

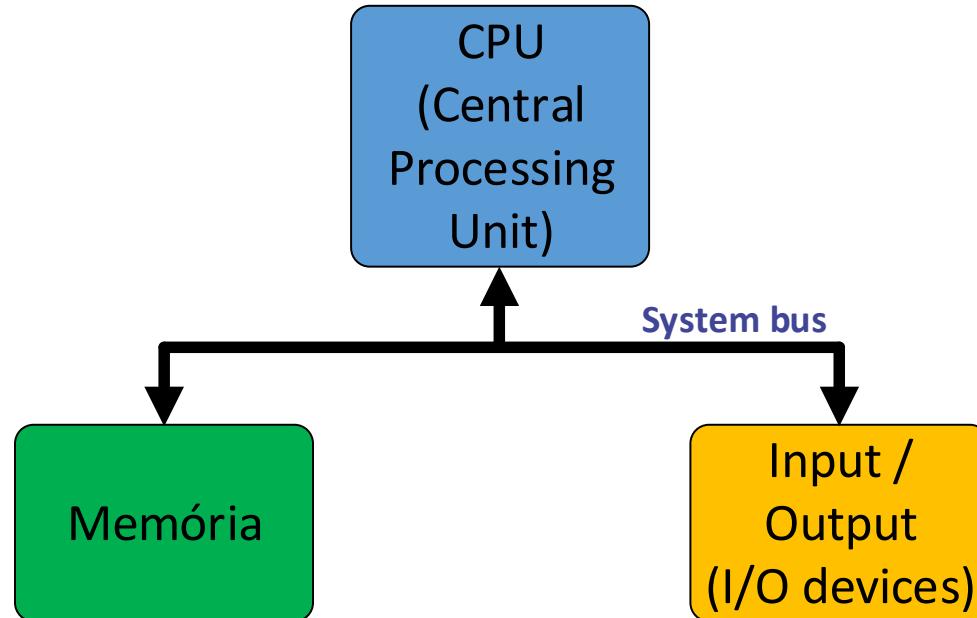
Aula prática 1

- Conceitos básicos de Arquitetura de Computadores.
- Programação em linguagem *assembly*: estrutura de um programa e instruções básicas do MIPS.
- Apresentação do MARS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira



Computador: the *big picture*



- **CPU** (ou microprocessador) – executa sequencialmente instruções
- **Memória** – armazena o programa (conjunto de instruções) e dados
- **I/O devices** – comunicação com o exterior
- **System Bus** – interliga os subsistemas

