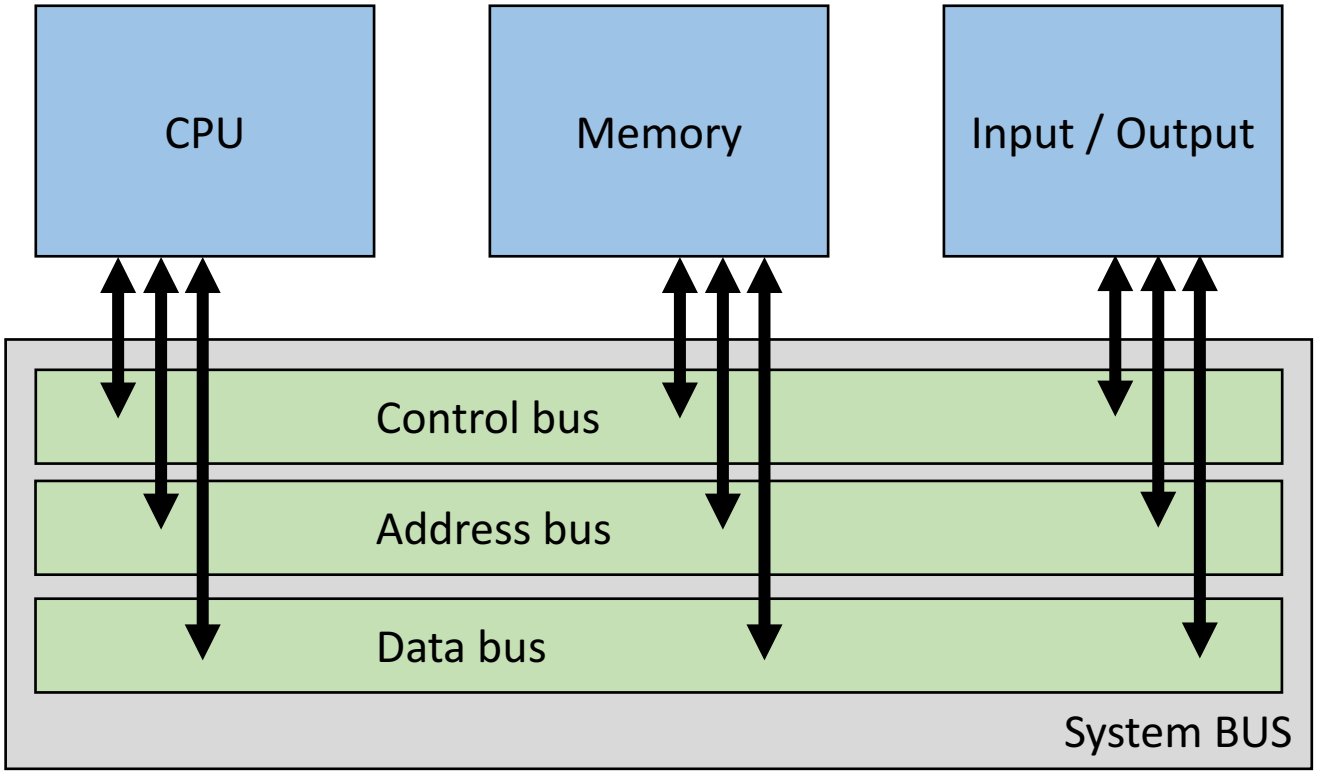
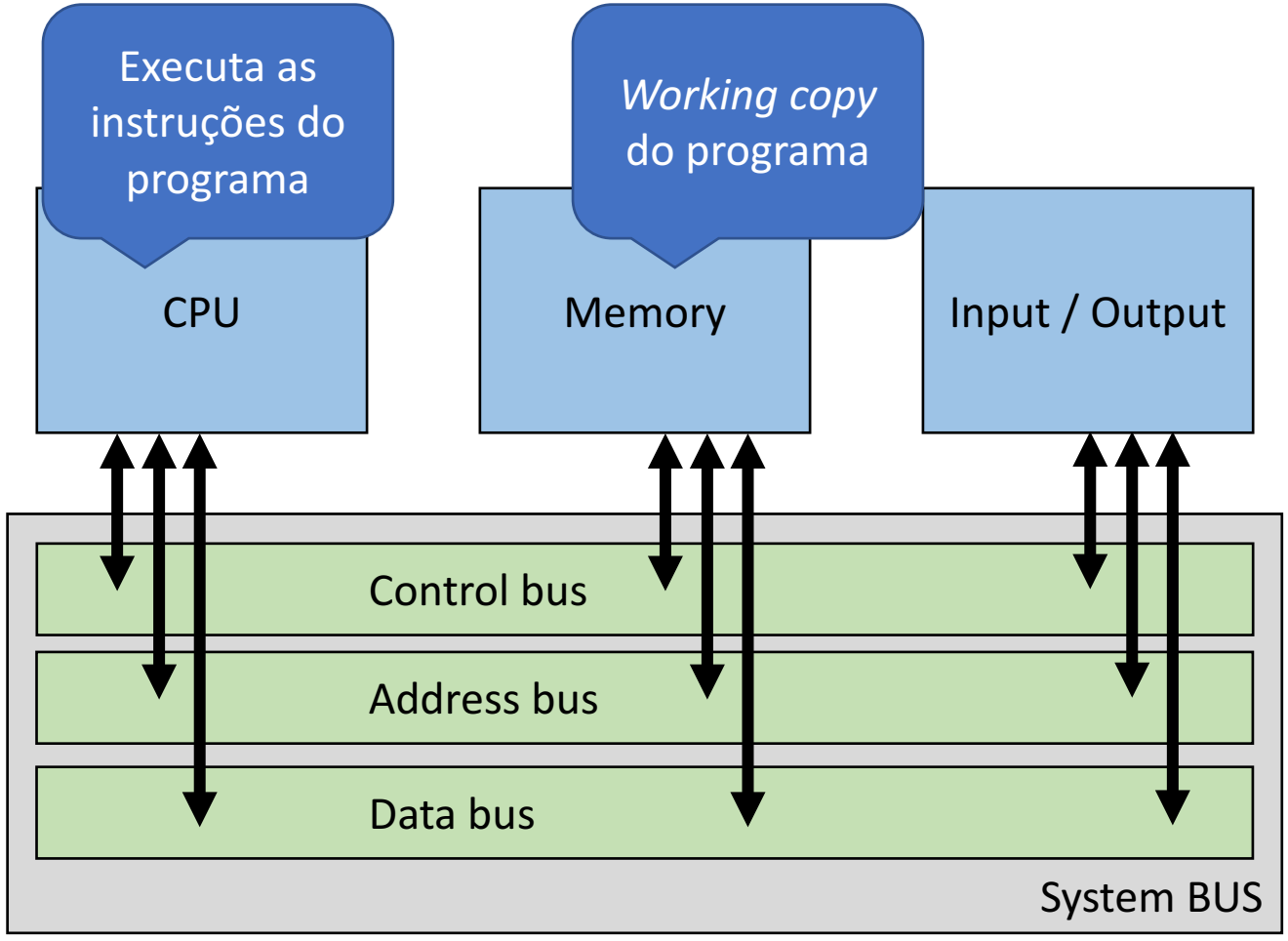
Arquitetura de Computadores

