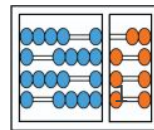


MC404AE - Organização Básica de Computadores e Ling. Montagem

Arquivos executáveis, objetos e programas em
linguagem de montagem

Prof. Allan M. de Souza



Codificação de programas

— — — —

- Programas podem ser codificados de diversas formas.

Ex: programa nativo, programa fonte, script, java bytecode, ...;

- **Programa nativo:** programas que podem ser executados diretamente pela CPU do computador, ou seja, que são codificados com instruções que a CPU entende!

Agenda

— — — —

- Geração de programas nativos
- Rótulos e símbolos
- Referências e relocação
- Símbolos globais vs locais
- Ponto de entrada
- Organização do programa em seções
- Arquivos objetos vs executáveis

Geração de programas nativos

— — — —

- **Compilador:** Converte programas de ling. de alto nível para linguagem de montagem.
 - Programas em alto nível e em linguagem de montagem são representados em arquivos texto!
- **Montador:** Converte programas em ling. de montagem para linguagem de máquina.
 - Produz um arquivo objeto (.o)
 - Arquivos objeto possuem código em ling. de máquina e são codificados de forma binária!
- **Ligador:** Liga o código de vários arquivos objeto (.o) e produz um arquivo executável.

Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

Arquivo main.c contém a função main(),
que chama a função func().

```
int func(int a) {  
    return 42+a;  
}
```

func.c

Arquivo func.c contém a implementação
da função func();

Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

```
int func(int a) {  
    return 42+a;  
}
```

func.c

chama clang para gerar o código em linguagem de montagem para **main.c**

```
$clang-15 --target=riscv32 -march=rv32g -mabi=ilp32d -mno-relax main.c -S -o main.s
```

Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

```
int func(int a) {  
    return 42+a;  
}
```

func.c

flag **-target** para compilar
programa para RISC-V

Flag **-march** para gerar código para
RV32g (versão de 32bits do RISC-V)

```
$clang-15 --target=riscv32 -march=rv32g -mabi=ilp32d -mno-relax main.c -S -o main.s
```

Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

```
int func(int a) {  
    return 42+a;  
}
```

func.c

Flag **-S** informa ao GCC para gerar código em linguagem de montagem (assembly).

```
$clang-15 --target=riscv32 -march=rv32g -mabi=ilp32d -mno-relax main.c -S -o main.s
```


Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

```
int func(int a) {  
    return 42+a;  
}
```

func.c

Flag **-mno-relax** bloqueia otimizações
que seriam realizadas pelo montador

Flag **-o** especifica o nome do
arquivo de saída a ser produzido.

```
$clang-15 --target=riscv32 -march=rv32g -mabi=ilp32d -mno-relax main.c -S -o main.s
```

Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

```
7f 45 4c 46 01 01 01  
00 00 00 00 00 00 00  
00 00 01 00 f3 00 01  
00 00 00 00 00 00 00  
...
```

main.o

```
int func(int a) {  
    return 42+a;  
}
```

func.c

chama o programa para montar o
código em linguagem de montagem

```
$clang-15 --target=riscv32 -march=rv32g -mabi=ilp32d -mno-relax main.s -c -o main.o
```

Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

```
7f 45 4c 46 01 01 01  
00 00 00 00 00 00 00  
00 00 01 00 f3 00 01  
00 00 00 00 00 00 00  
...
```

main.o

```
int func(int a) {  
    return 42+a;  
}
```

func.c

```
.globl func  
func:  
    addi a0, a0, 42  
    ret
```

func.s

```
$clang-15 --target=riscv32 -march=rv32g -mabi=ilp32d -mno-relax func.c -S -o func.s
```

