INTRODUCTION TO RISC-V

RISC-V入门教程

ISA | 汇编指令 | 系统编程 | 组成原理 | 嵌入式应用

汇编过程

主讲 邢建国

中国开放指令生态（RISC-V）联盟 | 浙江中心
China RICS-V Alliance | Zhejiang Center

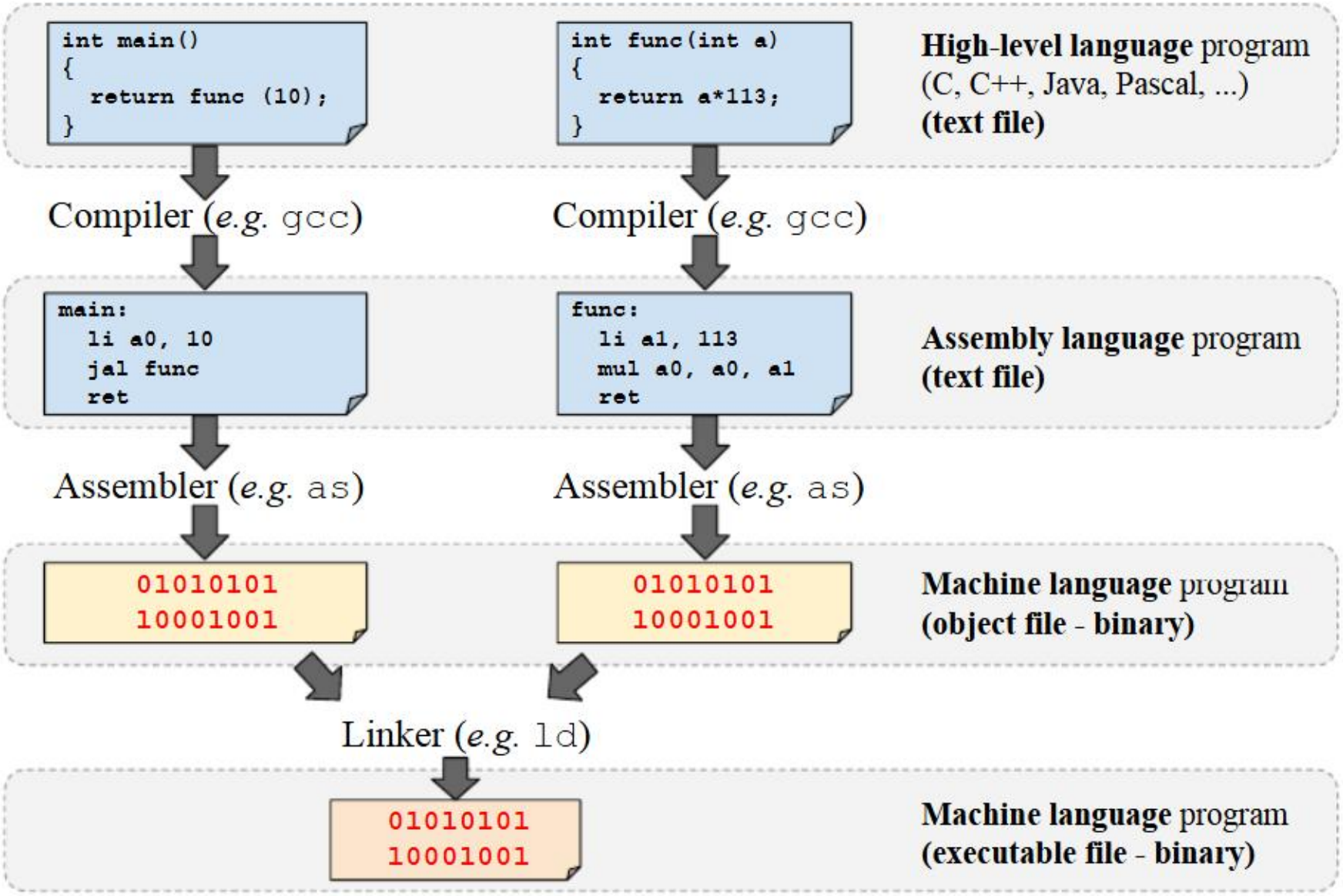
浙江图灵算力研究院
ZHEJIANG TURING INSTITUTE

RISC-V®

汇编过程

- 生成本地代码
- 标号、符号、符号表
- 节(section)
- 目标代码与可执行程序

生成本地程序


int main()
{
return func (10);
}
int func(int a)
{
return a*113;
}
High-level language program
(C, C++, Java, Pascal, ...)
(text file)
Compiler (e.g. gcc)
Compiler (e.g. gcc)
main:
li a0, 10
jal func
ret
func:
li a1, 113
mul a0, a0, a1
ret
Assembly language program
(text file)
Assembler (e.g. as)
Assembler (e.g. as)
01010101
10001001
01010101
10001001
Machine language program
(object file - binary)
Linker (e.g. ld)
01010101
10001001
Machine language program
(executable file - binary)

生成本地程序

main.c

```
1 int main()
2 {
3   int r = func (10)
4   return r+1;
5 }
```

(1) gcc

main.s

```
1   .text
2   .align        2
3 main:
4   addi sp,sp,-16
5   li   a0,10
6   sw   ra,12(sp)
7   jal  func
8   lw   ra,12(sp)
9   addi a0,a0,1
10  addi sp,sp,16
11  ret
```

riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S main.c -o main.s

生成本地程序

main.s

```
1    .text
2    .align        2
3  main:
4    addi sp,sp,-16
5    li   a0,10
6    sw   ra,12(sp)
7    jal  func
8    lw   ra,12(sp)
9    addi a0,a0,1
10   addi sp,sp,16
11   ret
```