MIPS

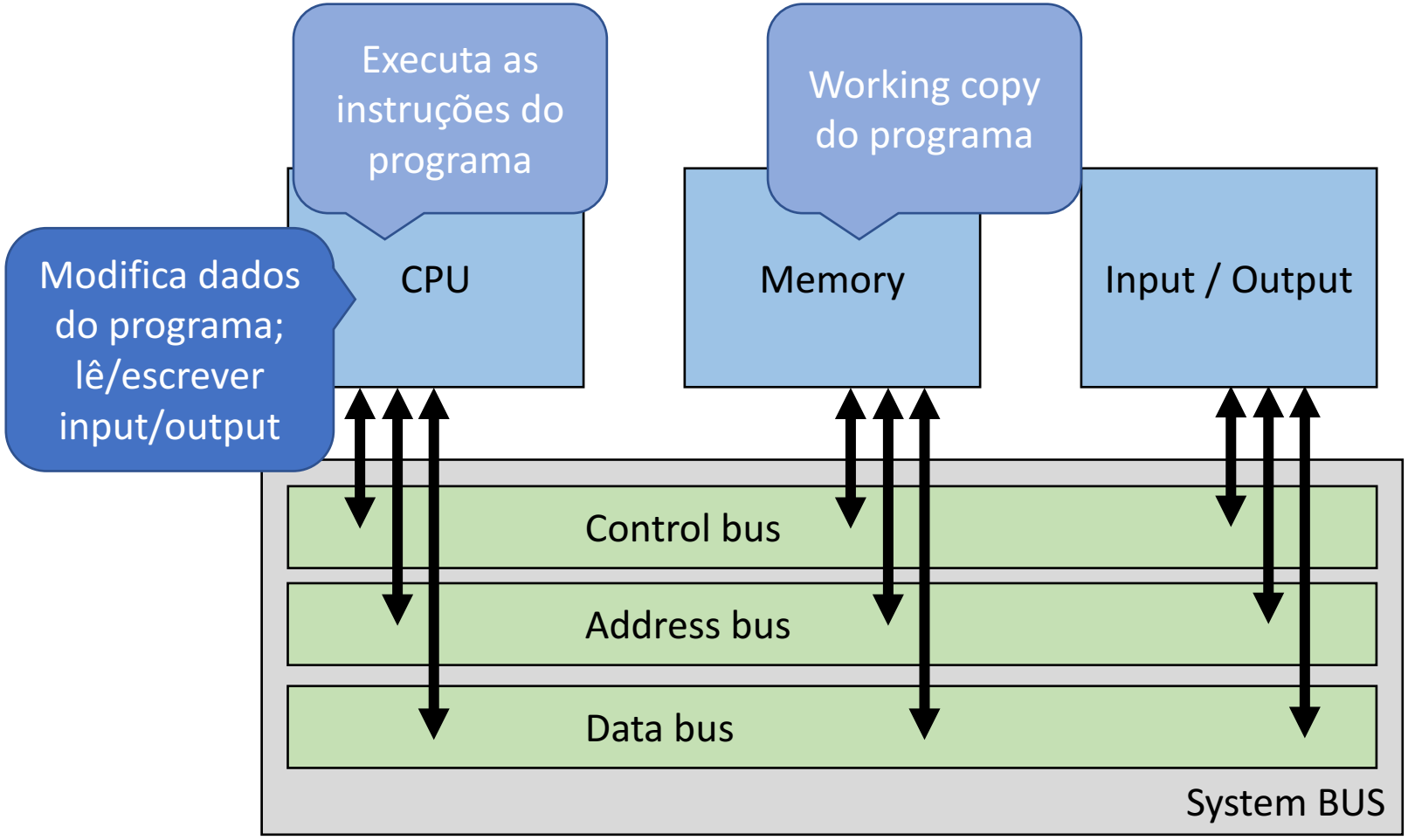
Arquitetura de von Neumann



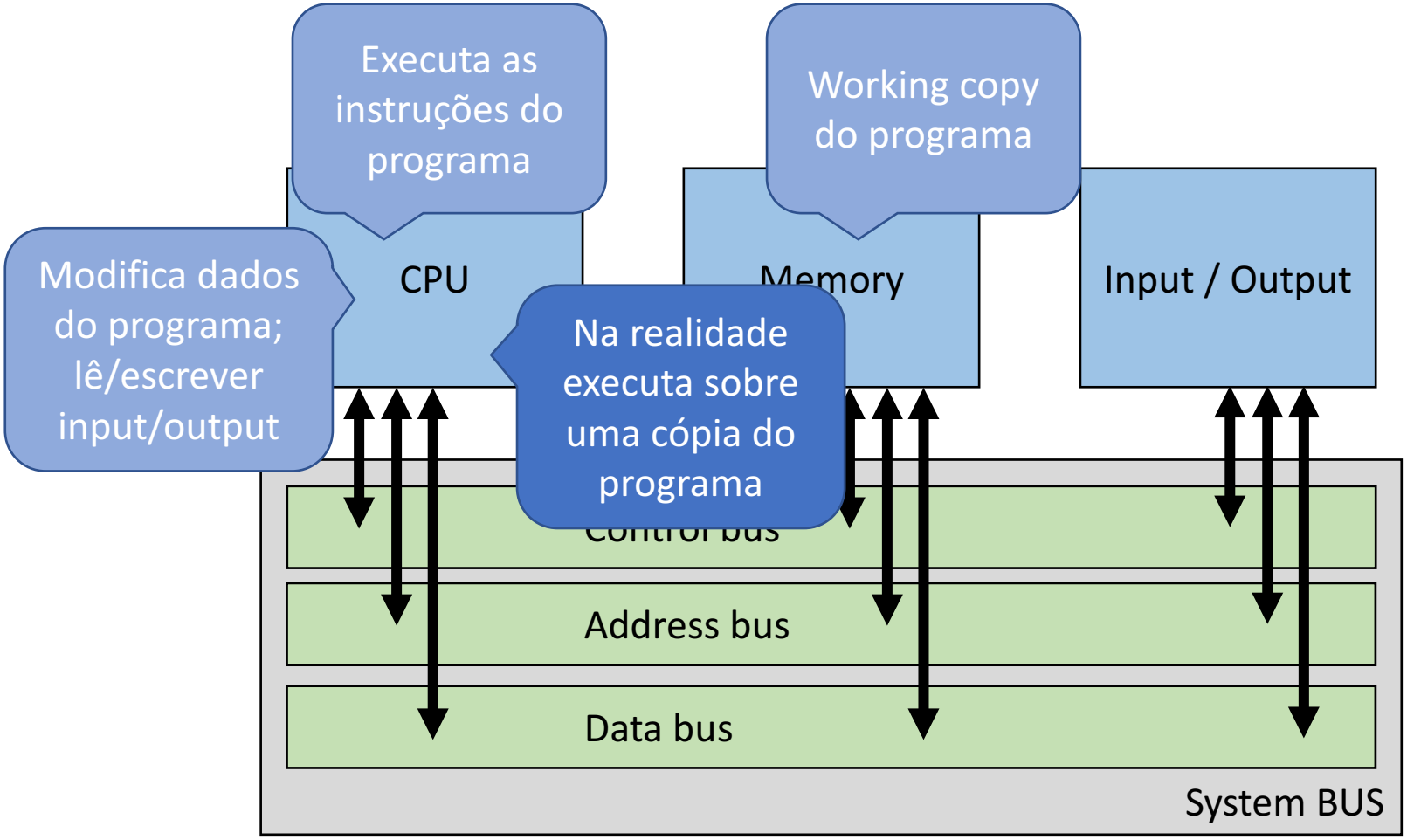
Arquitetura de von Neumann



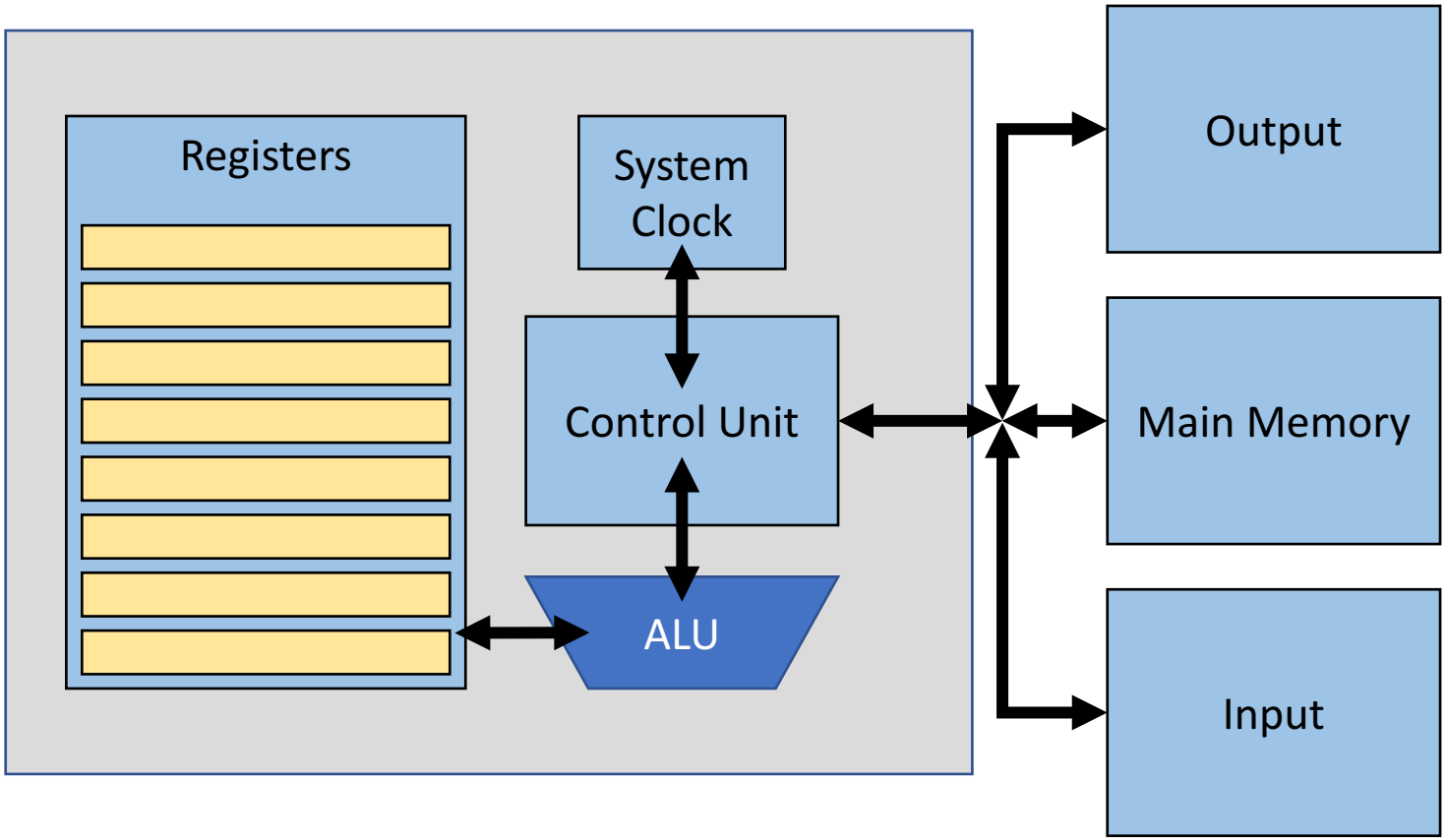
Arquitetura de von Neumann



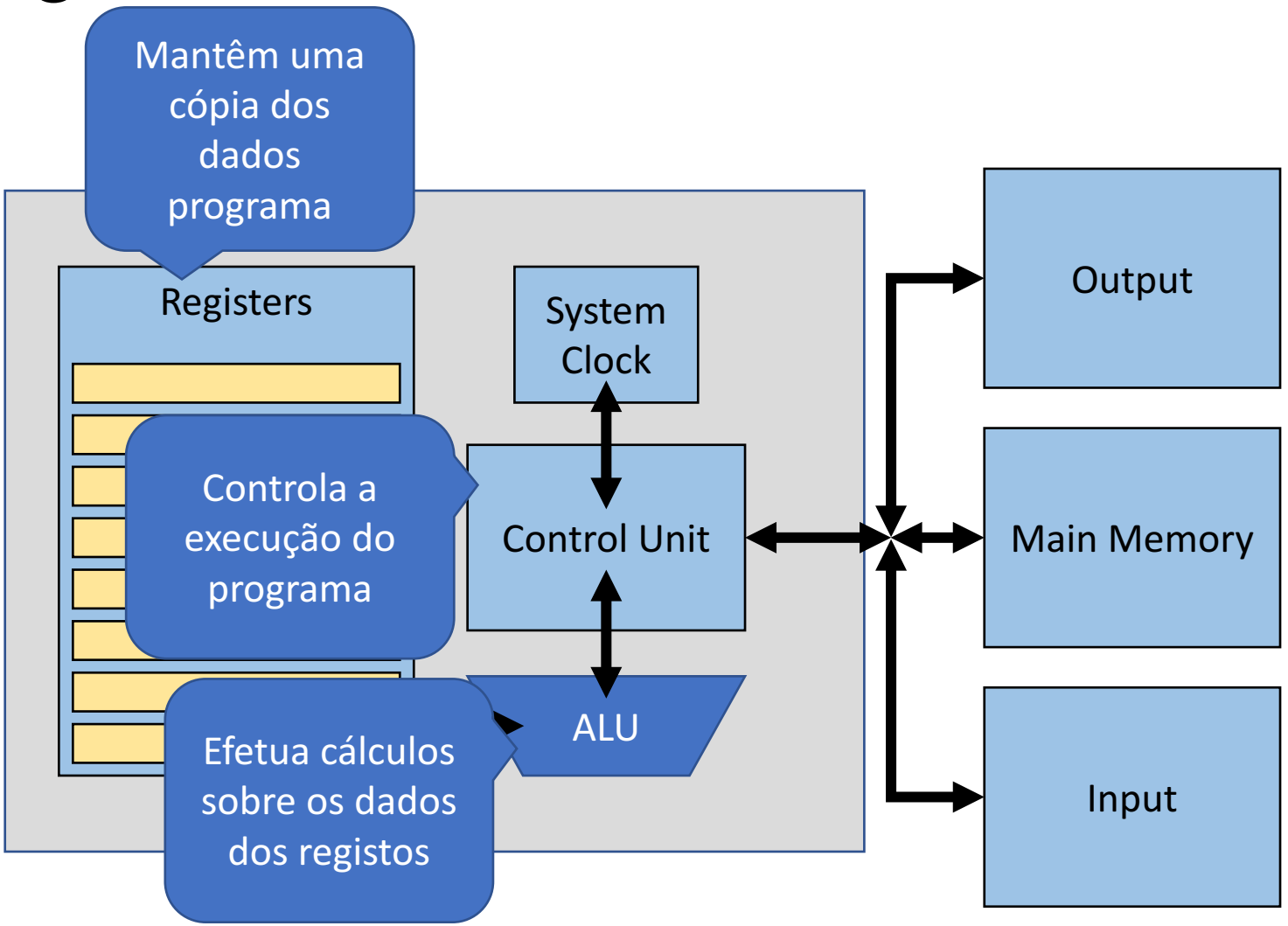
Arquitetura de von Neumann



CPU


Registers
System Clock
Control Unit
ALU
Output
Main Memory
Input

CPU


Mantêm uma cópia dos dados programa
Registers
System Clock
Controla a execução do programa
Control Unit
ALU
Efetua cálculos sobre os dados dos registos
Output
Main Memory
Input

ISA – Instruction Set Architecture

- Interface entre o hardware e o software
  - Linguagem máquina
  - Define as regras para codificar e interpretar instruções máquina

- ISA define
  - Instruções
  - Regras de endereçamento
  - Tipos de dados
  - Registos
  - Arquitetura da memória
  - *Interrupt & exception handling*
  - *External I/O*

MIPS

- Processador de 32 bits
  - 32 registos de 32 bits
  - Versões mais recentes de 64 bits
- Arquitetura RISC
  - *Reduced instruction set computer*
- Cache
  - 32 kb dados e 63 kb de instruções

