Arquitectura de Computadores I

Assembly RISC-V

Miguel Barão

ESCOLA DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Resumo

Organização de um programa

Linguagem Assembly RISC-V

Organização de um programa

Assembler e loader

Assembly:

```
.data
.word 332, 4, -3
.text
main:
      addi t0, zero, 100
B:    addi t0, t0, -1
      blt zero, t0, B
```

assembler/linker

Ficheiro executável (binário):

```
01100011100010001000010010101001
11010010000100101011110101110101
11110010100010000100100000010010
00000100010010001110000100010110
```

loader

Programa em memória (instruções e dados)

Secções de código e dados

Um programa executável é composto por várias secções:

`.text` contém o código máquina das instruções.

`.data` contém as variáveis inicializadas do programa

- Variáveis globais e variáveis locais estáticas.
- Tamanho dos dados é fixo.
- Dados são de leitura/escrita.

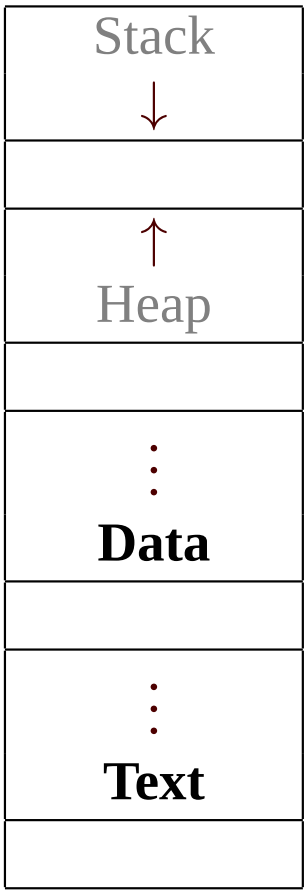
`.rodata` Contém as constantes do programa (*read only data*).

`.bss` Para as variáveis não inicializadas (poupa espaço no executável).

Nos nossos programas vamos usar apenas as secções *text* e *data*.

Layout de um programa em memória

Quando o programa é carregado em memória para execução, as secções *text* e *data* são copiadas para *segmentos* de memória.




```
.data # daqui em diante são dados

      .string "Hello World"
      .word 1, 2, 3, 4, 5
      .byte 7, 0xff


.text # daqui em diante é código

main:
      addi t0, zero, 1
      add t0, a0, t0
      ...
```

Linguagem Assembly RISC-V

Linguagem Assembly

Um programa em Assembly é um ficheiro de texto (`.s`) constituído por

- directivas
- etiquetas (*labels*)
- instruções
- pseudo-instruções
- comentários

O ficheiro é lido e interpretado por um programa chamado assembler.

O assembler converte da linguagem Assembly para um ficheiro objecto (`.o`) que contém os dados e o código máquina das instruções.

Cada arquitectura tem uma linguagem Assembly diferente.

directivas dão informação ao assembler sobre o que vem a seguir, e como o assembler se deve comportar daí em diante.

```
.data .string .word .half .byte .zero
.text
```

directivas dão informação ao assembler sobre o que vem a seguir, e como o assembler se deve comportar daí em diante.

```
.data .string .word .half .byte .zero
.text
```

labels evitam que o programador se tenha de preocupar com endereços. Os endereços são calculados pelo assembler.

```
A:    addi t0, t0, -1
      bge t0, zero, A
```

Linguagem Assembly

directivas dão informação ao assembler sobre o que vem a seguir, e como o assembler se deve comportar daí em diante.

```
.data .string .word .half .byte .zero
.text
```

labels evitam que o programador se tenha de preocupar com endereços. Os endereços são calculados pelo assembler.

```
A:    addi t0, t0, -1
      bge t0, zero, A
```

instruções são convertidas em código máquina pelo assembler.

```
addi t0, t0, 354   -->   0x16228293
```

directivas dão informação ao assembler sobre o que vem a seguir, e como o assembler se deve comportar daí em diante.

```
.data .string .word .half .byte .zero
.text
```

labels evitam que o programador se tenha de preocupar com endereços. Os endereços são calculados pelo assembler.

```
A:    addi t0, t0, -1
      bge t0, zero, A
```

instruções são convertidas em código máquina pelo assembler.

```
addi t0, t0, 354   -->   0x16228293
```

pseudo-instruções são convertidas em instruções pelo assembler, não fazem parte do programa executável. Não têm código máquina.

```
li t0, 0x12345678 | lui t0, 0x12345
                  | addi t0, t0, 0x678
```

Directivas para o assembler

```
.data                   # daqui em diante são dados

.string "Hello"         # string ASCII terminada com nulo '\0'
.word   0x12345678      # números de 32 bits (word)
.half   0x1234          # números de 16 bits (half-word)
.byte   0x48            # números de 8 bits (byte)
.word   -3, 0, 1        # sequência de words
.zero   120             # 120 bytes com zeros

.text                   # daqui em diante é código
```

Directivas para o assembler

```
.data                   # daqui em diante são dados

.string "Hello"         # string ASCII terminada com nulo '\0'
.word   0x12345678      # números de 32 bits (word)
.half   0x1234          # números de 16 bits (half-word)
.byte   0x48            # números de 8 bits (byte)
.word   -3, 0, 1        # sequência de words
.zero   120             # 120 bytes com zeros

.text                   # daqui em diante é código
```

Atenção

As directivas não são instruções RISC-V, são ordens para o assembler.

