ARCOS Group

uc3m | Universidad Carlos III de Madrid

L3: Fundamentals of assembler programming (3)
Computer Structure

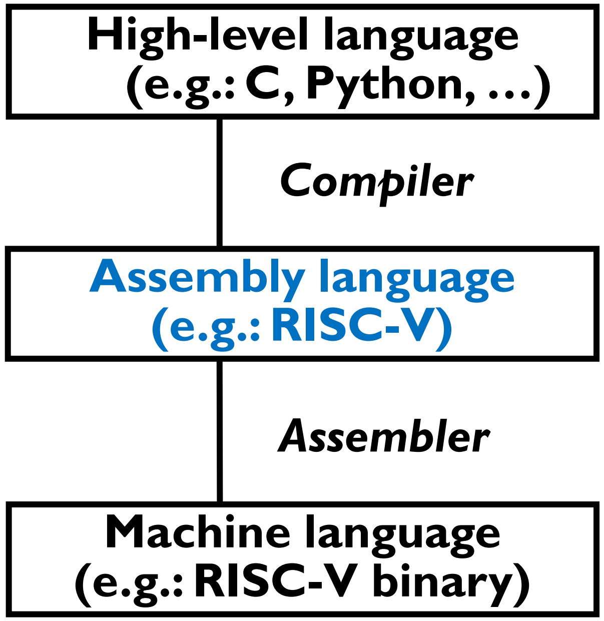
Bachelor in Computer Science and Engineering
Bachelor in Applied Mathematics and Computing
Dual Bachelor in Computer Science and Engineering and Business Administration

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats, addressing modes and instruction sets
- Procedure calls and stack convention

Different language levels

Reminder



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   t0, 0(x2)
lw   t1, 4(x2)
sw   t1, 0(x2)
sw   t0, 4(x2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

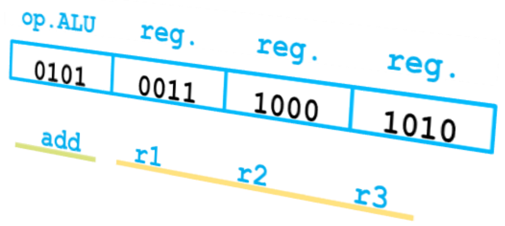
- An **assembly instruction** corresponds to a **machine instruction**
  - Example:
    - Assembly: `add x5, x6, x2`
    - Machine: `0x002302B3`

Instructions and pseudo-instructions RISC-V$_{32}$

Reminder

- An **assembler pseudo-instruction** corresponds to **one or more assembler instructions**
  - Example1:
    - Instruction: `mv    x2, x1`
    - Equivalent to: `add  x2, zero, x1`
  - Example2:
    - Instruction: `li   t1,   0x00800010`
    - Does not fit in 32 bits but can be used as a pseudo-instruction.
    - Equivalent to:
      - `lui t1, 0x00800`
      - `ori t1, t1, 0x010`
- An **assembly instruction** corresponds to a **machine instruction**
  - Example:
    - Assembly: `add x5, x6, x2`
    - Machine: `0x002302B3`

Instruction format

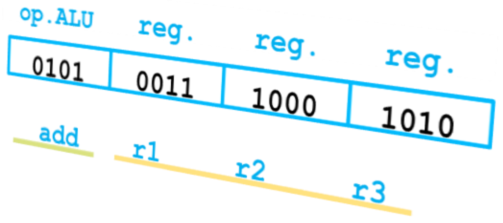


- **A machine instruction is encoded in binary**:
  - Size fits one or more words.
  - An instruction is divided into fields:
    - Each field encodes an element that includes the instruction
    - There may be implicit elements
  - Example of fields in an RISC-V instruction:

31 25	24 20	19 15	14 12	11 7	6 0
funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**

- The format specifies, for each field of the instruction:
  - The **meaning** of **each field**
  - **Coding used** in each field:
    - Binary, one's complement, etc.
  - The **number of bits** of each field:
    - The size of the fields limits the number of values to be encoded

Instruction format



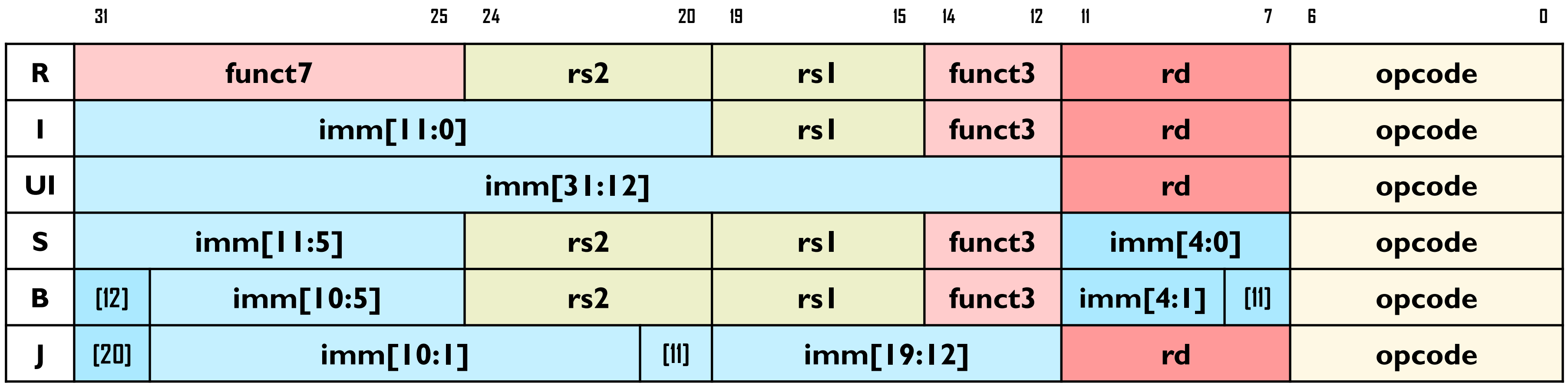
- A **machine instruction** is self-contained and **includes**:
  - Operation code
  - Operands (value or location of the value to be used)
  - Results (location where to store results)
  - Address of the next instruction
    - Implicit: PC ← PC + '4' (point to the following 32-bit instruction)
    - Explicit: j 0x01004 (modifies the PC)

- Usually:
  - One **architecture** offers a **few instruction formats**.
    - Simplicity in the design of the control unit.
  - **Fields** of the **same type** always **equal length**.
  - **Selection together** with the **operation code** (e.g.: add, addi)
    - **Usually the first field.**

Format length

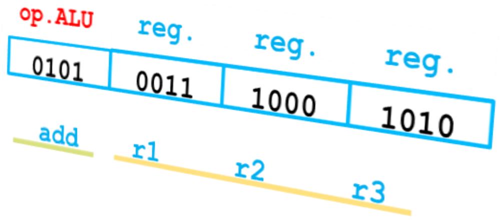
- The format length is number of bits to encode the instruction
  - The size of an instruction is usually one word (or multiples words)
  - In RISC-V_{32} the size of all instructions is one word (32 bits)
- Two types:
  - Fixed/Unique length:
    - All instructions with the same size
    - Examples:
      - MIPS32 (32 bits), PowerPC (32 bits), …
  - Variable length:
    - Different instructions can have different sizes
    - How to know the instruction length? → Op. code
    - Examples:
      - IA32 (Intel processors): variable number of bytes

Example: instruction format for RISC-V

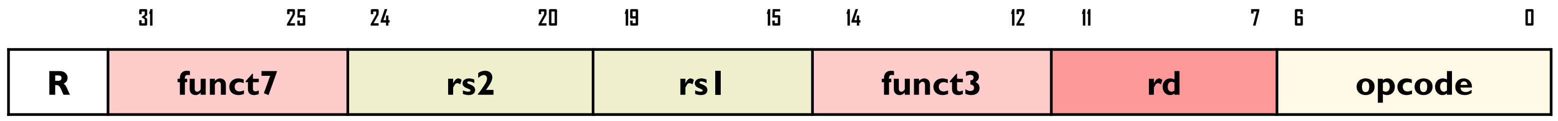


- **opcode** (7 bits): partially indicates the type of instruction format.
  - **R**egister, **I**mmediate, **U**pper **I**mmediate, **S**tore, **B**ranch, **J**ump
- **funct7+funct3** (10 bits): together with opcode, describe the op. to perform.
- **rs1** (5 bits): specifies the register as first operand.
- **rs2** (5 bits): specifies the register as second operand.
- **rd** (5 bits): specifies the target register.

