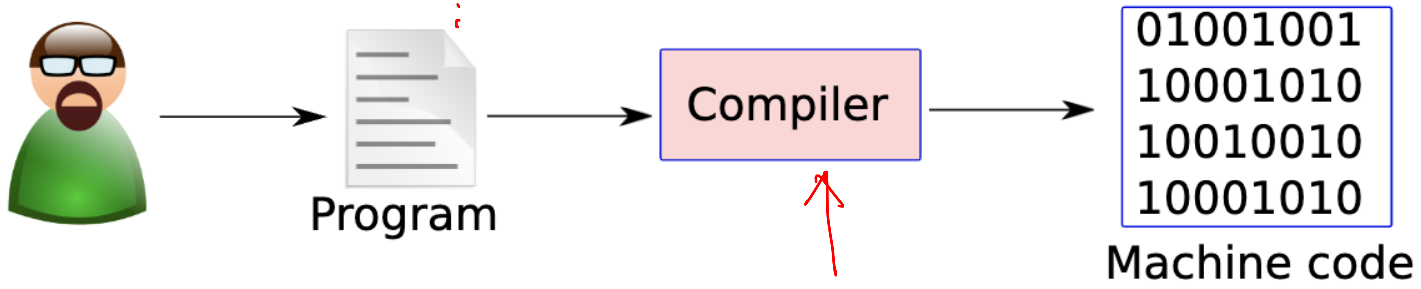
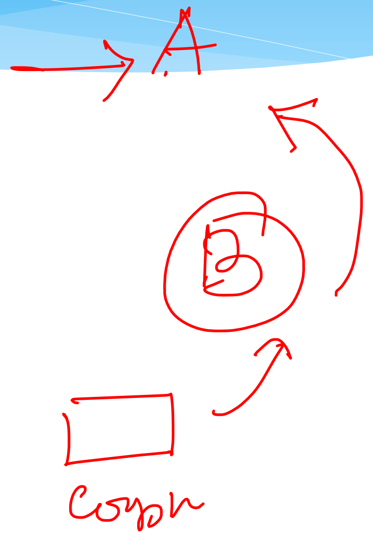
Basic Computer Architecture

Chapter 3: Assembly Language

Outline

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





0 1 - - -

$$\text{Assembly} \rightarrow \boxed{\text{Assembler}} \rightarrow \underline{\text{MC}}$$

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) and a list of operands.

Why learn Assembly Language ?

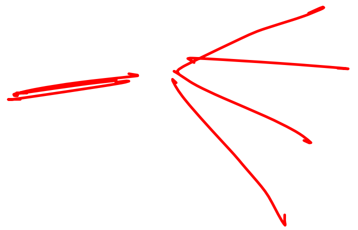
https://www.tiobe.com/tiobe-index/

- 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)

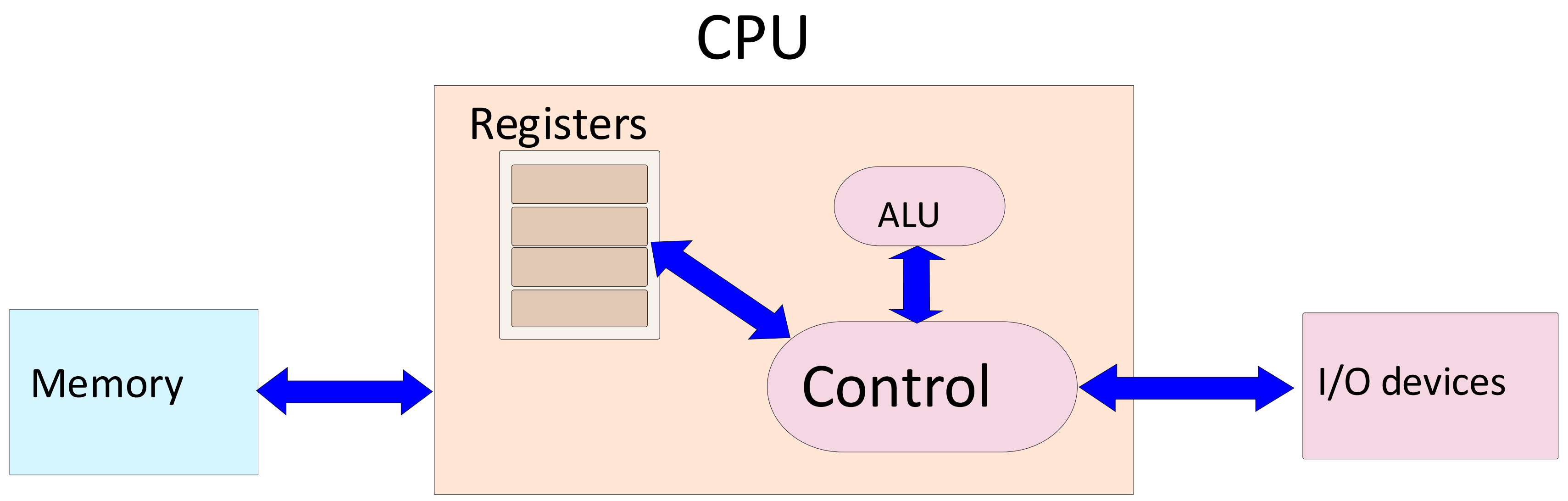
C, C++, Java



Hardware Designers Perspective

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

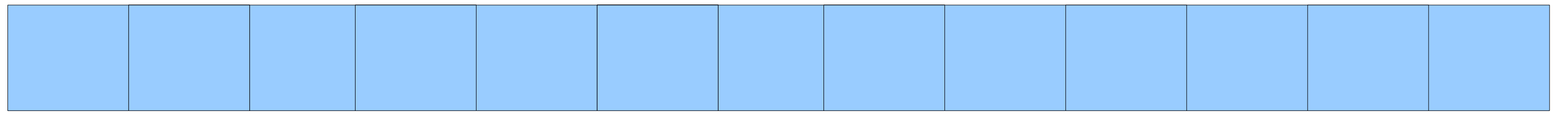
Machine Model – Von Neumann Machine with Registers



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

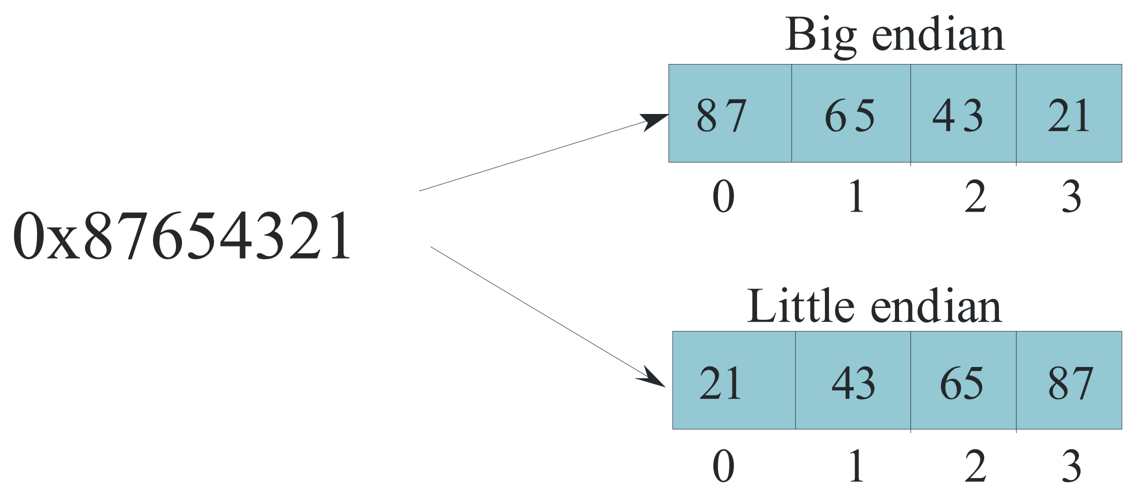


- Memory
  - One large array of bytes
  - Each location has an address
  - The address of the first location is 0, and increases by 1 for each subsequent location
- The program is stored in a part of the memory
- The program counter contains the address of the current instruction

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



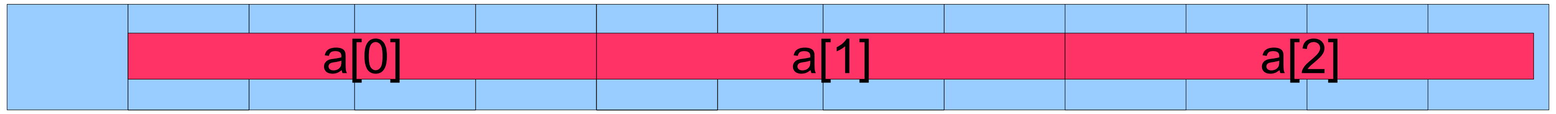
- Note the order of the storage of bytes

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][100]
  - float b[100][100]
  - Two methods : row major and column major

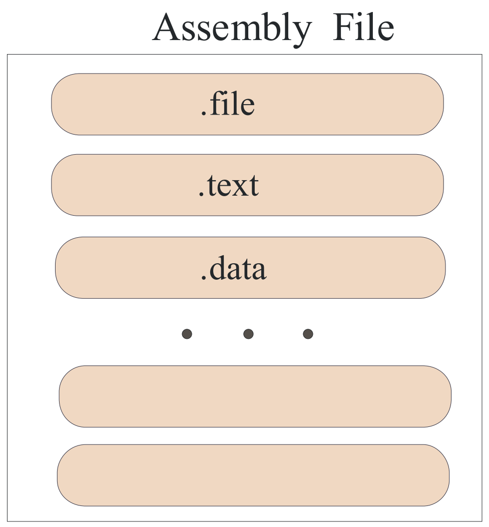
Row Major vs Column Major

- 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

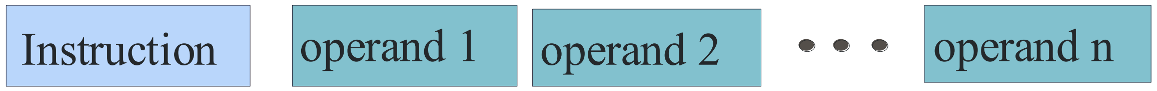


- Divided into different sections
- Each section contains some data, or assembly instructions