MIPS – Registos

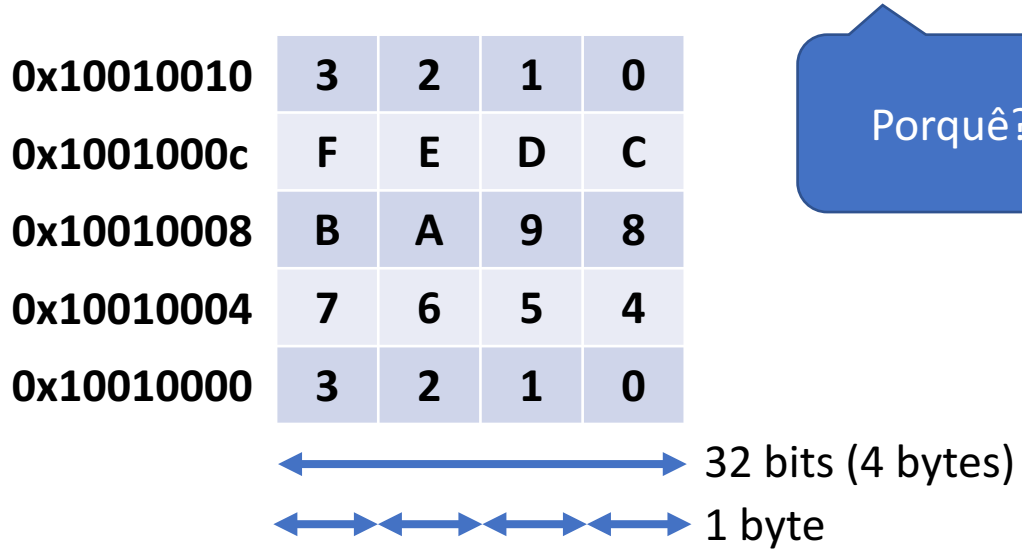
Nome	Número	Utilização
$zero	$0	Constante 0
$at	$1	Reservado ao assembler
$v0 .. $v1	$2 .. $3	Resultado de uma função/procedimento
$a0 .. $a3	$4 .. $7	Argumentos 1, 2, 3 e 4
$t0 .. $t7	$8 .. $15	Temporários (não preservados entre chamadas)
$s0 .. $s7	$16 .. $23	Persistentes (preservados entre chamadas)
$t8 .. $t9	$24 .. $25	Temporários (não preservados entre chamadas)
$k0 .. $k1	$26 .. $27	Reservados ao kernel do S.O.
$gp	$28	Ponteiro para a área global (dados estáticos)
$sp	$29	Ponteiro da stack
$fp	$30	Ponteiro da frame
$ra	$31	Endereço de retorno (usado pela chamada de uma função)

MIPS – Tipos de dados

- .word – 4 bytes (32 bits)
- .half – (half-word) 2 bytes (16 bits)
- .byte – 1 byte (8 bits)
- .char – 1 byte (8 bits)

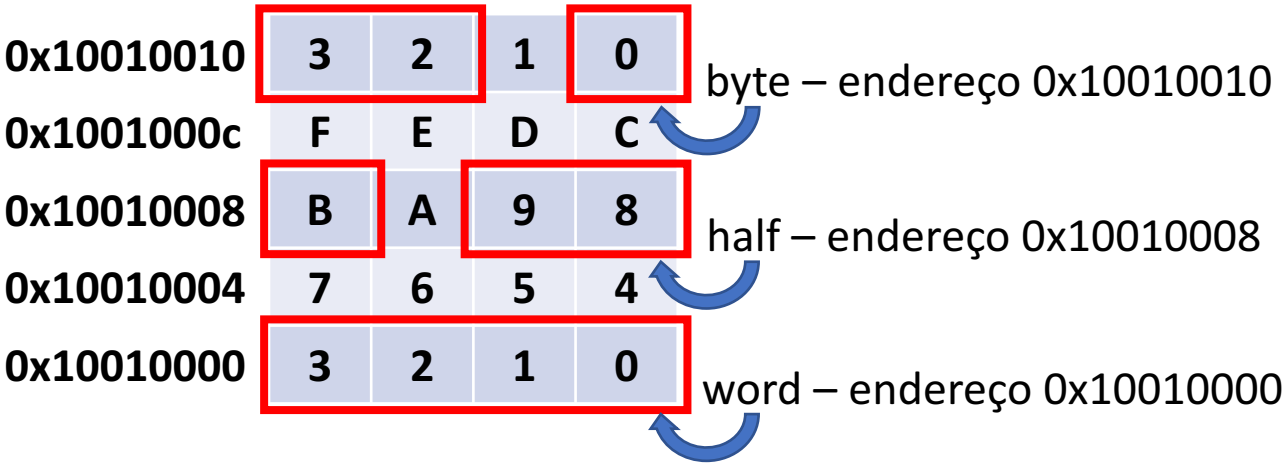
MIPS – Endereçamento

- Endereços de 32 bits (4 bytes)
- *Little endian*
  - Bit menos significativo está no endereço do byte menor
- Endereçamento ao byte
  - Tamanho máximo de um programa: $(2^{32} - 1)$ bytes