Operation code

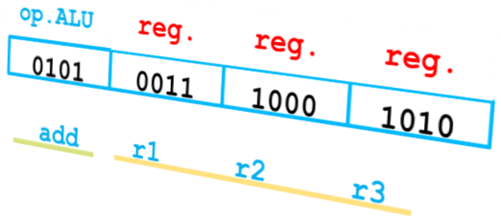


- Fixed size:
  - n bits → 2^n operation codes
  - m operation codes → $\lceil \log_2 m \rceil$ bits.
- Extension fields
  - RISC-V (arithmetic-logic instructions)
  - `Op` = 0; the instruction is encoded in `funct`*`X`*



- Variable sizes:
  - More frequent instructions= shorter sizes

Locations of operands



1. In the instruction

   li t0 0x123

2. In registers (processor)

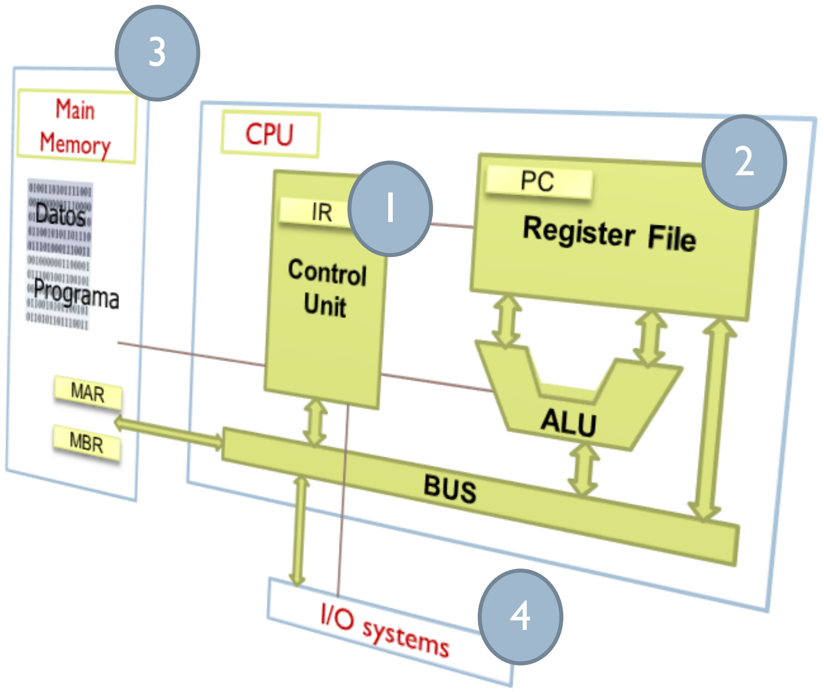
   li t0 0x123

3. Main memory

   lw t0 address(x0)

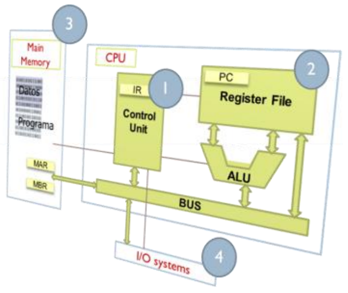
4. Input/output modules

   in t0 0xFEB



Locations of operands

Reminder



1. In the instruction

   li   t0 0x123

2. In registers (processor)

   li   t0 0x123

3. Main memory

   lw t0 address(x0)

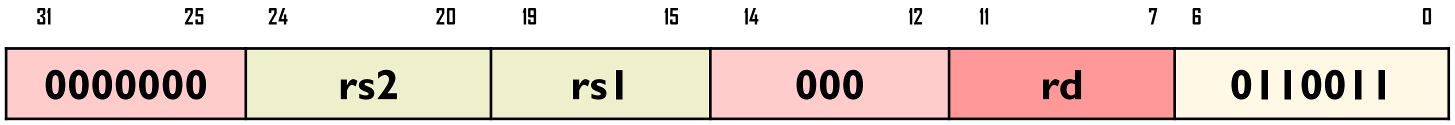
   - num(registro): represents the address obtained by summing num with the address stored in the register

4. Input/output modules

   in   t0 0xFEB

Example instruction and associated format in RISC-V

- `add rd rs1 rs2`



Exercise

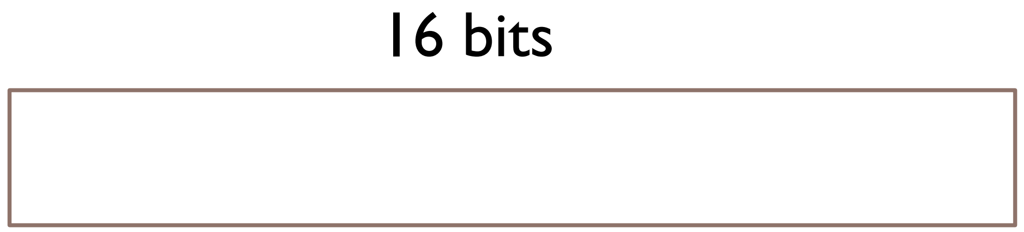
- A 16-bit computer
  has an instruction set of 60 instructions and
  a register file of 8 registers.

  The following is requested:
  Define the format of this instruction: ADDx R1 R2 R3, where R1, R2 and R3 are registers.

Exercise (solution)

word-> 16 bits
60 instructions
8 registers (in RB)
ADDx R1(reg.), R2(reg.), R3(reg.)

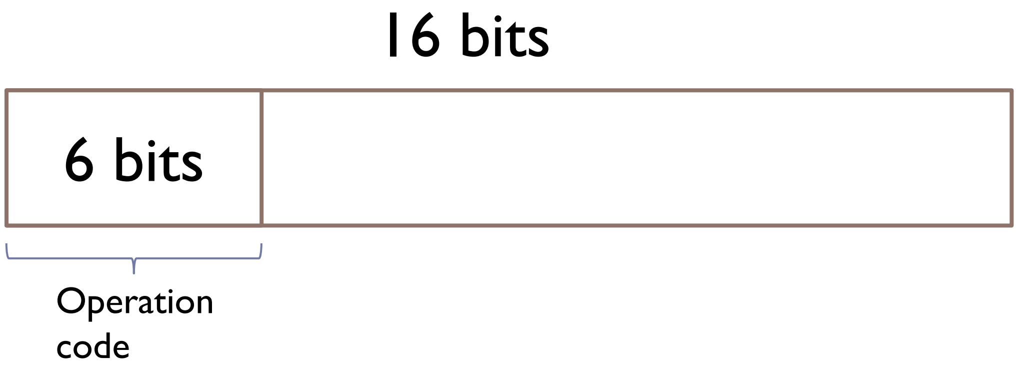
- 16-bit word defines the size of the instruction



Exercise (solution)

word-> 16 bits
60 instructions
8 registers (in RB)
ADDx R1(reg.), R2(reg.), R3(reg.)

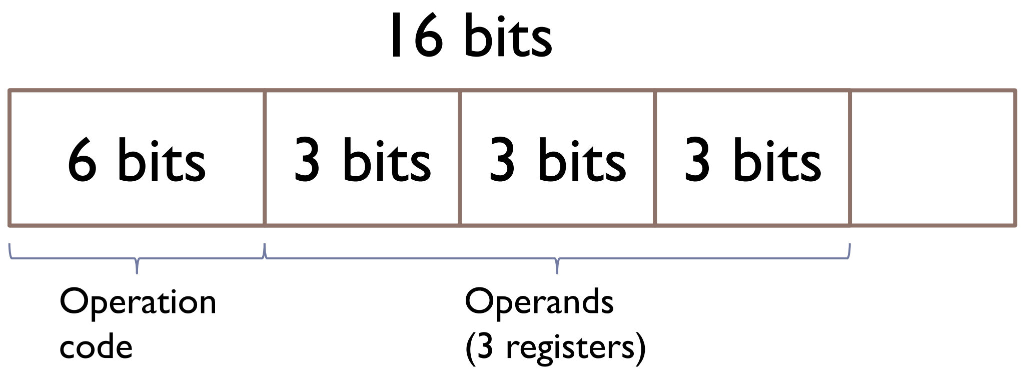
- To encode 60 instructions, 6 bits are required for the operation code (minimum)



Exercise (solution)

word-> 16 bits
60 instructions
8 registers (in RB)
ADDx R1(reg.), R2(reg.), R3(reg.)