Geração de programas nativos

Exemplo

```
int main() {  
    return func(10);  
}
```

main.c

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

```
7f 45 4c 46 01 01 01  
00 00 00 00 00 00 00  
00 00 01 00 f3 00 01  
00 00 00 00 00 00 00  
...
```

main.o

```
int func(int a) {  
    return 42+a;  
}
```

func.c

```
.globl func  
func:  
    addi a0, a0, 42  
    ret
```

func.s

```
7f 45 4c 46 01 01 01  
00 00 00 00 00 00 00  
00 00 01 00 f3 00 01  
00 00 00 00 00 00 00  
...
```

func.o

```
7f 45 4c 46 01 01 01  
00 00 00 00 00 00 00  
00 00 01 00 f3 00 01  
00 00 00 00 00 00 00  
...
```

main.x

```
$ld.lld main.o func.o -o main.x
```

chama o programa ligar o código dos dois arquivos objetos

Geração de programas nativos

- **Linguagem de montagem**

- Linguagem simbólica => texto
- Em inglês: Assembly language

- **Arquivos texto:**

- sequências de bytes que representam caracteres;
- o valor de cada byte depende do padrão de codificação (p.ex: UTF-8 ou ASCII).
- o programa "hexdump" pode ser usado para imprimir os valores dos bytes de um arquivo qualquer.

```
$ hexdump main.s
00000000 6d 61 69 6e 3a 0a 20 20 6c 69 20 61 30 2c 20 31
00000010 30 0a 20 20 6a 61 6c 20 66 75 6e 63 0a 20 20 72
00000020 65 74 0a
00000023
```

main:

```
li a0, 10
jal func
ret
```

main.s

Geração de programas nativos

- **Linguagem de montagem**

- Linguagem simbólica => texto
- Em inglês: Assembly language

- **Arquivos texto:**

- os comandos "cat" ou "less" do Linux podem ser usados para inspecionar o conteúdo de arquivos texto.
- Eles mostram os caracteres em vez dos valores de cada byte.

```
$ cat main.s  
main:  
    li a0, 10  
    jal func  
    ret
```

```
main:  
    li a0, 10  
    jal func  
    ret
```

main.s

Geração de programas nativos

— — — —

- **Linguagem de máquina**

- Sequência de bits que a CPU entende (arquivo binário)
- Em inglês: Machine language

- **Montador**

- Ferramenta que converte programa em linguagem de montagem para linguagem de máquina
- Em inglês: Assembler ○ Código em linguagem de máquina produzido pelo montador é armazenado em um **arquivo objeto**!

Geração de programas nativos

Arquivo objeto

- Diversos formatos:
 - Unix/Linux:
 - a.out
 - COFF: Common Object File Format
 - **ELF: Executable and Linking Format**
 - Windows:
 - PE: Portable Executable
- Todos estes formatos podem ser usados para codificar tanto arquivos objeto quanto arquivos executáveis!
- Arquivo objeto e executáveis são arquivos binários!

Geração de programas nativos

- Linguagem de máquina
 - Sequência de bits que a CPU entende (arquivo binário)
 - Em inglês: Machine language
- Arquivos binários
 - Ferramentas especiais podem ser usadas para decodificar a informação de arquivos binários e mostrá-las de forma fácil de se visualizar.
 - a ferramenta "objdump" decodifica informações de arquivos objeto e mostra de forma textual!

```
$objdump -d main.o
main.o: file format elf32-littleriscv
Disassembly of section .text:
00000000 <main>:
    0: 00a00513 li a0,10
    4: ffdff0ef jal ra,0 <main>
    8: 00008067 ret
```

```
7f 45 4c 46 01 01 01
00 00 00 00 00 00 00
00 00 01 00 f3 00 01
00 00 00 00 00 00 00
...
```

main.o

Geração de programas nativos

- **objdump:** decodifica arquivos objetos e executáveis e mostra informação de forma amigável (textual)

```
$objdump -d main.o
main.o: file format elf32-littleriscv
Disassembly of section .text:
00000000 <main>:
    0: 00a00513 li a0,10
    4: ffdff0ef jal ra,0 <main>
    8: 00008067 ret
```

```
$objdump -d func.o
func.o: file format elf32-littleriscv
Disassembly of section .text:
00000000 <func>:
    0: 02a50513 addi a0,a0,42
    4: 00008067 ret
```