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



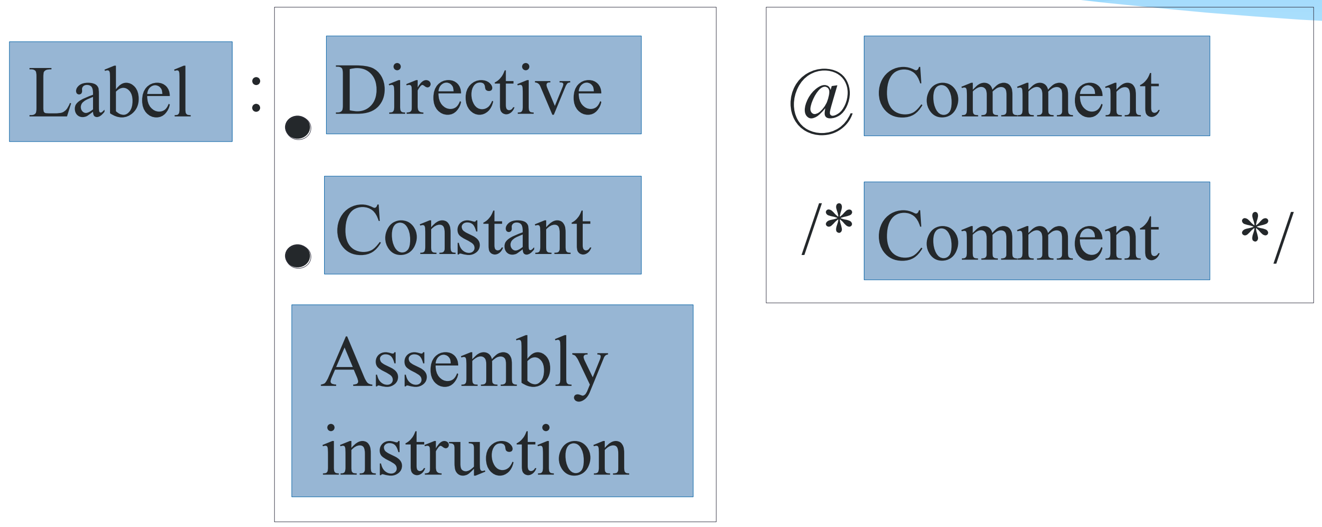
- instruction
  - textual identifier of a machine instruction
- operand
  - constant (also known as an immediate)
  - register
  - memory location

Examples of Instructions

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

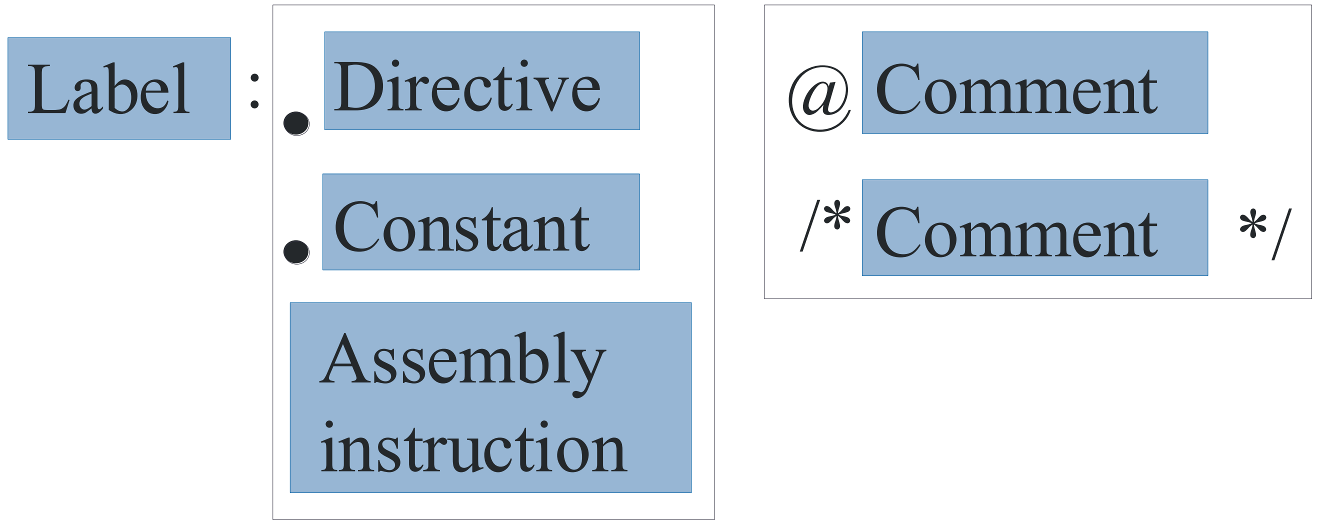
- subtract the contents of *r2* from the contents of *r1*, and save the result in *r3*
- multiply the contents of *r2* with the contents of *r1*, and save the results in *r3*

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

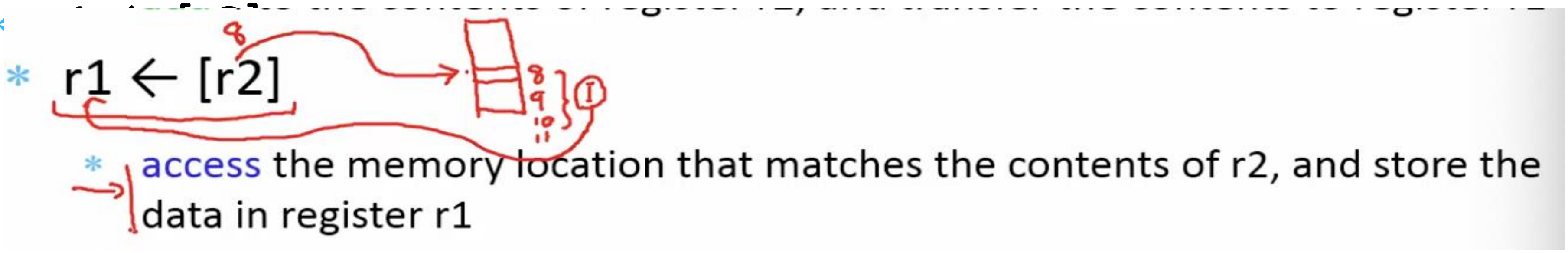
- Data Processing Instructions
  - add, subtract, multiply, divide, compare, logical or, logical and
- Data Transfer Instructions
  - transfer values between registers, and memory locations
- Branch instructions
  - branch to a given label
- Special instructions
  - interact with peripheral devices, 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)
- 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 ← r2
  - transfer the contents of register r2 to register r1
- r1 ← r2 + 4
  - add 4 to the contents of register r2, and transfer the contents to register r1
-



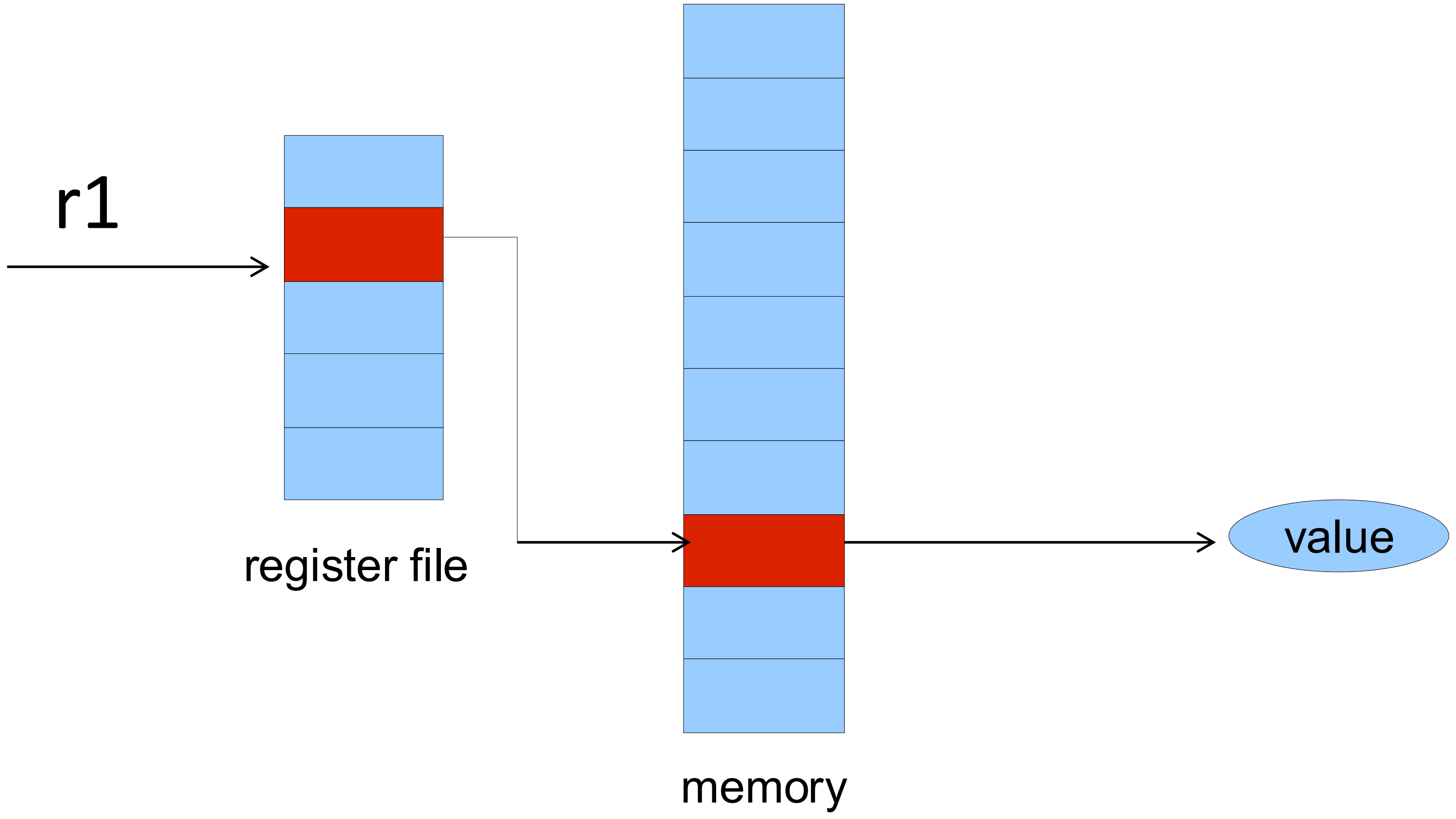
- r1 ← [r2]
  - access the memory location that matches the contents of r2, and store the data in register r1