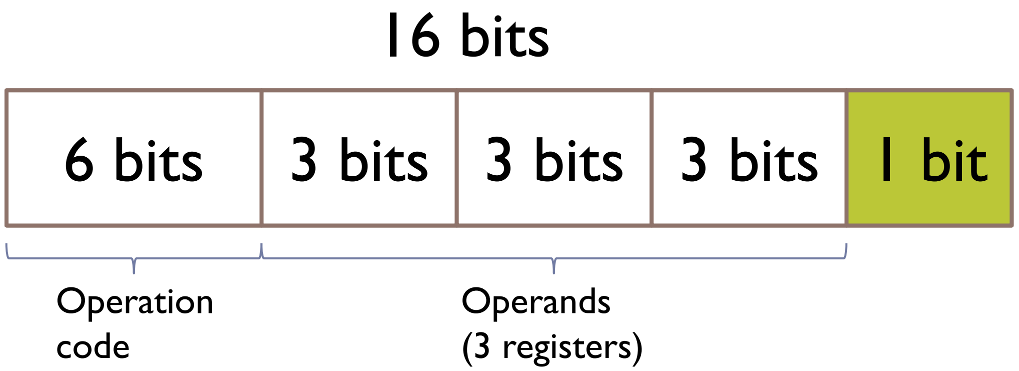
- For 8 registers, 3 bits are required (minimum)



Exercise (solution)

word-> 16 bits
60 instructions
8 registers (in RB)
ADDx R1(reg.), R2(reg.), R3(reg.)

- 1 bit left over (16-6-3-3-3 = 1), used for padding



Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats, addressing modes and instruction sets
- Procedure calls and stack convention

Addressing modes

- The addressing mode is a procedure for determining the location of an operand, a result or an instruction

- Implicit
- Immediate
- Direct
  - to register
  - to memory
- Indirect
  - to register
  - to memory
- Relative
  - to index register
  - base register
  - to PC
  - to stack

Modos de direccionamiento en RISC-V

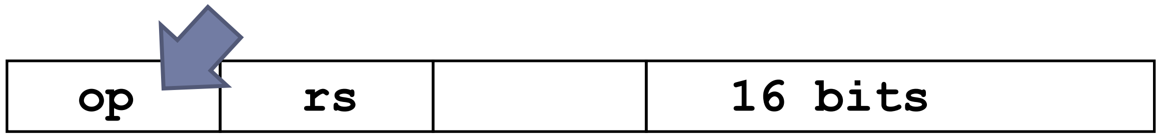
Immediate	value
Direct	
To memory	address
To register	xr
Indirect	
To memory	
To register	(xr)
Relative to	
□ register	offset(xr)
□ stack	offset(sp)
□ PC	beq … label1

Addressing modes

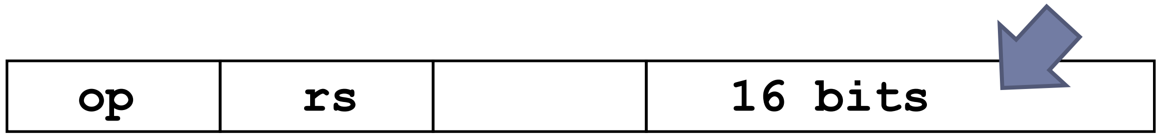
- The addressing mode is a procedure for determining the location of an operand, a result or an instruction

  - **Implicit**
  - **Immediate**
  - **Direct**
    - **to register**
    - **to memory**
  - Indirect
    - to register
    - to memory
  - Relative
    - to index register
    - base register
    - to PC
    - to stack

Implicit addressing

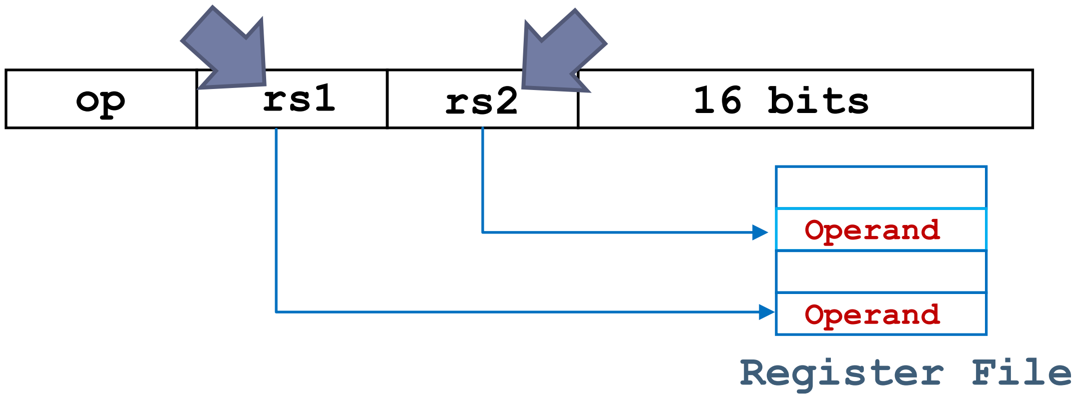
Description	▸ The operand is not coded in the instruction but is part of the instruction.
Example	▸ auipc a0 0x12345 ▸ a0 = PC + (0x12345 << 12). ▸ a0 is one operand, **PC is the other (implicit)** 
(V/I) Advantages / Disadvantages	✓ Fast: no need to access memory. ✗ But it is only possible in a few cases.

Immediate addressing

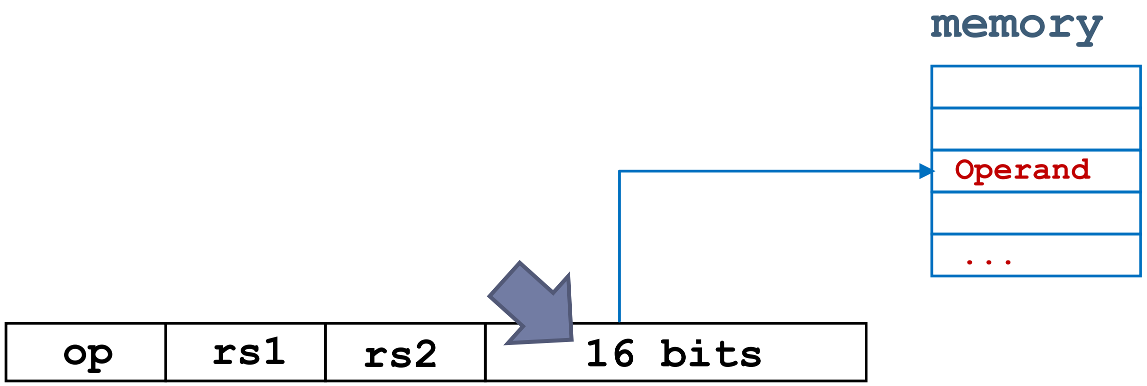
Description	▸ The operand is part of the instruction.
Example	▸ li a0 0x4f51 ▸ Loads in register a0 the immediate value 0x4f51. ▸ The **value 0x00004f51** is in an **immediate** field. 
(V/I) Advantages / Disadvantages	✓ It is fast: no need to access memory. ✗ Value does not always fit in a word: ▸ It does not fit in 32-bits, it is equivalent to: □ `lui t1, 0x87654` □ `ori t1, t1, 0x321`

Direct to register addressing

register addressing

Description	Operand is in a register.
Example	mv a0 a1 • Copy in the a0 register the value stored in the a1 register. • The identifier of a0 and a1 is encoded in the instruction.  op rs1 rs2 16 bits Operand Operand Register File
(V/I) Advantages / Disadvantages	✗ The number of records is limited. ✓ Access to registers is fast. ✓ The number of registers is small => few bits for encoding, shorter instructions.