Visão simplificada do CPU

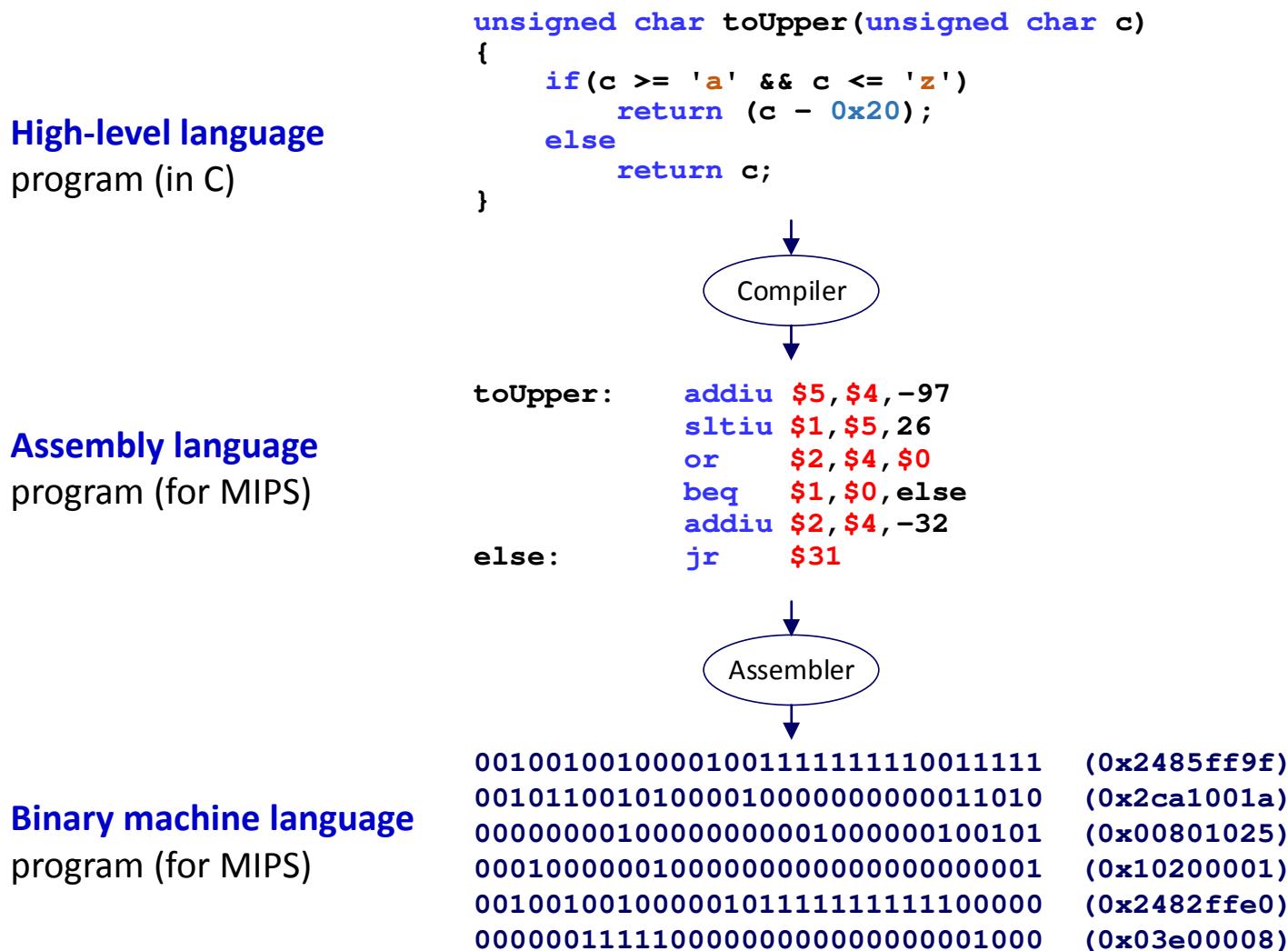
- O CPU é um sistema digital complexo. No entanto, numa visão simplificada, podemos descrevê-lo como contendo três blocos fundamentais:
 - **ALU** (Unidade Aritmética e Lógica)
 - **Registros**
 - **Unidade de controlo**
- **ALU** – realiza as operações aritméticas e lógicas mais comuns (por exemplo, adição, multiplicação, divisão, AND, OR, NOR, XOR)
- **Registros** – elementos de armazenamento (memória) localizados dentro do CPU
 - Usados para diversos fins
 - Um registo armazena uma única unidade de informação (ex. se o registo for de 8 bits pode armazenar 1 byte)
- **Unidade de controlo** - responsável pela coordenação dos vários blocos do CPU, durante a execução de uma instrução



Visão simplificada do CPU – Registros

- Na perspetiva do utilizador, os registos mais importantes são:
 - **Program Counter (PC)**
 - **Registros de utilização geral**, para armazenamento de dados (geralmente em número muito reduzido, por exemplo 32)
- **Program Counter**
 - Usado para guardar o endereço da memória onde se situa a próxima instrução a ser executada
 - No CPU, após a leitura do código de uma instrução, o valor do PC é atualizado para apontar para a instrução seguinte
- Os **regtos de utilização geral** são, habitualmente, referenciados por nomes (e.g., no MIPS: \$0, \$1,...,\$31)

Níveis de Representação



Assembly

- Linguagem básica de programação de microprocessadores, legível por humanos
- Conjunto de instruções que realizam operações simples
 - Somar o conteúdo de 2 registos
 - Subtrair o conteúdo de dois registos
 - Inicializar um registo com um valor
 - Transferir um valor de um registo interno para a memória
- Exemplos:

add \$1, \$5, \$7	# \$1 = \$5 + \$7
sub \$3, \$4, \$2	# \$3 = \$4 - \$2
ori \$6, \$0, 0x1234	# \$6 = \$0 0x1234
	# \$6 = 0x1234