Geração de programas nativos

- **objdump:** decodifica arquivos objetos e executáveis e mostra informação de forma amigável (textual)

```
$objdump -d main.x  
main.x: file format elf32-littleriscv
```

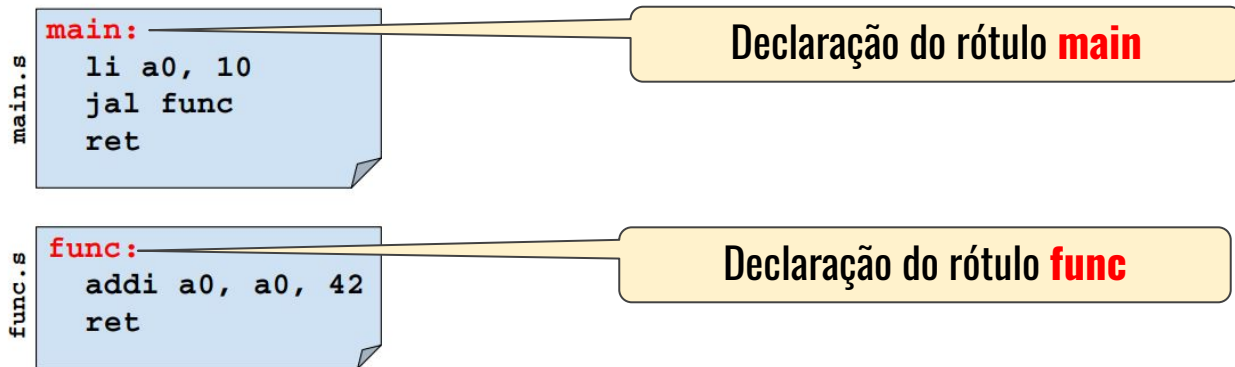
```
Disassembly of section .text:
```

```
00010054 <main>:  
    10054: 00a00513 li a0,10  
    10058: 008000ef jal ra,10060 <func>  
    1005c: 00008067 ret  
  
00010060 <func>:  
    10060: 02a50513 addi a0,a0,42  
    10064: 00008067 ret
```

Rótulos e símbolos

Rótulos e símbolos

- Rótulos são marcadores que representam posições no programa, i.e, **posições de memória**;
- São convertidos para endereços pelo montador e pelo ligador; Ex: rótulos **main** e **func**



Rótulos e símbolos

- Rótulos são marcadores que representam posições no programa, i.e, **posições de memória**;
- São convertidos para endereços pelo montador e pelo ligador; Ex: rótulos **main** e **func**

main.s

```
main:
    li a0, 10
    jal func
    ret
```

func.s

```
func:
    addi a0, a0, 42
    ret
```

referência para o rótulo **func**

Rótulos e símbolos

- Rótulos são marcadores que representam posições no programa, i.e, **posições de memória**;
- São convertidos para endereços pelo montador e pelo ligador; Ex: rótulos **main** e **func**

main.s

```
main:
    li a0, 10
    jal func
    ret
```

func.s

```
func:
    addi a0, a0, 42
    ret
```

```
00010054 <main>:
    10054: 00a00513 li    a0,10
    10058: 008000ef jal   ra,10060 <func>
    1005c: 00008067 ret

00010060 <func>:
    10060: 02a50513 addi a0,a0,42
    10064: 00008067 ret
```

Montador e ligador
convertem rótulos em
endereços e ajustam as
referências

Rótulos e símbolos

- Rótulos são geralmente usados para demarcar a posição inicial de variáveis globais e de rotinas do programa. ligador;
- Ex: Rotinas **int main();** e **int func(int);**

main.s

```
main:
    li a0, 10
    jal func
    ret
```

func.s

```
func:
    addi a0, a0, 42
    ret
```

```
00010054 <main>:
    10054: 00a00513 li    a0,10
    10058: 008000ef jal   ra,10060 <func>
    1005c: 00008067 ret

00010060 <func>:
    10060: 02a50513 addi  a0,a0,42
    10064: 00008067 ret
```

Rótulos e símbolos

- Símbolos são "**nomes**" que são associados a valores numéricos.
- A **tabela de símbolos** é a estrutura de dados que **mapeia** os **nomes** dos símbolos nos **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