Addressing Modes

- Let *V* be the value of an operand, and let r1, r2 specify registers
- Immediate addressing mode
  - V $\leftarrow$ imm , e.g. 4, 8, 0x13, -3
- Register direct addressing mode
  - V $\leftarrow$ r1
  - e.g. r1, r2, r3 …
- Register indirect
  - V $\leftarrow$ [r1]
- Base-offset : V $\leftarrow$ [r1 + offset], e.g. 20[r1] (V $\leftarrow$ [20+r1])

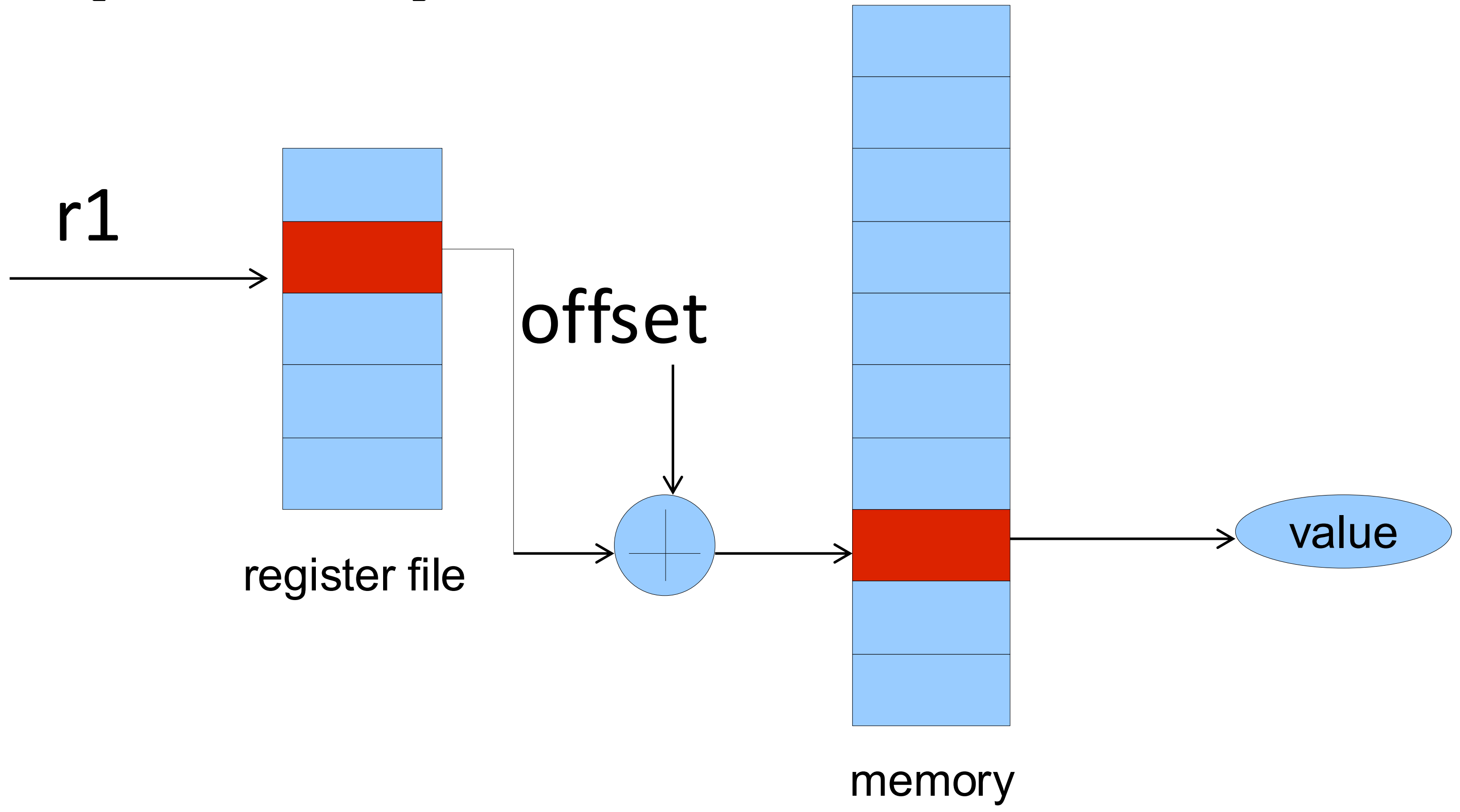
Register Indirect Mode

- V ← [r1]



Base-offset Addressing Mode

- V ← [r1+offset]

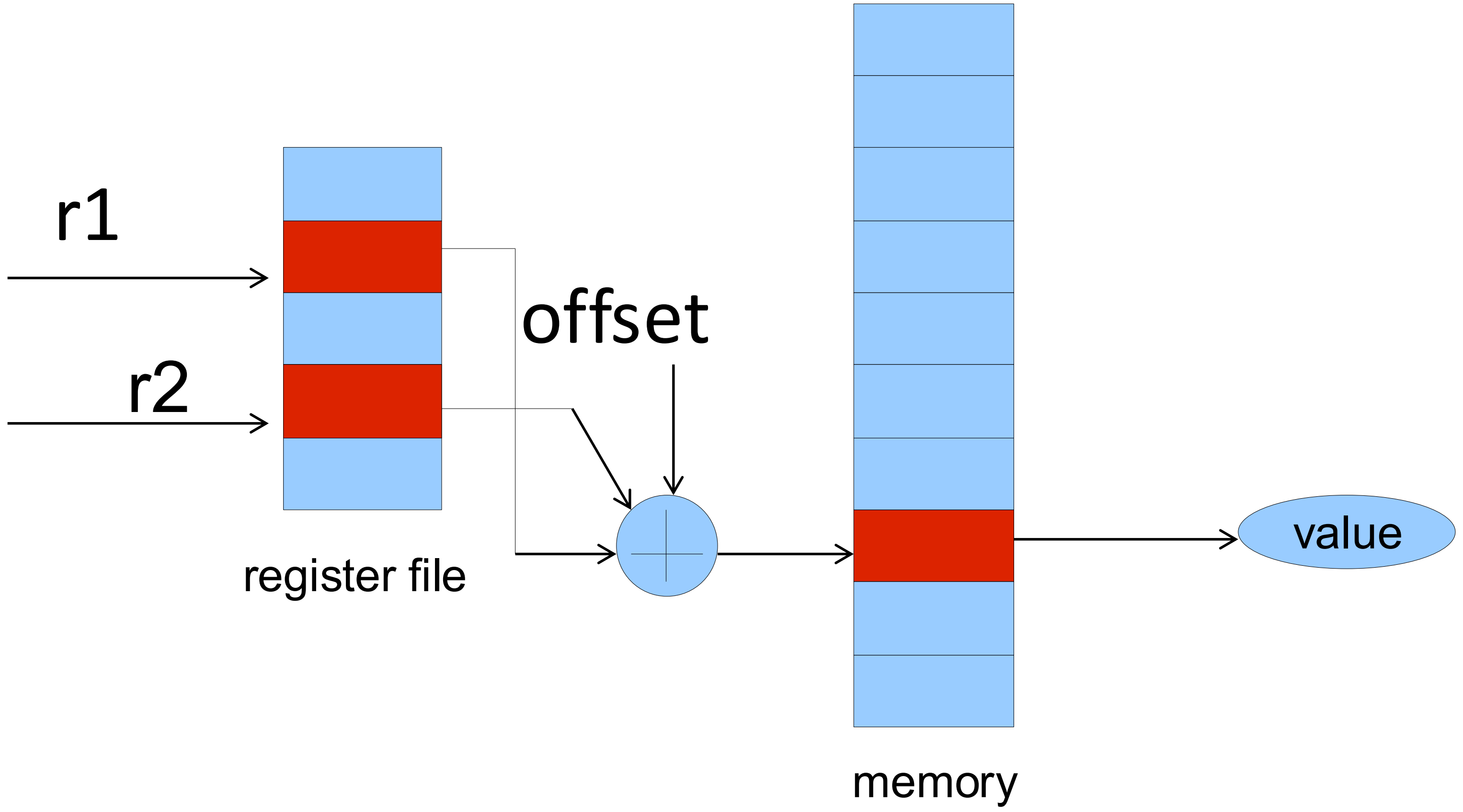


Addressing Modes - II

- Base-index-offset
  - V ← [r1 + r2 + offset]
  - example: 100[r1,r2] (V ← [r1 + r2 + 100])
- Memory Direct
  - V ← [addr]
  - example : [0x12ABCD03]
- PC Relative
  - V ← [pc + offset]
  - example: 100[pc] (V ← [pc + 100])

Base-Index-Offset Addressing Mode

- V ← [r1+r2 +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)

mov instruction

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

- 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
  - 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	$r1 \leftarrow \mathrm{r}2 - r3$
mul r1, r2, r3	$r1 \leftarrow r2 \times r3$
div r1, r2, r3	$r1 \leftarrow r2/r3$ (quotient)
mod r1, r2, r3	$r1 \leftarrow r2\ mod\ r3$ (remainder)
cmp r1, r2	set flags

Examples of Arithmetic Instructions

- Convert the following code to assembly

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

- Assign the variables to registers
  - a ← r0, b ← r1, c ← r2, d ← 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
```

- Assign the variables to registers
  - a ← r0, b ← r1, c ← r2, d ← 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
```