（2）as

目标代码：main.o

riscv64-unknown-elf-as -mabi=ilp32 -march=rv32i main.s -o main.o

生成本地程序

目标代码：main.o、mylib.o

（3）ld

执行文件：main

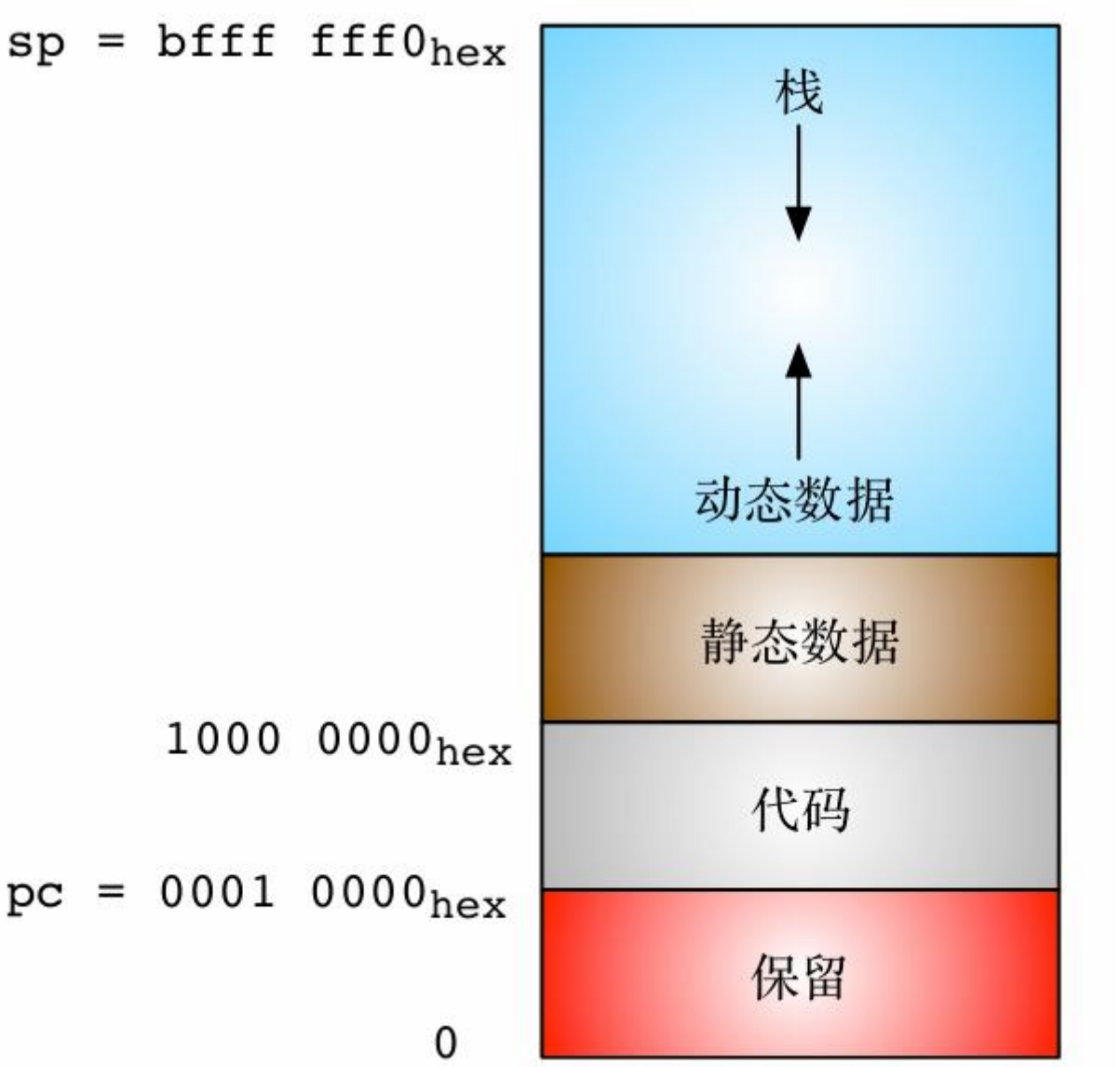
```
riscv64-unknown-elf-ld -m elf32lriscv main.o mylib.o -o main
```

生成本地程序

源文件：main.c func.c  目标文件：main.o func.o  执行文件：main.x

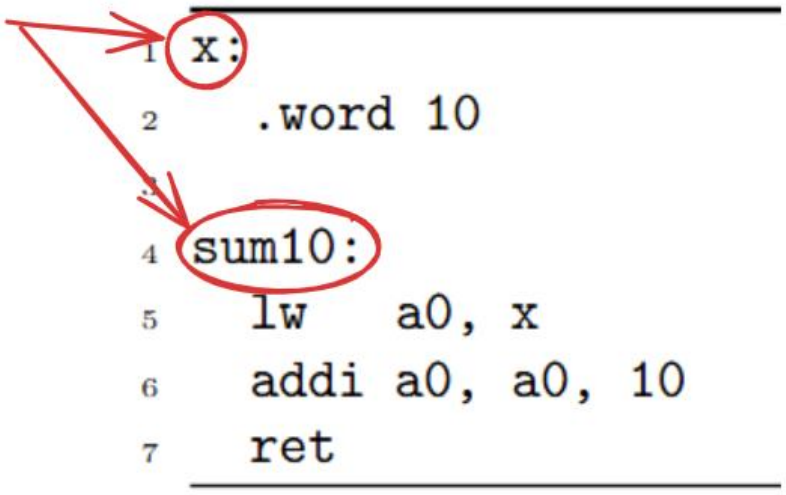
```
riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S main.c -o main.s
riscv64-unknown-elf-as -mabi=ilp32 -march=rv32i main.s -o main.o
riscv64-unknown-elf-gcc -mabi=ilp32 -march=rv32i -S func.c -o func.s
riscv64-unknown-elf-as -mabi=ilp32 -march=rv32i func.s -o func.o
riscv64-unknown-elf-ld -m elf32lriscv main.o func.o -o main.x
```

