ARCOS Group

uc3m | Universidad Carlos III de Madrid

Lesson 3 (III)
Fundamentals of assembler programming

Computer Structure

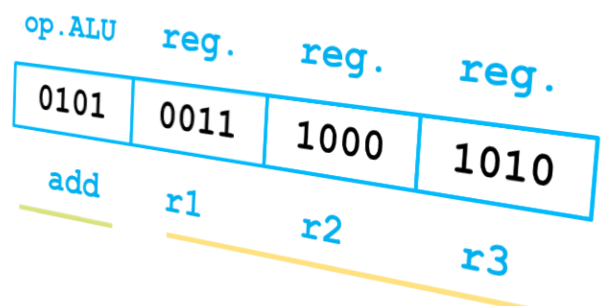
Bachelor in Computer Science and Engineering

Contents

- Basic concepts on assembly programming
- MIPS32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention

Information of an instruction

- The instructions:
  - Its size is adjusted to one word or multiples words
  - They are divided in fields:
    - Operation to do
    - Operands
      - There can be implicit operands



- The instruction format:
  - Form of representation of an instruction composed of fields of binary numbers:
    - The field size limits the number of values to encode

Information of an instruction

- Instruction set based on very few formats:
  - Each instruction belongs to one of this available formats

- Example: 3 instruction formats in MIPS

Type R
arithmetic

op.	rs	rt	rd	shamt	func.
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Type I
Immediate
transfer

op.	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

Type J
branches

op.	offset
6 bits	26 bits

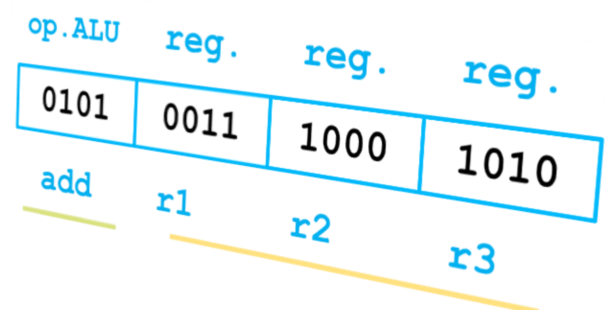
Instruction and pseudoinstruction in MIPS32

- An assembly instruction corresponds to a machine instruction
  - A machine instruction occupies 32 bits
  - Example: `addi $t1, $t1, 2`

- A pseudo-assembler instruction corresponds to one or several machine instructions.
  - Example 1:
    - The instruction: `move reg2, reg1`
    - It is equivalent to: `add reg2, $zero, reg1`
  - Example 2:
    - The instruction : `li $t1, 0x00800010`
    - It does not fit in 32 bits, but it can be used as a pseudoinstruction.
    - It is equivalent to:
      - `lui $t1, 0x0080`
      - `ori $t1, $t1, 0x0010`

Instruction fields

- Each field encodes:
  - Operation (Operation code)
    - Instruction and format used
  - Operands
    - Location of operands
    - Location for results
    - Location of next instruction *(in branches)*
      - Implicit: PC $\longleftarrow$ PC + ‘4’ (next instruction)
      - Explicit: j 0x01004 (PC modified)



ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

Locations of operands

1. In the instruction

   li $t0 0x123

2. In registers (processor)

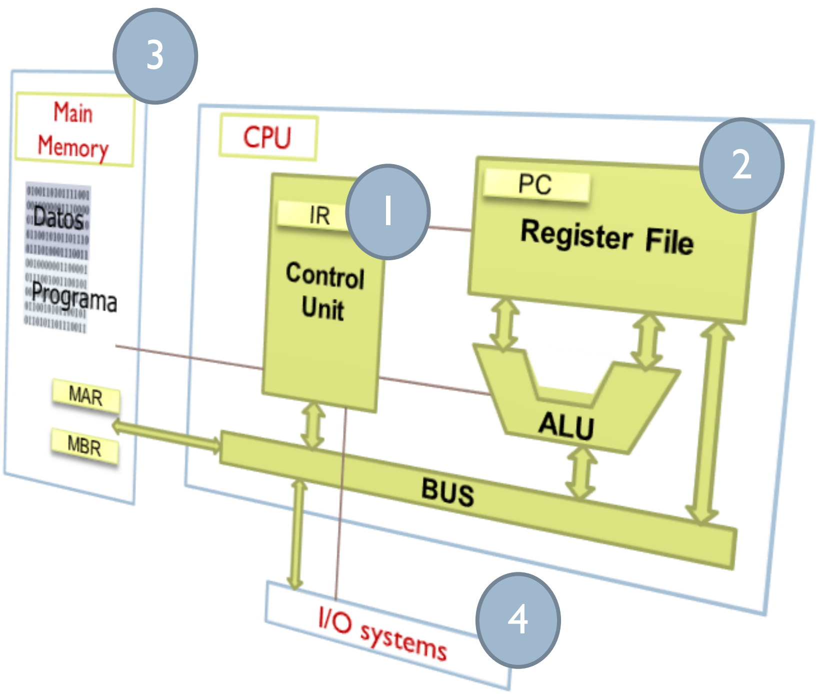
   li $t0 0x123

3. Main memory

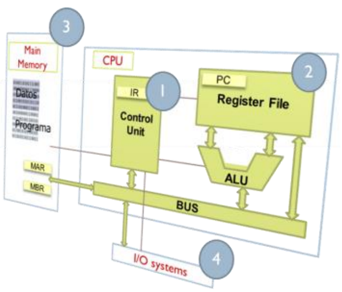
   lw $t0 address

4. Input/output modules

   in $t0 0xFEB



Kinds of ways to said the locations of operands: addressing modes



1. In the instruction

   li $t0 0x123

2. In registers (processor)

   li $t0 0x123

3. Main memory

   lw $t0 address

4. Input/output modules

   in $t0 0xFEB

- Number representing an address
- Symbolic label representing an address
- (register): represents the address stored in the register
- num(register): represents the address obtained by adding num with the address stored in the register
- label + num: represents the address obtained by adding label with num

ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

Contents

- Basic concepts on assembly programming
- MIPS32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention

Addressing modes

- Procedure that allows to localize the operands
- Types:
  - Implicit
  - Immediate
  - Direct
    - To register
    - To memory
  - Indirect
    - To register
    - To memory
  - Relative
    - To index register
    - Base register
    - To PC
    - To stack

Addressing modes

- Procedure that allows to localize the operands
- Types:
  - **Implicit**
  - **Immediate**
  - Direct
    - **To register**
    - **To memory**
  - Indirect
    - To register
    - To memory
  - Relative
    - To index register
    - Base register
    - To PC
    - To stack

Implicit addressing

- The operand is not encoded in one field of instruction.
- Example (mips32): beqz $a0 label1
  - If $a0 is zero, then branch to label
  - $a0 is an operand, $zero is the other one

op	rs		16 bits

- G/B
  - Good: It is fast because no extra memory access is required.
  - Good: Instructions shorter
  - Bad: limited options

Immediate addressing

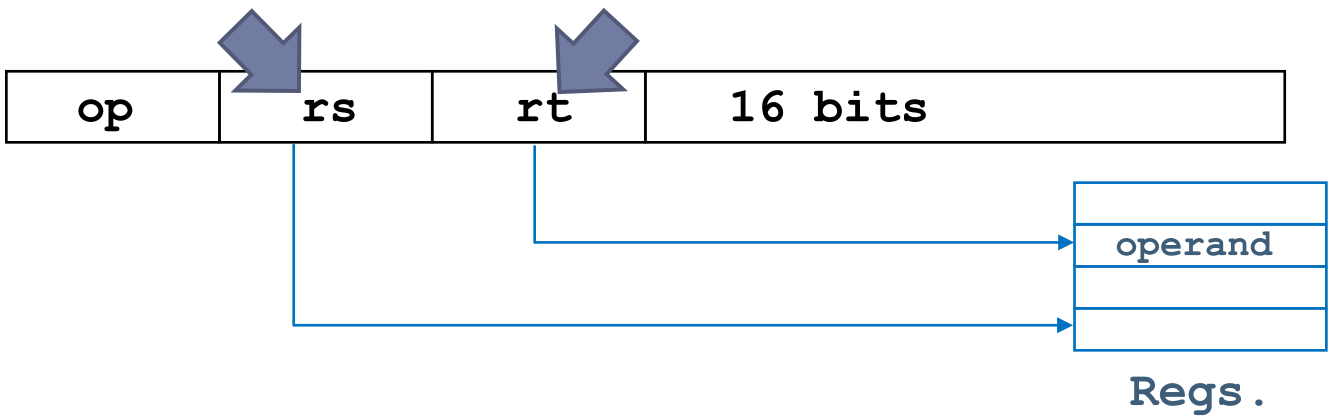
- Operand is in one field of the instruction of instruction.
- Example (mips32): `li $a0 0x25`
  - Load 0x25 in register $a0
  - 0x25 is the immediate value

op	rs		16 bits

- G/B
  - Good: It is fast because no extra memory access is required.
  - Bad: not always fits within a word
    - `li $t1, 0x00800010`
    - Equivalent to:
      - `lui $t1, 0x0080`
      - `ori $t1, $t1, 0x0010`

Register addressing

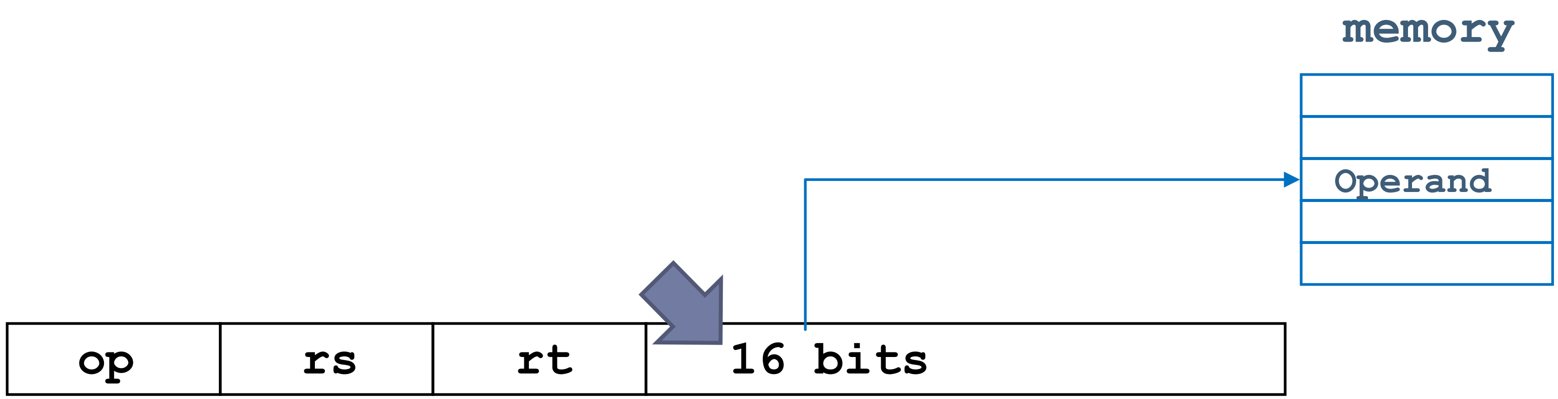
- Operand is in a register.
- Example (MIPS): move $a0 $a1
  - $a0 and $a1 are encoded in the instruction