Logical Instructions

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

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

Compute (a | b). Assume that a is stored in r0, and b is stored in r1. Store the result in r2.
Answer:

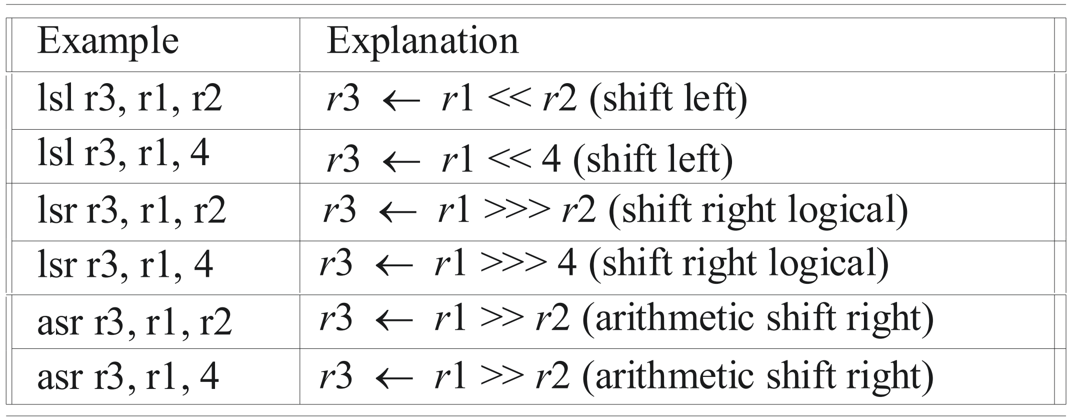
SimpleRisc
```
or r2, r0, r1
```

Shift Instructions

- Logical shift left (lsl) (<< operator)
  - 0010 << 2 is equal to 1000
  - (<< n) is the same as multiplying by 2^n
- Arithmetic shift right (asr) (>> operator)
  - 0010 >> 1 = 0001
  - 1000 >> 2 = 1110
  - same as dividing a signed number by 2^n

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	$r3 \leftarrow r1 << r2$ (shift left)
lsl r3, r1, 4	$r3 \leftarrow r1 << 4$ (shift left)
lsr r3, r1, r2	$r3 \leftarrow r1 >>> r2$ (shift right logical)
lsr r3, r1, 4	$r3 \leftarrow r1 >>> 4$ (shift right logical)
asr r3, r1, r2	$r3 \leftarrow r1 >> r2$ (arithmetic shift right)
asr r3, r1, 4	$r3 \leftarrow r1 >> r2$ (arithmetic shift right)

Example with Shift Instructions

- Compute 101 * 6 with shift operators

```
mov r0, 101
lsl r1, r0, 1
lsl r2, r0, 2
add r3, r1, r2
```

Example - II

- Compute 102 * 7.5 with shift operators

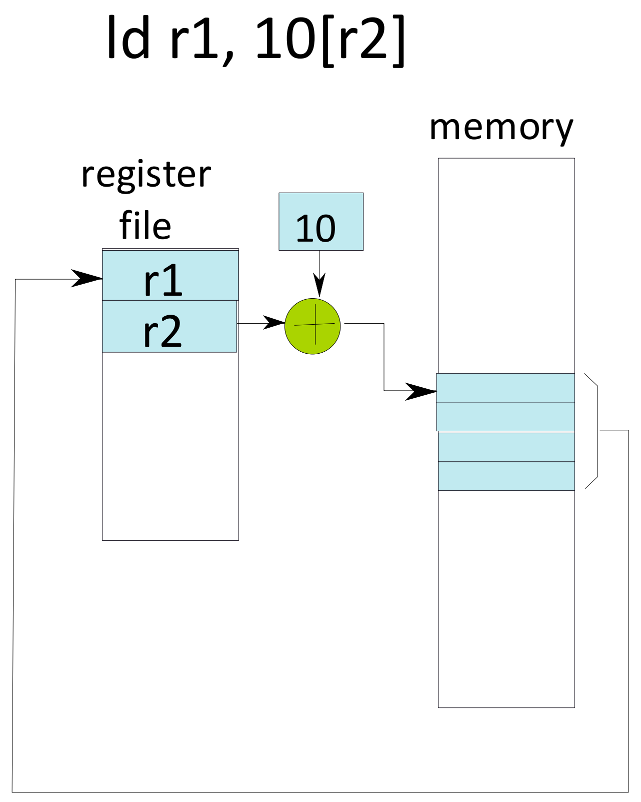
```
mov r0, 102
lsl r1, r0, 3
lsr r2, r0, 1
sub r3, r1, r2
```

Load-store instructions

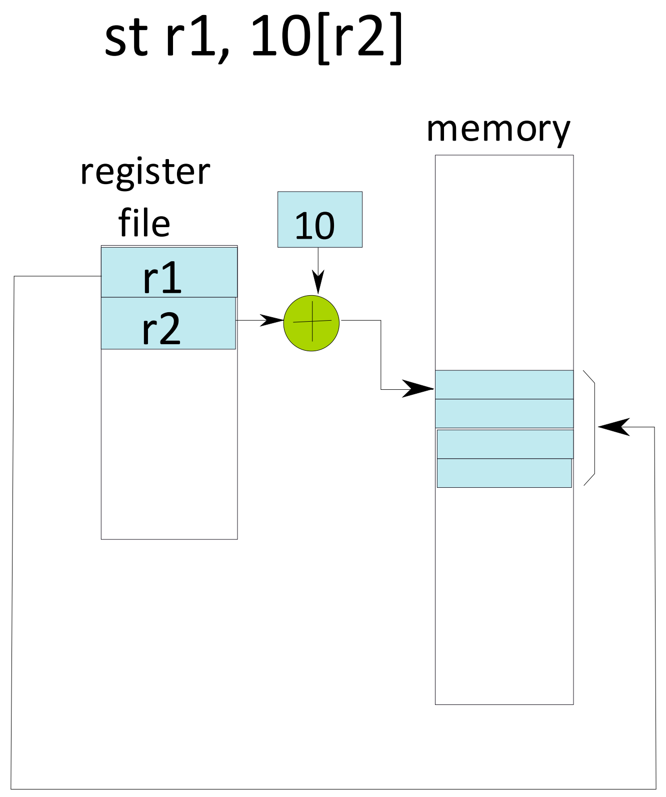
ld r1, 10[r2]	$r1 \leftarrow [r2 + 10]$
st r1, 10[r2]	$[r2+10] \leftarrow r1$

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

Load-Store



(a)



(b)

Example – Load/Store

* Translate :

```
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
st r1, 12[r0]
mov r2, 8
st r2, 16[r0]
add r3, r1, r2
st r3, 20[r0]
```

Branch Instructions

* Unconditional branch instruction

b .foo	branch to .foo

```
add r1, r2, r3
b .foo
...
...
.foo:
       add r3, r1, r4
```

Conditional Branch Instructions

beq .foo	branch to .foo if $flags.E = 1$
bgt .foo	branch to .foo if $flags.GT = 1$

- 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
```

Example - II

Answer: Compute the factorial of the variable num.

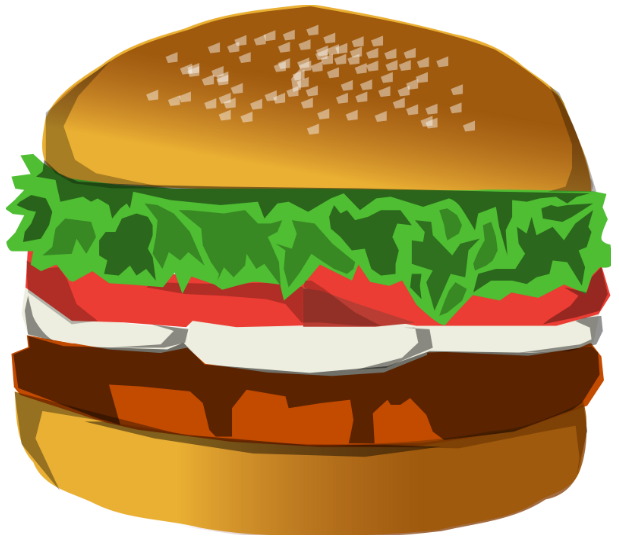
C

```
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 → 1729

Modifiers

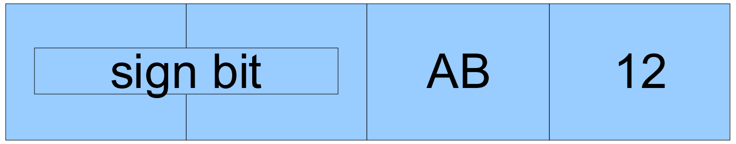
- We can add the following modifiers to an instruction that has an immediate operand
- Modifier :
  - default : mov → treat the 16 bit immediate as a signed number (automatic sign extension)
  - (u) : movu → treat the 16 bit immediate as an unsigned number
  - (h) : movh → left shift the 16 bit immediate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

Mechanism

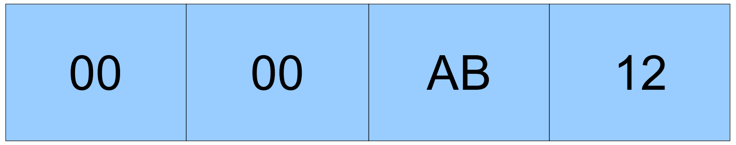
- The processor internally converts a 16 bit immediate to a 32 bit number
- It uses this 32 bit number for all the computations
- Valid only for arithmetic/logical insts
- We can control the generation of this 32 bit number
  - sign extension (default)
  - treat the 16 bit number as unsigned (u suffix)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

