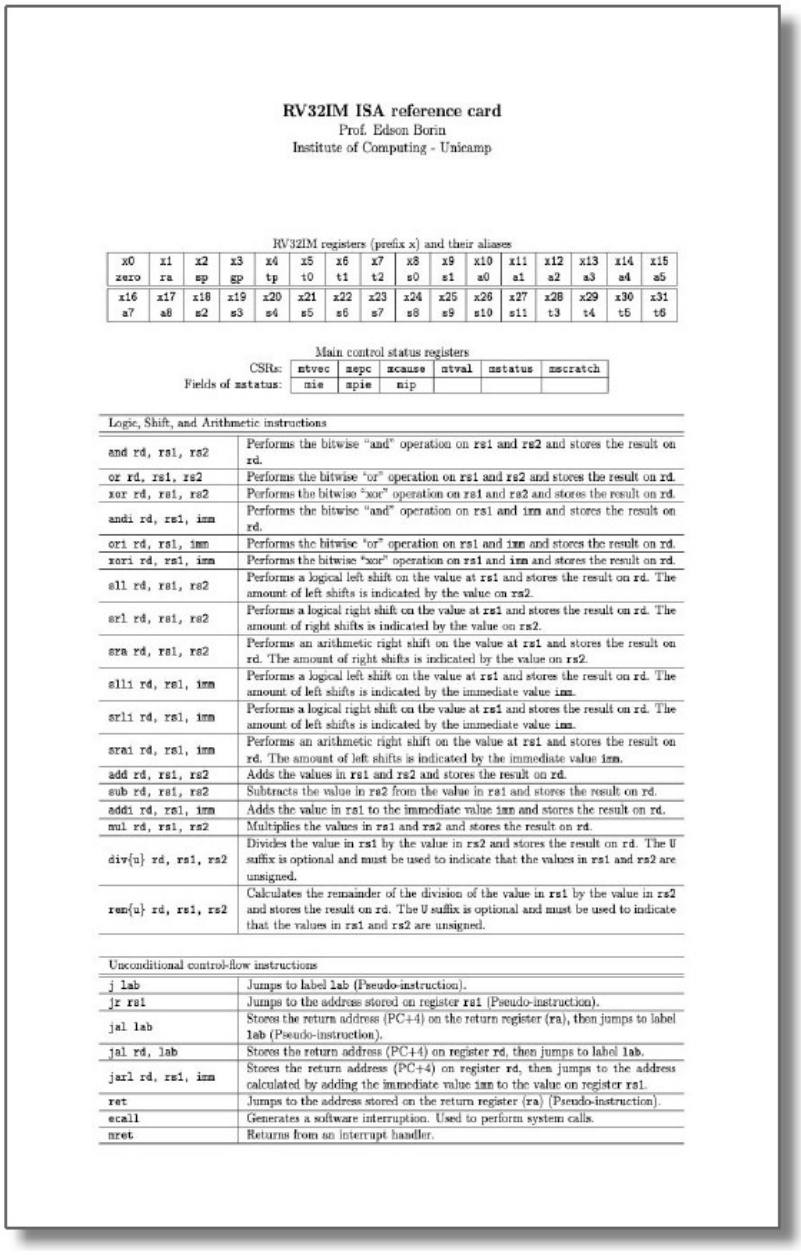
Arquitetura de Computadores II

Linguagem de máquina e instruções

Professor Matheus Alcântara Souza

RISC-V – Linguagem de montagem – Instruções de montagem

- Instruções RV32IM estão no cartão de referência



RV32IM ISA reference card

Prof. Edson Borin
Institute of Computing - Unicamp

RV32IM registers (prefix x) and their aliases

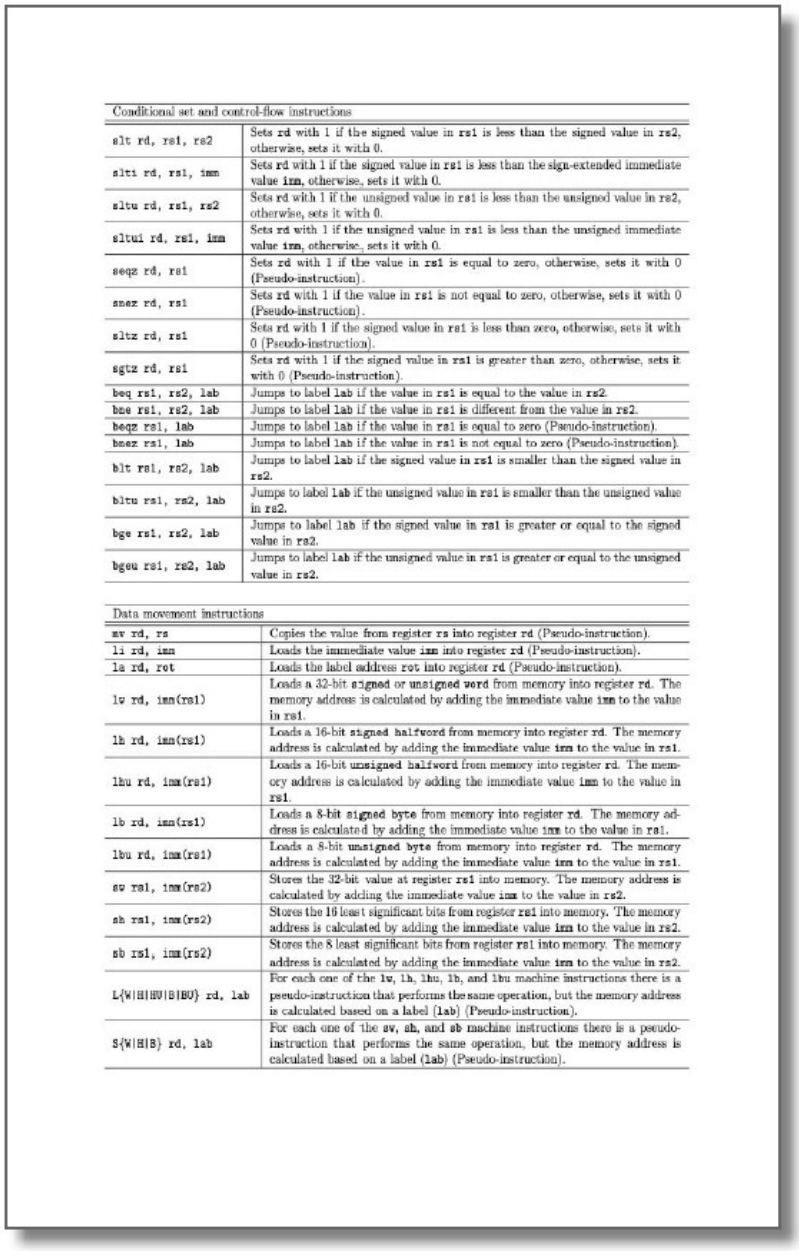
x0 zero	x1 ra	x2 sp	x3 gp	x4 tp	x5 t0	x6 t1	x7 t2	x8 s0	x9 s1	x10 a0	x11 a1	x12 a2	x13 a3	x14 a4	x15 a5
x16 a7	x17 s8	x18 s2	x19 s3	x20 s4	x21 s5	x22 s6	x23 s7	x24 s8	x25 s9	x26 s10	x27 s11	x28 t3	x29 t4	x30 t5	x31 t6