Direct to memory addressing

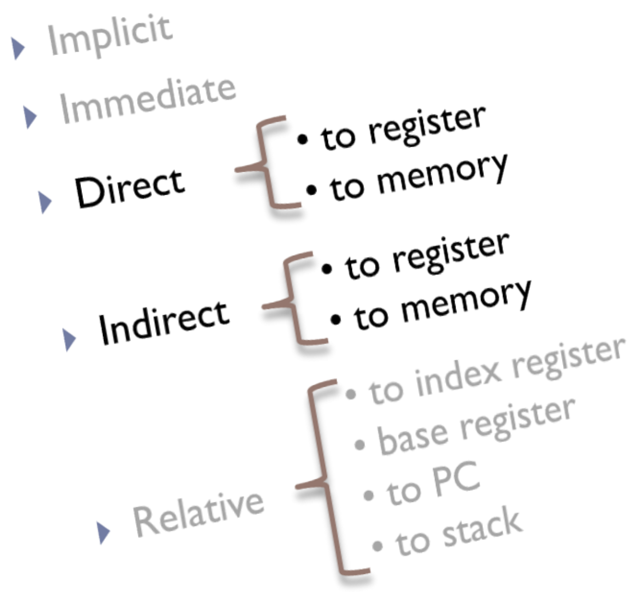
Description	The operand is in memory, and the address is encoded in the instruction (not availabe for RISC-V).
Example	LD .R1 #0xFFF0 # IEEE 694 Loads in R1 the word stored in 0xFFF0. 
(V/I) Advantages / Disadvantages	✗ Memory access is slower compared to registers ✗ Long addresses => longer instructions ✓ Access to a large address space (capacity greater than R.F.)

Addressing modes

- The addressing mode is a procedure for determining the location of an operand, a result or an instruction

- Implicit
- Immediate
- Direct
  - to register
  - to memory
- **Indirect**
  - **to register**
  - **to memory**
- Relative
  - to index register
  - base register
  - to PC
  - to stack

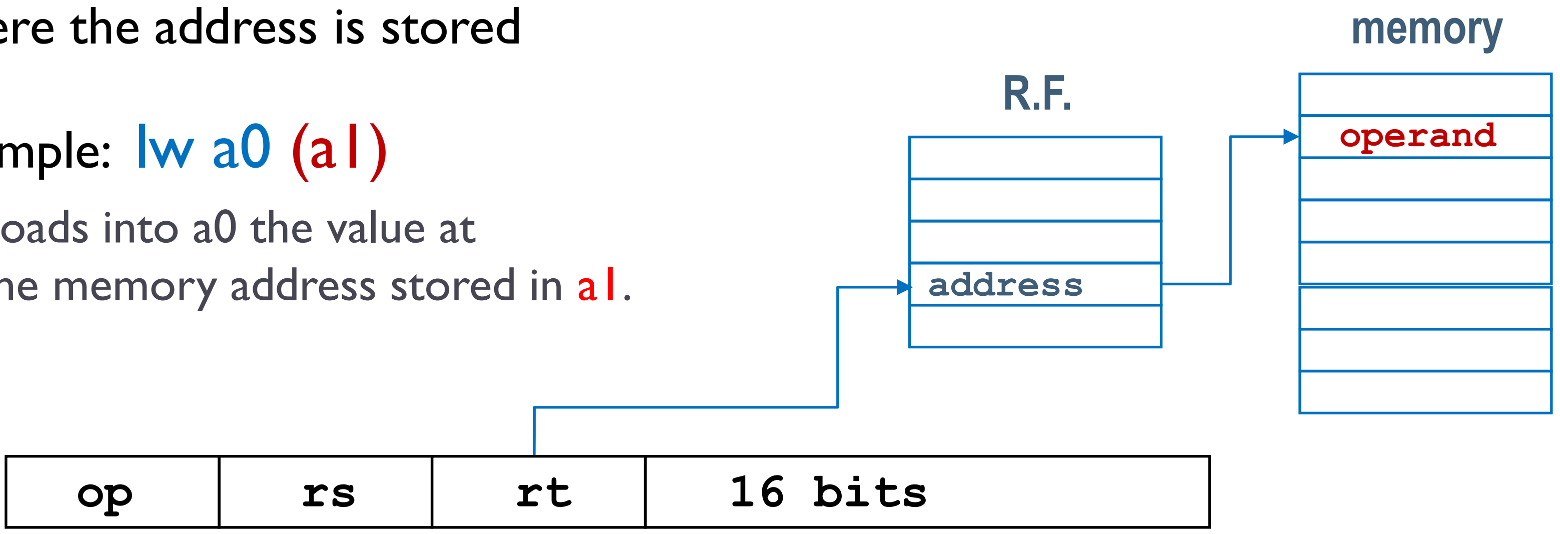
Direct vs. indirect addressing



- Direct addressing indicates where the operand is located:
  - In which register or memory location
- Indirect indicates where the address of the operand is located:
  - This address must be accessed in memory
  - A level (or several) of addressing is incorporated

Register indirect addressing

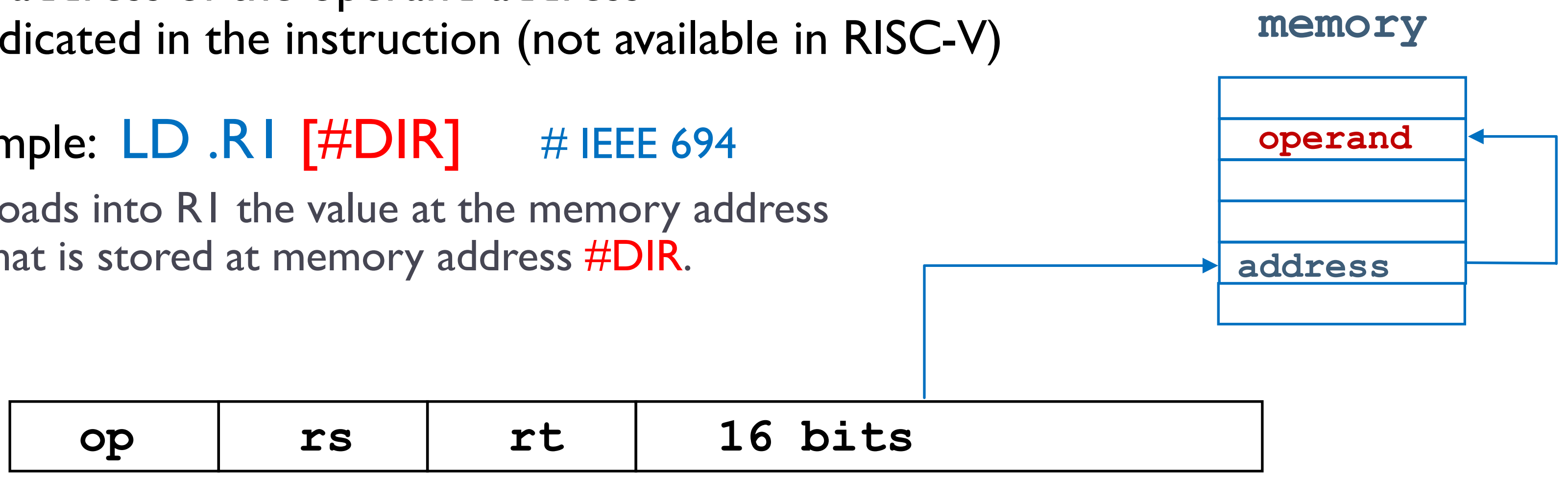
- The instruction has the register where the address is stored
- Example: lw a0 (a1)
  - Loads into a0 the value at the memory address stored in a1.



- A/D
  - ✓ Wide address space, short instructions
  - Pseudo-instruction equivalent to lw a0 0(a1)

Indirect addressing

- The address of the operand address is indicated in the instruction (not available in RISC-V)
- Example: LD .R1 [#DIR] # IEEE 694
  - Loads into R1 the value at the memory address that is stored at memory address #DIR.