More about Modifiers

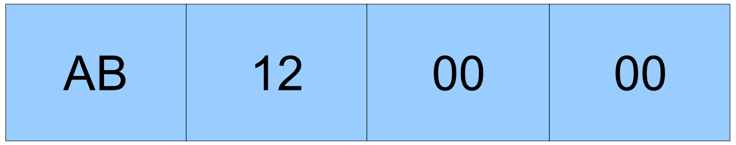
- default : mov r1, 0xAB 12



- unsigned : movu r1, 0xAB 12



- high: movh r1, 0xAB 12



Examples

- Move : 0x 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
```

Example

- Set r0 ← 0x 12 AB A9 2D

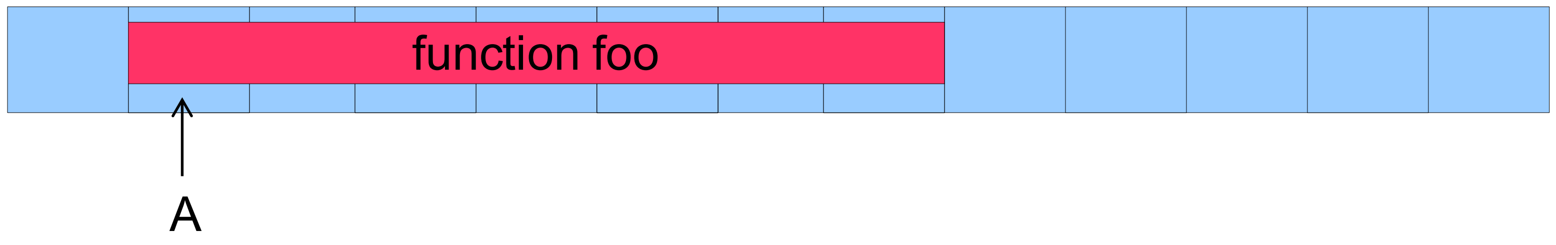
```
movh r0, 0x 12 AB
addu  r0, 0x A9 2D
```

Outline

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

Implementing Functions

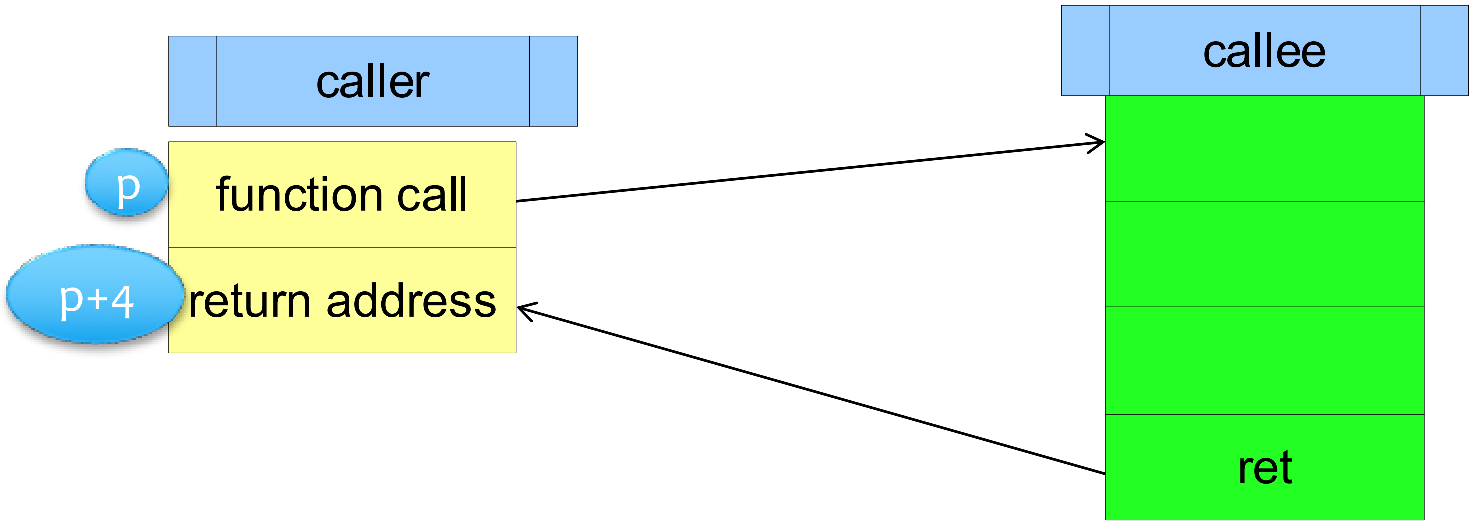
- Functions are blocks of assembly instructions that can be repeatedly invoked to perform a certain action
- Every function has a starting address in memory (e.g. foo has a starting address A



Implementing Functions - II

- To call a function, we need to set :
  - pc ← 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

Notion of the Return Address

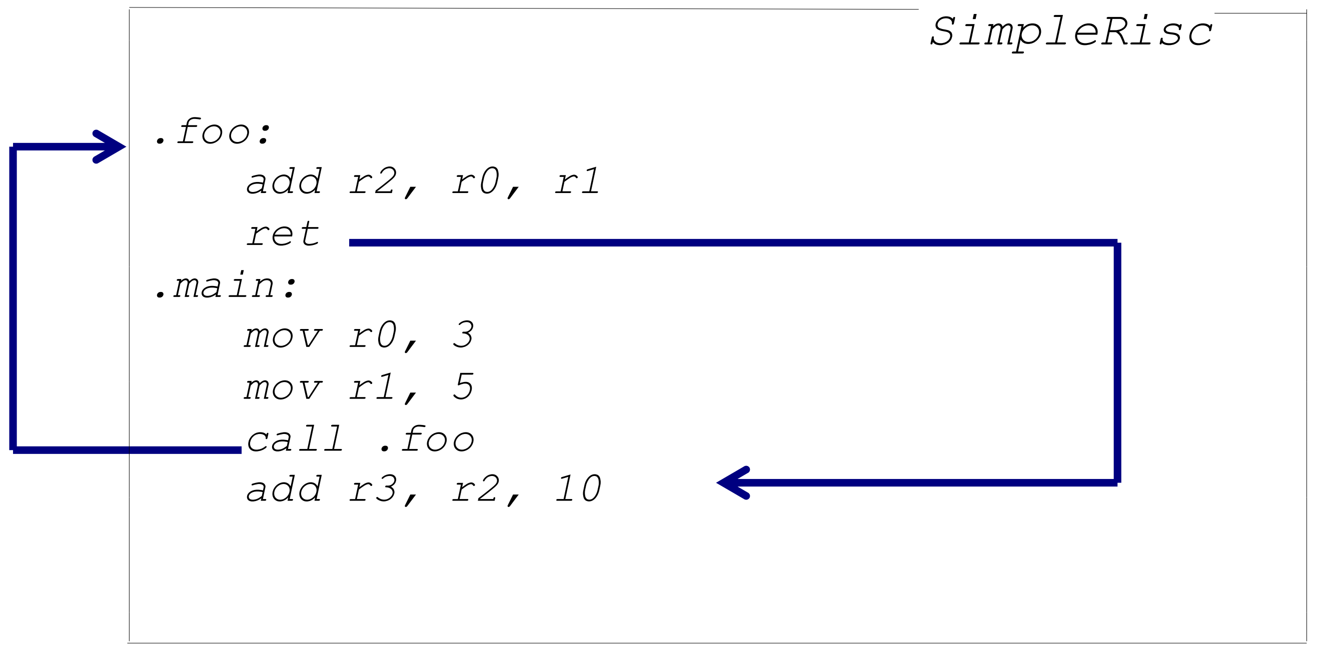


- PC of the call instruction → p
- PC of the return address → p + 4

because, every instruction takes 4 bytes

How do we pass arguments/ return values

* Solution : use registers