Main control status registers

CSRs:	mtvec	mepc	mcause	mtval	mstatus	mscratch
Fields of mstatus:	mie	mpie	mip			

Logic, Shift, and Arithmetic instructions	
`and rd, rs1, rs2`	Performs the bitwise "and" operation on `rs1` and `rs2` and stores the result on `rd`.
`or rd, rs1, rs2`	Performs the bitwise "or" operation on `rs1` and `rs2` and stores the result on `rd`.
`xor rd, rs1, rs2`	Performs the bitwise "xor" operation on `rs1` and `rs2` and stores the result on `rd`.
`andi rd, rs1, imm`	Performs the bitwise "and" operation on `rs1` and `imm` and stores the result on `rd`.
`ori rd, rs1, imm`	Performs the bitwise "or" operation on `rs1` and `imm` and stores the result on `rd`.
`xori rd, rs1, imm`	Performs the bitwise "xor" operation on `rs1` and `imm` and stores the result on `rd`.
`sll rd, rs1, rs2`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the value on `rs2`.
`srl rd, rs1, rs2`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`sra rd, rs1, rs2`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of right shifts is indicated by the value on `rs2`.
`slli rd, rs1, imm`	Performs a logical left shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srli rd, rs1, imm`	Performs a logical right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`srai rd, rs1, imm`	Performs an arithmetic right shift on the value at `rs1` and stores the result on `rd`. The amount of left shifts is indicated by the immediate value `imm`.
`add rd, rs1, rs2`	Adds the values in `rs1` and `rs2` and stores the result on `rd`.
`sub rd, rs1, rs2`	Subtracts the value in `rs2` from the value in `rs1` and stores the result on `rd`.
`addi rd, rs1, imm`	Adds the value in `rs1` to the immediate value `imm` and stores the result on `rd`.
`mul rd, rs1, rs2`	Multiplies the values in `rs1` and `rs2` and stores the result on `rd`.
`div{u} rd, rs1, rs2`	Divides the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.
`rem{u} rd, rs1, rs2`	Calculates the remainder of the division of the value in `rs1` by the value in `rs2` and stores the result on `rd`. The `U` suffix is optional and must be used to indicate that the values in `rs1` and `rs2` are unsigned.

Unconditional control-flow instructions	
`j lab`	Jumps to label `lab` (Pseudo-instruction).
`jr rs1`	Jumps to the address stored on register `rs1` (Pseudo-instruction).
`jal lab`	Stores the return address (PC+4) on the return register (`ra`), then jumps to label `lab` (Pseudo-instruction).
`jal rd, lab`	Stores the return address (PC+4) on register `rd`, then jumps to label `lab`.
`jarl rd, rs1, imm`	Stores the return address (PC+4) on register `rd`, then jumps to the address calculated by adding the immediate value `imm` to the value on register `rs1`.
`ret`	Jumps to the address stored on the return register (`ra`) (Pseudo-instruction).
`ecall`	Generates a software interruption. Used to perform system calls.
`mret`	Returns from an interrupt handler.



Conditional set and control-flow instructions	
`slt rd, rs1, rs2`	Sets `rd` with 1 if the signed value in `rs1` is less than the signed value in `rs2`, otherwise, sets it with 0.
`slti rd, rs1, imm`	Sets `rd` with 1 if the signed value in `rs1` is less than the sign-extended immediate value `imm`, otherwise, sets it with 0.
`sltu rd, rs1, rs2`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned value in `rs2`, otherwise, sets it with 0.
`sltui rd, rs1, imm`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned immediate value `imm`, otherwise, sets it with 0.
`seqz rd, rs1`	Sets `rd` with 1 if the value in `rs1` is equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`snez rd, rs1`	Sets `rd` with 1 if the value in `rs1` is not equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`sltz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is less than zero, otherwise, sets it with 0 (Pseudo-instruction).
`sgtz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is greater than zero, otherwise, sets it with 0 (Pseudo-instruction).
`beq rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is equal to the value in `rs2`.
`bne rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is different from the value in `rs2`.
`beqz rs1, lab`	Jumps to label `lab` if the value in `rs1` is equal to zero (Pseudo-instruction).
`bnez rs1, lab`	Jumps to label `lab` if the value in `rs1` is not equal to zero (Pseudo-instruction).
`blt rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is smaller than the signed value in `rs2`.
`bltu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is smaller than the unsigned value in `rs2`.
`bge rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is greater or equal to the signed value in `rs2`.
`bgeu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is greater or equal to the unsigned value in `rs2`.