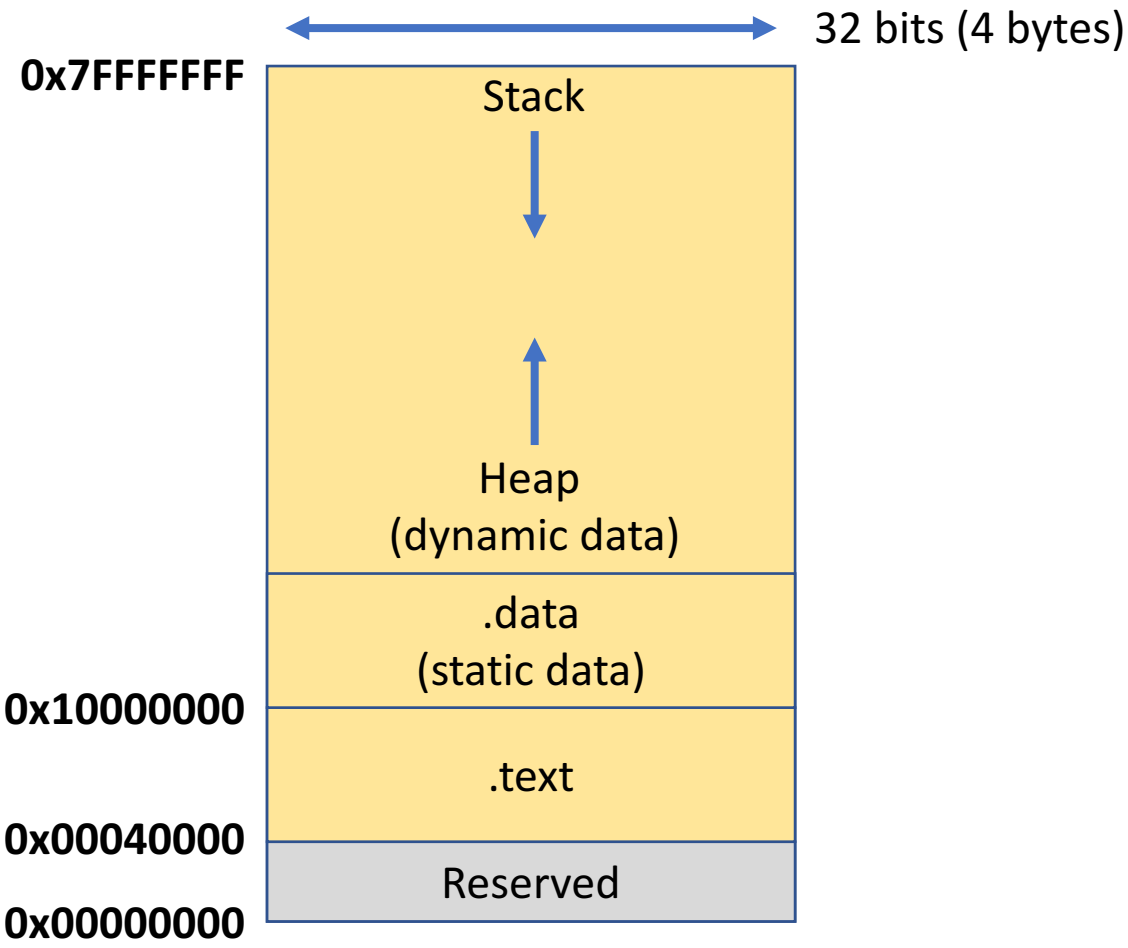


MIPS – Regras de endereçamento

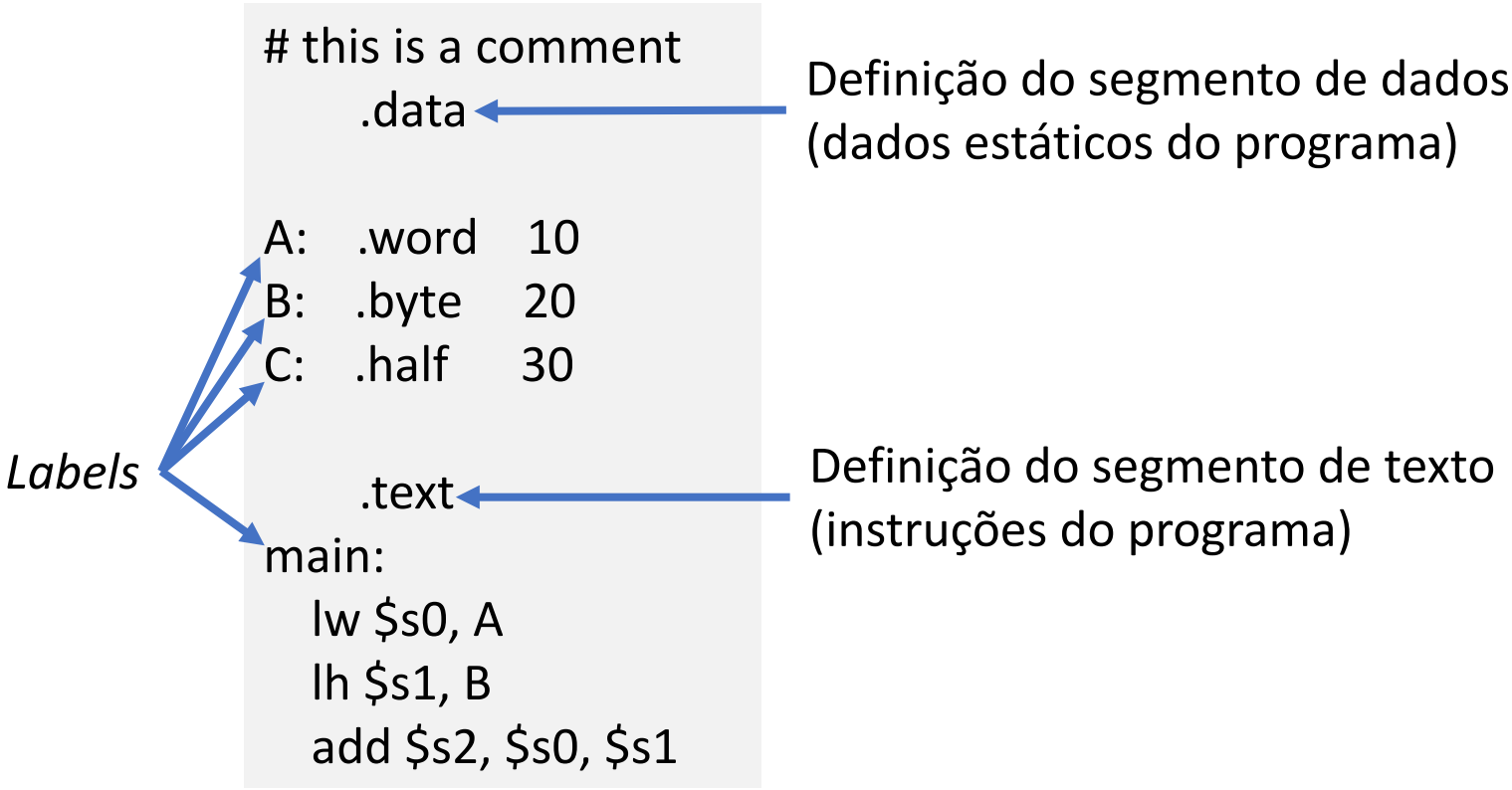
- word – tem que ocupar uma linha de memória
  - Tem que ter endereço múltiplo de 4
- half-word – ocupa os 2 primeiros/últimos bytes
  - Tem que ter endereço par
- byte – ocupa o 1º byte livre



MIPS – Memória



Program structure

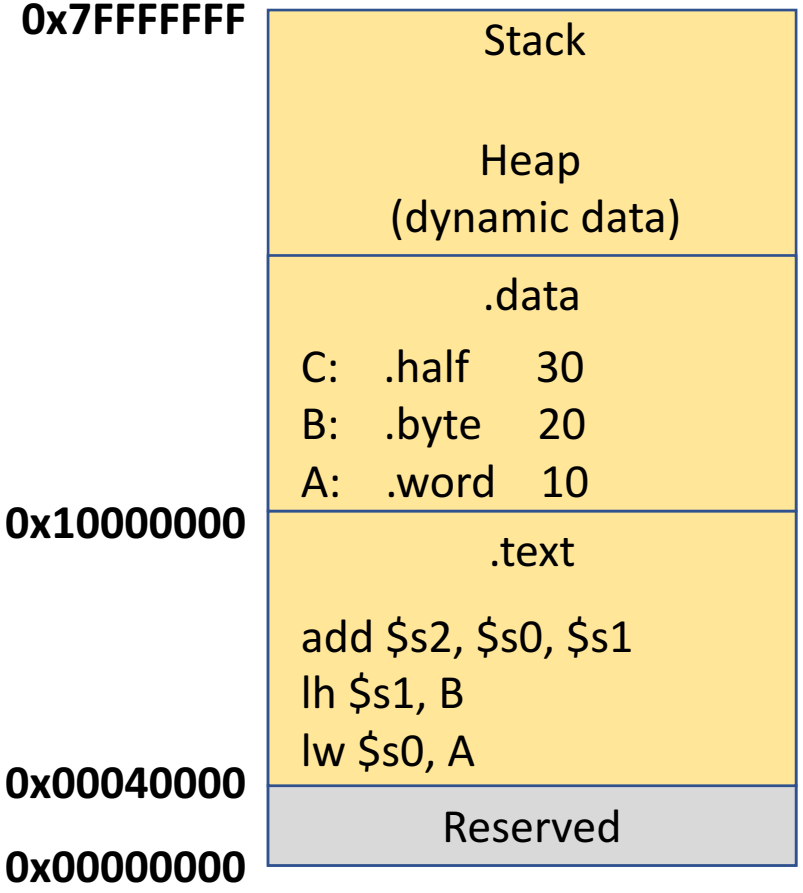


Program structure

```
# this is a comment
        .data

A:    .word    10
B:    .byte    20
C:    .half    30

        .text
main:
    lw $s0, A
    lh $s1, B
    add $s2, $s0, $s1
```



Instruções

- Load (l?) / Store (s?)

  lw registo_destino, endereço_memória
  lh registo_destino, endereço_memória
  lb registo_destino, endereço_memória

  sw registo_origem, endereço_memória
  sh registo_origem, endereço_memória
  sb registo_origem, endereço_memória

Instruções

- Aritmética

  add $s0, $s1, $s2 ## $s0 = $s1 + $s2

  sub $s0, $s1, $s2 ## $s0 = $s1 - $s2

  …

- Controlo (branches e jumps)

  bgt $s0, $s1, target ## branch to *target* if $s0 > $s1

  blt $s0, $s1, target ## branch to *target* if $s0 < $s1

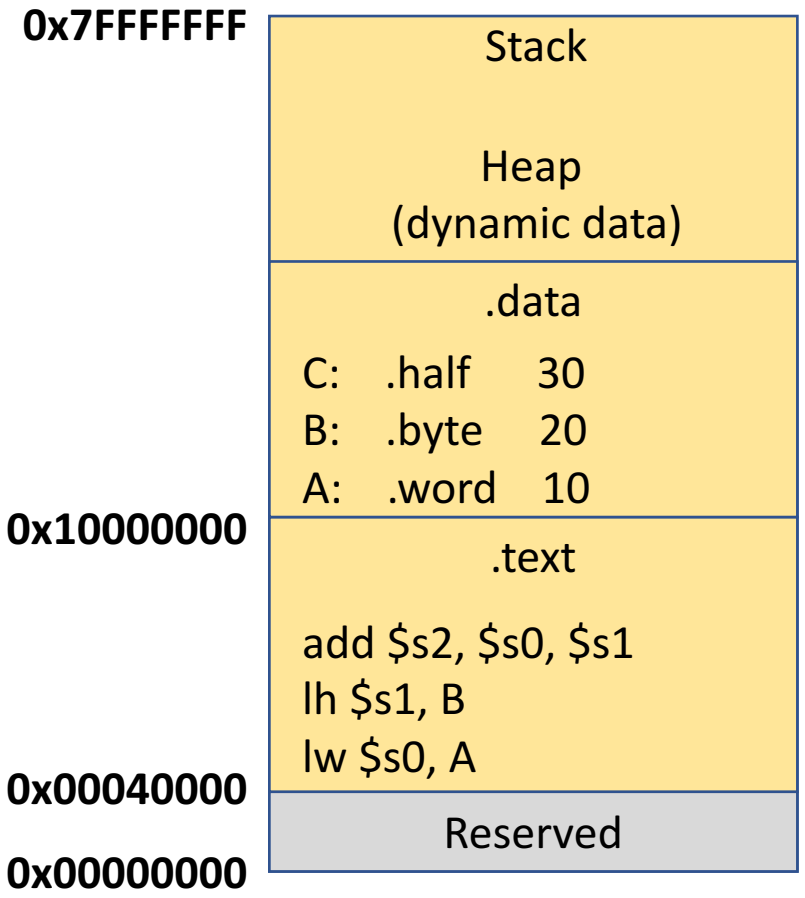
  beq $s0, $s1, target ## branch to *target* if $s0 = $s1