Labels

As *labels* marcam posições no código. São usadas pelo assembler para calcular endereços.

```
.data

NOME:   .string "James Bond"
PESO:   .word 78

.text

main:   la t0, NOME         # t0 = endereço da string
        lb a0, 0(t0)        # a0 = 'J'
        call putchar
```

Que labels estão a ser usadas neste pedaço de código?

Labels

As *labels* marcam posições no código. São usadas pelo assembler para calcular endereços.

```
.data

NOME:   .string "James Bond"
PESO:   .word 78

.text

main:   la t0, NOME         # t0 = endereço da string
        lb a0, 0(t0)        # a0 = 'J'
        call putchar
```

Que labels estão a ser usadas neste pedaço de código?

`NOME, PESO, main, putchar`

Pseudo-instruções

As pseudo-instruções são convertidas em instruções RISC-V pelo assembler. Existem para facilitar a programação em assembly.

Pseudo-instrução	Instruções RISC-V	Nome
`nop`	`addi x0, x0, 0`	No-operation
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`call offset`	`auipc x1, offset[31:12]+offset[11]` `jalr x1, offset[11:0](x1)`	Call faraway subroutine
`ret`	`jalr x0, 0(x1)`	Return from subroutine
`mv rd, rs`	`addi rd, rs, 0`	Move (copy)
`li rd, imm`	`lui rd, imm[31:12]+imm[11]` `addi rd, rd, imm[11:0]`	Load 32 bit immediate
`la rd, symbol`	`auipc rd, delta[31:12]+delta[11]` `addi rd, rd, delta[11:0]`	Load absolute address
`not rd, rs`	`xori rd, rs, -1`	One's complement

Exemplo: array de words

```
.data
.word 1, -4, 1000
```

0x1001000f	0x00
0x1001000e	0x00
0x1001000d	0x03
0x1001000c	0xe8
0x1001000b	0xff
0x1001000a	0xff
0x10010009	0xff
0x10010008	0xfc
0x10010007	0x00
0x10010006	0x00
0x10010005	0x00
0x10010004	0x01

- Words alinhadas em endereços múltiplos de 4.
- Ordenação de bytes *little endian*.

Exemplo: array de bytes

```
.data
.byte 10, -1, 255, 0
```

0x10010007	0x00
0x10010006	0xff
0x10010005	0xff
0x10010004	0x0a

- Bytes ocupam endereços consecutivos.
- Problema da ordenação de bytes não se coloca.

Exemplo: string

```
.data
.string "Hello World!"
```

0x10010010	0x00 ('\0')
0x1001000f	0x21 ('!')
0x1001000e	0x64 ('d')
0x1001000d	0x6c ('l')
0x1001000c	0x72 ('r')
0x1001000b	0x6f ('o')
0x1001000a	0x57 ('W')
0x10010009	0x20 (' ')
0x10010008	0x6f ('o')
0x10010007	0x6c ('l')
0x10010006	0x6c ('l')
0x10010005	0x65 ('e')
0x10010004	0x48 ('H')

- Cada carácter ASCII ocupa 1 byte.
- String termina com carácter nulo.
- Problema da ordenação de bytes não se coloca.

Exemplo: múltiplos dados

Podemos carregar vários dados em memória:

```
            .data
NOME:       .string "John"
LATLON:     .word -1, 31
WELCOME:    .string "Welcome"
```

Conteúdo da memória:

```
0x10010000: 4a  6f  68  6e  00  00  00  00
            J   o   h   n   NUL **  **  **

0x10010008: ff  ff  ff  ff  1f  00  00  00
            _____(-1)_____  ______31______

0x10010010: 57  65  6c  63  6f  6d  65  00
            W   e   l   c   o   m   e   NUL
```

As words são automaticamente alinhadas em endereços múltiplos de 4.

Exemplo: percorrer um array

```
.data
A:    .word 1, 2, -4, 1000, 0, -333

.text
main:
      la t0, A          # t0 = endereço do array
      li t1, 6          # t1 = tamanho do array

R:    lw t2, 0(t0)
      addi t0, t0, 4
      addi t1, t1, -1
      bne t1, zero, R
```

`la` e `li` são pseudo-instruções. Permitem obter o endereço do array e definir o comprimento.

Exemplo: modificar uma string

Trocar pares de caracteres: `"Hello World!" --> "eHll ooWlr!d"`

```
.data
S:    .string "Hello World!"

.text
main:
      la t0, S            # t0 = endereço da string

R:    lb t1, 0(t0)        # le primeiro char
      lb t2, 1(t0)        # le segundo char

      sb t1, 1(t0)        # guarda na outra posição
      sb t2, 0(t0)        # idem

      addi t0, t0, 2      # avanca na string
      j R
```

Qual é o erro?

Exemplo: modificar uma string

Trocar pares de caracteres: `"Hello World!" --> "eHll ooWlr!d"`

```
.data
S:    .string "Hello World!"

.text
main:
      la t0, S            # t0 = endereço da string

R:    lb t1, 0(t0)        # le primeiro char
      lb t2, 1(t0)        # le segundo char

      sb t1, 1(t0)        # guarda na outra posição
      sb t2, 0(t0)        # idem

      addi t0, t0, 2      # avanca na string
      j R
```

Qual é o erro? Ciclo infinito, destrói toda a memória... corrija!