Data movement instructions	
`mv rd, rs`	Copies the value from register `rs` into register `rd` (Pseudo-instruction).
`li rd, imm`	Loads the immediate value `imm` into register `rd` (Pseudo-instruction).
`la rd, rot`	Loads the label address `rot` into register `rd` (Pseudo-instruction).
`lw rd, imm(rs1)`	Loads a 32-bit `signed` or `unsigned word` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lh rd, imm(rs1)`	Loads a 16-bit `signed halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lhu rd, imm(rs1)`	Loads a 16-bit `unsigned halfword` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lb rd, imm(rs1)`	Loads a 8-bit `signed byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`lbu rd, imm(rs1)`	Loads a 8-bit `unsigned byte` from memory into register `rd`. The memory address is calculated by adding the immediate value `imm` to the value in `rs1`.
`sw rs1, imm(rs2)`	Stores the 32-bit value at register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sh rs1, imm(rs2)`	Stores the 16 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`sb rs1, imm(rs2)`	Stores the 8 least significant bits from register `rs1` into memory. The memory address is calculated by adding the immediate value `imm` to the value in `rs2`.
`L{W|H|HU|B|BU} rd, lab`	For each one of the `lw`, `lh`, `lhu`, `lb`, and `lbu` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`) (Pseudo-instruction).
`S{W|H|B} rd, lab`	For each one of the `sw`, `sh`, and `sb` machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (`lab`) (Pseudo-instruction).

RISC-V – Linguagem de montagem – Instruções de montagem

- Instruções RV32IM estão no cartão de referência

Logic, Shift, and Arithmetic instructions	
`and rd, rs1, rs2`	Performs the bitwise "and" operation on **rs1** and **rs2** and stores the result on **rd**.
`or rd, rs1, rs2`	Performs the bitwise "or" operation on **rs1** and **rs2** and stores the result on **rd**.
`xor rd, rs1, rs2`	Performs the bitwise "xor" operation on **rs1** and **rs2** and stores the result on **rd**.
`andi rd, rs1, imm`	Performs the bitwise "and" operation on **rs1** and **imm** and stores the result on **rd**.
`ori rd, rs1, imm`	Performs the bitwise "or" operation on **rs1** and **imm** and stores the result on **rd**.
`xori rd, rs1, imm`	Performs the bitwise "xor" operation on **rs1** and **imm** and stores the result on **rd**.
`sll rd, rs1, rs2`	Performs a logical left shift on the value at **rs1** and stores the result on **rd**. The amount of left shifts is indicated by the value on **rs2**.
`srl rd, rs1, rs2`	Performs a logical right shift on the value at **rs1** and stores the result on **rd**. The amount of right shifts is indicated by the value on **rs2**.
`sra rd, rs1, rs2`	Performs an arithmetic right shift on the value at **rs1** and stores the result on **rd**. The amount of right shifts is indicated by the value on **rs2**.
`slli rd, rs1, imm`	Performs a logical left shift on the value at **rs1** and stores the result on **rd**. The amount of left shifts is indicated by the immediate value **imm**.
`srli rd, rs1, imm`	Performs a logical right shift on the value at **rs1** and stores the result on **rd**. The amount of left shifts is indicated by the immediate value **imm**.
`srai rd, rs1, imm`	Performs an arithmetic right shift on the value at **rs1** and stores the result on **rd**. The amount of left shifts is indicated by the immediate value **imm**.
`add rd, rs1, rs2`	Adds the values in **rs1** and **rs2** and stores the result on **rd**.
`sub rd, rs1, rs2`	Subtracts the value in **rs2** from the value in **rs1** and stores the result on **rd**.
`addi rd, rs1, imm`	Adds the value in **rs1** to the immediate value **imm** and stores the result on **rd**.
`mul rd, rs1, rs2`	Multiplies the values in **rs1** and **rs2** and stores the result on **rd**.
`div{u} rd, rs1, rs2`	Divides the value in **rs1** by the value in **rs2** and stores the result on **rd**. The **U** suffix is optional and must be used to indicate that the values in **rs1** and **rs2** are unsigned.
`rem{u} rd, rs1, rs2`	Calculates the remainder of the division of the value in **rs1** by the value in **rs2** and stores the result on **rd**. The **U** suffix is optional and must be used to indicate that the values in **rs1** and **rs2** are unsigned.

RISC-V – Linguagem de montagem – Instruções de montagem

- Instruções RV32IM estão no cartão de referência

Unconditional control-flow instructions	
`j lab`	Jumps to label **`lab`** (Pseudo-instruction).
`jr rs1`	Jumps to the address stored on register **`rs1`** (Pseudo-instruction).
`jal lab`	Stores the return address (PC+4) on the return register (**`ra`**), then jumps to label **`lab`** (Pseudo-instruction).
`jal rd, lab`	Stores the return address (PC+4) on register **`rd`**, then jumps to label **`lab`**.
`jarl rd, rs1, imm`	Stores the return address (PC+4) on register **`rd`**, then jumps to the address calculated by adding the immediate value **`imm`** to the value on register **`rs1`**.
`ret`	Jumps to the address stored on the return register (**`ra`**) (Pseudo-instruction).
`ecall`	Generates a software interruption. Used to perform system calls.
`mret`	Returns from an interrupt handler.