内存布局


sp = bfff fff0hex
栈
动态数据
静态数据
1000 0000hex
代码
pc = 0001 0000hex
保留
0

■ 汇编程序中的标号(label)、符号(symbol)

标号是代表程序位置的“标记”



```
1  x:
2      .word 10
3
4  sum10:
5      lw   a0, x
6      addi a0, a0, 10
7      ret
```

符号是与数值相关联的“名称”

标号由汇编器自动转换为符号，并与表示其在程序中位置的数值相关联，该数值是内存地址。

符号(symbol)、符号表(symbol table)

“符号表”是将每个程序和符号映射到其值的数据结构。符号表在目标文件中

查看符号表：riscv64-unknown-elf-nm sum10.o

```
$ riscv64-unknown-elf-nm  sum10.o
00000004 t .L0
00000004 t sum10
00000000 t x
```

标号由汇编器自动转换为符号，并与表示其在程序中位置的数值相关联，该数值是内存地址。

程序员也可以使用指令.set显式定义符号

```
1 .set answer, 42
2 get_answer:
3   li a0, answer
4   ret
```

```
$ riscv64-unknown-elf-as -march=rv32im get_answer.s -o get_answer.o
$ riscv64-unknown-elf-nm  get_answer.o
0000002a a answer
00000000 t get_answer
```

引用标号和重定位Relocation

在汇编和链接过程中，对标号的每个引用都必须替换为地址。

```
1 trunk42:
2   li    t1, 42
3   bge   t1, a0, done
4   mv    a0, t1
5 done:
6   ret
```

当汇编这个程序时，汇编程序将每个汇编指令翻译成一条4个字节大小的机器指令。因此，程序总共占用16个内存字。此外，汇编程序将第一条指令映射到地址0，将第二条指令映射到地址4，依此类推。标号trunk42代表地址0，done代表地址12.

```
$ riscv64-unknown-elf-objdump -D trunk.o

trunk.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <trunk42>:
   0: 02a00313          li t1,42
   4: 00a35463          bge t1,a0,c <done>
   8: 00030513          mv a0,t1

0000000c <done>:
   c: 00008067          ret
```

引用标号和重定位Relocation

重定位是为代码和数据分配新内存地址的过程。

在重定位过程中，链接器需要调整代码和数据以反映新地址，即调整符号表上与标号关联的地址以及对标号的引用。重定位表是一种数据结构，其中包含描述需要如何修改程序指令和数据以反映地址重新分配的信息。每个目标文件都包含一个重定位表，链接器在执行重定位过程时使用它们来调整代码。

```
$ riscv64-unknown-elf-objdump -r trunk.o

trunk.o:     file format elf32-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_RISCV_BRANCH    done
```

```
$ riscv64-unknown-elf-ld -m elf32lriscv trunk.o -o trunk.x
$ riscv64-unknown-elf-objdump -D trunk.x

trunk.x:     file format elf32-littleriscv

Disassembly of section .text:

00010054 <trunk42>:
   10054: 02a00313          li t1,42
   10058: 00a35463          bge t1,a0,10060 <done>
   1005c: 00030513          mv a0,t1

00010060 <done>:
   10060: 00008067          ret
```

未定义引用

汇编代码依赖于标号来引用程序位置。在某些情况下，汇编代码引用了不在在同一文件中定义的标号。这在调用在另一个文件上实现的例程或访问在另一个文件上声明的全局变量时很常见。

```
1 # Contents of the main.s file
2 start:
3   li   a0, 10
4   li   a1, 20
5   jal exit
```

```
$ riscv64-unknown-elf-nm main.o
00000000 t start
         U exit
```