- V/I
  - ✓ Large address space
  - ✓ Addressing can be nested, multilevel or cascading
    - Example: LD .R1 [[[.R1]]]
  - ✗ May require several memory accesses
  - ✗ slower instructions to execute

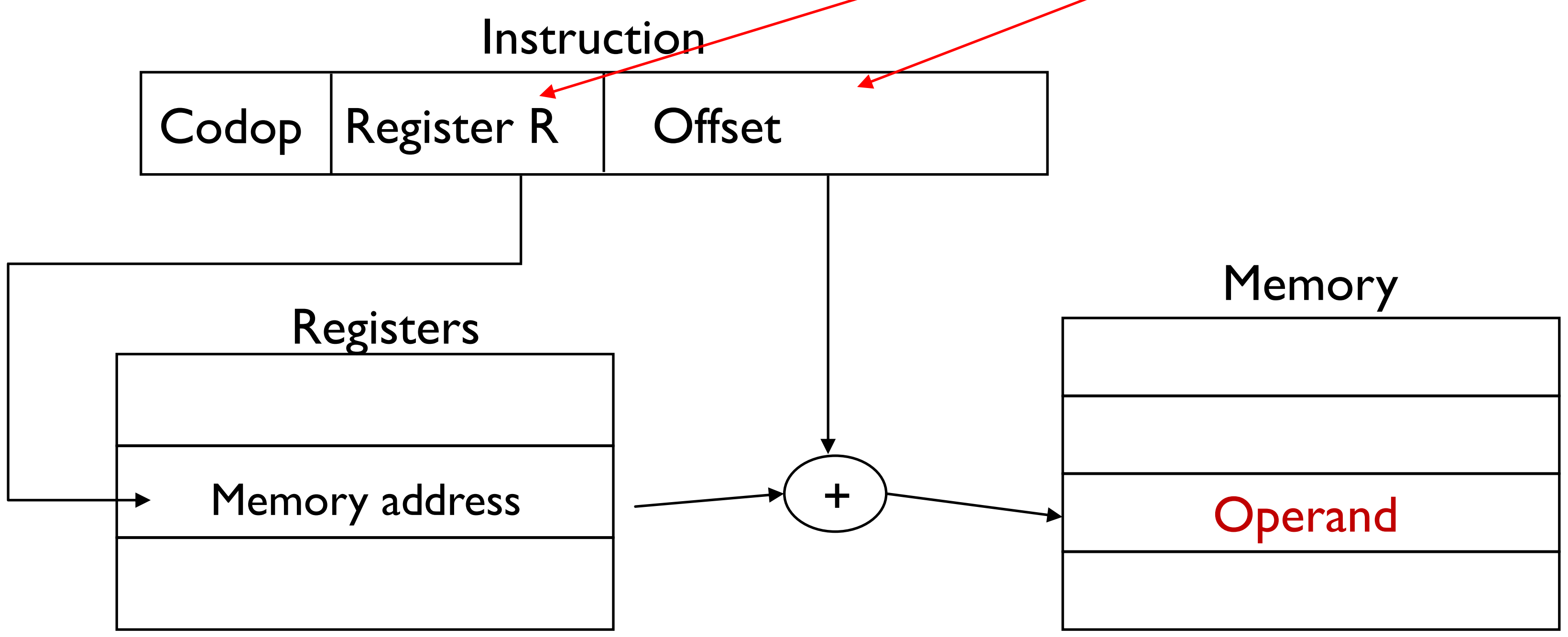
Addressing modes

- The addressing mode is a procedure for determining the location of an operand, a result or an instruction

  - Implicit
  - Immediate
  - Direct
    - to register
    - to memory
  - Indirect
    - to register
    - to memory
  - **Relative**
    - **to index register**
    - **base register**
    - **to PC**
    - **to stack**

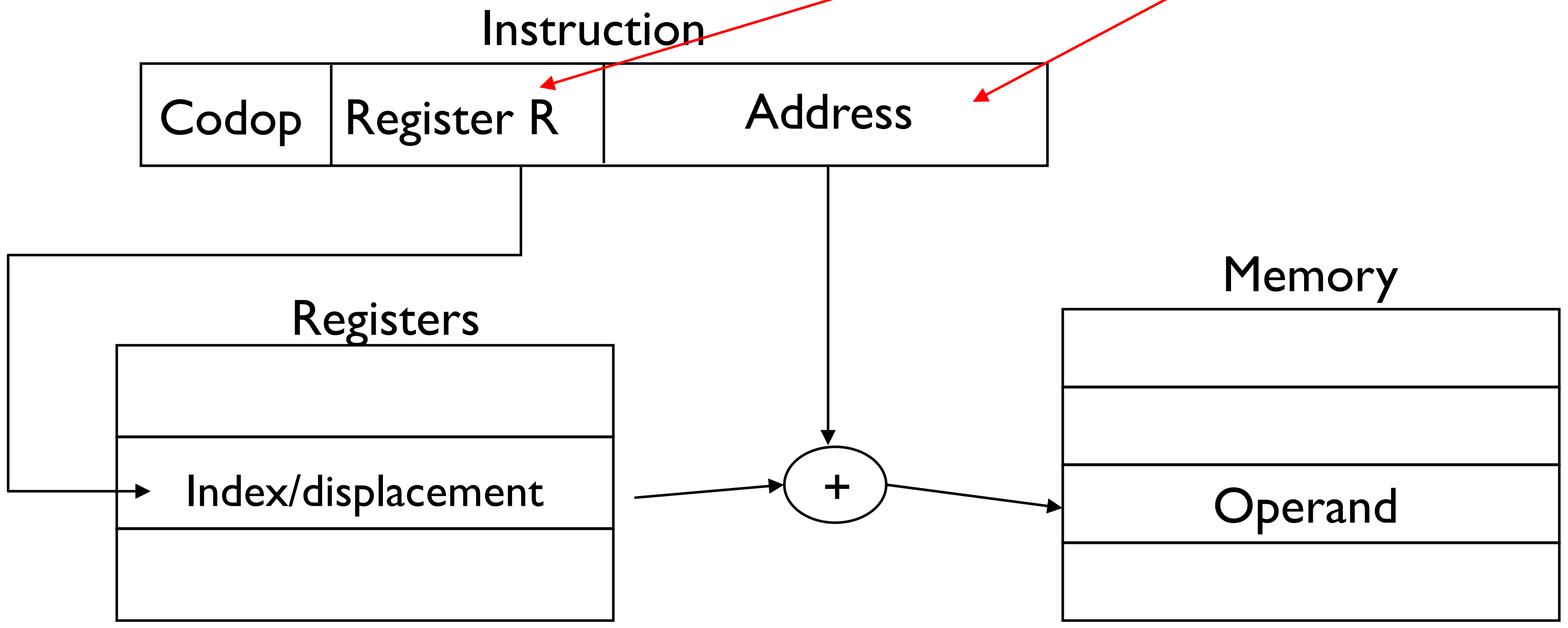
Base-register addressing

- Example: lw a0 12(t1)
  - Loads into a0 the contents of the memory location given by t1 + 12
  - Uses two fields of the instruction, **t1 has the base address**



Index-register addressing

- Example: lw a0 dir(t1)
  - Loads into a0 the contents of the memory location given by t1 + dir
  - Uses two fields: t1 **represents** the displacement (**index**) with respect to the direction dir



Utility: access to vectors

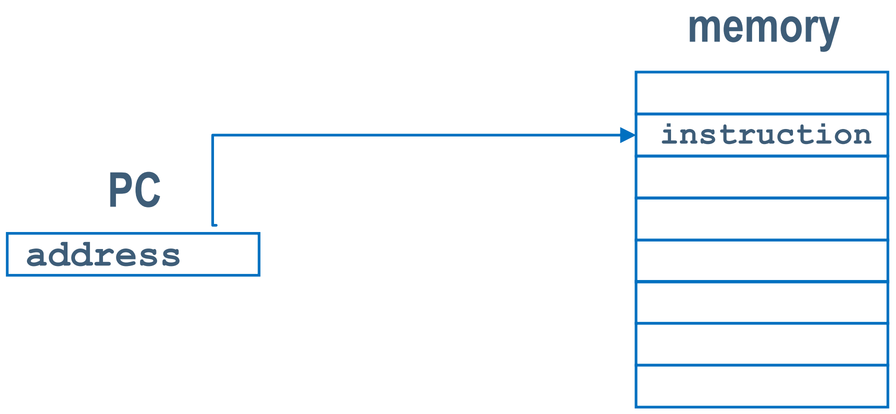
```
int v[5] ;

main ( )
{
   v[3] = 5 ;

   v[4] = 8 ;

   v[1] = 3 ;
}
```

```
.data
  v:  .zero 20    # 5int*4bytes/int

.text
main:
        la   t0 v
        li   t1 5
        sw   t1 12(t0)

        li   t0 16
        li   t1 8
        sw   t1 v(t0)

        la    t0 v
        addi  t0 t0 4
        li    t1 3
        sw    t1 (t0)
```