RISC-V – Linguagem de montagem – Instruções de montagem

- Instruções RV32IM estão no cartão de referência

Conditional set and control-flow instructions	
`slt rd, rs1, rs2`	Sets `rd` with 1 if the signed value in `rs1` is less than the signed value in `rs2`, otherwise, sets it with 0.
`slti rd, rs1, imm`	Sets `rd` with 1 if the signed value in `rs1` is less than the sign-extended immediate value `imm`, otherwise, sets it with 0.
`sltu rd, rs1, rs2`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned value in `rs2`, otherwise, sets it with 0.
`sltui rd, rs1, imm`	Sets `rd` with 1 if the unsigned value in `rs1` is less than the unsigned immediate value `imm`, otherwise, sets it with 0.
`seqz rd, rs1`	Sets `rd` with 1 if the value in `rs1` is equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`snez rd, rs1`	Sets `rd` with 1 if the value in `rs1` is not equal to zero, otherwise, sets it with 0 (Pseudo-instruction).
`sltz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is less than zero, otherwise, sets it with 0 (Pseudo-instruction).
`sgtz rd, rs1`	Sets `rd` with 1 if the signed value in `rs1` is greater than zero, otherwise, sets it with 0 (Pseudo-instruction).
`beq rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is equal to the value in `rs2`.
`bne rs1, rs2, lab`	Jumps to label `lab` if the value in `rs1` is different from the value in `rs2`.
`beqz rs1, lab`	Jumps to label `lab` if the value in `rs1` is equal to zero (Pseudo-instruction).
`bnez rs1, lab`	Jumps to label `lab` if the value in `rs1` is not equal to zero (Pseudo-instruction).
`blt rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is smaller than the signed value in `rs2`.
`bltu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is smaller than the unsigned value in `rs2`.
`bge rs1, rs2, lab`	Jumps to label `lab` if the signed value in `rs1` is greater or equal to the signed value in `rs2`.
`bgeu rs1, rs2, lab`	Jumps to label `lab` if the unsigned value in `rs1` is greater or equal to the unsigned value in `rs2`.

- Instruções RV32IM estão no cartão de referência

Data movement instructions	
`mv rd, rs`	Copies the value from register **rs** into register **rd** (Pseudo-instruction).
`li rd, imm`	Loads the immediate value **imm** into register **rd** (Pseudo-instruction).
`la rd, rot`	Loads the label address **rot** into register **rd** (Pseudo-instruction).
`lw rd, imm(rs1)`	Loads a 32-bit **signed** or **unsigned word** from memory into register **rd**. The memory address is calculated by adding the immediate value **imm** to the value in **rs1**.
`lh rd, imm(rs1)`	Loads a 16-bit **signed halfword** from memory into register **rd**. The memory address is calculated by adding the immediate value **imm** to the value in **rs1**.
`lhu rd, imm(rs1)`	Loads a 16-bit **unsigned halfword** from memory into register **rd**. The memory address is calculated by adding the immediate value **imm** to the value in **rs1**.
`lb rd, imm(rs1)`	Loads a 8-bit **signed byte** from memory into register **rd**. The memory address is calculated by adding the immediate value **imm** to the value in **rs1**.
`lbu rd, imm(rs1)`	Loads a 8-bit **unsigned byte** from memory into register **rd**. The memory address is calculated by adding the immediate value **imm** to the value in **rs1**.
`sw rs1, imm(rs2)`	Stores the 32-bit value at register **rs1** into memory. The memory address is calculated by adding the immediate value **imm** to the value in **rs2**.
`sh rs1, imm(rs2)`	Stores the 16 least significant bits from register **rs1** into memory. The memory address is calculated by adding the immediate value **imm** to the value in **rs2**.
`sb rs1, imm(rs2)`	Stores the 8 least significant bits from register **rs1** into memory. The memory address is calculated by adding the immediate value **imm** to the value in **rs2**.
`L{W|H|HU|B|BU} rd, lab`	For each one of the **lw**, **lh**, **lhu**, **lb**, and **lbu** machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (**lab**) (Pseudo-instruction).
`S{W|H|B} rd, lab`	For each one of the **sw**, **sh**, and **sb** machine instructions there is a pseudo-instruction that performs the same operation, but the memory address is calculated based on a label (**lab**) (Pseudo-instruction).

RISC-V – Linguagem de montagem

- Valores imediatos
  - São valores numéricos
  - Codificados na própria instrução

```
li a0, 10     # carrega dez em a0
li a0, -10    # carrega menos dez em a0
li a1, 0xa    # carrega dez em a1
li a2, 0b1010 # carrega dez em a2
li a3, 012    # carrega dez em a3
li a4, '0'    # carrega quarenta e oito em a4
li a5, 'a'    # carrega noventa e sete em a5
li a5, -'a'   # carrega menos noventa e sete em a5
```

RISC-V – Linguagem de montagem

- Valores imediatos
  - São valores numéricos
  - Podem ser usados em diretivas também

```
.set temp, 100
.word 10
.byte 'a'
```

RISC-V – Linguagem de montagem

- Símbolos
  - São “nomes” aos quais se associam valores numéricos
  - A tabela de símbolos é a estrutura de dados que mapeia o nome dos símbolos aos valores
  - O montador transforma rótulos em símbolos e os armazena na tabela de símbolos
    - O símbolo criado é associado a um endereço que representa a posição do rótulo no programa