- G/B:
  - Good: Faster access (no extra memory access required)
  - Good: Small address field
  - Bad: limited number of registers

Direct addressing

- Operand in memory. The instruction encodes the address.
- Example (MIPS): lw $t1, 0xFFF0
  - Load in $t1 the word stored in address 0xFFF0



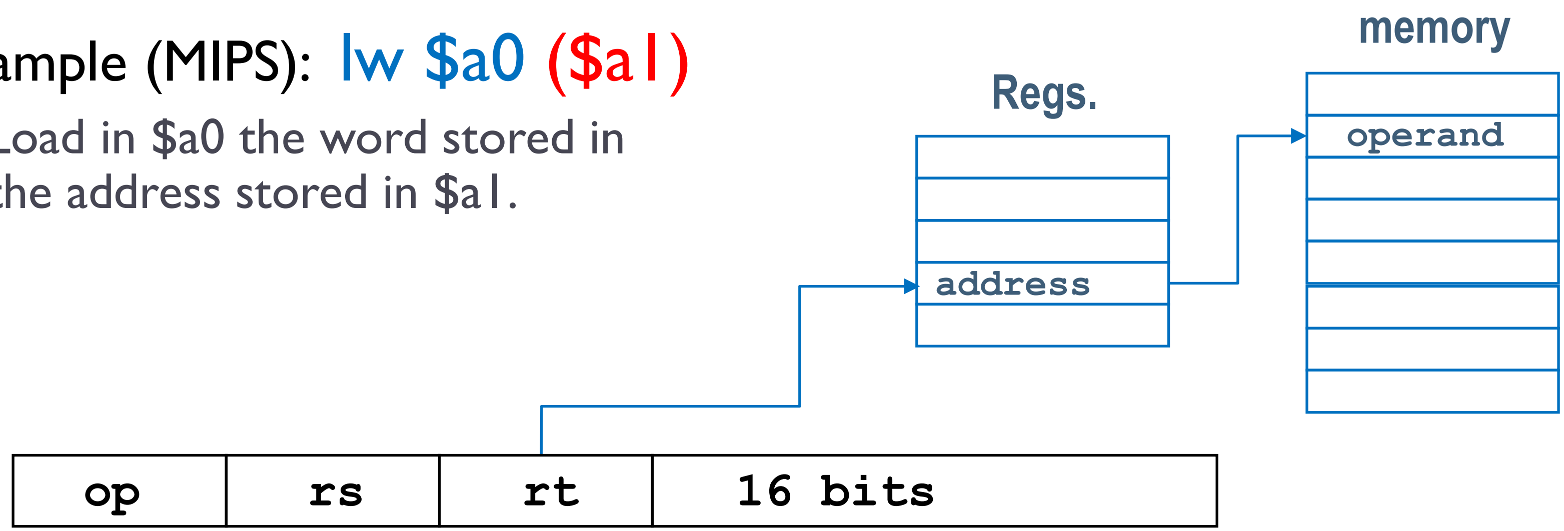
- G/B:
  - Good: capacity is bigger than register file
  - Bad: memory access time is larger than register access time
  - Bad: large fields => large instructions

Addressing modes

- Procedure that allows to localize the operands
- Types:
  - Implicit
  - Immediate
  - Direct
    - To register
    - To memory
  - Indirect
    - **To register**
    - **To memory**
  - Relative
    - To index register
    - Base register
    - To PC
    - To stack

Register indirect addressing

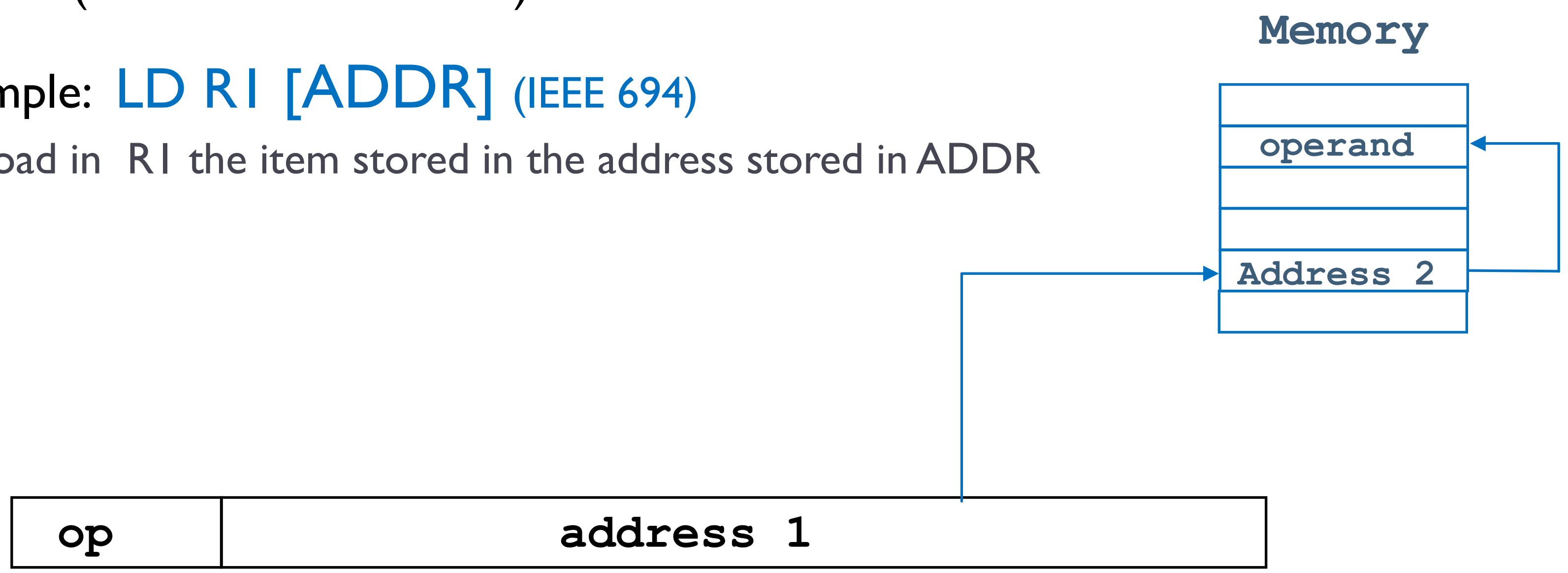
- The instruction has the register where the address is stored
- Example (MIPS): lw $a0 ($a1)
  - Load in $a0 the word stored in the address stored in $a1.