链接目标文件时，链接器必须解析未定义的符号，即它必须找到符号定义并使用符号值调整符号表和代码。
在上面的示例中，链接器将查找符号exit，以便它可以调整指令以引用正确的地址。如果找不到符号的定义，它会停止链接过程并发出错误消息。

```
$ riscv64-unknown-elf-ld -m elf32lriscv main.o -o main.x
riscv64-unknown-elf-ld: warning: cannot find entry symbol start; ...
riscv64-unknown-elf-ld: main.o: in function ‘start’:
(.text+0x8): undefined reference to ‘exit’
```

全局与本地符号

符号被分类为本地或全局符号。
本地符号仅在同一文件上可见，链接器不使用它们来解析其他文件上的未定义引用。
链接器使用全局符号来解析其他文件上的未定义引用。
默认情况下，汇编器将标号注册为本地符号。指令. global是一个汇编指令，指示汇编器将标号注册为全局符号。

```
1 # Contents of the exit.s file
2 .globl exit
3 exit:
4   li a0, 0
5   li a7, 93
6   ecall
```

```
$ riscv64-unknown-elf-as -march=rv32im main.s -o main.o
$ riscv64-unknown-elf-as -march=rv32im exit.s -o exit.o
$ riscv64-unknown-elf-ld -m elf32lriscv main.o exit.o -o main.x
riscv64-unknown-elf-ld: warning: cannot find entry symbol start; ...
```

链接器松弛选项（linker relax）

“跳转并链接”指令的PC相对地址字段有20位，因此一条指令能跳得很远。

尽管编译器为每个外部函数的引用都生成两条指令，但很多时候只有一条是必须的。这种优
化能同时节省时间和空间，因此链接器会扫描数趟代码，尽可能将两条指令替换为一条。由于每趟可能会缩短调用点和函数之间的距离，使该距离可在一条指令中容纳，所以链接器会不断优化代码，直到代码不再变化。

该过程称为链接器松弛，其名源于求解方程组的松弛技术。除了过程调用，对于gp±2KiB范围内的数据寻址，RISC-V链接器也会使用全局指针进行松弛，从而消除一条lui或 auipc。

类似地，链接器也会对位于 tp±2KiB 范围内的线程本地存储寻址进行松弛。

程序入口点

每个程序都有一个入口点，CPU必须从该点开始执行程序。
入口点由一个地址定义，该地址是必须执行的第一条指令的地址。可执行文件有一个包含有关可执行程序头部包含入口地址字段。当操作系统将程序加载到主存储器中，它就会使用入口地址设置PC，以便程序开始执行

```
1 # Contents of the main.s file
2 .globl start
3 start:
4   li  a0, 10
5   li  a1, 20
6   jal exit
```

```
$ riscv64-unknown-elf-objdump -D main.x

main.x:     file format elf32-littleriscv
Disassembly of section .text:
00010054 <exit>:
   10054: 00000513          li a0,0
   10058: 05d00893          li a7,93
   1005c: 00000073          ecall

00010060 <start>:
    10060: 00a00513          li a0,10
    10064: 01400593          li a1,20
    10068: fedff0ef          jal ra,10054 <exit>
    ...
```

```
$ riscv64-unknown-elf-as -march=rv32im main.s -o main.o
$ riscv64-unknown-elf-as -march=rv32im exit.s -o exit.o
$ riscv64-unknown-elf-ld -m elf32lriscv exit.o main.o -o main.x
```