```
.foo:
    add r2, r0, r1
    ret
.main:
    mov r0, 3
    mov r1, 5
    call .foo
    add r3, r2, 10
```

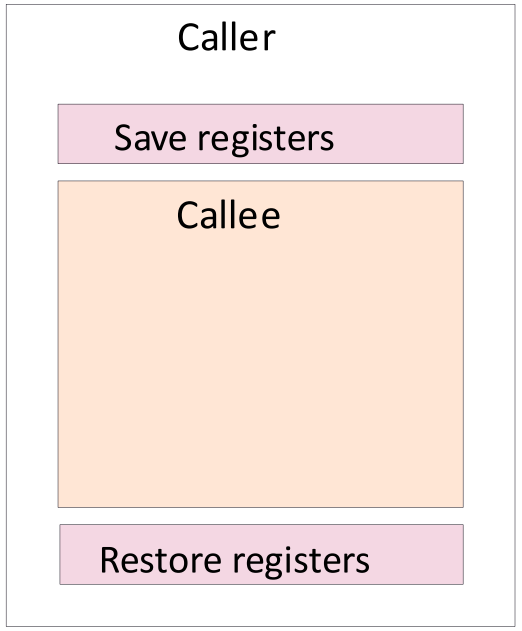
Problems with this Mechanism

- 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

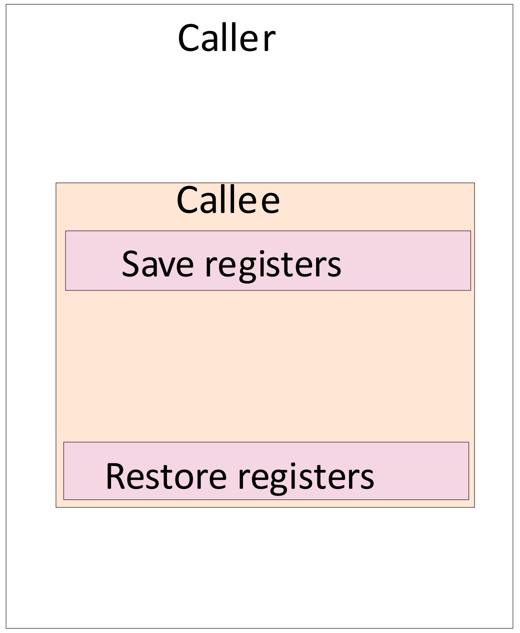
Register Spilling

- The notion of spilling
  - The caller can save the set of registers its 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



(a) Caller saved

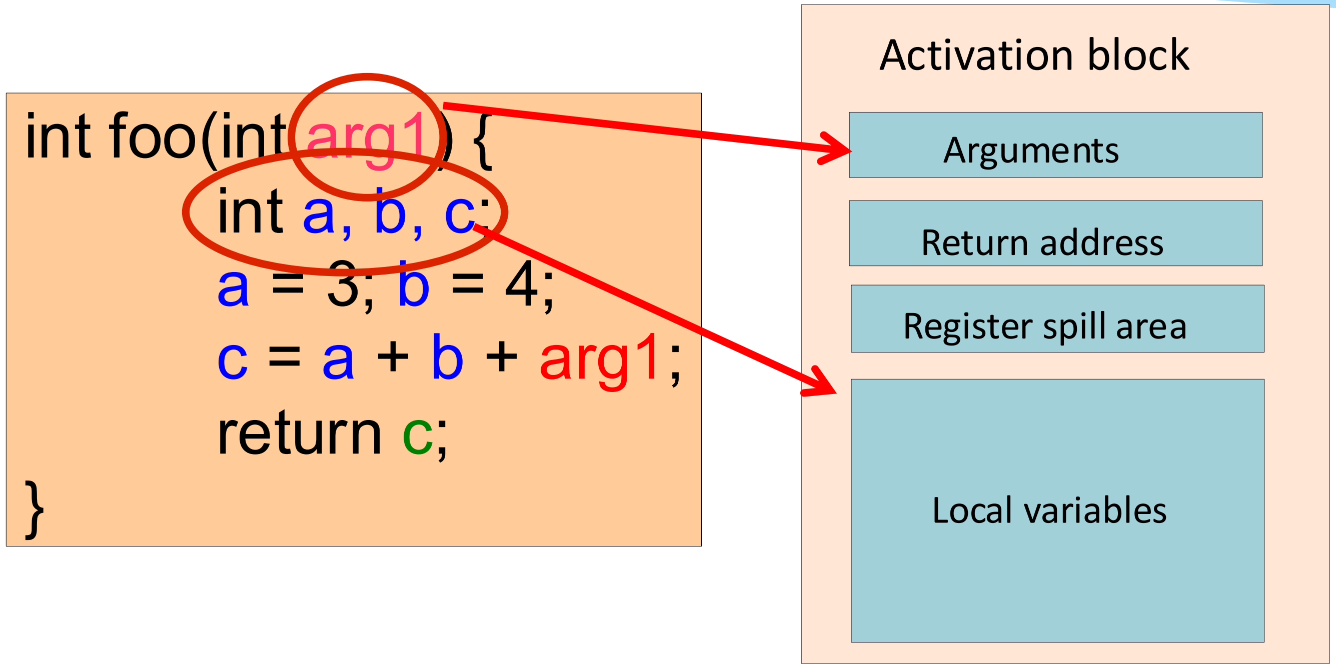


(b) Callee saved

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


```
int foo(int arg1) {
        int a, b, c;
        a = 3; b = 4;
        c = a + b + arg1;
        return c;
}
```

* 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

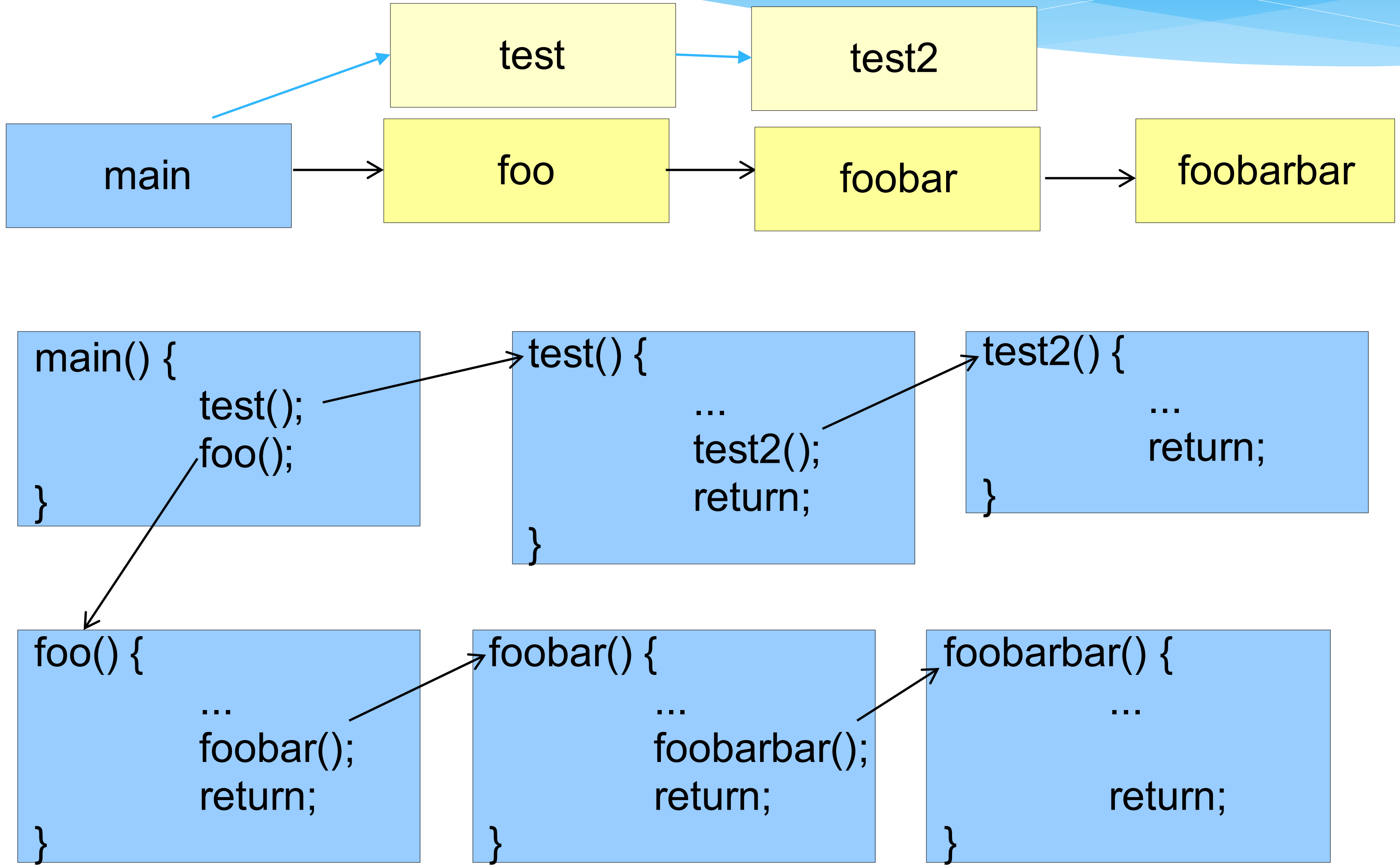
Using Activation Blocks - III

- 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



$$\begin{tikzcd}
 & \text{test} \arrow[r] & \text{test2} & \\
\text{main} \arrow[ur] \arrow[r] & \text{foo} \arrow[r] & \text{foobar} \arrow[r] & \text{foobarbar}
\end{tikzcd}$$

```
main() {
            test();
            foo();
}
```

```
test() {
            ...
            test2();
            return;
}
```

```
test2() {
            ...
            return;
}
```

```
foo() {
            ...
            foobar();
            return;
}
```

```
foobar() {
            ...
            foobarbar();
            return;
}
```

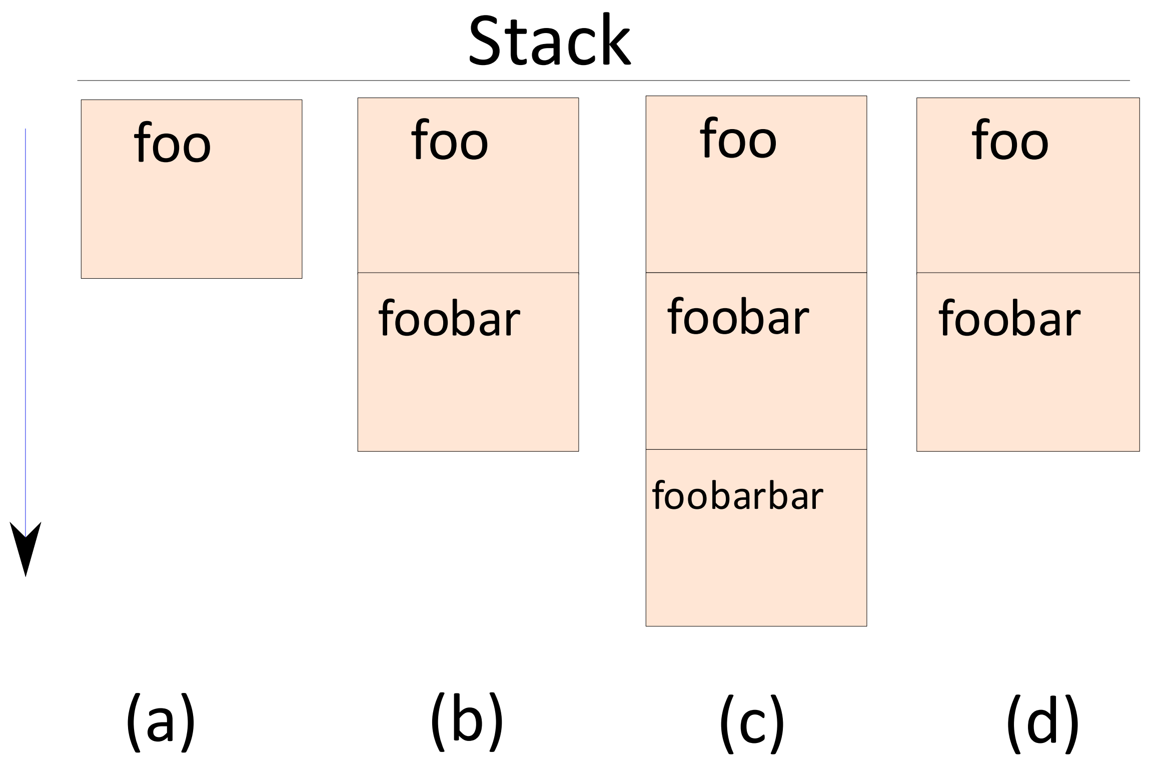
```
foobarbar() {
            ...