- G/B:
  - Good: Small fields
  - Good: address in a register can addressing the entire memory (32-bits registers)
  - Bad: extra memory access (slow execution)

Indirect addressing

- The instruction has the address where the operand address is stored (not available in MIPS)
- Example: LD R1 [ADDR] (IEEE 694)
  - Load in R1 the item stored in the address stored in ADDR



- G/B:
  - Bad: several memory accesses are required
  - Bad: slower instructions
  - Good: large address space, addressing can be multilevel (e.g.: [[[.R1]]])

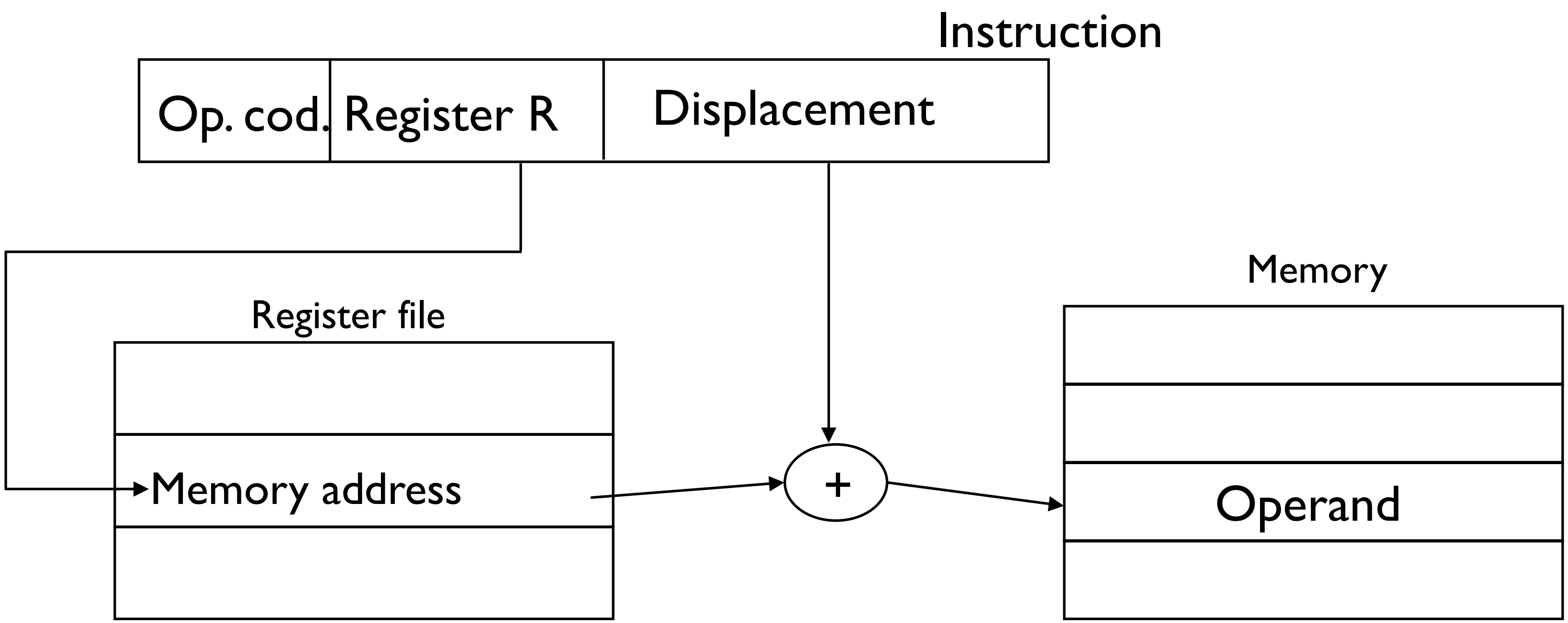
Addressing modes

- Procedure that allows to localize the operands
- Types:
  - 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)