Rótulos e símbolos

A tabela de símbolos pode ser visualizada com a ferramenta objdump (opção -t)

```
objdump -t main.x
main.x: file format elf32-littleriscv
SYMBOL TABLE:
00010054 l d .text 00000000 .text
...
00010054 l .text 00000000 main
00011068 g .text 00000000 __bss_start
00010060 g .text 00000000 func
00011068 g .text 00000000 __DATA_BEGIN__
00011068 g .text 00000000 _edata
00011068 g .text 00000000 _end
```

Símbolos main e func. Criados automaticamente pelo montador para os rótulos "main:" e "func:"

Referências e relocação

Referências e relocação

- Instruções ou diretivas de um programa em linguagem de montagem podem **referenciar** símbolos pelo **nome**
 - O montador e o ligador substituem as referências pelo valor do símbolo

main.s

```
main:
    li a0, 10
    jal func
    ret
```

func.s

```
func:
    addi a0, a0, 42
    ret
```

referência para o símbolo **func**

Referências e relocação

- Instruções ou diretivas de um programa em linguagem de montagem podem referenciar símbolos pelo nome
 - **O montador e o ligador substituem as referências pelo valor do símbolo**

main.s

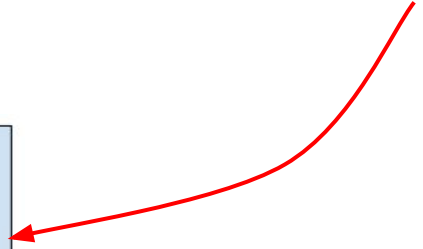
```
main:
    li a0, 10
    jal func
    ret
```

func.s

```
func:
    addi a0, a0, 42
    ret
```

```
00010054 <main>:
    10054: 00a00513 li    a0,10
    10058: 008000ef jal   ra,10060 <func>
    1005c: 00008067 ret

00010060 <func>:
    10060: 02a50513 addi  a0,a0,42
    10064: 00008067 ret
```



Referências e relocação

O montador e o ligador substituem as referências pelo valor do símbolo. **Como?**

Referências e relocação

O montador e o ligador substituem as referências pelo valor do símbolo. **Como?**

- A **tabela de relocação** contém informações sobre itens do programa (instruções e dados) que referenciam símbolos e necessitam ajustes quando os valores dos símbolos mudam.
- Para cada referência, o montador gera uma entrada na tabela de relocação e, ao final, gera o arquivo objeto.
- O ligador usa a tabela de relocação para ajustar as referências!

Relocação: é o processo de re-associar endereços às instruções e dados do programa.

- O ligador realiza a relocação quando está ligando múltiplos arquivos objetos
- Como consequência, as referências para símbolos que estão associados a endereços devem ser ajustadas

Referências e relocação

Podemos inspecionar a tabela de relocação com a ferramenta objdump (opção -r)

```
main.s  
main:  
    li a0, 10  
    jal func  
    ret
```

```
$objdump -r main.o  
main.o: file format elf32-littleriscv  
  
RELOCATION RECORDS FOR [.text]:  
  
OFFSET      TYPE          VALUE  
00000004 R_RISCV_JAL func
```

Referências e relocação

Referências não definidas são referências para símbolos que não foram definidos.

- Exemplo: referência para o símbolo `func` em `"main.s"`

```
main.s  
main:  
    li a0, 10  
    jal func  
    ret
```

```
$objdump -t main.o  
main.o: file format elf32-littleriscv  
SYMBOL TABLE:  
00000000 l    d    .text    00000000 .text  
00000000 l    d    .data    00000000 .data  
00000000 l    d    .bss     00000000 .bss  
00000000 l          .text    00000000 main  
...  
00000000          *UND*    00000000 func
```

Referências e relocação

Referências não definidas são referências para símbolos que não foram definidos.

- O ligador tenta resolver referências não definidas olhando para símbolos em outros arquivos objetos.
- Exemplo: procura pelo símbolo `_start` durante a ligação.
 - Se não encontrar o símbolo, o ligador interrompe o processo de ligação e emite um erro.

