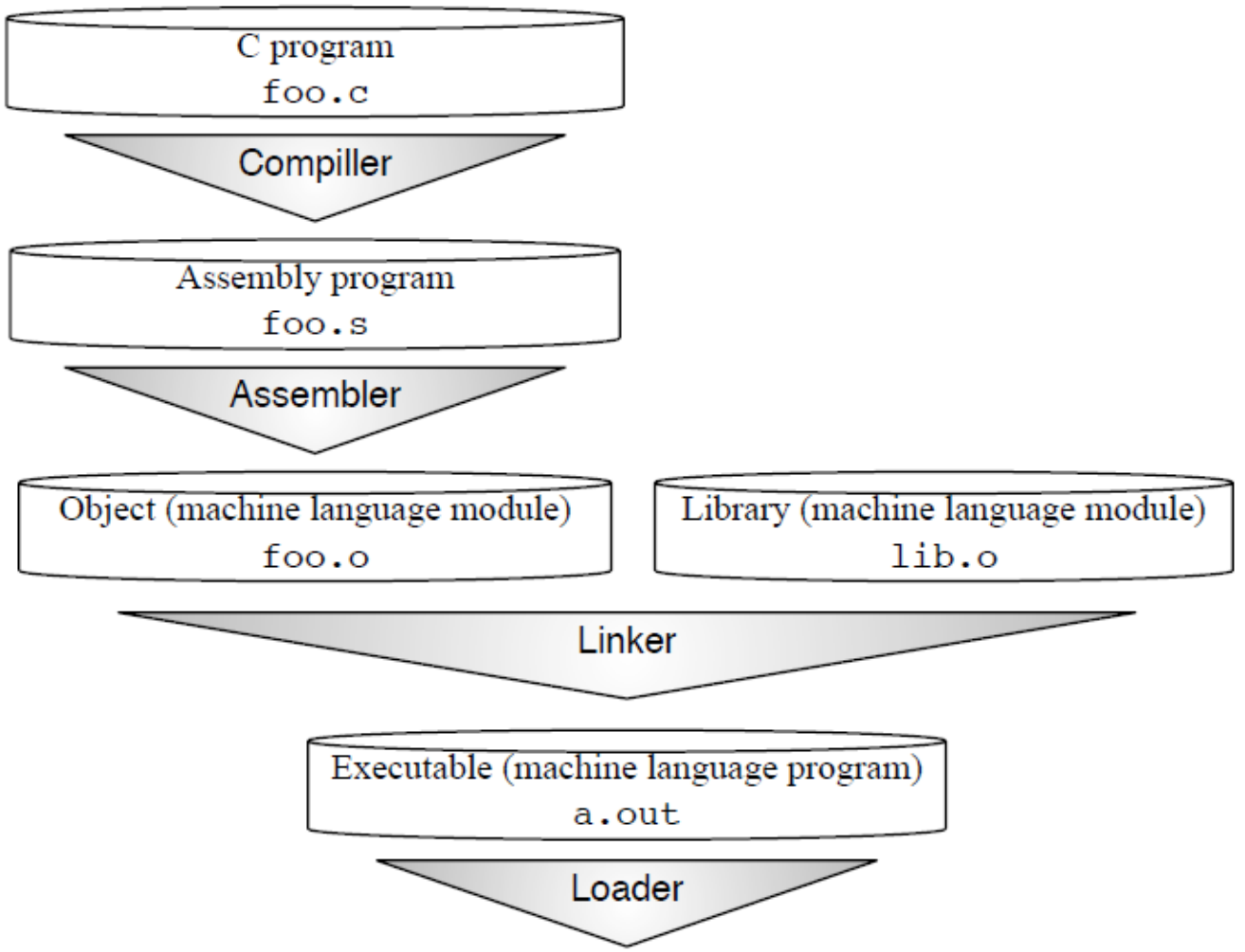
RISC-V: Program Execution *(Compiling, Assembling, Linking, and Loading)*

Illustrated with “Hello World” Program

Sparsh Mittal

Courtesy: "The RISC-V Reader: An Open Architecture Atlas" by David Patterson and Andrew Waterman

Steps of translation from C source code to a running program.
Some of these steps may be combined for speed.
The file extensions shown are for unix.
The equivalent suffixes in MS-DOS are .C, .ASM, .OBJ, .LIB, and .EXE.