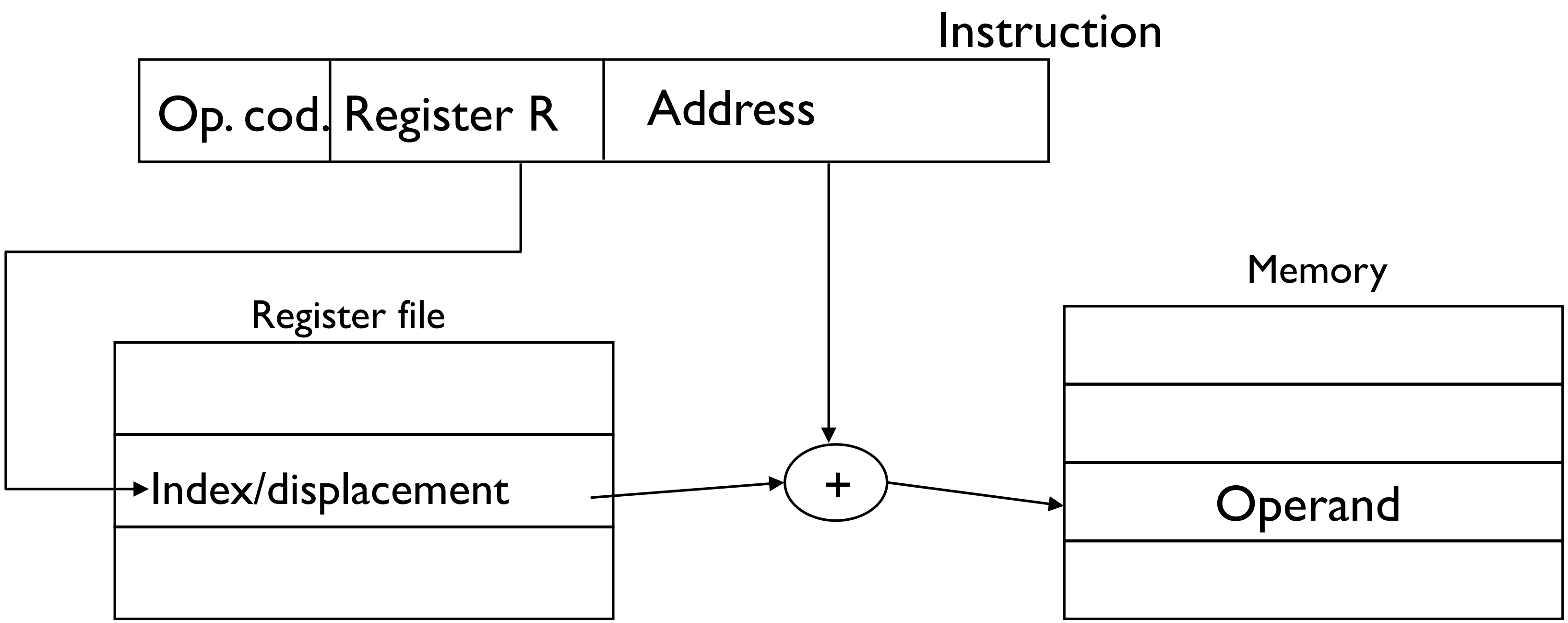
- Load in $a0 the word stored in address: $t1 + 12
- $t1 represents the base address



Index-register addressing

Example: lw $a0   address($t1)

- Load in $a0 the word stored in address: $t1 + address
- $t1 represents an index (offset from address)



ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

Used in arrays

```
int vec[5] ;
...

main ()
{
   v[4] = 8;



}
```

```
.data
.align 2          # next item align to 4
vec: .space 20 # 5 elem.*4 bytes

.text
.globl main
 main:  li $t2  8
        la $t1 vec
        sw $t2 16($t1)

...
```

Used in arrays

```c
int vec[5] ;
...

main ()
{
    v[4] = 8;
    v[3] = 8;


}
```

```
.data
.align 2          # next item align to 4
vec: .space 20 #  5 elem.*4 bytes

.text
.globl main
 main:   li $t2  8
         li $t1 16
         sw $t2 vec($t1)

         addi $t1 $t1 -4
         sw $t2 vec($t1)

...
```

Program counter in MIPS 32

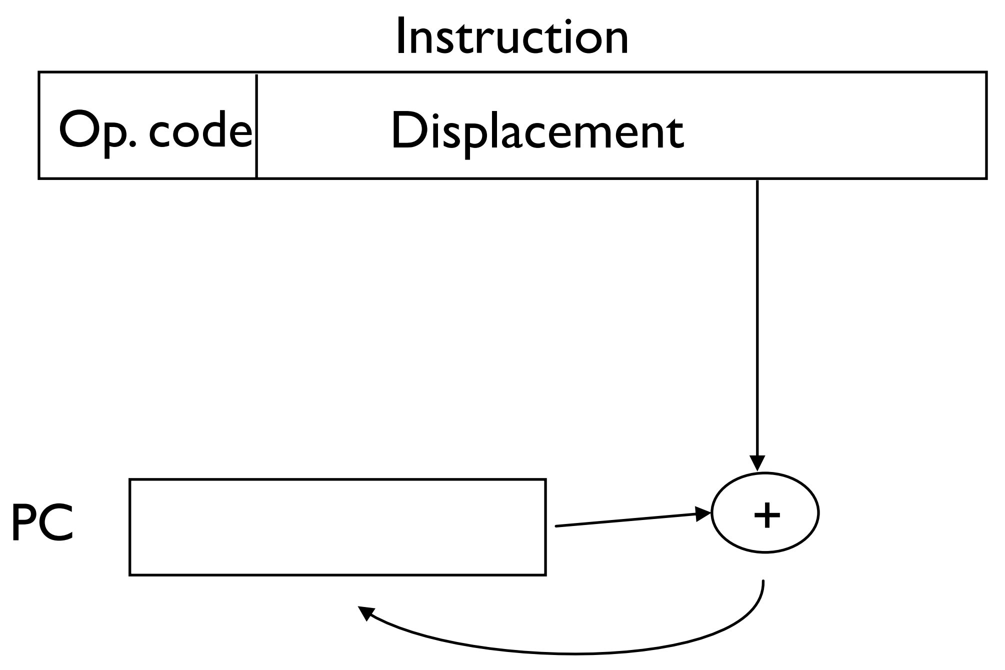
- Registers of 32 bits
- Program counter of 32 bits
- Instructions of 32 bits (one word)
- Program counter stores instruction address
- Next instruction is 4 bytes beyond
- Program counter is updated:
  - PC = PC + 4