RISC-V – Linguagem de montagem

- Símbolos
  - Nomes podem ser usados como parâmetro em algumas diretivas e instruções de montagem

```
x: .word 10      # Rótulo x: define o símbolo x
.set temp, 100   # Diretiva .set define um símbolo

la a0, x         # carrega o endereõ de x em a0
li a1, [temp]    # carrega a constante temp em a1

y: .word x       # Inicia o conteúdo da variável y
                 # com o endereço da variável x
```

RISC-V – Linguagem de montagem

- Rótulos
  - São marcadores no código que serão convertidos em endereços pelo montador
  - O montador GNU para RV32I aceita dois tipos de rótulos:
    - Simbólicos
    - Numéricos

RISC-V – Linguagem de montagem

- Rótulos **simbólicos**
  - São convertidos para símbolos e adicionados na tabela de símbolos
    - Usados geralmente para anotar a posição (endereço) de variáveis globais e rotinas do código.
  - A sintaxe de um rótulo simbólico é uma palavra com letras, dígitos numéricos e “underscore” (_), terminada com o caractere “dois pontos” (:)
    - Não pode começar com dígito numérico

- Rótulos **simbólicos**
  - A sintaxe de um rótulo simbólico é uma palavra com letras, dígitos numéricos e “underscore” (_), terminada com o caractere “dois pontos” (:)
    - Não pode começar com dígito numérico
    - É case sensitive!

```
# Rótulos válidos

_x:
_y____z:
Teste123:
Var_1:
var_1:
```

```
# Rótulos inválidos

1x:
var-1:
var 1:
@y:
```

RISC-V – Linguagem de montagem

- Rótulos **numéricos**
  - São rótulos locais úteis para referencia trechos de código próximos.
    - São referenciados de forma relativa e um rótulo numérico pode ter o mesmo nome que outros rótulos numéricos no mesmo arquivo
  - Sintaxe: um dígito numérico seguido de “dois pontos”

```
# Exemplos de rótulos numéricos

1:
2:
1:
8:
```

- Rótulos **numéricos**
  - Referências: devem incluir o dígito que identifica o rótulo e um sufixo
    - f : *forward* – ou seja, próximo rótulo numérico
    - b : backward – ou seja, rótulo numérico anterior

```
# Exemplos de referências para rótulos numéricos

1:
beq a0, zero, 1f    # retorna da função
beq a0, a1,   1b    # salta para trás
1:
ret
```

RISC-V – Linguagem de montagem

- Exemplo com uso de rótulos **simbólicos** e **numéricos**

```
# Pow function – computes a^b
# Inputs: a0=a, a1=b
# Output: a0=a^b

pow:
  mv   a2, a0       # Saves a0 in a2
  li   a0, 1        # Sets a0 to 1
1:
  beqz a1, 1f       # if a1 = 0 the done
  mul  a0, a0, a2   # else, multiply
  addi a1, a1, -1   # Decrements the counter
  j    1b
1:
  ret
```

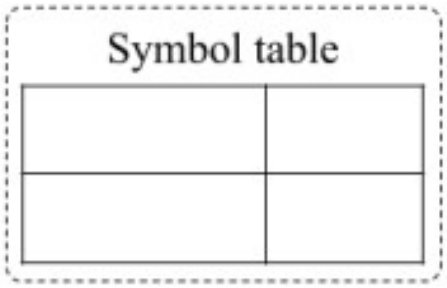
- **Contador de localização**
  - O *location counter* é um contador interno do montador que auxilia no processo de atribuir endereços às instruções e símbolos
    - Contém o endereço da próxima posição livre de memória – a posição onde será montado o próximo elemento do programa
  - Cada seção do programa tem seu próprio contador de localização e todos são iniciados com zero no início do processo de montagem