程序入口点

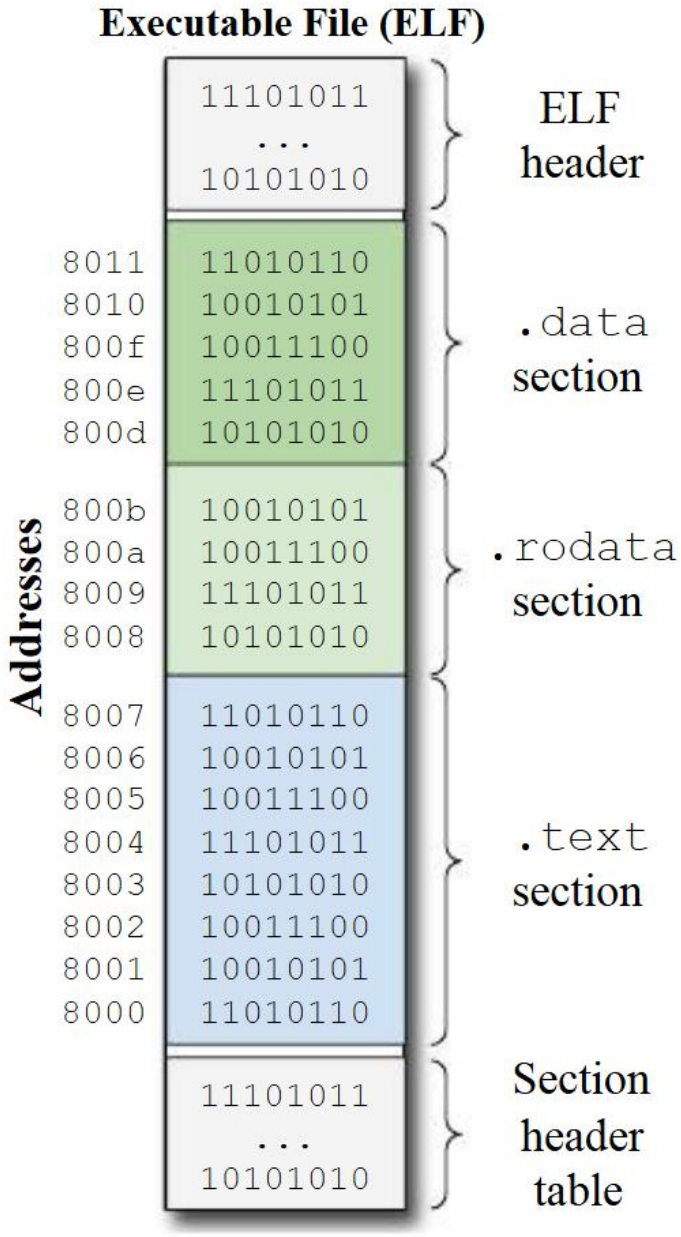
```
$ riscv64-unknown-elf-readelf -h main.x
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x10060
  Start of program headers:          52 (bytes into file)
  Start of section headers:          476 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 5
```

程序的节(section)

- 可执行文件、目标文件以及汇编程序通常按section组织。一个section可能包含数据或指令，每个section都映射到一组连续的主存储器地址。
- 常用的section
  - .text: 用于存储程序指令的部分；
  - .data: 用于存储初始化全局变量的部分，需要在程序开始执行之前初始化其值的变量。
  - .bss: 用于存储未初始化全局变量的部分；
  - .rodata: 用于存储常量的部分，即程序读取但在执行期间未修改的值。

程序的节(section)