Nesse caso, o erro aconteceu porque tentamos realizar a ligação do arquivo **main.o** sem o arquivo **func.o**

```
$ld.lld main.o -o main.x
warning: cannot find entry symbol _start; defaulting to 0000000000010054
main.o: in function `main': (.text+0x4): undefined reference to `func'
```

Símbolos globais vs locais

Símbolos globais vs locais

Os símbolos do programa são classificados como globais ou locais.

- **Locais:** apenas visíveis dentro do mesmo arquivo.
- **Globais:** visíveis externamente => Usados pelo ligador para resolver referências não definidas.

Por padrão os símbolos são locais.

- A diretiva **".globl nome"** transforma o símbolo nome em global.

se o símbolo func não for declarado como global o ligador não conseguirá resolver a referência a este símbolo no arquivo **main.s**

func.s

```
.globl func  
func:  
    addi a0, a0, 42  
    ret
```

Símbolos globais vs locais

Exemplo: Arquivo "func.s" sem a diretiva **.globl**

```
$ cat main.s
```

```
main:  
    li a0, 10  
    jal func  
    ret
```

```
$ cat func.s
```

```
func:  
    addi a0, a0, 42  
    ret
```

```
$ld.lld main.o -o main.x
```

```
warning: cannot find entry symbol _start; defaulting to 0000000000010054  
main.o: in function `main': (.text+0x4): undefined reference to `func'
```

Símbolos globais vs locais

Exemplo: Arquivo "func.s" com a diretiva **.globl**

```
$ cat main.s
main:
    li a0, 10
    jal func
    ret

$ cat func.s
.globl func
func:
    addi a0, a0, 42
    ret
```

```
$ld.lld main.o -o main.x
warning: cannot find entry symbol _start; defaulting to 0000000000010054
```

Ponto de entrada

Ponto de entrada do programa

O ponto de entrada é o endereço da primeira instrução que deve ser executada quando um programa é iniciado

- Este endereço é gravado pelo ligador em um campo no cabeçalho do arquivo executável
- Para executar uma aplicação, o sistema operacional carrega o código e os dados da aplicação na memória principal e executa uma instrução de salto para o ponto de entrada do programa.

Ponto de entrada do programa

Podemos inspecionar o **cabeçalho** de um arquivo executável **ELF** com a ferramenta **readelf**.

```
$ riscv64-unknown-elf-readelf -h main.x
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                   EXEC (Executable file)
  Machine:                               RISC-V
  Version:                               0x1
  Entry point address:                   0x10054
  Start of program headers:              52 (bytes into file)
  Start of section headers:              516 (bytes into file)
  ...
```

Endereço de entrada 0x10054

Ponto de entrada do programa

Este endereço é gravado pelo ligador em um campo no cabeçalho do arquivo executável!

Como o ligador sabe qual é o ponto de entrada?

- Ele procura pelo símbolo **_start**, se não encontrar, então ele ajusta o ponto de entrada como sendo o endereço da primeira instrução que ele colocou no programa executável.

Ponto de entrada do programa

Este endereço é gravado pelo ligador em um campo no cabeçalho do arquivo executável!

```
$ cat main.s
```

```
main:  
    li a0, 10  
    jal func  
    ret
```

```
$ cat func.s
```

```
.globl func  
func:  
    addi a0, a0, 42  
    ret
```

```
$ld.lld main.o -o main.x (sem _start)
```

```
warning: cannot find entry symbol _start; defaulting to 0000000000010054
```

E se trocarmos o rótulo main por **start**?