Código máquina

- Sequência de bits que codifica cada uma das instruções *assembly*
- Exemplos:

Instrução *assembly*

`add $1, $5, $7`

`sub $3, $4, $2`

`ori $6, $0, 0x1234`

Código máquina

`0x00A70820`

`0x00821822`

`0x34061234`

- É gerado
 - Por um **compilador**, quando o programa é escrito numa linguagem de alto nível (por exemplo C)
 - Por um **assembler** quando o programa é escrito em linguagem **assembly**



O MIPS

- É um **microprocessador de 32 bits**, isto é:
 - cada **registro interno** armazena uma *word* de **32 bits**
 - a **ALU** opera sobre quantidades de **32 bits**
- Tem **32 registos** internos de uso geral, com a designação nativa **\$0, \$1, \$2, ..., \$31**
- Estes registos são normalmente referenciados nos programas por um nome lógico (facilita a aplicação de uma convenção de utilização, a ver mais tarde)
 - **\$a0, \$a1, \$a2, \$a3**
 - **\$t0, \$t1, \$t2, ..., \$t9**
 - **\$s0, \$s1, \$s2, ..., \$s7**
 - **\$v0, \$v1**
 - **\$ra**
- O registo **\$0** é um caso particular, uma vez que não permite armazenamento e, quando lido, **retorna sempre o valor 0**



Exemplos de algumas instruções do MIPS

- Operações **aritméticas**

add Rdst, Rsrc1, Rsrc2 # $Rdst = Rsrc1 + Rsrc2$

- Ex: **add \$t0, \$a0, \$t1**

sub Rdst, Rsrc1, Rsrc2 # $Rdst = Rsrc1 - Rsrc2$

- Ex: **sub \$a1, \$s0, \$t2**

addi Rdst, Rsrc1, Imm # $Rdst = Rsrc1 + Imm$

- Ex: **addi \$t5, \$a3, 0x13F4**

- Operações **lógicas bitwise**

and Rdst, Rsrc1, Rsrc2 # $Rdst = Rsrc1 \& Rsrc2$

or Rdst, Rsrc1, Rsrc2 # $Rdst = Rsrc1 | Rsrc2$

ori Rdst, Rsrc1, Imm # $Rdst = Rsrc1 | Imm$

- Ex: **ori \$v0, \$0, 0x12** # $$v0 = 0x12$ (zero é o
elemento neutro do OR)



Anatomia de um programa Assembly

```
.data  
...  
... } Dados  
.text  
.globl main  
# label # Instrução      # comentário  
main:   ori $t0,$0,3       # $t0 = 3  
        ori $t2,$0,8       # $t2 = 8  
        add $t1,$t0,$t0     # $t1 = $t0 + $t0  
        add $t1,$t1,$t2     # $t1 = $t1 + $t2  
        jr $ra              # fim do programa } Instruções
```

.text, .data -> ordens para o Assembler (diretivas)

nome: -> label (nome dado a um endereço, e.g., main, str1,...)