Compiler

Input: High-Level Language Code (e.g., **`foo.c`**)

Output: Assembly Language Code
(e.g., **`foo.s`** for RISC-V)

```
gcc -O2 -S -c foo.c
```

Use this flag to see assembly language code

Note: Output *may* contain pseudo-instructions

Pseudo-instructions: instructions that assembler understands but not in machine

For example (copy the value from t2 to t1):

`mv t1,t2` ⇒ **`addi t1,t2,0`**

Assembler

- Input: Assembly Language Code (includes pseudo ops)
  (e.g., `foo.s` for RISC-V)
- Output: Object Code, information tables (true assembly only)
  (e.g., `foo.o` for RISC-V)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

Pseudoinstructions, which are clever variations of real instructions, make it easier to write assembly language programs without having to complicate the ISA

32 RISC-V pseudoinstructions that rely on x0, the zero register.

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

Figure 3.3: 32 RISC-V pseudoinstructions that rely on x0, the zero register. Appendix A includes includes the RISC-V pseudoinstructions as well as the real instructions. Those that read the 64-bit counters can read by upper 32 bits in RV32I by using the "h" version of the instructions and the lower 32 bits using the plain version. (Tables 20.2 and 20.3 of [Waterman and Asanović 2017] are the basis of this figure.).

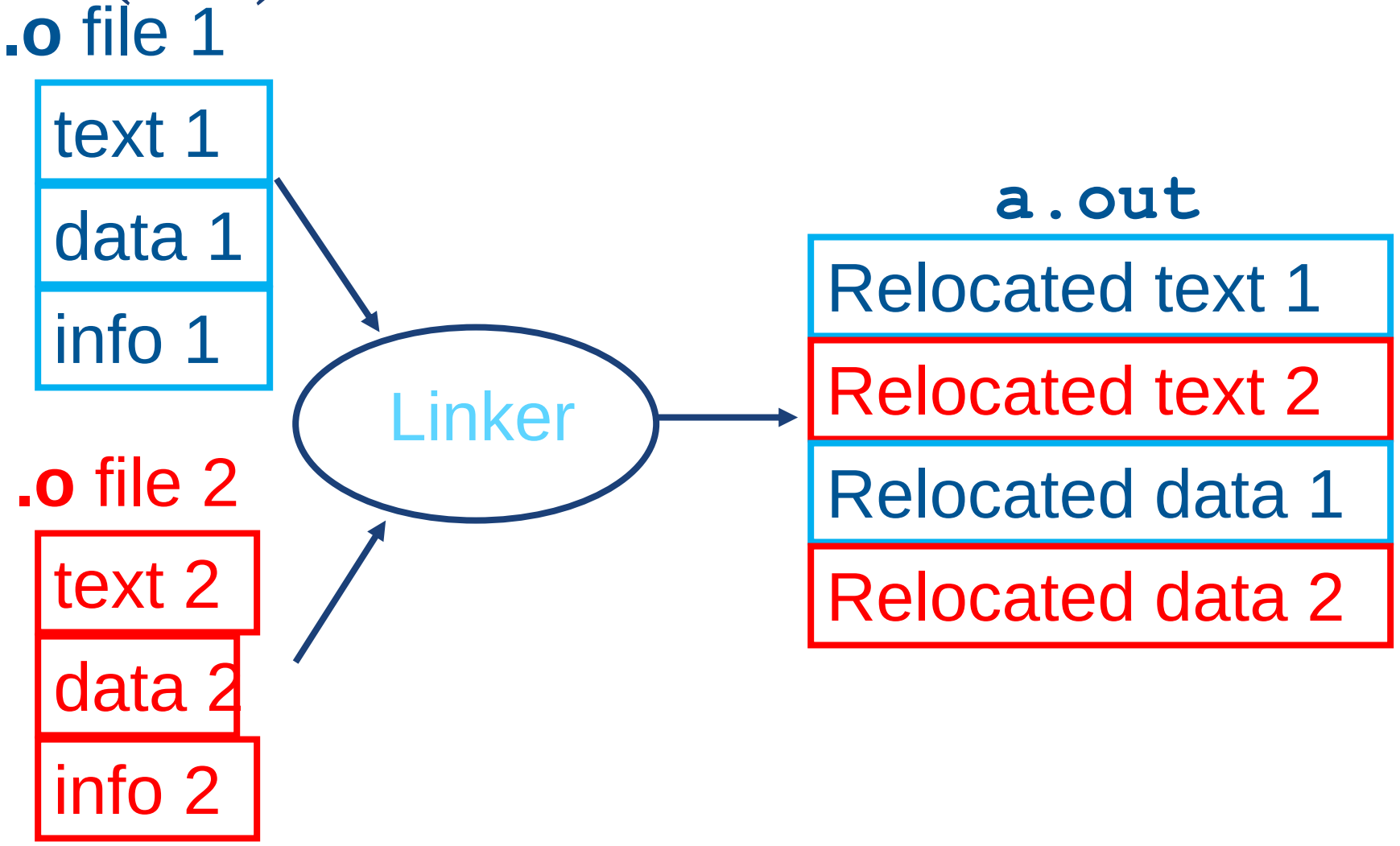
28 RISC-V pseudoinstructions that are independent of x0, the zero register.