Ponto de entrada do programa

Este endereço é gravado pelo ligador em um campo no cabeçalho do arquivo executável!

```
$ cat main.s
_start:
    li a0, 10
    jal func
    ret

$ cat func.s
.globl func
func:
    addi a0, a0, 42
    ret
```

```
$ld.lld main.o -o main.x (sem _start)
warning: cannot find entry symbol _start; defaulting to 0000000000010054
```

```
$ld.lld main.o -o main.x (com _start)
warning: cannot find entry symbol _start; defaulting to 0000000000010054
```

Ainda continuamos com erro, pois **_start** precisa ser um **rótulo global**

Ponto de entrada do programa

Este endereço é gravado pelo ligador em um campo no cabeçalho do arquivo executável!

```
$ cat main.s
.globl _start
_start:
    li a0, 10
    jal func
    ret

$ cat func.s
.globl func
func:
    addi a0, a0, 42
    ret
```

```
$ld.lld main.o -o main.x (sem _start)
warning: cannot find entry symbol _start; defaulting to 0000000000010054
```

```
$ld.lld main.o -o main.x (com _start)
warning: cannot find entry symbol _start; defaulting to 0000000000010054
```

```
$ld.lld main.o -o main.x (com _start global)
$
```

Nenhum erro ocorre! 😊

Organização do programa em seções

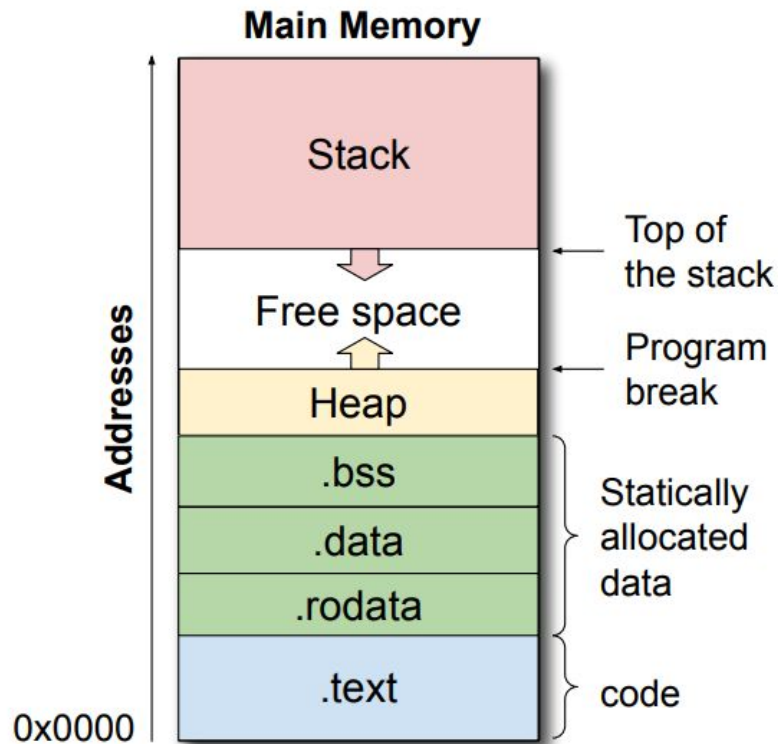
Organização do programa em seções

Programas executáveis são organizados em seções.

- Seção **".text"**: dedicada ao armazenamento do código do programa (as instruções)
- Seção **".data"**: dedicada ao armazenamento das variáveis globais inicializadas
- Seção **".bss"**: dedicada ao armazenamento das variáveis globais não inicializadas
- Seção **".rodata"**: dedicada ao armazenamento de constantes (ready-only).

Organização do programa em seções

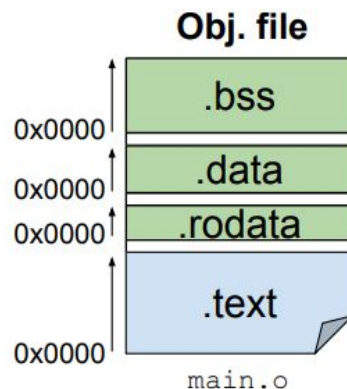
Programas executáveis são organizados em seções.



Organização do programa em seções

Arquivos objeto também são organizados em seções.

- O montador produz um **arquivo objeto** com uma ou mais seções.
- Cada seção tem seu próprio espaço de endereçamento e se inicia no endereço 0.