ori -> mnemónica de uma instrução

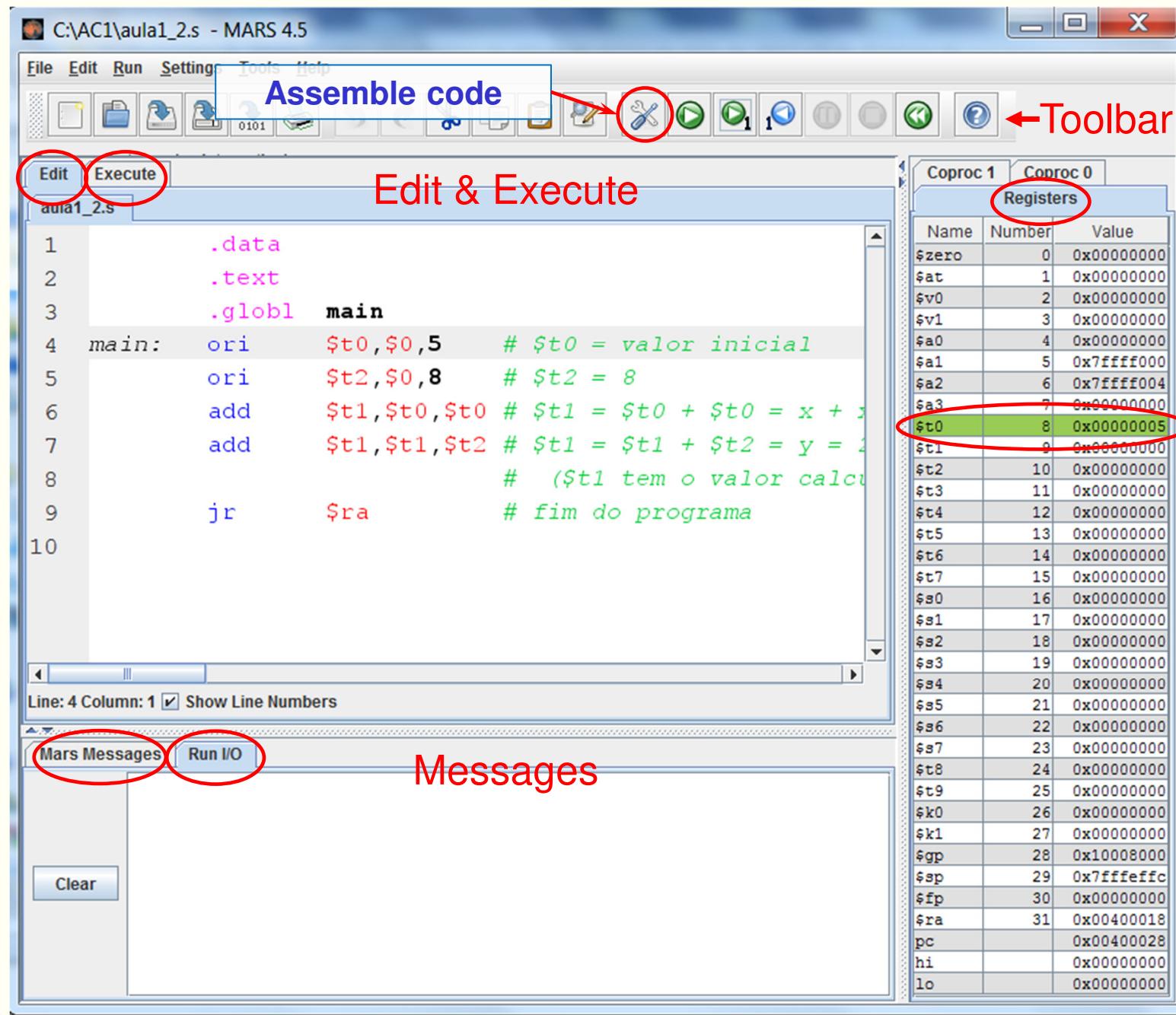
\$t0, \$0, 3 -> operandos de uma instrução