  bne $s0, $s1, target ## branch to *target* if $s0 != $s1

  …

Program structure

- Tanto os dados como as instruções (programas) são mantidas em memória
- Dados podem ter tamanhos diferentes (byte; half; word)
- **Todas as instruções são codificadas em 4 bytes** (1 word)



Codificação de instruções

- Trabalho produzido pelo *assembler*
  - *Uma das fases da compilação*
  - Compilar um programa, transformar programa A -> B

- Pré-processamento
  - Inclui substituição de macros, remoção de comentários,
- Processamento (ou compilação)
  - Tradução do código fonte em código ***assembly***
    - Para a arquitectura de CPU correspondente (x86, x86_64, mips, arm64, powerpc, etc)
- ***Assembler***
  - Tradução do código ***assembly*** em código máquina
- *Linker*
  - Rearranjo do código de forma a incluir código não fornecido (ex: funções externas)

MIPS – Codificação de instruções

- Todas as instruções têm o mesmo tamanho
  - 1 word — 4 bytes — 32 bits
- ISA define 3 formatos de instruções
  - R-Type (register)
  - I-Type (immediate)
  - J-Type (jump)
- Todos os formato são consistente
  - *opcode* ocupa sempre os mesmos bits

MIPS – Codificação de instruções

- R-type instructions (register instructions)

opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- opcode – código da operação
- rd – destination register
- rs – source register
- rt – source/destination register (transient)
- shamt – used for shift operations
- func – used for special functions

MIPS – Codificação de instruções

- R-typ s (reg tions)

Porquê 5 bits

Porquê 5 bits

opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Porquê 5 bits

- opcode – c eração
- rd – destin er
- rs – source
- rt – source/destination register (transient)
- shamt – used for shift operations
- func – used for special functions

MIPS – Codificação de instruções

- R-type instructions (register instructions)

opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Formato

XXX rd, rt, rs

MIPS – Codificação de instruções

- I-type instructions (immediate instructions)

opcode	rs	rt	immed
6 bits	5 bits	5 bits	16 bits

- rs – source register
- rt – source/destination register (transient)
- immed – 16 bit immediate value

- Formato

  XXXi rt, rs, immed

MIPS – Codificação de instruções

- J-type instructions (Jump intructions)

opcode	addr
6 bits	26 bits

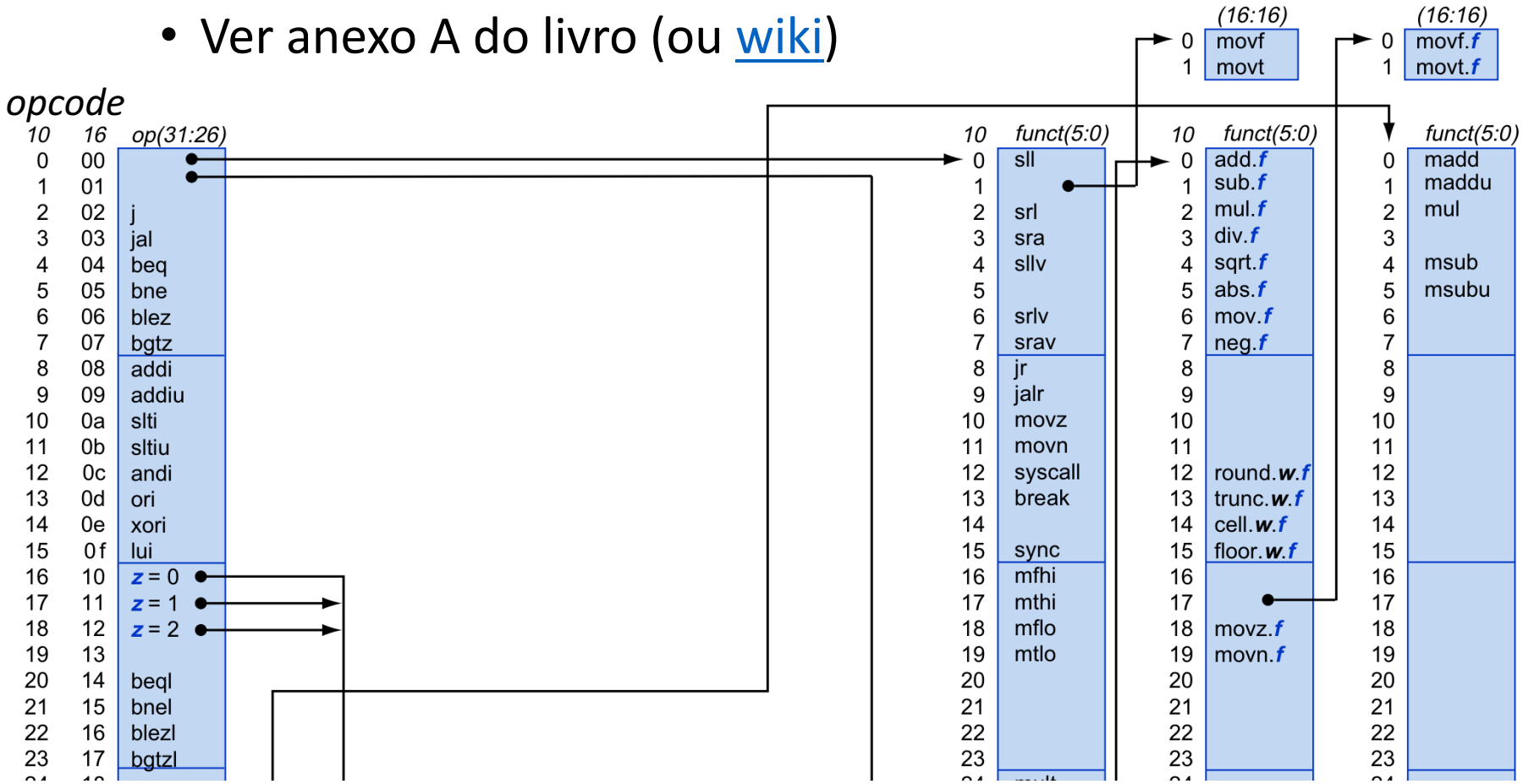
- addr – address

- Formato

j? **addr**

MIPS – Codificação de instruções

- *opcode* permite diferenciar as instruções
  - Ver anexo A do livro (ou wiki)



MIPS – Codificação de instruções

- *opcode* permite diferenciar as instruções
  - Ver anexo A do livro (ou wiki)

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x20
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA
addu	Add Unsigned	R	0x00	0x21
and	Bitwise AND	R	0x00	0x24
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
bne	Branch if Not Equal	I	0x05	NA
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
j	Jump to Address	J	0x02	NA
…	…	…	…	…

MIPS – Codificação de instruções

- Exemplo: add $s0, $s1, $s2
  - opcode -> 0x00 –> 000000_2
  - funct -> 0x20 –> 100000_2
  - $s0 –> $16 –> 10000_2
  - $s1 –> $17 –> 10001_2
  - $s2 –> $18 –> 10010_2