address:	Instruction:
0x00400000	or $2,$0,$0
0x00400004	slt $8,$0,$5
0x00400008	beq $8,$0,3
0x0040000c	add $2,$2,$4
0x00400010	addi $5,$5,-1
0x00400014	j 0x100001

PC-relative addressing

Example: beqz $t1 label

- If $t1 == 0, then PC = PC + "label"
- Label encoded as displacement (address -> # instructions to jump)



PC-relative addressing in MIPS

- Instruction `beq $9, $0, label` is encoded as:

6	5	5	16
CO	rs	rt	immediate

- When $t0 == $1, what is the value for `end` label?

```
loop:    beq   $t0,$1, end
         add   $t8,$t4,$t4
         addi  $t0,$0,-1
         j     loop
end:     …
```

- Label must be encoded in an "immediate" field

PC-relative addressing in MIPS

- Instruction `beq $9, $0, label` is encoded as:

6	5	5	16
CO	rs	rt	immediate

- When $t0 == $1, `end` is 3 ("number of instructions to skip")

```
loop:      beq   $t0,$1, end
           ...
end:       ...
```

- Reasons:
  - When an instruction is going to be executed, PC has the addresss of next instruction in memory
  - When the condition in satisfied: PC = PC + (`label`* 4)

Use in loops

- end represents the address where the instruction `move` is stored

```
        li   $t0 8
        li   $t1 4
        li   $t2 1
        li   $t4 0
while:  bge  $t4 $t1 end
        mul  $t2 $t2 $t0
        addi $t4 $t4 1
        b    while
end:    move $t2  $t4
```

Use in loops

```
        li    $t0 8
        li    $t1 4
        li    $t2 1
        li    $t4 0
while:  bge   $t4 $t1 end
        mul   $t2 $t2 $t0
        addi  $t4 $t4 1
        b     while
end:    move  $t2  $t4
```

Address	Content
0x0000100	li $t0 8
0x0000104	li $t1 4
0x0000108	li $t2 1
0x000010C	li $t4 0
0x0000110	bge $t4 $t1 end
0x0000114	mul $t2 $t2 $t0
0x0000118	addi $t4 $t4 1
0x000011C	b while
0x0000120	move $t2 $t4

Use in loops

```
        li   $t0 8
        li   $t1 4
        li   $t2 1
        li   $t4 0
while:  bge  $t4 $t1 end
        mul  $t2 $t2 $t0
        addi $t4 $t4 1
        b    while
end:    move $t2  $t4
```

Address	Content
0x0000100	li $t0 8
0x0000104	li $t1 4
0x0000108	li $t2 1
0x000010C	li $t4 0
0x0000110	bge $t4 $t1 end
0x0000114	mul $t2 $t2 $t0
0x0000118	addi $t4 $t4 1
0x000011C	b while
0x0000120	move $t2 $t4

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

Use in loops

```
            li    $t0  8
            li    $t1  4
            li    $t2  1
            li    $t4  0
while:      bge   $t4  $t1  end
            mul   $t2  $t2  $t0
            addi  $t4  $t4  1
            b     while
end:        move  $t2   $t4
```

Address	Content
0x0000100	li $t0 8
0x0000104	li $t1 4
0x0000108	li $t2 1
0x000010C	li $t4 0
0x0000110	bge $t4 $t1 **3**
0x0000114	mul $t2 $t2 $t0
0x0000118	addi $t4 $t4 1
0x000011C	b **-4**
0x0000120	move $t2 $t4