            return;
}
```

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 ← sp - <constant>
- De-allocating an activation block :
  - sp ← sp + <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) ← return address register
- call instruction
  - Puts *pc + 4* in ra, and jumps to the function
- ret instruction
  - Puts *ra* in *pc*

Recursive Factorial Program

C

```c
int factorial(int num) {
    if (num <= 1) return 1;
    return num * factorial(num - 1);
}
void main() {
    int result = factorial(10);
}
```

Factorial in SimpleRisc

```
.factorial:
    cmp r0, 1              /* compare (1,num) */
    beq .return
    bgt .continue
    b .return
.continue:
    sub sp, sp, 8          /* create space on the stack */

    st r0, [sp]            /* push r0 on the stack */
    st ra, 4[sp]           /* push the return address register */
    sub r0, r0, 1          /* num = num - 1 */
    call .factorial        /* result will be in r1 */
    ld r0, [sp]            /* pop r0 from the stack */
    ld ra, 4[sp]           /* restore the return address */
    mul r1, r0, r1         /* factorial(n) = n * factorial(n-1) */
    add sp, sp, 8          /* delete the activation block */
    ret
.return:
    mov r1, 1
    ret
.main:
    mov r0, 10
    call .factorial
```

nop instruction

- nop → does nothing
- Example : nop

Outline

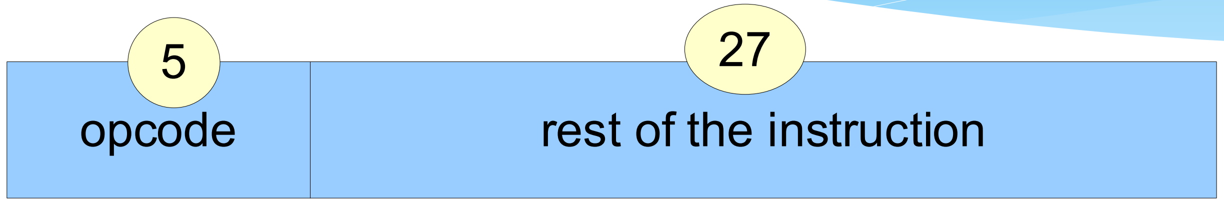
* Overview of Assembly Language
* Assembly Language Syntax
* SimpleRisc ISA
* Functions and Stacks
* SimpleRisc Encoding

Encoding Instructions

* Encode the SimpleRisc ISA using 32 bits.
* We have 21 instructions. Let us allot each instruction an unique code (opcode)

Instruction	Code	Instruction	Code	Instruction	Code
add	00000	not	01000	beq	10000
sub	00001	mov	01001	bgt	10001
mul	00010	lsl	01010	b	10010
div	00011	lsr	01011	call	10011
mod	00100	asr	01100	ret	10100
cmp	00101	nop	01101		
and	00110	ld	01110		
or	00111	st	01111		

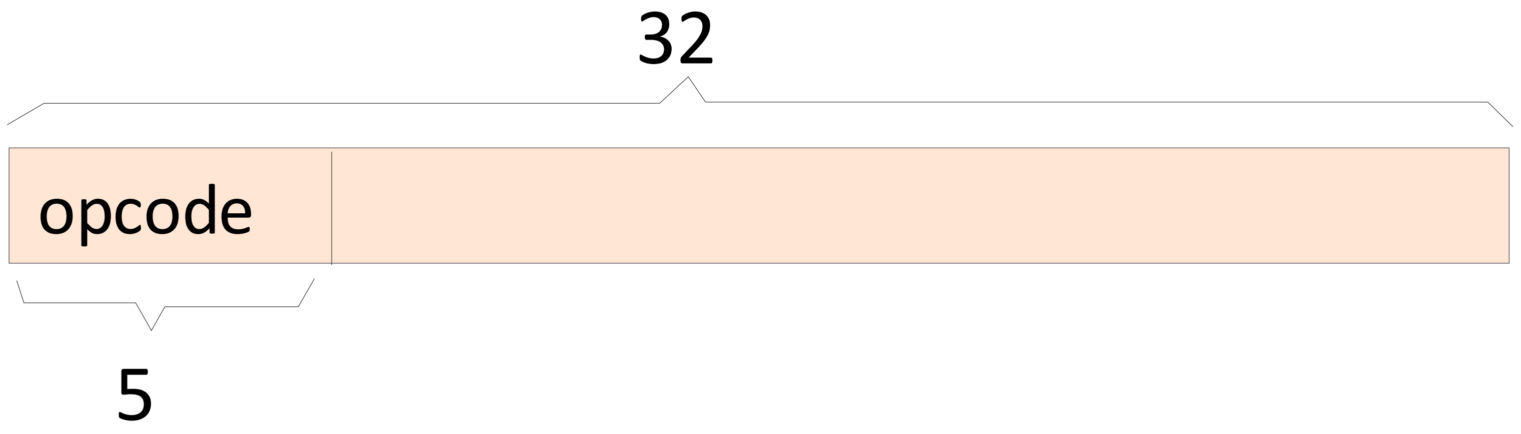
Basic Instruction Format



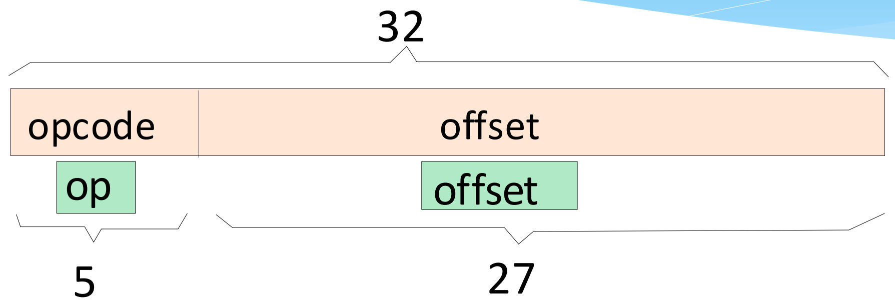
Inst.	Code	Format	Inst.	Code	Format
add	00000	add rd,rs1, (rs2/imm)	lsl	01010	lsl rd, rs1, (rs2/imm)
sub	00001	sub rd,rs1, (rs2/imm)	lsr	01011	lsr rd, rs1, (rs2/imm)
mul	00010	mul rd, rs1, (rs2/imm)	asr	01100	asr rd, rs1, (rs2/imm)
div	00011	div rd, rs1, (rs2/imm)	nop	01101	nop
mod	00100	mod rd, rs1, (rs2/imm)	ld	01110	ld rd.imm[rs1]
cmp	00101	cmp rs1, (rs2/imm)	st	01111	st rd. imm[rs1]
and	00110	and rd, rs1, (rs2/imm)	beq	10000	beq offset
or	00111	or rd,rs1, (rs2/imm)	bgt	10001	bgt offset
not	01000	not rd, (rs2/imm)	b	10010	b offset
mov	01001	mov rd, (rs2/imm)	call	10011	call offset
			ret	10100	ret