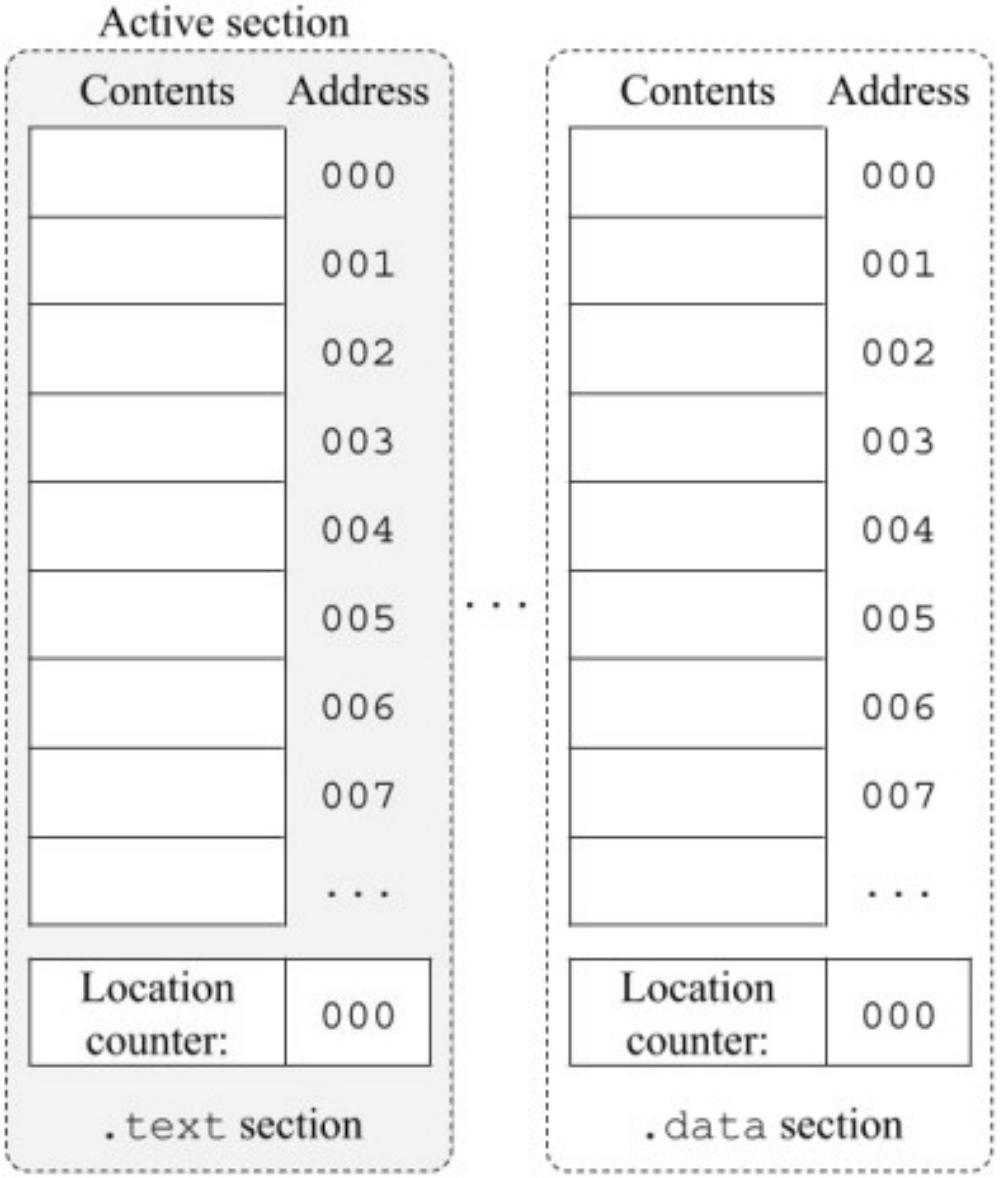
- **Contador de localização**

Exemplo:

Input file

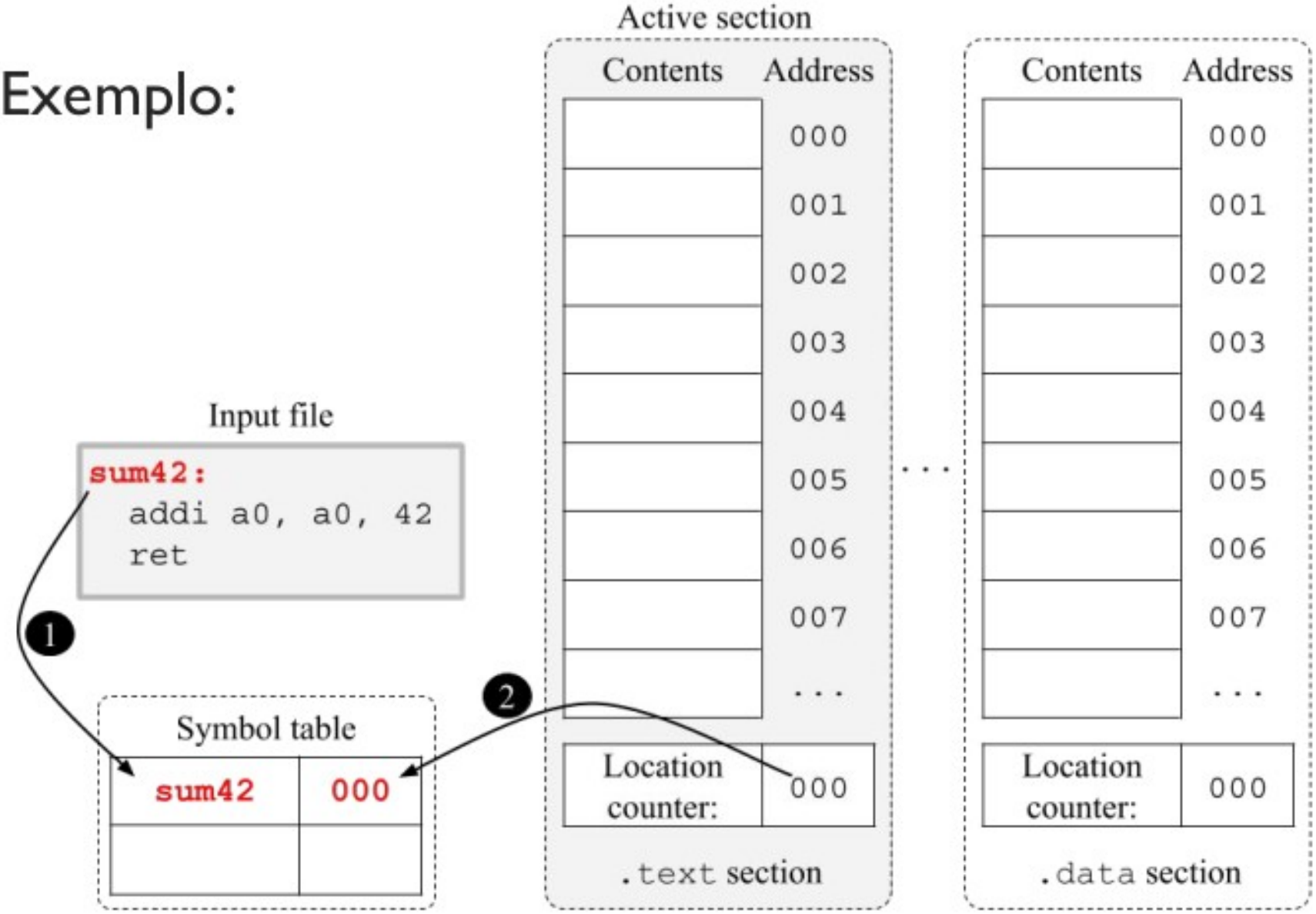
```
sum42:
  addi a0, a0, 42
  ret
```



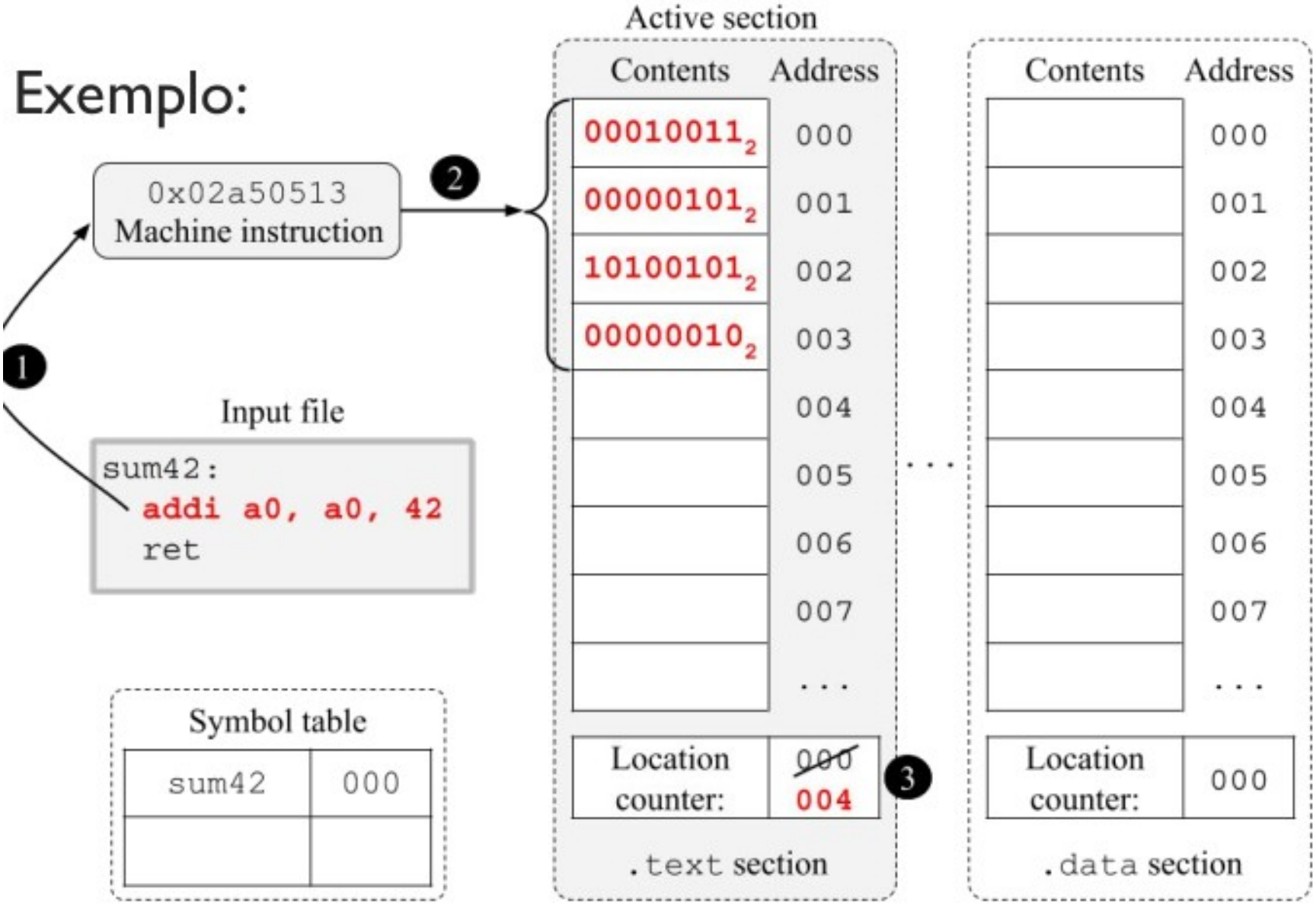


- **Contador de localização**

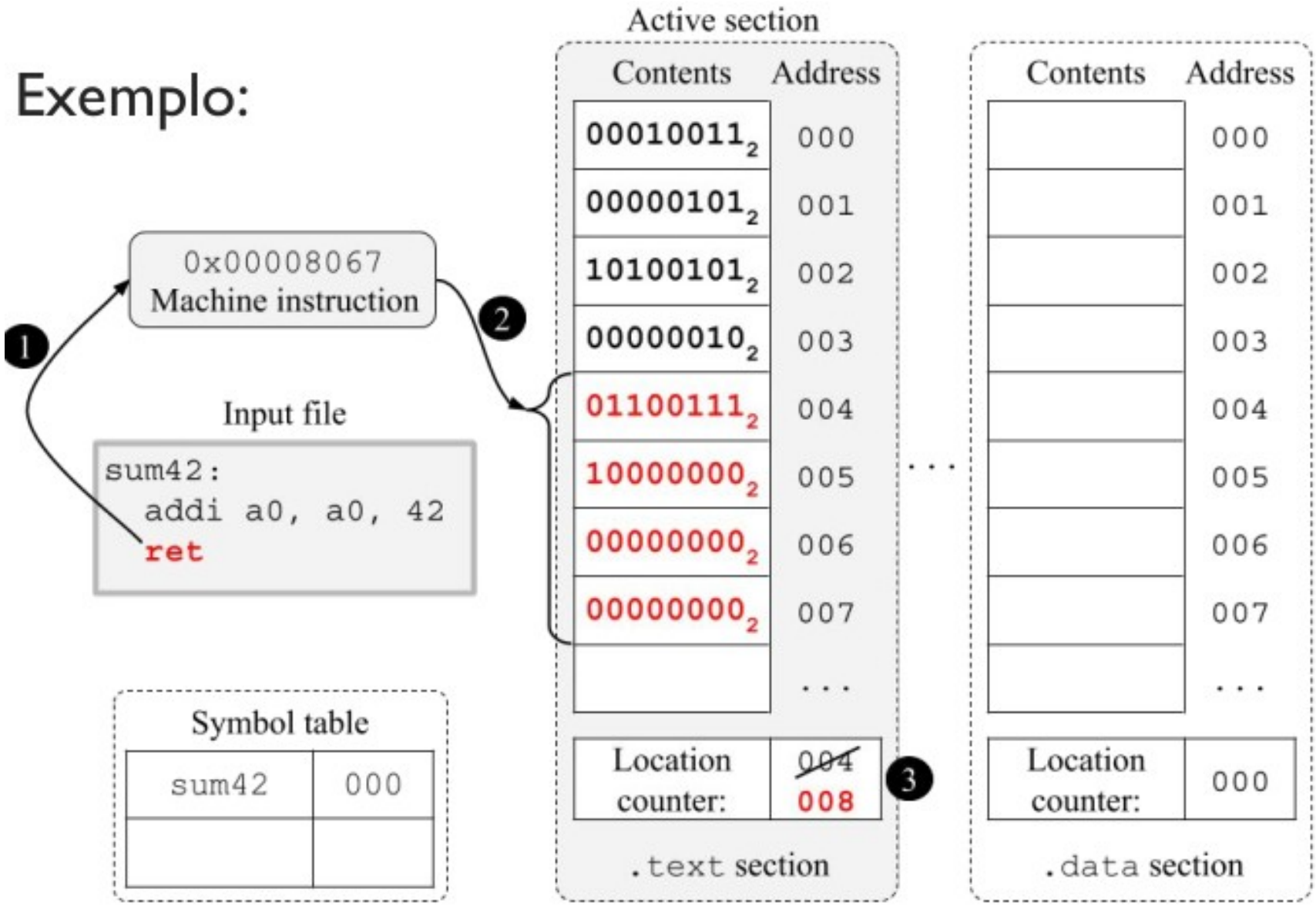
Exemplo:



- **Contador de localização**



- **Contador de localização**



RISC-V – Linguagem de montagem – Diretivas de montagem

- **Diretivas de montagem**
  - São comandos para controlar o processo de montagem
    - Interpretados pelo montador durante o processo de montagem!

  - Exemplo: a diretiva **“.byte 45”** instrui o montador a colocar um *byte* com valor 45 no programa

- **Inserção de valores**

Directive	Arguments	Description
`.string`	string	Emit NULL terminated string
`.asciz`	string	Emit NULL terminated string (alias for .string)
`.ascii`	string	Emit string without NULL character
`.byte`	expression [, expression]*	Emit one or more 8-bit comma separated words
`.half`	expression [, expression]*	Emit one or more 16-bit comma separated words
`.word`	expression [, expression]*	Emit one or more 32-bit comma separated words
`.dword`	expression [, expression]*	Emit one or more 64-bit comma separated words

- **Inserção de valores**

  .string e **.asciz** : adicionam uma string codificada em ASCII e terminada em zero no ponto atual de montagem do programa.

  O exemplo abaixo adiciona 3 bytes no programa, sendo os bytes 111 (“o”), 105 (“i”) e o byte com valor 0:

  .string “oi”

- **Inserção de valores**

  .ascii : adiciona uma string codificada em ASCII no ponto atual de montagem do programa, sem o byte 0 no final.