The Program Counter register (PC)…

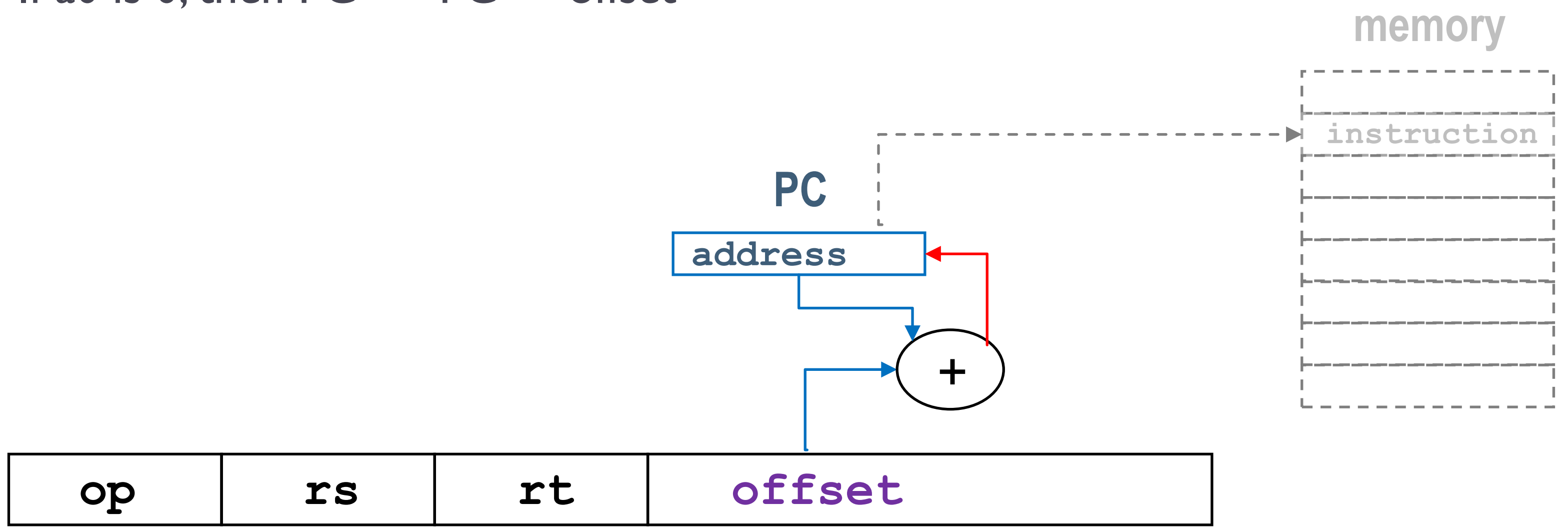
Reminder

- It is a 32-bit (4-byte) register in a 32-bit computer.
- It stores the address of the next instruction to be executed
  - Points to a word (4 bytes) with the instruction to be executed
  - PC on a 32-bit computer is upgraded by default as PC = PC + 4



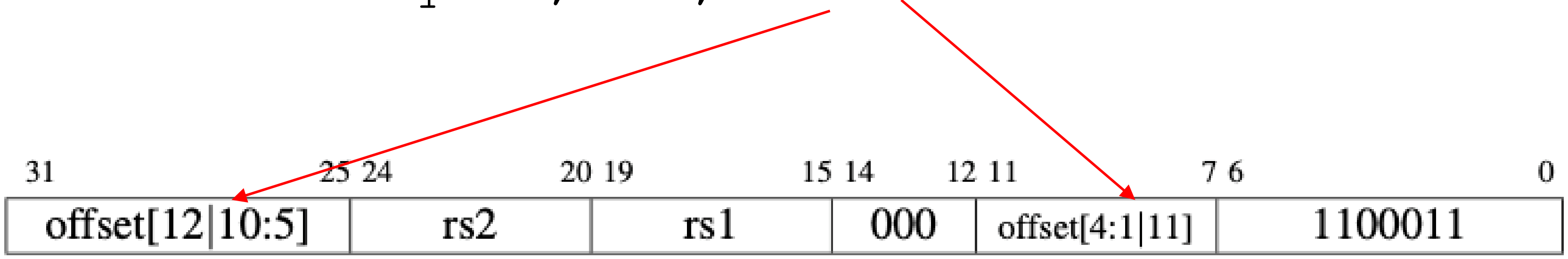
PC-relative addressing

- Example: beq a0 x0 label
  - The assembler encode label as the offset from the beq instruction to the memory position associated to label.
    - **Label encoded as displacement** (address -> # instructions to jump)
  - If a0 is 0, then PC <= PC + “offset”



PC-relative addressing in RISC-V

- Instruction `beq t0, x1, offset` is encoded as:



- Label has to be encoded in the "offset" field as an offset relative to the beq instruction at the time of execution (PC points to the first byte of the following instruction)
  - The offset value can be positive or negative
- How is the PC updated if t0 == x1?
  - If the condition is met:
    - PC = PC + offset
  - If the condition is not met:
    - PC = PC + 4

PC-relative addressing in RISC-V

- How much is **fin** value when generating machine code?

```
bucle:    beq   t0, x1, fin
          add   t8, t4, t4
          addi  t0, x0, -1
          j     bucle
fin:      mv    t1 x0
          . . .
```

- The value of fin is:
  - `fin == 12`
  - When an instruction is executed, the PC points to the next instruction
  - Skip 3 instructions ("addi", "j" y "mv")
  - Each instruction to be skipped is 4 bytes

Used in loops

```
        li    t0  8
        li    t1  4
        li    t2  1
        li    t4  0
while:  bge   t4  t1  fin
        mul   t2  t2  t0
        addi  t4  t4  1
        j     while
fin:    mv    t2  t4
```

- end represents the address where the instruction `mv` is stored
- while represents the address where the instruction `bge` is stored

Used in loops

```
        li   t0 8
        li   t1 4
        li   t2 1
        li   t4 0
while:  bge  t4 t1 fin
        mul  t2 t2 t0
        addi t4 t4 1
        j    while
fin:    mv   t2 t4
```

Address	Content
0x0000100	`li t0 8`
0x0000104	`li t1 4`
0x0000108	`li t2 1`
0x000010C	`li t4 0`
0x0000110	`bge t4 t1 fin`
0x0000114	`mul t2 t2 t0`
0x0000118	`addi t4 t4 1`
0x000011C	`j while`
0x0000120	`mv t2 t4`

Used in loops

```
        li   t0 8
        li   t1 4
        li   t2 1
        li   t4 0
while:  bge  t4 t1 fin
        mul  t2 t2 t0
        addi t4 t4 1
        j    while
fin:    mv   t2 t4
```

Address	Content
0x0000100	`li   t0 8`
0x0000104	`li   t1 4`
0x0000108	`li   t2 1`
0x000010C	`li   t4 0`
0x0000110	`bge  t4 t1 fin`
0x0000114	`mul  t2 t2 t0`
0x0000118	`addi t4 t4 1`
0x000011C	`j    while`
0x0000120	`mv   t2  t4`

- end encoded as displacement relative to current PC => 3
  PC = PC + 3 * 4
- while encoded as displacement relative to current PC => -4
  PC = PC + (-4)*4

Used in loops

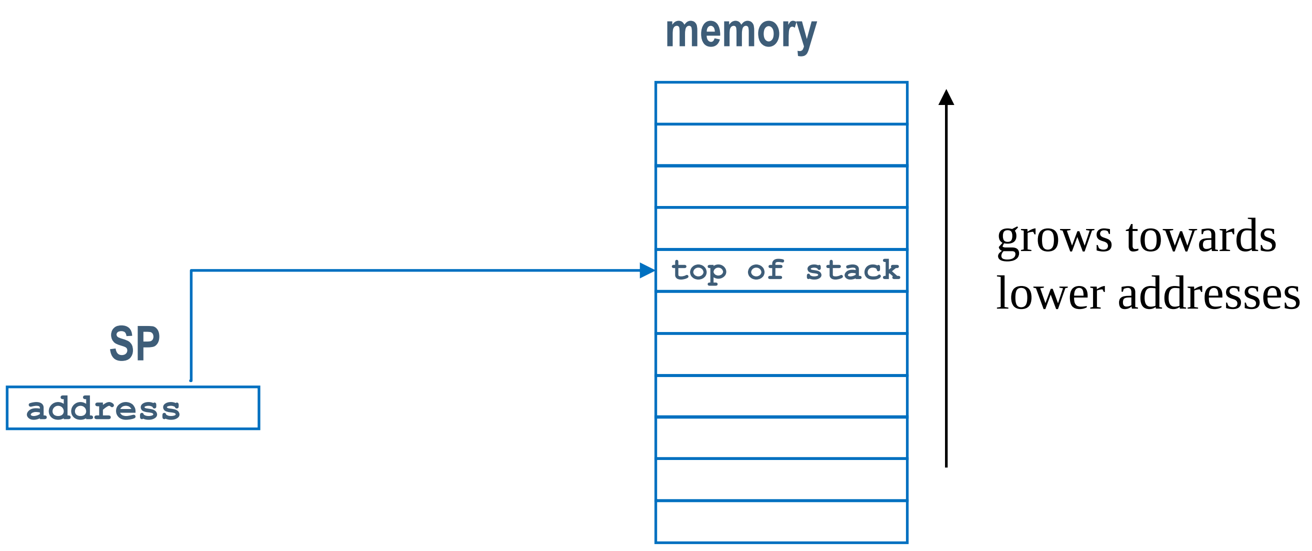
```
        li   t0 8
        li   t1 4
        li   t2 1
        li   t4 0
while:  bge  t4 t1 fin
        mul  t2 t2 t0
        addi t4 t4 1
        j    while
fin:    mv   t2 t4
```