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

Differences between b and j instructions

- Instruction j address

op.	address
6 bits	26 bits

Branch address => PC = address

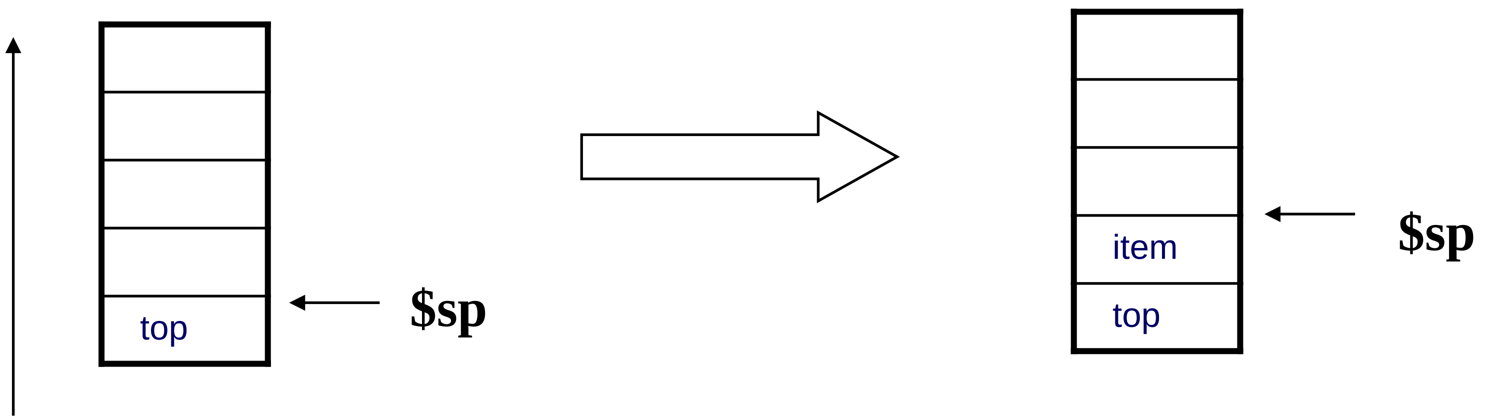
- Instruction b displacement

op.			displacement
6 bits	5 bits	5 bits	16 bits

Branch address => PC = PC + displacement

Stack addressing

PUSH Reg Push the content of a register (item)

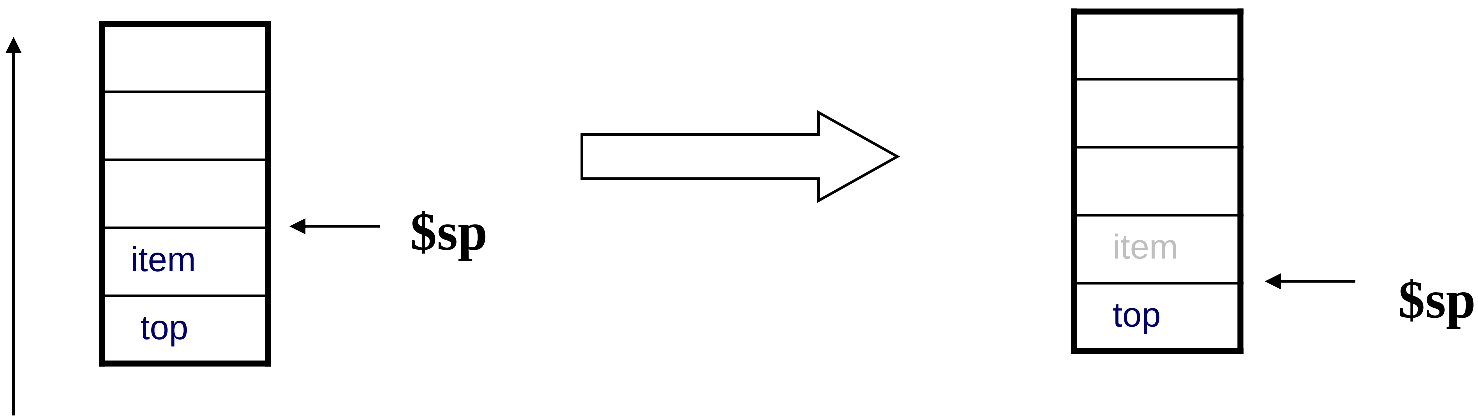


Stack grows to lower memory addresses

Stack addressing

POP Reg

Pop the top of the stack (item)
Copy the element in a Reg



Stack grows to lower memory addresses

Stack addressing

- MIPS does not have PUSH or POP instructions
  - The stack pointer ($sp) is visible
  - We assume that $sp points to the last element in the stack
    - Push -> allocate new word + store a value
    - Pop -> retrieve a value + deallocate a word

PUSH $t0

```
sub $sp, $sp, 4
sw $t0, ($sp)
```

POP $t0

```
lw $t0, ($sp)
add $sp, $sp, 4
```

Examples of addressing types

- la $t0 label immediate
  - The second operand is an address
  - But this address is not accessed, the address is the operand
- lw $t0 label direct
  - The second operand is an address
  - A memory access is required to obtain the final operand
- bne $t0 $t1 label PC-relative
  - Last operand represents a displacement
  - Label is encoded as a number that represents a displacement relative to PC

Addressing modes in MIPS

Immediate	value
Register	$r
Direct	dir
Register indirect	($r)
Register relative	displacement($r)
PC-relative	beq label
Stack-relative	displacement($sp)

Contents

- Basic concepts on assembly programming
- MIPS32 assembly language, memory model and data representation
- Instruction formats and addressing modes
- Procedure calls and stack convention

Instruction format

- A machine instruction is divided in **fields**
- A machine instruction includes:
  - Operation code
  - Operand addresses
  - Result address
  - Address of the next instruction
  - How the operands are represented
- Example in MIPS:

Type R arithmetic	op.	rs	rt	rd	shamt	func.

Instruction format

- The format specifies, for each field in the instruction:
  - The meaning of each field
  - The number of bits of each field
  - How is encoded each field
    - binary, 1's complement, displacement w.r.t. …
- Usually:
  - Very few formats in order to simplify the control unit design.
  - Fields of the same type have the same length.
  - Operation code is the first field (of first word)

Félix García Carballeira, Alejandro Calderón Mateos

Format length

- The format length is number of bits used to encode the instruction

- Two types:
  - 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 length of an instruction? → Op. code
    - Examples:
      - IA32 (Intel processors): variable number of bytes

Instruction format

- The size of an instruction is usually one word, but there can be instructions of several words
  - In MIPS the size of all instructions is one word (32 bits)
- Operation code:
  - With n bits we can encode 2^n instructions
  - We can use extensions fields to encode more instructions
  - Example: in MIPS, arithmetic instructions have the operation code 0. The operation is encoded in func. field

Type R arithmetic	op.	rs	rt	rd	shamt	func.