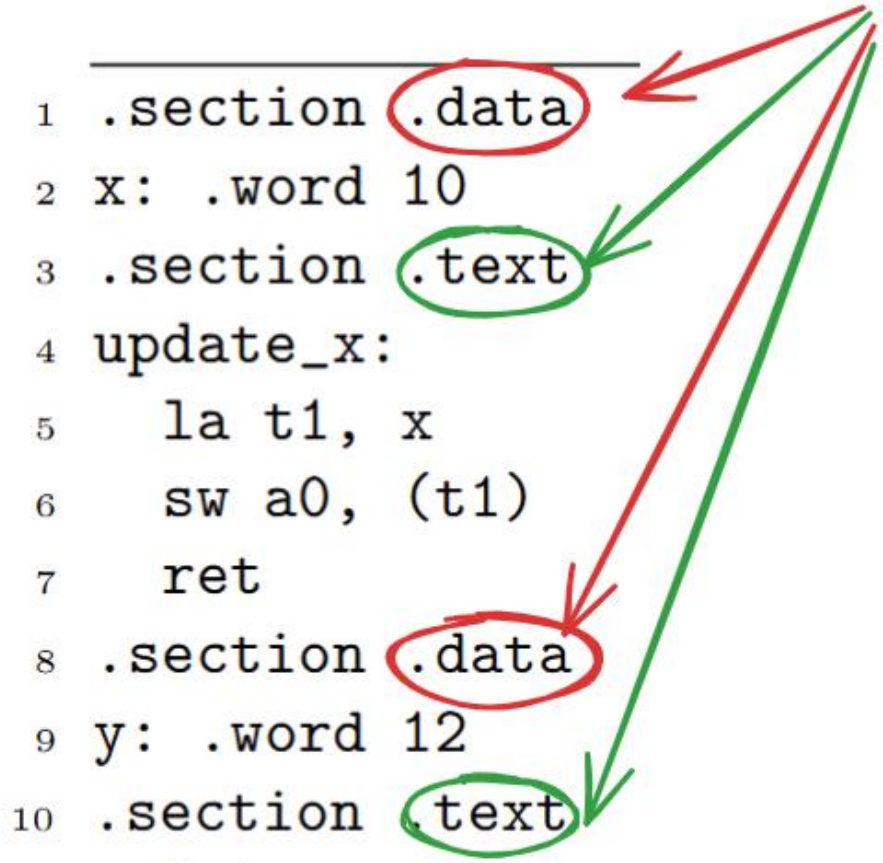
- 当链接多个目标文件时，链接器（LD）将来自具有相同名称的section的信息合并，将它们放在可执行文件的单个section中。



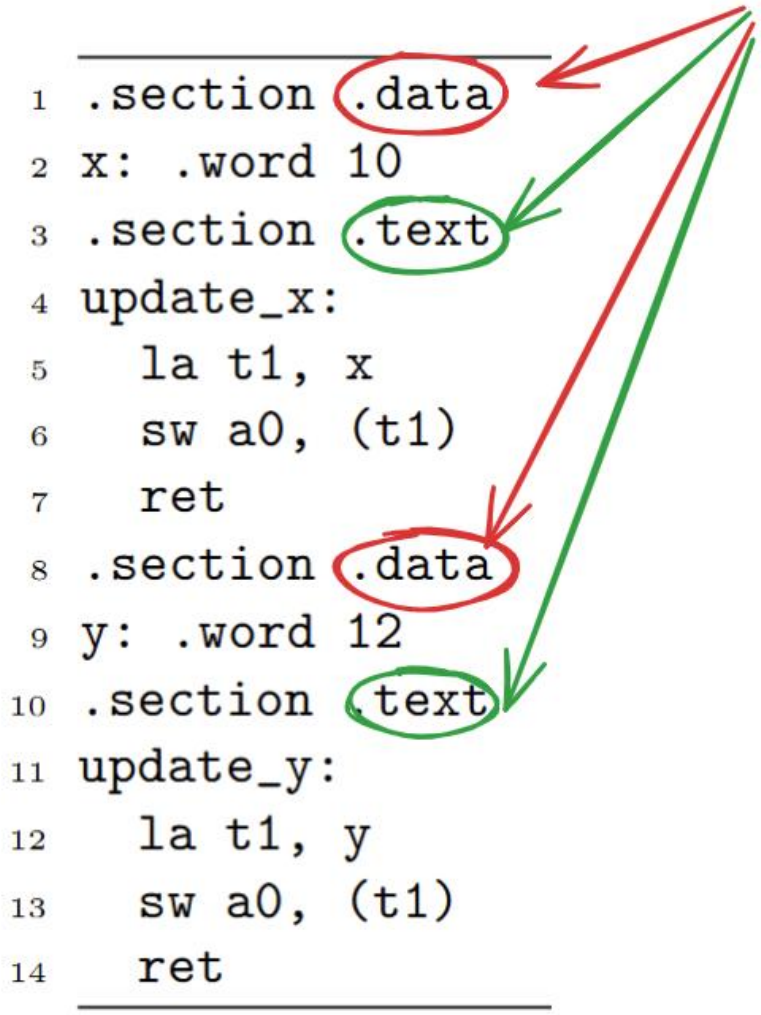
程序的节(section)