$000000\ 10001\ 10010\ 10000\ 00000\ 100000_2$

ou **0x02328020**

Registos

Nome	Número
$zero	$0
$at	$1
$v0 .. $v1	$2 .. $3
$a0 .. $a3	$4 .. $7
$t0 .. $t7	$8 .. $15
$s0 .. $s7	$16 .. $23
$t8 .. $t9	$24 .. $25
$k0 .. $k1	$26 .. $27
$gp	$28
$sp	$29
$fp	$30
$ra	$31

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x20
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA

Machine, Assembly, and C Code

- 00010000100001010000000000000111
  00000000101001000001000000101010
  00010100010000000000000000000011
  00000000101001000010100000100011
  00000100000000011111111111111100
  00000000100001010010000000100011
  00000100000000011111111111111010
  00000000000001000001000000100001
  00000011111000000000000000001000

Machine, Assembly, and C Code

00010000100001010000000000000111	**beq $4, $5, 28**
00000000101001000001000000101010	**slt $2, $5, $4**
00010100010000000000000000000011	**bne $2, $0, 12**
00000000101001000010100000100011	**subu $5, $5, $4**
00000100000000001111111111111100	**bgez $0 -16**
00000000100001010010000000100011	**subu $4, $4, $5**
00000100000000001111111111111010	**bgez $0 -24**
00000000000001000001000000100001	**addu $2, $0, $4**
00000011111000000000000000001000	**jr $31**

Machine, Assembly, and C Code

- 00010000100001010000000000000111
  00000000101001000001000000101010
  000 0011
  000 0011
  000 1100
  000 0011
  000 1010
  000 0001
  000 1000

```
gcd:
beq $a0, $a1, .L2
slt $v0, $a1, $a0
bne $v0, $zero, .L1
subu $a1, $a1, $a0
b gcd
.L1:
subu $a0, $a0, $a1
b gcd
.L2: move $v0, $a0
j $ra
```

```
beq $4, $5, 28
slt $2, $5, $4
bne $2, $0, 12
subu $5, $5, $4
bgez $0 -16
subu $4, $4, $5
bgez $0 -24
addu $2, $0, $4
jr $31
```

Machine, Assembly, and C Code

- 00010000100001010000000000000111
  00000000101001000001000000101010
  000
  000
  000
  000
  000
  000
  000

```
beq $4, $5, 28
slt $2, $5, $4
bne $2, $0, 12
```

```
gcd:
beq $a0, $a1, .L2
slt $v0, $a1, $a0
bne $v0, $zero, .L1
subu $a1, $a1, $a0
b gcd
.L1:
subu $a0, $a0, $a1
b gcd
.L2: move $v0, $a0
j $ra
```

```
int gcd (int a, int b) {
   while (a != b) {
      if (a > b)
         a = a - b;
      else
         b = b - a;
   }
   return a;
}
```

MIPS – Intruções

- Load

l* $reg, address

lb –> load byte

lh –> load half-word

lw –> load word

la –> load address

li –> load immediate

MIPS – Intruções

- Load (exemplo)

```
    .data
A:  .word   10
B:  .byte   30
C:  .half   40
    .text
main:
    lw $s0, A     # s0 = valor de A
    lb  $s1, B    # s1 = valor de B
    lh  $s2, C    # s2 = valor de c
    li  $s3, 25   # s3 = 25
    la  $s4, A    # s4 = endereço de A
```

MIPS – Intruções

- Store

s* $reg, address

sb –> store byte

sh –> store half-word

sw –> store word

MIPS – Intruções

- Store (exemplo)

```
    .data
A:  .word   0
B:  .byte   0
C:  .half   0
    .text
main:
    li   $s0, 15
    sw $s0, A     # A = valor de s0
    sb  $s0, B    # B = valor de s0
    sh  $s0, C    # C = valor de s0
```

MIPS – Intruções

- Aritmética

add $reg, $reg, $reg -> add $s0, $s1, $s2 # s0=s1+s2

addi $reg, $reg, value -> addi $s0, $s1, 20 # s0=s1+20

…

sub $reg, $reg, $reg -> sub $s0, $s1, $s2 # s0=s1-s2

…

mult $reg, $reg -> mult $s0, $s1 # (hi,lo) = s0*s1

div $reg, $reg -> div $s0, $s1 # hi s0%s1, lo = s0/s1

MIPS – Intruções

- Controlo

b** $reg1, $reg2, address

bgt $reg1, $reg2, addr **# jump to *addr* if reg1 > reg2**

bge $reg1, $reg2, addr **# jump to *addr* if reg1 >= reg2**

blt $reg1, $reg2, addr **# jump to *addr* if reg1 < reg2**

ble $reg1, $reg2, addr **# jump to *addr* if reg1 <= reg2**

beq $reg1, $reg2, addr **# jump to *addr* if reg1 == reg2**

bne $reg1, $reg2, addr **# jump to *addr* if reg1 != reg2**

MIPS – Branches e Jumps

- Permitem definir estruturas de controlo e ciclos

Exemplo:

```
if (x < 0)
        x = 0
else
        y += x
```

```
    .data
    .text
main:  …
     blt $s0, $zero, LT
     add $s1, $s1, $s0
     j END
LT: li $s0, 0
END: ....
```

MIPS – Branches e Jumps

- Permitem definir estruturas de controlo e ciclos

Exemplo:

```
while (x < 0)
        x += 1
```

```
    .data
    .text
main:  …
INIT: bge $s0, $zero, END
        addi $s0, $s0, 1
        j INIT
END: ....
```

MIPS – Arrays

- Array -> coleção de valores do mesmo tipo acedidos por indexação
  - Mantidos em memória em posições contiguas

Exemplos:

```
    .data
A:  .word   10, 20, 30, 40, 50
B:  .word   5:10
C:  .space  40
    .text
main:
    …
```

MIPS – Arrays

Exemplos:

```
    .data
A:  .word   10, 20, 30, 40, 50
B:  .word   5:10
C:  .space  40
    .text
main:
    …
```

MIPS – Arrays e Ciclos

Exemplo 1:

```
int x = [10, 20, 30, 40, 50];
int i, sum = 0;
for (i = 0; i < 5; i++) {
        sum += x[i];
}
```

```
        .data
A:  .word   10, 20, 30, 40, 50
B:  .word   0
        .text
main:   la $s0, A
        li $s1, 0
        li $s2, 0
init:   bge  $s1, 5, end
        add $t2, $s1, $s1
        add $t2, $t2, $t2
        add $t2, $s0, $t2
        lw $s3, 0($t2)
        add  $s2, $s2, $s3
        addi $s1, $s1, 1
        j init
end:    sw $s2, B
```

MIPS – Arrays e Ciclos

Exemplo 1:

```
int x = [10, 20, 30, 40, 50];
int i, sum = 0;
for (i = 0; i < 5; i++) {
        sum += x[i];
}
```

```
                .data
A:    .word    10, 20, 30, 40, 50
B:    .word    0
                .text
main:   la $s0, A
        addi $s1, $s0, 20
        li $s2, 0
init:   bge  $s0, $s1, end
        lw $s3, 0($s0)
        add  $s2, $s2, $s3
        addi $s0, $s0, 4
        j init
end:    sw $s2, B
```

MIPS – Arrays e Ciclos

Exemplo 2:

```
int x = [10, 20, 30, 40, 50];
int i, sum = 0;
for (i = 0; i < 5; i++) {
        x[i] += 1;
}
```

```
                .data
A:    .word    10, 20, 30, 40, 50
B:    .word    0
                .text
main:   la $s0, A
        addi $s1, $s0, 20
        li $s2, 0
init:   bge  $s0, $s1, end
        lw $s3, 0($s0)
        addi  $s3, $s3, 1
        sw $s3, 0($s0)
        addi $s0, $s0, 4
        j init
end:    sw $s2, B
```

MIPS – Syscalls

- Chamadas ao sistema permitem interagir com o sistema
  - Ler do input
  - Escrever p/ output
  - Terminar o programa
  - …
- O contexto da execução do programa muda
- A execução do programa só continua após a execução da chamada

MIPS – Syscalls

- São definidas pelo código da operação

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	
exit	10		

MIPS – Syscalls

Exemplo:

```
int x;
scanf("%d", &x);
x += 1;
printf("new x value: %d", x);
```

```
                .data
txt:    .asciiz "new x value: "
                .text
main:
        li $v0, 5
        syscall
        move $s0, $v0
        addi $s0, $s0, 1
        li $v0, 4
        la $a0, txt
        syscall
        li $v0, 1
        move $a0, $s0
        syscall
```