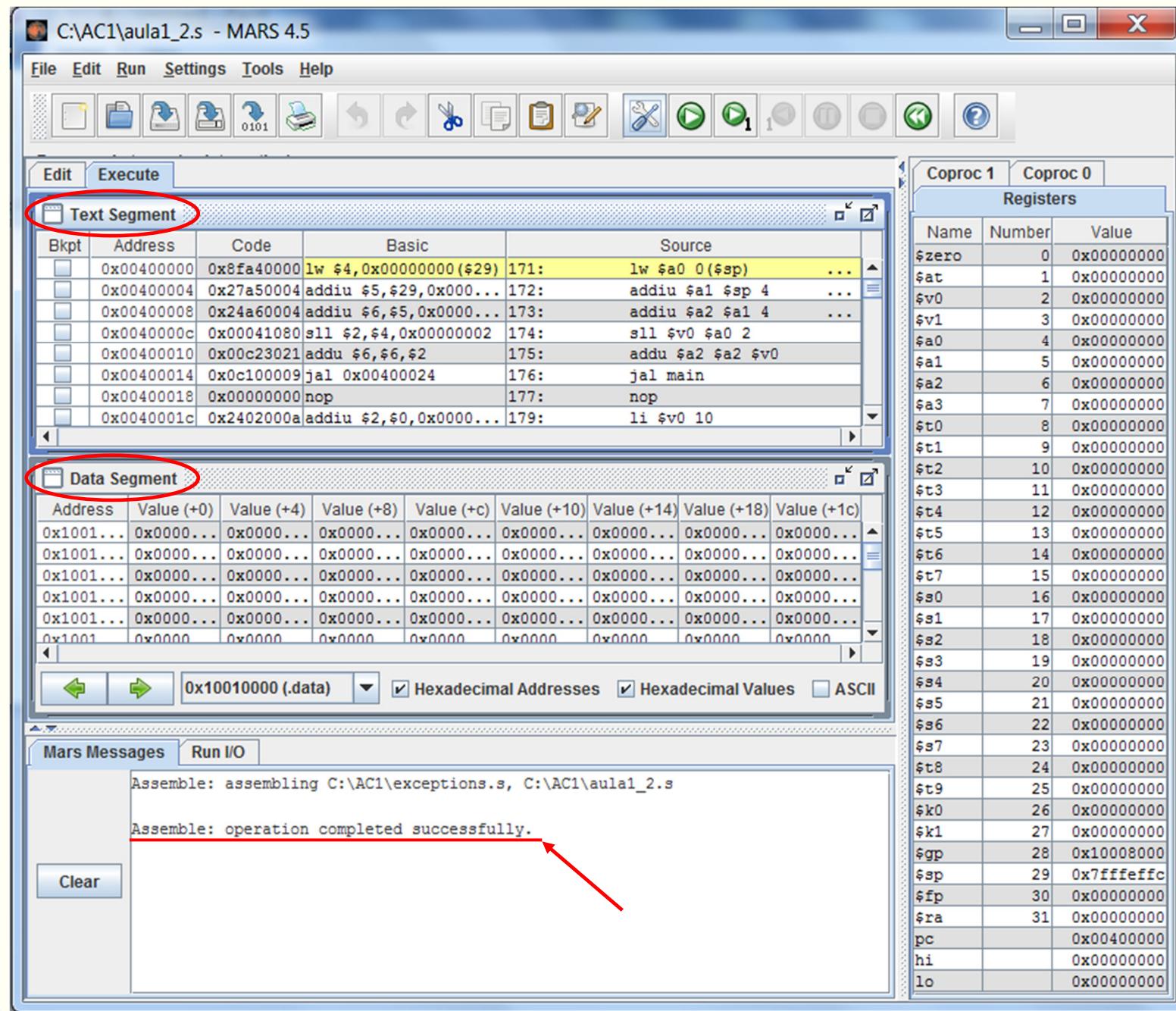


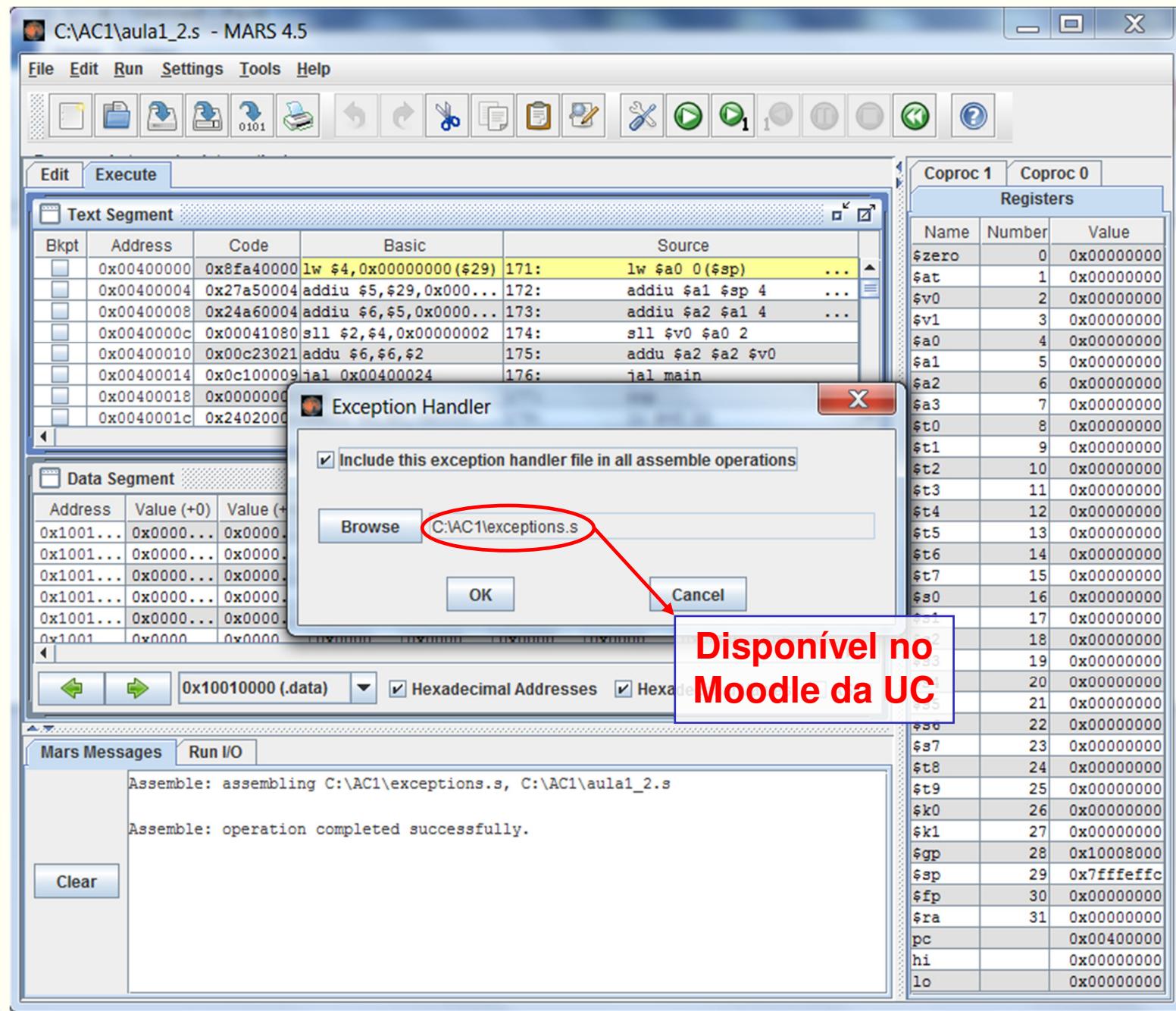
MARS – um ambiente de simulação para o MIPS