- 默认情况下，GNU汇编器工具将所有信息添加到.text中。要指示汇编器将组装信息添加到其他section中，程序员（或编译器）可以使用指令：
  - .section section_name
  - 指示汇编器将以下信息放入命名的节中

```
1  .section .data
2  x:  .word 10
3  .section .text
4  update_x:
5    la t1, x
6    sw a0, (t1)
7    ret
8  .section .data
9  y:  .word 12
10 .section .text
11 update_y:
12   la t1, y
13   sw a0, (t1)
14   ret
```



程序的节(section)



```
1  .section .data
2  x: .word 10
3  .section .text
4  update_x:
5    la t1, x
6    sw a0, (t1)
7    ret
8  .section .data
9  y: .word 12
10 .section .text
11 update_y:
12   la t1, y
13   sw a0, (t1)
14   ret
```

```
$ riscv64-unknown-elf-as -march=rv32im prog.s -o prog.o
$ riscv64-unknown-elf-objdump -D prog.o

prog.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <update_x>:
   0: 00000317            auipc t1,0x0
   4: 00030313            mv t1,t1
   8: 00a32023            sw a0,0(t1) # 0 <update_x>
   c: 00008067            ret

00000010 <update_y>:
  10: 00000317            auipc t1,0x0
  14: 00030313            mv t1,t1
  18: 00a32023            sw a0,0(t1) # 10 <update_y>
  1c: 00008067            ret

Disassembly of section .data:

00000000 <x>:
   0: 000a                    c.slli zero,0x2
...

00000004 <y>:
   4: 000c                    0xc
...
```

程序的节(section)

```
1  .section .data
2  x: .word 10
3  .section .text
4  update_x:
5    la t1, x
6    sw a0, (t1)
7    ret
8  .section .data
9  y: .word 12
10 .section .text
11 update_y:
12   la t1, y
13   sw a0, (t1)
14   ret
```

```
$ riscv64-unknown-elf-ld -m elf32lriscv prog.o -o prog.x
$ riscv64-unknown-elf-objdump -D prog.x

prog.x:     file format elf32-littleriscv

Disassembly of section .text:

00010074 <update_x>:
   10074: 00001317            auipc t1,0x1
   10078: 01c30313            addi t1,t1,28 # 11090 <__DATA_BEGIN__>
   1007c: 00a32023            sw a0,0(t1)
   10080: 00008067            ret

00010084 <update_y>:
   10084: 80418313            addi t1,gp,-2044 # 11094 <y>
   10088: 00a32023            sw a0,0(t1)
   1008c: 00008067            ret

Disassembly of section .data:

00011090 <__DATA_BEGIN__>:
   11090: 000a                c.slli zero,0x2
...

00011094 <y>:
   11094: 000c                0xc
...
```

可执行文件中地址已分配，目标文件中地址为相对地址

可执行文件和目标文件

- ELF: Executable and Linking, linux等系统中用于编码目标文件和可执行文件。
- 目标文件和可执行文件的区别
  - 目标文件中的地址不是最终的，来自不同部分的元素可能会被分配相同的地址。不同部分的元素可能不会同时驻留在主内存中；
  - 目标文件通常包含对未定义符号的多个引用，这些引用预计将由链接器解析；
  - 目标文件包含重定位表，以便对象文件上的指令和数据可以在链接时重新定位。可执行文件上的地址通常是最终的；
  - 目标文件没有入口点（entry）。

RISC-V®

汇编过程

- 生成本地代码
- 标号、符号、符号表
- 节(section)
- 目标代码与可执行程序