Pseudoinstruction	Base Instruction(s)	Meaning
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags

Linker

- Input: Object code files, information tables (e.g., foo.o,libc.o for RISC-V)
- Output: Executable code (e.g., a.out for RISC-V)
- Combines several object (.o) files into a single executable ("linking")
- Enable separate compilation of files
  - Changes to one file do not require recompilation of the whole program
    - Linux source > 20 M lines of code!

Linker (2/3)


.o file 1
text 1
data 1
info 1
.o file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

Loader Basics

Input: Executable Code (e.g., **a.out** for RISC-V)
Output: (program is run)
Executable files are stored on disk
When one is run, loader's job is to load it into memory and start it running
In reality, loader is the operating system (OS)
    Loading is one of the OS tasks

Hello World Program in C

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

Hello World Program in RISC-V assembly language (hello.s)

```
  .text                    # Directive: enter text section
  .align 2                 # Directive: align code to 2^2 bytes
  .globl main              # Directive: declare global symbol main
main:                      # label for start of main
  addi sp,sp,-16           # allocate stack frame
  sw   ra,12(sp)           # save return address
  lui  a0,%hi(string1)     # compute address of
  addi a0,a0,%lo(string1)  #   string1
  lui  a1,%hi(string2)     # compute address of
  addi a1,a1,%lo(string2)  #   string2
  call printf              # call function printf
  lw   ra,12(sp)           # restore return address
  addi sp,sp,16            # deallocate stack frame
  li   a0,0                # load return value 0
  ret                      # return
  .section .rodata         # Directive: enter read-only data section
  .balign 4                # Directive: align data section to 4 bytes
string1:                   # label for first string
  .string "Hello, %s!\n"   # Directive: null-terminated string
string2:                   # label for second string
  .string "world"          # Directive: null-terminated string
```

Assembler directives

The commands that start with a period are assembler directives.

They are commands to the assembler rather than code to be translated by it.

They tell the assembler where to place code and data, specify text and data constants for use in the program, and so forth.

```
  .text                     # Directive: enter text section
  .align 2                  # Directive: align code to 2^2 bytes
  .globl main               # Directive: declare global symbol main
main:                       # label for start of main
  addi sp,sp,-16            # allocate stack frame
  sw   ra,12(sp)            # save return address
  lui  a0,%hi(string1)      # compute address of
  addi a0,a0,%lo(string1)   #   string1
  lui  a1,%hi(string2)      # compute address of
  addi a1,a1,%lo(string2)   #   string2
  call printf               # call function printf
  lw   ra,12(sp)            # restore return address
  addi sp,sp,16             # deallocate stack frame
  li   a0,0                 # load return value 0
  ret                       # return
  .section .rodata          # Directive: enter read-only data section
  .balign 4                 # Directive: align data section to 4 bytes
string1:                    # label for first string
  .string "Hello, %s!\n"    # Directive: null-terminated string
string2:                    # label for second string
  .string "world"           # Directive: null-terminated string
```

- .text—Enter text section.
- .align 2—Align following code to 2^2 bytes.
- .globl main—Declare global symbol “main”.
- .section .rodata—Enter read-only data section.
- .balign 4—Align data section to 4 bytes.
- .string “Hello, %s!\n”—Create this null-terminated string.
- .string “world”—Create this null-terminated string

HelloWorld program in RISC-V machine language (hello.o).