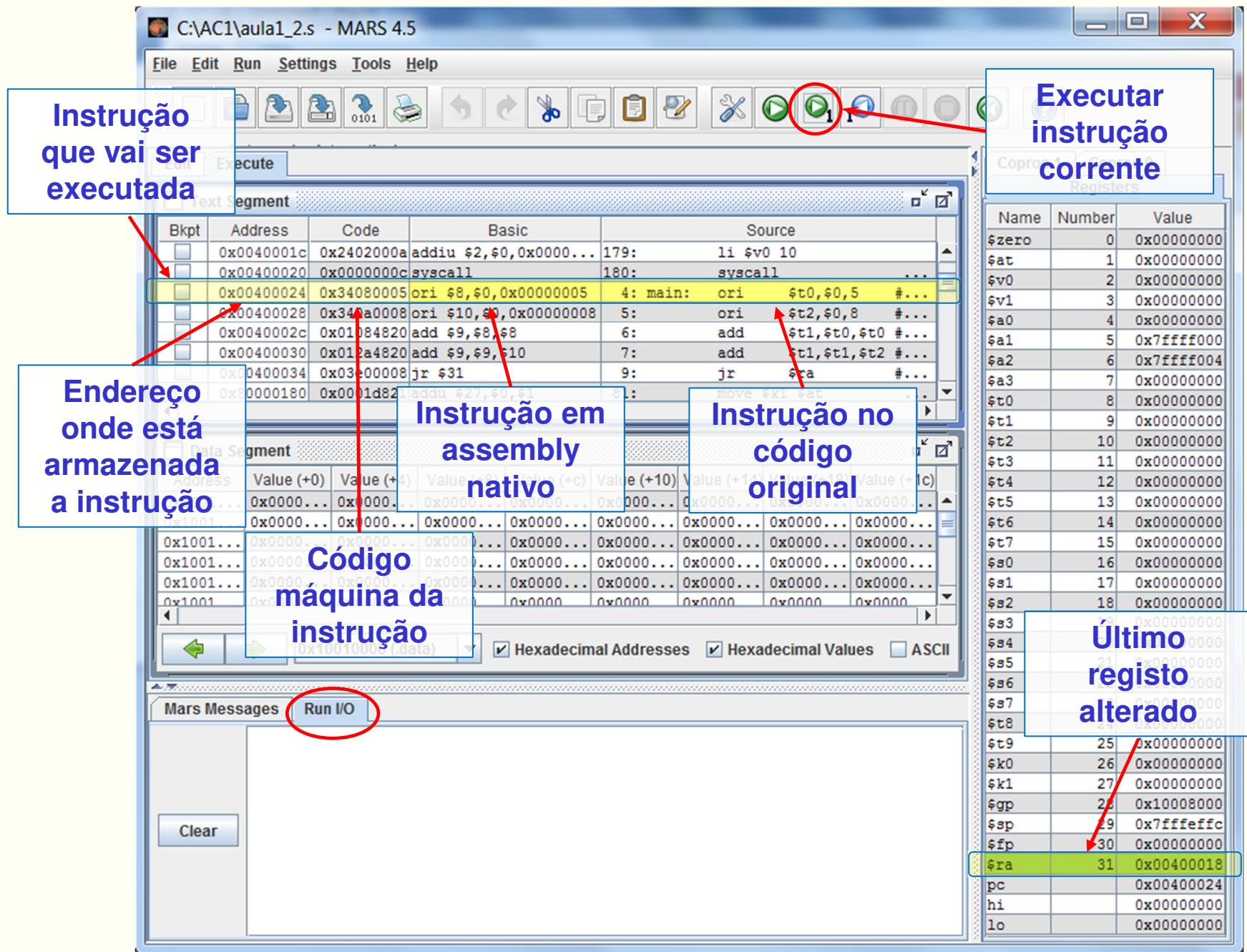
- MARS - MIPS Assembler and Runtime Simulator
- Ambiente integrado de Desenvolvimento (IDE), com:
 - Editor
 - Assembler
 - Simulador
- O simulador permite:
 - Execução do programa *assembly* de uma só vez, ou instrução a instrução (*single step execution*)
 - Acesso aos registos internos do CPU para visualizar/alterar o seu valor
 - Acesso à memória para visualizar/alterar o seu conteúdo
 - Interagir com o exterior (através de *system calls*)