Example: MIPS instruction formats

```
add $t0, $t0, $t1
```

6	5	5	5	5	6
CO	rs	rt	rd	sa	func

6	26
CO	immediate

```
addi $t0, $t0, 1
```

6	5	5	16
CO	rs	rt	immediate

Example of MIPS formats

- MIPS Instruction:
  - `add    $8,$9,$10`
  - Format :

6	5	5	5	5	6
CO	rs	rt	rd	sa	func

Decimal representation:

0	9	10	8	0	32

Binary representation

000000	01001	01010	01000	00000	100000

Example of MIPS formats

- MIPS Instruction:
  - `addi    $21,$22,-50`
  - Format:

CO (6)	rs (5)	rt (5)	immediate (16)

Decimal representation:

8	22	21	-50

Binary representation

001000	10110	10101	1111111111001110

How to use the addi instruction with 32 bits values?

- What happens when this instruction is used in a program?
  - `addi   $t0,$t0, 0xABABCDCD`
  - The immediate value has 32 bits.
    This instruction cannot be encoded in one word (32 bits)

Félix García Carballeira, Alejandro Calderón Mateos

How to use the addi instruction with 32 bits values?

- What happens when this instruction is used in a program?
  - `addi   $t0,$t0, 0xABABCDCD`
  - The immediate value has 32 bits.
    This instruction cannot be encoded in one word (32 bits)
- Solution:
  - This instruction is translated to:

```
lui     $at, 0xABAB
ori     $at, $at, 0xCDCD
add     $t0, $t0, $at
```

  - The `$at` is reserved to the assembler

Questions

- How does the unit control know the format of an instruction?

- How does the unit control know the number of operands of an instruction?

- How does the unit control know the format of each operand?

Operation code

- Fixed size:
  - n bits ➔ 2^n operation codes
  - m operation codes ➔ $\log_2 m$ bits.
- Extension fields
  - MIPS (arithmetic-logic instructions)
  - `Op` = 0; The instruction is encoded in `func`

Type R
arithmetic

op.	rs	rt	rd	shamt	func.

- Variable sizes:
  - More frequent instructions= shorter sizes

Example

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

- Define the format of this instruction: ADDx R1 R2 R3, where R1, R2 and R3 are registers.

ARCOS @ UC3M
Félix García Carballeira, Alejandro Calderón Mateos

Solution

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

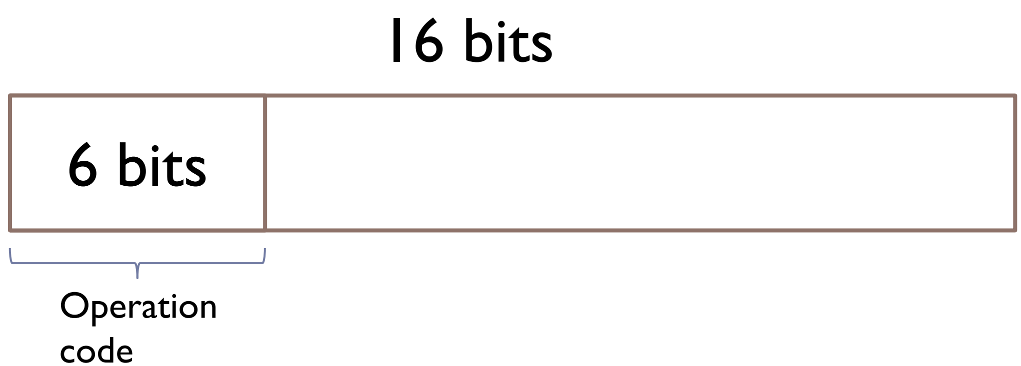
- Word of 16 bits

16 bits

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



Solution

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

- To encode 8 registers, 3 bits are required

16 bits

6 bits	3 bits	3 bits	3 bits	
Operation code	Operands (3 registers)			

Solution

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

- Spare one bit (16-6-3-3-3 = 1)

16 bits

6 bits	3 bits	3 bits	3 bits	1 bit
Operation code	Operands (3 registers)			

Instruction sets

- There are different ways for the classification of the instructions sets.

- For example:
  - By complexity of the instruction set
    - CISC vs RISC
  - By available execution modes

CISC-RISC

- CISC: *Complex Instruction Set Architecture* (http://es.wikipedia.org/wiki/RISC)
  - Many instructions
  - Complex instructions
  - Irregular design
- RISC: *Reduced Instruction Set Code* (*http://es.wikipedia.org/wiki/CISC*)
  - Simple instructions
  - Very few instructions
  - Instructions with fixed size
  - Many registers
  - Most of instructions use registers
  - Parameters are passed using registers
  - Pipelined architectures

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

Exercise

- A 16-bit computer, with byte memory addressing has 60 machine instructions and a register bank with 8 registers. What is the format for the instruction ADDV R1, R2, M, where R1 y R2 are registers and M represents a memory address?

Exercise

- A 32-bit computer with byte memory addressing has 64 machine instructions and 128 registers. Consider the instruction SWAPM addr1, addr2, that swaps the content of two memory addresses addr1 and addr2.
  - What is the memory address space of this computer?
  - Define the format for this instruction.
  - Write a program fragment in MIPS32 equivalent to the above instruction
  - If the instruction has to be encoded in one word, what is the addresses range assuming that memory addresses are represented in binary?