  O exemplo abaixo adiciona 2 bytes no programa, sendo os bytes 111 (“o”) e 105 (“i”), somente:

  .ascii “oi”

RISC-V – Linguagem de montagem – Diretivas de montagem

- **Inserção de valores**

  .byte : adiciona um ou mais bytes no ponto atual de montagem do programa.

  O exemplo abaixo adiciona 3 bytes no programa, com os valores 10, 20 e 30:

  .byte 10, 20, 30

RISC-V – Linguagem de montagem – Diretivas de montagem

- **Inserção de valores**

  .half, **.word** e **.dword** : adicionam um ou mais valores de 16, 32 e 64 bits, respectivamente, no ponto atual de montagem do programa.

  O exemplo abaixo adiciona dois valores de 32 bits (4 bytes) no programa:

  .word 20, 30

- **Inserção de valores**

Directive	Arguments	Description
.string	string	Emit NULL terminated string
.asciz	string	Emit NULL terminated string (alias for .string)
.ascii	string	Emit string without NULL character
.byte	expression [, expression]*	Emit one or more 8-bit comma separated words
.half	expression [, expression]*	Emit one or more 16-bit comma separated words
.word	expression [, expression]*	Emit one or more 32-bit comma separated words
.dword	expression [, expression]*	Emit one or more 64-bit comma separated words

- Quando combinadas com rótulos, podem ser usadas para declarar e inicializar variáveis globais

```
# Exemplo

x: .word 12   # variável x iniciada com valor 12 (4 bytes)
y: .byte 12   # variável y iniciada com valor 12 (1 byte)
msg: .string “ACII”   # variável msg com string “ACII”
```