Funções

Funções

- Funções permitem criar abstrações, bem como reutilizar código
  - A assinatura da função abstrai a sua utilização dos detalhes de implementação
  - O mesmo código pode ser usado em diferentes zonas do programa (sempre que a função é chamada)
- Quando uma função é chamada
  - Os seus argumentos são avaliados e passados a valores
  - O fluxo de execução do programa passa para o corpo da função
  - Quando o resultado é obtido, o fluxo de execução do programa retorna para o endereço após a chamada da função

Funções

- As funções têm o seu próprio espaço de memória
  - Variáveis locais não são visíveis do exterior, e vice-versa
  - Mesmo quando funções se chamam a elas próprias (recursividade)

```
int max (int[] array, int n) {
        int i, max = array[0];
        for (i = 1; i < n; i ++) {
                if (array[i] > max)
                        max = array[i];
        }
        return max;
}
```

MIPS – Funções

- No entanto há um nº limitado de registos do processador
  - Como é que se garante que uma função não altera valores de registos utilizados fora da função?

MIPS – Funções (stack)

- Stack – Segmento de memória usado como pilha de dados (comportamento LIFO)
  - Permite empilhar (*push*) e desempilhar (*pop*) dados
- Stack cresce no sentido dos endereços menores
  - Tem como base o “maior” endereço do pograma

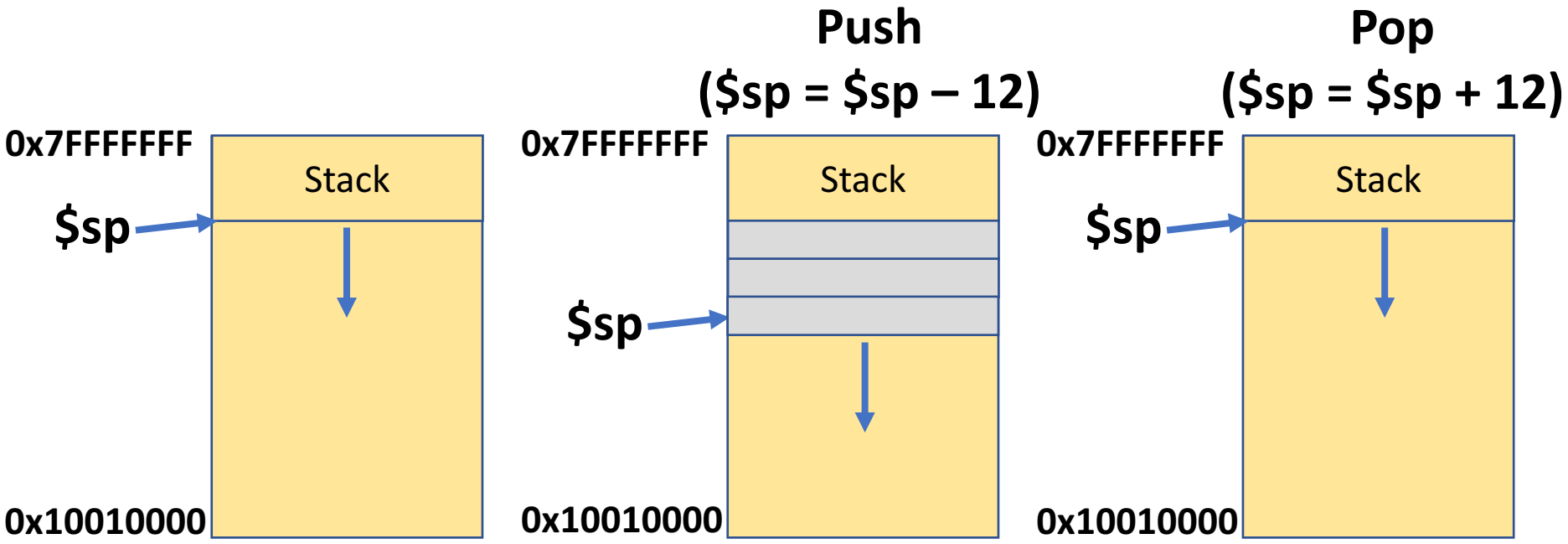
- *Permite guardar valores dos registos, para que possam ser reutilizados, sem comprometer a execução do programa*

MIPS – Funções (stack)

- ISA do MIPS não oferece operações de *push* e *pop*
  - Mas permite manipular o registo *stack pointer* ($sp)

- $sp mantêm o endereço atual do topo da *stack*
  - O endereço do topo da pilha **diminui** ao fazer *push* e **aumenta** quando se faz *pop*

MIPS – Funções (stack)



MIPS – Funções (stack)

- Push -> permite empilhar dados na *stack*
  - *Guardar valores dos registos pré chamadas a funções*

```
push: addi $sp, $sp, -8
      sw $s0, 0($sp)
      sw $s1, 4($sp)
```

- Pop -> permite desempilhar dados na *stack*
  - *Restaurar valores dos registos após chamadas a funções*

```
pop:  lw $s0, 0($sp)
      lw $s1, 4($sp)
      addi $sp, $sp, 8
```

MIPS – Funções (chamada)

- *Caller* deve passar os argumentos utilizando os registos $a0 … $a3
- *Caller* chama a função usando a instrução `jal`
  - Realiza o salto e guarda em $ra o endereço de retorno (valor de $pc antes do salto)
- *Callee* calcula o resultado e guarda-o nos registos $v0 e $v1
- *Callee* retorna a execução para o *Caller* usando a instrução `jr`

MIPS – Funções (chamada)

- *Caller* é responsável por guardar o valor dos registos $t0..$t9
  - Convenção
- *Callee* é responsável por guardar o valor dos registos $s0..$s7
  - Convenção
- *Caller* deve guardar o valor do $ra e dos $a0..$a3
  - Caso ele próprio tenha que chamar outras funções

MIPS – Funções (exemplo)

```
int max (int[] array, int n) {
   int i, max = array[0];
   for (i = 1; i < n; i ++) {
      if (array[i] > max)
         max = array[i];
   }
   return max;
}

int main() {
   int max, x = [10, 20, 30, 40, 50];
   max = max(x, 5);
   printf(“max: %d”, max);
}
```

```
            .data
X: .word 10, 50, 30, 40, 20
Y: .asciiz "max: ”
            .text
main:       la $a0, X
            li $a1, 5
            jal max
            move $s0, $v0
            li $v0, 4
            la $a0, Y
            syscall
            move $a0, $s0
            li $v0, 1
            syscall
            li $v0, 10
            syscall
```

```
max:    addi $sp, $sp, -12
        sw $s0, 8($sp)
        sw $s1, 4($sp)
        sw $s2, 0($sp)
        lw $s0, ($a0)
        li $s1, 1
loop:   addi $a0, $a0, 4
        bge $s1, $a1, end
        lw $s2, ($a0)
        blt $s2, $s0, cont
        move $s0, $s2
cont:   addi $s1, $s1, 1
        j loop
end:    move $v0, $s0
        lw $s0, 8($sp)
        lw $s1, 4($sp)
        lw $s2, 0($sp)
        addi $sp, $sp, 12
        jr $ra
```

MIPS – Funções (exemplo)

```
int fact(int x) {
   if (x == 1)
      return 1;
   else
      return x * fact(x-1);
}

int main() {
      int x = fact(5);
      printf("%d", x);
}
```

```
      .data

      .text
main:
      li $a0, 5
      jal fact
      move $a0, $v0
      li $v0, 1
      syscall
      li $v0, 10
      syscall
```

```
fact:
      add $sp, $sp, -8
      sw $a0, 4($sp)
      sw $ra, 0($sp)
      bne $a0, 1, cont
      addi $v0, $zero, 1
      add $sp, $sp, 8
      jr $ra
cont:
      sub $a0, $a0, 1
      jal fact
      lw $a0, 4($sp)
      lw $ra, 0($sp)
      addi $sp, $sp, 8
      mul $v0, $v0, $a0
      jr $ra
```