Address	Content
0x0000100	li t0 8
0x0000104	li t1 4
0x0000108	li t2 1
0x000010C	li t4 0
0x0000110	bge t4 t1 12
0x0000114	mul t2 t2 t0
0x0000118	addi t4 t4 1
0x000011C	j -16
0x0000120	mv t2 t4

- end encoded as displacement relative to current PC => 3
  PC = PC + 3 * 4 = PC + 12
- while encoded as displacement relative to current PC => -4
  PC = PC + (-4)*4 = PC - 16

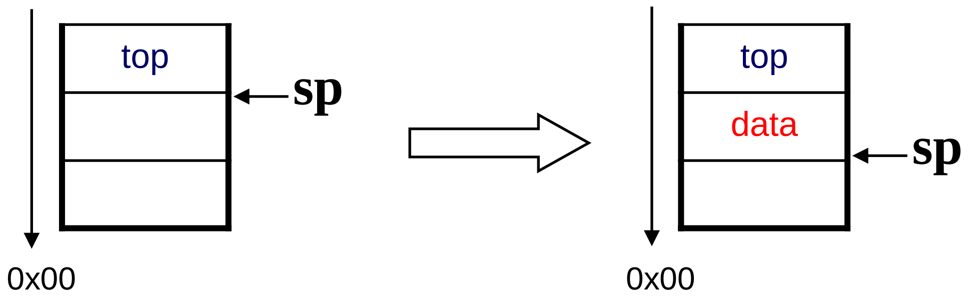
Stack addressing

- The Stack Pointer (SP) :
  - It is a 32-bit (4-byte) register in the RISC-V_{32}
  - Stores the address of the top of the stack.
    - Point to a word (4 bytes)
  - Two types of operations:
    - push
    - pop

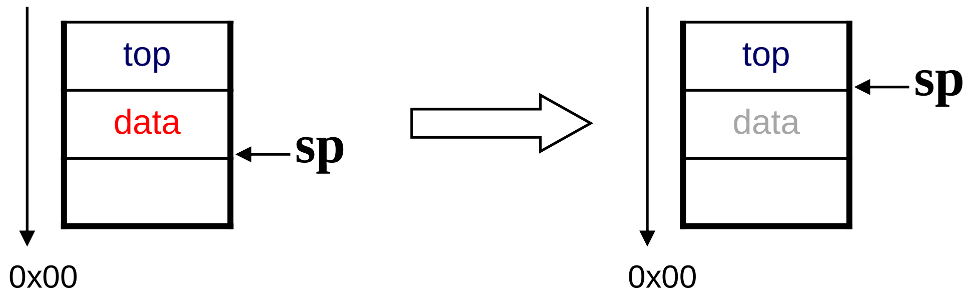


The stack grows towards lower addresses

PUSH Reg Stacks the contents of the register (data)



POP Reg Unstack the contents of the register (data)
Copy data in the Reg register



Stack addressing in RISC-V

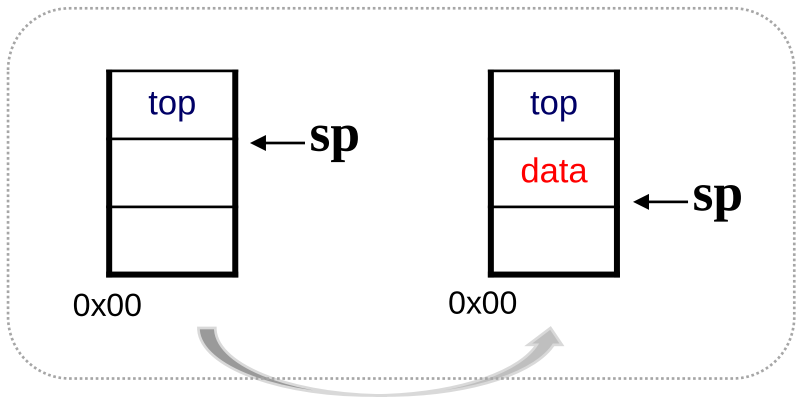
- RISC-V does not have PUSH or POP instructions.
- The stack pointer register (`sp`) is visible to the programmer:
  - It will be assumed that the stack pointer points to the last element on the stack

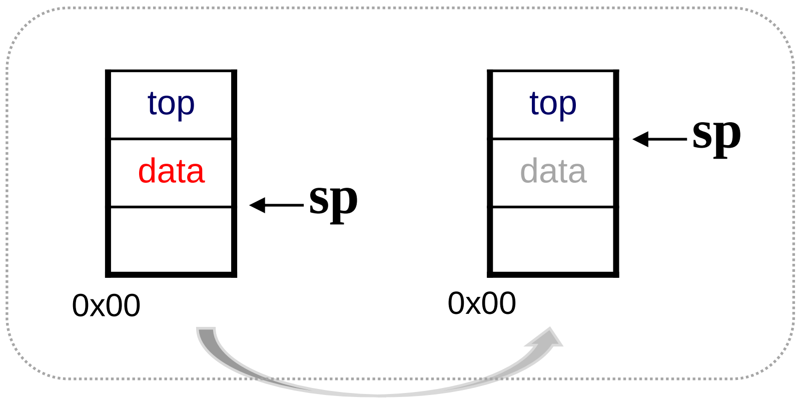
PUSH t0

```
addi  sp,  sp,  -4
sw    t0,  0(sp)
```

POP t0

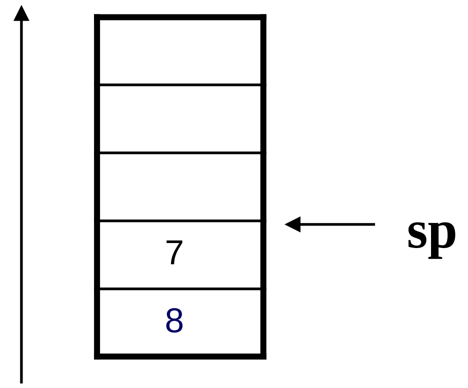
```
lw    t0,  0(sp)
addi  sp,  sp,  4
```





PUSH action in RISC-V_{32}

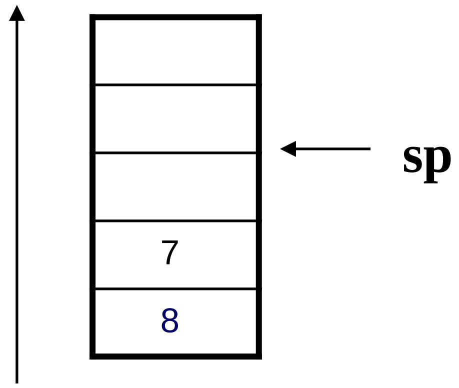
```
…
li   t2,  9
addi sp, sp, -4
sw   t2 0(sp)
…
```



- Initial state:
  - The stack pointer register (sp) points to the last element at the top of the stack.
  - The t2 register stores the value 9