0-Address Instructions

- nop and ret instructions



1-Address Instructions

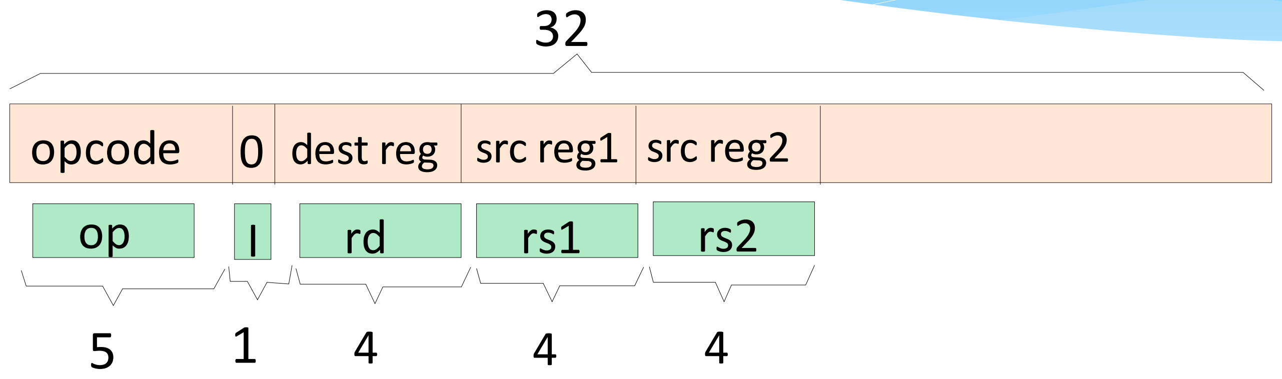


- Instructions – call, b, beq, bgt
- Use the branch format
- Fields :
  - 5 bit opcode
  - 27 bit offset (PC relative addressing)
  - Since the offset points to a 4 byte word address
  - The actual address computed is : PC + offset * 4

3-Address Instructions

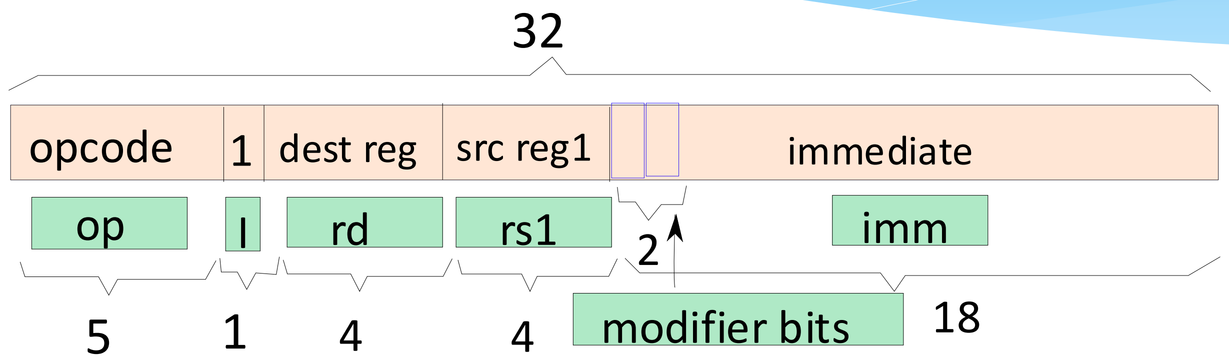
- Instructions – add, sub, mul, div, mod, and, or, lsl, lsr, asr
- Generic 3 address instruction
  - <opcode> rd, rs1, <rs2/imm>
- Let us use the I bit to specify if the second operand is an immediate or a register.
  - I = 0 → second operand is a register
  - I = 1 → second operand is an immediate
- Since we have 16 registers, we need 4 bits to specify a register

Register Format



- opcode → type of the instruction
- I bit → 0 (second operand is a register)
- dest reg → rd
- source register 1 → rs1
- source register 2 → rs2

Immediate Format

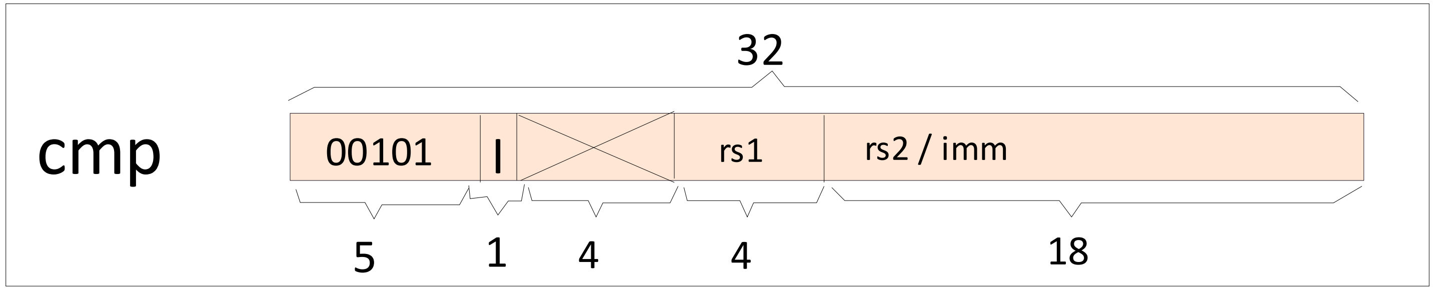


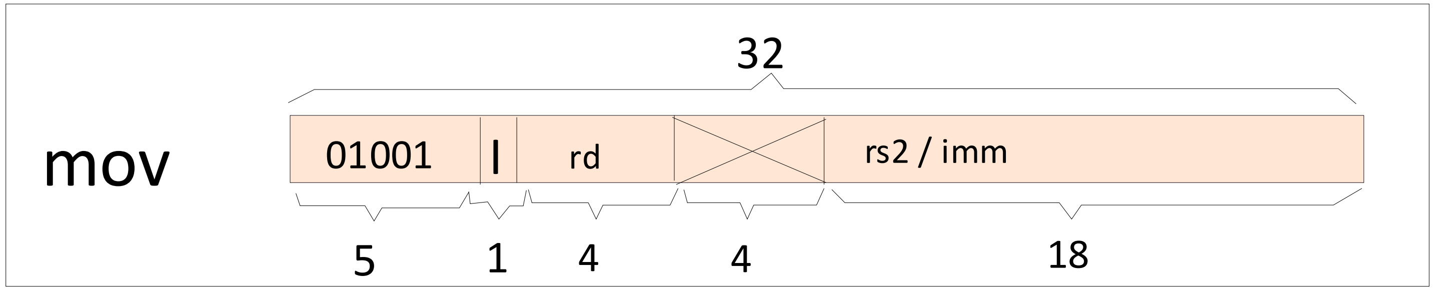
- opcode → type of the instruction
- I bit → 1 (second operand is an immediate)
- dest reg → rd
- source register 1 → rs1
- Immediate → imm
- modifier bits → 00 (default), 01 (u), 10 (h)

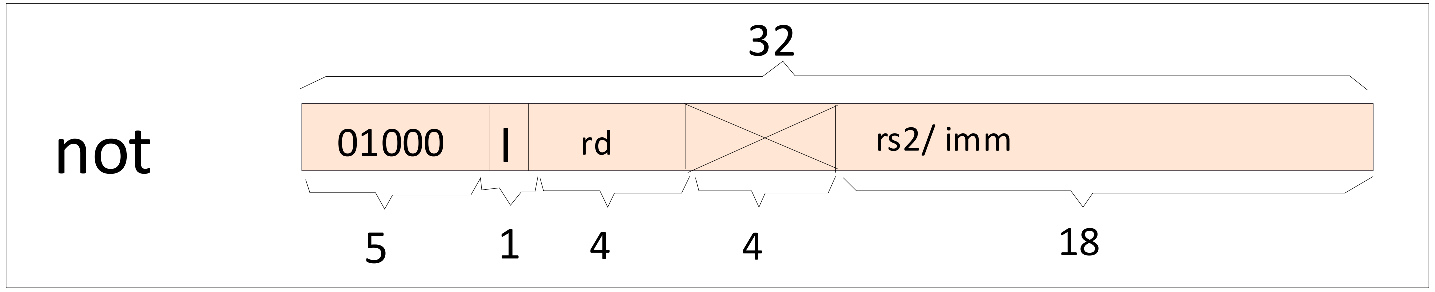
2 Address Instructions

- cmp, not, and mov
- Use the 3 address : immediate or register formats
- Do not use one of the fields

cmp, not, and mov

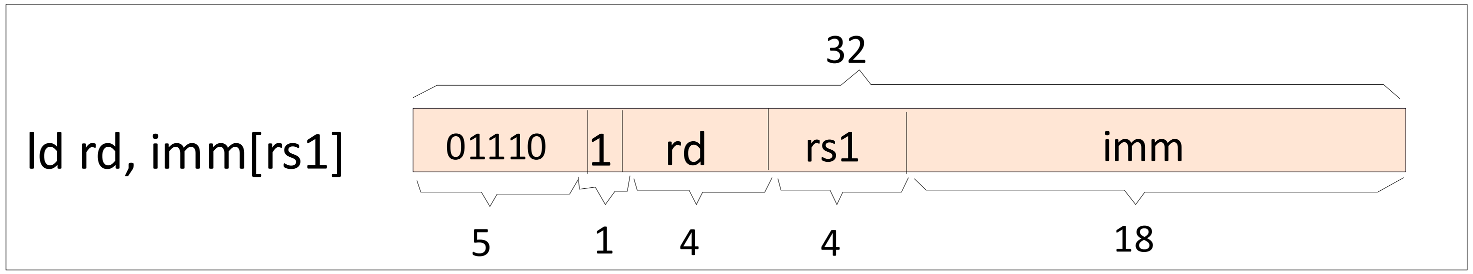






Load and Store Instructions

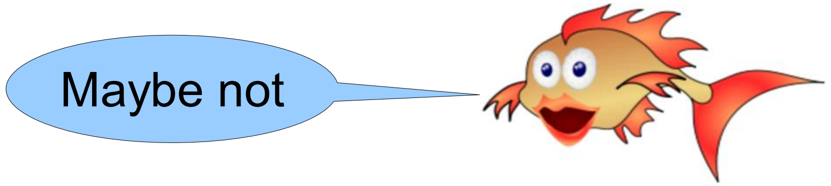
- ld rd, imm[rs1]
- rs1 → base register
- Use the immediate format.



Store Instruction

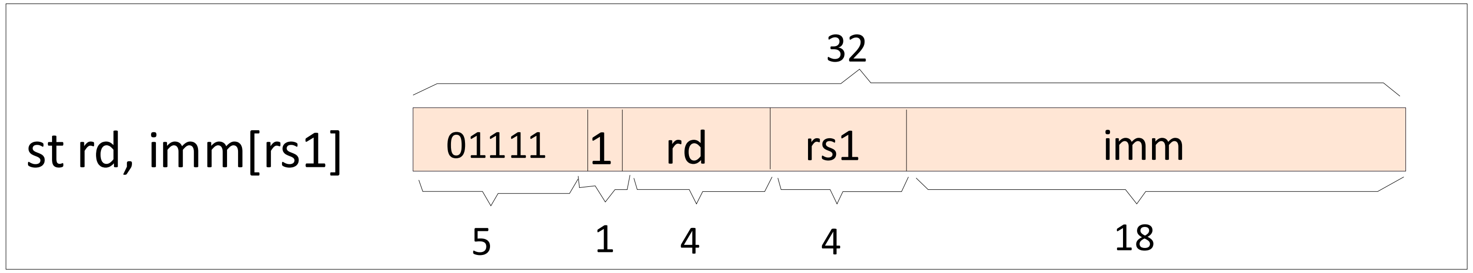
- Strange case of the store inst.
- st reg1, imm[reg2]
- has two register sources, no register destination, 1 immediate

- Cannot fit in the immediate format, because the second operand can be either be a register OR an immediate (not both)
  - Should we define a new format for store instructions ?



Store Instruction

- Let us make an exception and use the immediate format.
- We use the rd field to save one of the source registers
- st rd, imm[rs1]



Summary of Instruction Formats

Format	Definition				
branch	*op* (28-32)	*offset* (1-27)			
register	*op* (28-32)	I (27)	*rd* (23-26)	*rs* 1(19-22)	*rs* 2(15-18)
immediate	*op* (28-32)	I (27)	*rd* (23-26)	*rs* 1(19-22)	*imm* (1-18)
op → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register					
*rs*1 → source register 1, *rs*2 → source register 2, *imm* → immediate operand					

* branch format → nop, ret, call, b, beq, bgt
* register format → ALU instructions
* immediate format → ALU, ld/st instructions

THE END