Organização do programa em seções

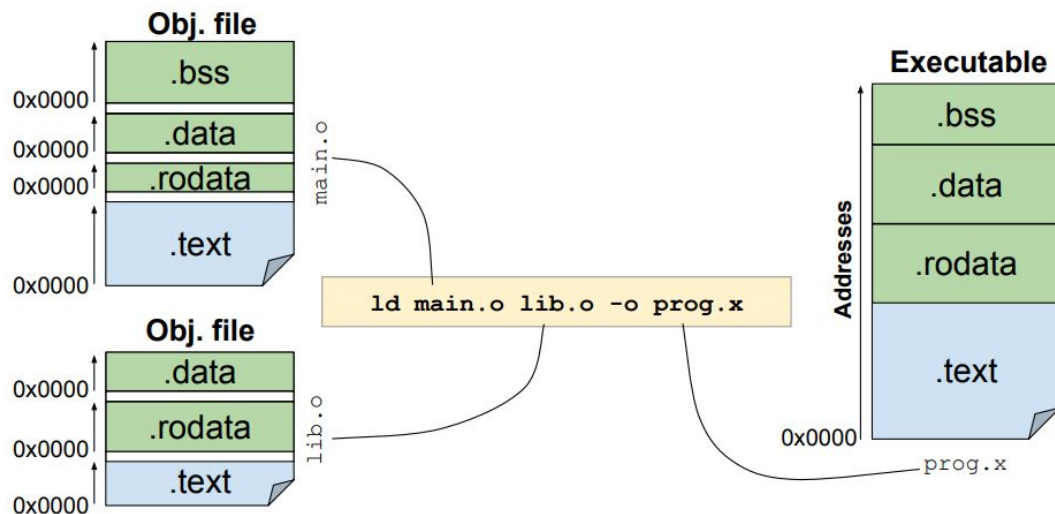
O arquivo objeto é dividido em partes e possui:

- **Um cabeçalho:** que indica o tamanho e a posição das partes restantes do arquivo;
- **Conjunto de seções.** Por exemplo:
 - Seção `.text`: contém o código em ling. de máquina.
 - Seção `.data`: contém variáveis globais inicializadas.
- **Tabela de símbolos:** lista de símbolos definidos e não resolvidos do arquivo.
- **Tabela de relocação:** lista de referências para símbolos.

Organização do programa em seções

0 Ligador:

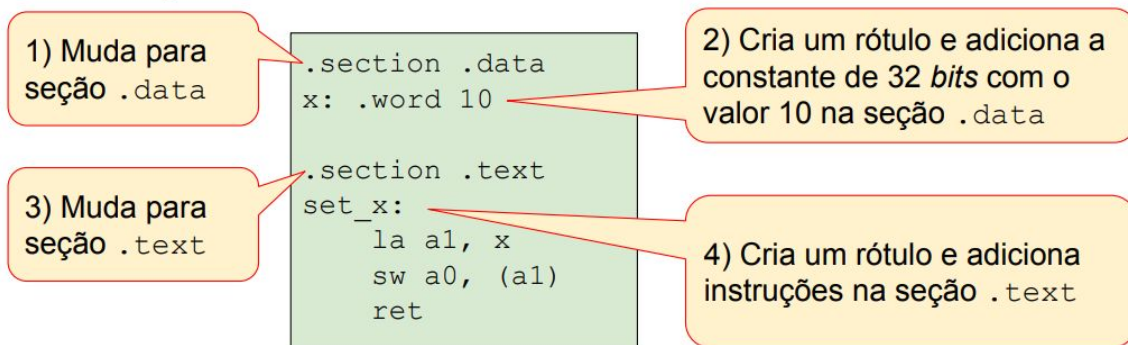
- agrupa as seções,
- ajusta (reloca) as referências a símbolos
 - (p.ex: campos de instruções que referenciam símbolos)
- liga os símbolos exportados e não definidos.



Diretiva `.section`

A diretiva `".section sec"` instrui o montador a mudar para a seção `sec`.