PUSH action in RISC-V$_{32}$

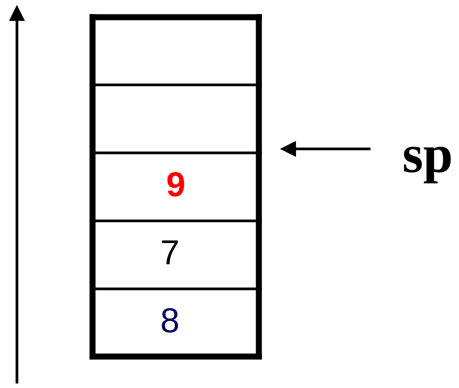
```
…
li    t2,  9
addi sp, sp, -4
sw    t2 0(sp)
…
```



- 4 is subtracted from the stack pointer register in order to insert a new word on the stack
  - addi sp, sp, -4

PUSH action in RISC-V$_{32}$

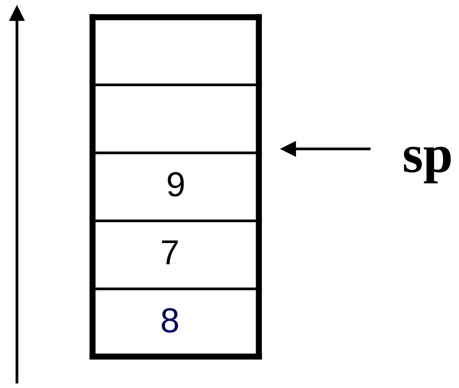
```
…
li   t2,  9
addi sp, sp, -4
sw   t2 0(sp)
…
```



- The contents of register t2 are inserted at the top of the stack:
  - sw t2 0(sp)

POP action in RISC-V$_{32}$

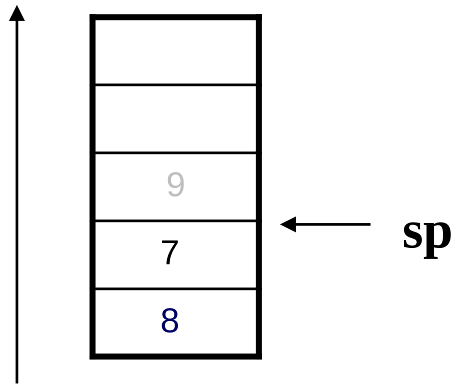
```
...
lw   t2 0(sp)
addi sp, sp, 4
...
```



- The data stored at the top of the stack is copied to t2 (9)
  - lw t2 0(sp)

POP action in RISC-V_{32}

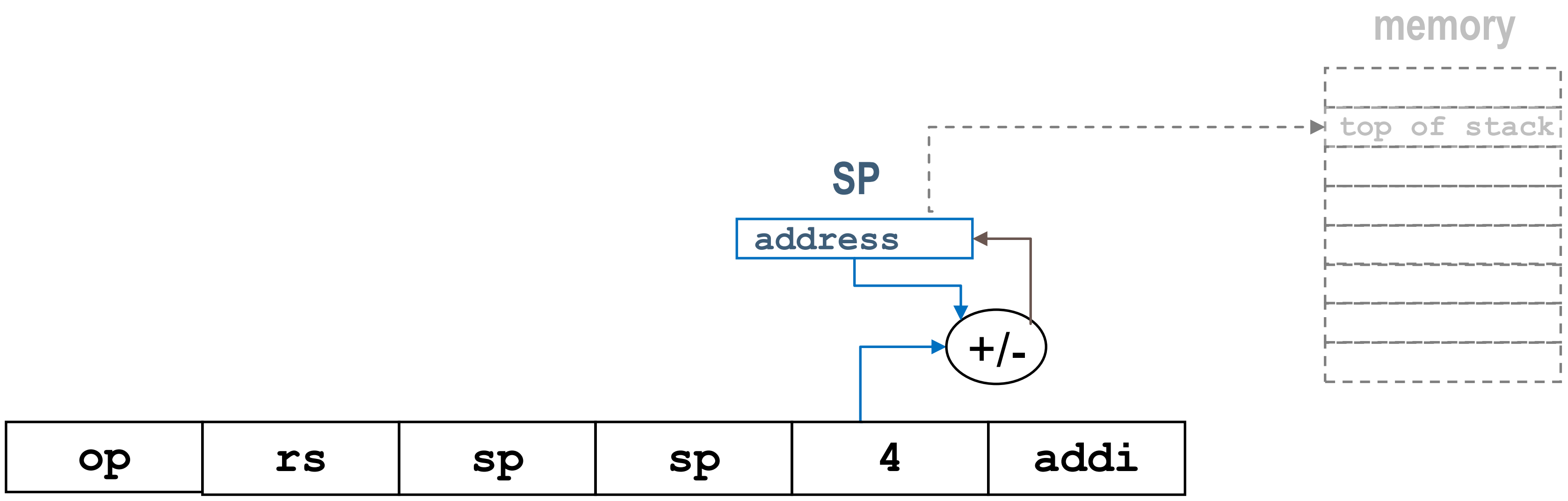
```
...
lw   t2 0(sp)
addi sp, sp, 4
...
```



- The sp register is updated to point to the new top of the stack.
  - addi sp, sp, 4
- The unstacked data (9) is still in memory but will be overwritten in future PUSH (or similar memory access) operation.

Stack addressing in RISC-V

- Example: push a0
  - addi sp sp -4 # SP = SP - 4
  - sw a0 0(sp) # memory[SP] = a0



Exercise

- Indicate the type of addressing used in the following instructions RISC-V:
  1. li t1 4
  2. lw t0 4(a0)
  3. bne x0 a0 label

Exercise (solution)

1. li t1 4
   - t1 -> direct to register
   - 4 -> immediate

1. lw t0 4(a0)
   - t0 -> direct to register
   - 4(a0) -> relative to base register

1. bne x0 a0 label
   - a0 -> direct to register
   - label -> relative to program counter

Examples of addressing modes

- la t0 label immediate
  - The second operand of the instruction is an address
  - BUT this address is not accessed, the address itself is the operand
- lw t0 label direct to memory (no ∃ in RV32)
  - The second operand of the instruction is an address
  - This address must be accessed in order to have the value to work with
- bne t0 t1 label relative to PC
  - The third operand of the instruction is offset relative to PC
  - label is encoded as a two's complement number representing the offset (as words) relative to the PC register

Example of instructions

- `la t0, 0x0F000002`
  - Direct to register + immediate.
    The `0x0F000002` value is loaded at t0

- `lbu t0, label(x0)`
  - Addresses direct to reg. + relative to base reg.
    The byte at memory address `label` is loaded at t0.

- `lb  t0, 0(t1)`
  - Addresses direct to reg. + relative to base reg.
    The byte in the memory location stored in t1+0 is loaded in t0.

Contents

- Basic concepts on assembly programming
- RISC-V 32 assembly language, memory model and data representation
- Instruction formats, addressing modes and instruction sets
- Procedure calls and stack convention

Instruction sets

- Queda definido por:
  - Instruction set
  - Instruction format
  - Registers
  - Addressing modes
  - Data types and formats

Instruction sets

- There are different ways for the classification of the instructions sets:
  - By complexity of the instruction set
    - CISC vs RISC
  - Execution modes
    - Stack
    - Register
    - Register-Memory, Memory-Register, …

CISC vs RISC

- *Complex Instruction Set Computer*
  - Many instructions
  - Complex instructions
    - More than one word
    - More complex control unit
    - Longer execution time
  - Irregular design

- About 20% of the instructions take up 80% of the total execution time of a program.
- 80% of the instructions are hardly ever used
- 80% of silicon underutilized, complex and costly

- *Reduced Instruction Set Computer*
  - Simple and orthogonal instructions:
    - Occupy one word
    - Instructions on registers
    - Use of the same addressing modes for all instructions (high degree of orthogonality)
  - More compact design:
    - Easier and faster control unit
    - Space left over for more registers and cache memory

Execution modes

- The execution modes indicates the number of operands and the type of operands that can be specified in an instruction.
  - 0 addresses → Stack.
    - PUSH 5; PUSH 7; ADD
  - 1 address → Accumulator register.
    - ADD R1 -> AC <- AC + R1
  - 2 addresses → Registers, Register-memory, Memory-memory.
    - ADD .R0, .R1 (R0 <- R0 + R1)
  - 3 addresses → Registers, Register-memory, memory-memory.
    - ADD .R0, .R1, .R2

ARCOS Group

uc3m | Universidad Carlos III de Madrid

L3: Fundamentals of assembler programming (3)
Computer Structure

Bachelor in Computer Science and Engineering
Bachelor in Applied Mathematics and Computing
Dual Bachelor in Computer Science and Engineering and Business Administration