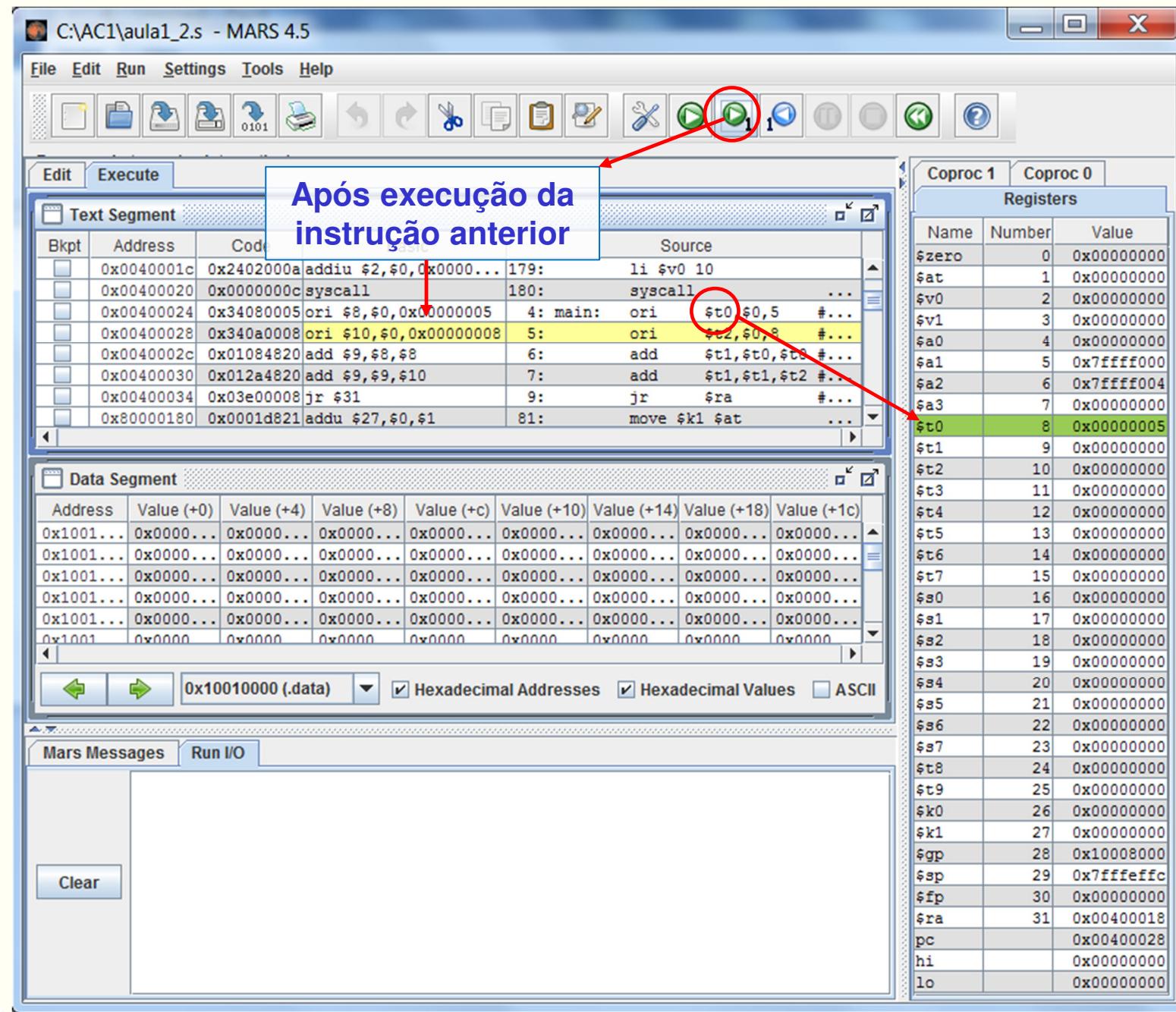












System Calls

- *System Calls* são funções do sistema operativo (SO) que implementam serviços básicos de I/O:
 - imprimir uma *string* no ecrã, ler um inteiro do teclado, ler uma *string* do teclado, imprimir um inteiro, etc.
- O MARS disponibiliza cerca de 50 *system calls*
 - O registo **\$v0** é usado para identificar a *system call*
 - Os registos **\$a0** a **\$a3** são usados para transferir valores (argumentos) para a *system call*
 - O *system call* pode usar **\$v0** para devolver um valor
- Exemplo

```
ori  $v0, $0, 11      # $v0=11 (system call
                      #   print_char())
ori  $a0, $0, 0x31    # $a0 = 0x31 = '1'
syscall               # chama a system call
```



System Calls

- Como funciona um *system call*, na perspetiva do utilizador:
 1. O Sistema Operativo verifica **\$v0** para saber qual a tarefa a realizar
 2. Se necessário, o Sistema Operativo lê os valores de entrada dos registos **\$a0 a \$a3** (e.g. imprimir um carácter no ecrã)
 3. O Sistema Operativo executa a tarefa
 4. O Sistema Operativo coloca o resultado no registo **\$v0** (se isso se aplicar, e.g. ler um inteiro do teclado)

```
ori  $v0,$0,11      # $v0=11 (system call
                      #   print_char()
ori  $a0,$0,0x31    # $a0 = 0x31 = '1'
syscall             # chama a system call
```