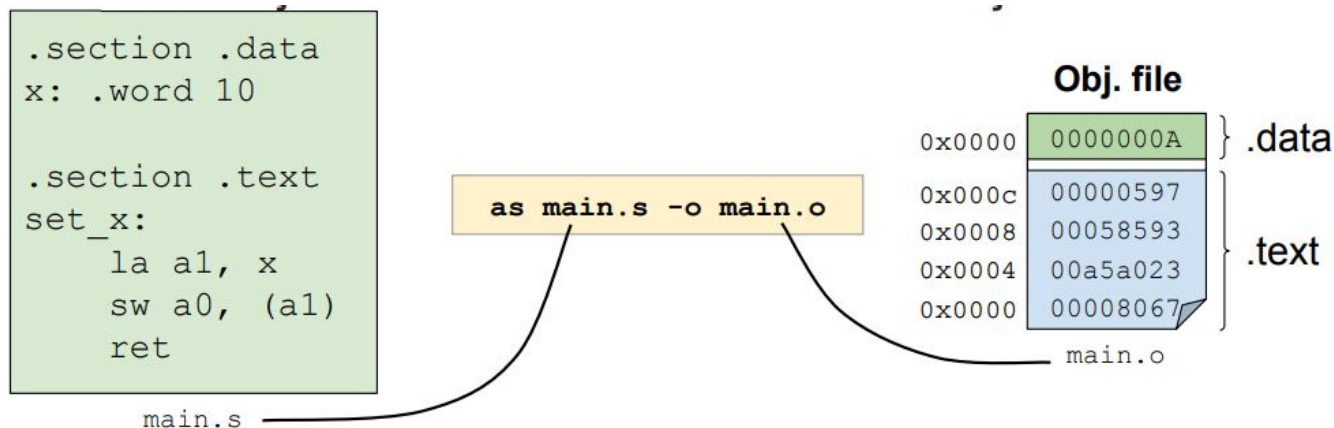
- A partir deste ponto, o conteúdo montado (instruções e dados) é colocado na seção `sec`



Diretiva .section

O arquivo objeto produzido pelo montador contém as seções definidas pela diretiva .section.

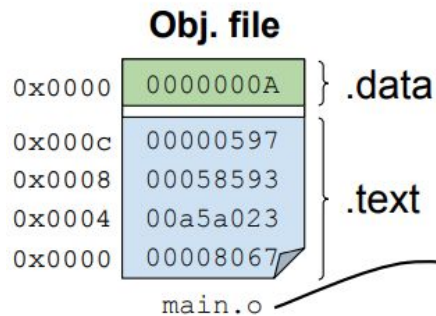
- Cada seção tem seu próprio espaço de endereçamento e se inicia no endereço 0.



Diretiva .section

O desmontador (objdump) pode ser usado para inspecionar o conteúdo das seções do arquivo objeto produzido pelo montador.

Exemplo:



objdump -D main.o

Disassembly of section .text:

```
00000000 <set_x>:  
0: 00000597 auipc a1,0x0  
4: 00058593 mv a1,a1  
8: 00a5a023 sw a0,0(a1) # 0 <set_x>  
c: 00008067 ret
```

Disassembly of section .data:

```
00000000 <x>:  
0: 000a  
...
```

Arquivos objetos vs executáveis

Sistemas Linux geralmente usam o formato **ELF** para armazenar tanto arquivos objeto quanto arquivos executáveis. No entanto:

- Arquivos objeto não possuem ponto de entrada.
- Endereços de código e variáveis em arquivos objeto são geralmente modificados pelo ligador. (**não são end. finais**)

Arquivos objetos vs executáveis

Sistemas Linux geralmente usam o formato ELF para armazenar tanto arquivos objeto quanto arquivos executáveis. No entanto:

- Arquivos objeto podem ter múltiplas referências para símbolos não definidos.
 - Espera-se que o ligador resolva estas referências durante o processo de ligação.
- O ligador geralmente não produz arquivos executáveis com referências não definidas.

Próxima aula

Linguagem de montagem para a arquitetura RV32I

- Sintaxe da linguagem de montagem
- Comentários
- Instruções de montagem
- Valores imediatos e símbolos
- Rótulos
- Contador de localização
- Diretivas de montagem