The assembler produces the following object file using the Executable and Linkable Format (ELF) standard format

```
00000000 <main>:
 0: ff010113   addi   sp,sp,-16
 4: 00112623   sw     ra,12(sp)
 8: 00000537   lui    a0,0x0
 c: 00050513   mv     a0,a0
10: 000005b7   lui    a1,0x0
14: 00058593   mv     a1,a1
18: 00000097   auipc  ra,0x0
1c: 000080e7   jalr   ra
20: 00c12083   lw     ra,12(sp)
24: 01010113   addi   sp,sp,16
28: 00000513   li     a0,0
2c: 00008067   ret
```

Assembly Language

```
main:
  addi sp,sp,-16
  sw   ra,12(sp)
  lui  a0,%hi(string1)
  addi a0,a0,%lo(string1)
  lui  a1,%hi(string2)
  addi a1,a1,%lo(string2)
  call printf
  lw   ra,12(sp)
  addi sp,sp,16
  li   a0,0
  ret
```

Machine Language

```
00000000 <main>:
 0: ff010113  addi  sp,sp,-16
 4: 00112623  sw    ra,12(sp)
 8: 00000537  lui   a0,0x0
 c: 00050513  mv    a0,a0
10: 000005b7  lui   a1,0x0
14: 00058593  mv    a1,a1
18: 00000097  auipc ra,0x0
1c: 000080e7  jalr  ra
20: 00c12083  lw    ra,12(sp)
24: 01010113  addi  sp,sp,16
28: 00000513  li    a0,0
2c: 00008067  ret
```

```
00000000 <main>:
 0: ff010113   addi   sp,sp,-16
 4: 00112623   sw     ra,12(sp)
 8: 00000537   lui    a0,0x0
 c: 00050513   mv     a0,a0
10: 000005b7   lui    a1,0x0
14: 00058593   mv     a1,a1
18: 00000097   auipc  ra,0x0
1c: 000080e7   jalr   ra
20: 00c12083   lw     ra,12(sp)
24: 01010113   addi   sp,sp,16
28: 00000513   li     a0,0
2c: 00008067   ret
```

- The six instructions that are later patched by the linker (locations 8 to 1c) have zero in their address fields.
- The symbol table included in the object file records the labels and addresses of all the instructions that need to be edited by the linker.

Hello World program as RISC-V machine language program after linking (a.out).

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

Machine Language (before linking)

```
00000000 <main>:
 0: ff010113   addi   sp,sp,-16
 4: 00112623   sw     ra,12(sp)
 8: 00000537   lui    a0,0x0
 c: 00050513   mv     a0,a0
10: 000005b7   lui    a1,0x0
14: 00058593   mv     a1,a1
18: 00000097   auipc  ra,0x0
1c: 000080e7   jalr   ra
20: 00c12083   lw     ra,12(sp)
24: 01010113   addi   sp,sp,16
28: 00000513   li     a0,0
2c: 00008067   ret
```

Machine Language (after linking)

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

With lui 0x21, the value becomes 0x21000, which is 135168. Subtracting 1520, we get, 133648 which is 0x20A10.

Common RISC-V assembler directives

Directive	Description
`.text`	Subsequent items are stored in the text section (machine code).
`.data`	Subsequent items are stored in the data section (global variables).
`.bss`	Subsequent items are stored in the bss section (global variables initialized to 0).
`.section .foo`	Subsequent items are stored in the section named `.foo`.
`.align n`	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a word boundary.
`.balign n`	Align the next datum on a n-byte boundary. For example, `.balign 4` aligns the next value on a word boundary.
`.globl sym`	Declare that label sym is global and may be referenced from other files.
`.string "str"`	Store the string str in memory and null-terminate it.
`.byte b1,..., bn`	Store the n 8-bit quantities in successive bytes of memory.
`.half w1,...,wn`	Store the n 16-bit quantities in successive memory halfwords.
`.word w1,...,wn`	Store the n 32-bit quantities in successive memory words.
`.dword w1,...,wn`	Store the n 64-bit quantities in successive memory doublewords.
`.float f1,..., fn`	Store the n single-precision floating-point numbers in successive memory words.
`.double d1,..., dn`	Store the n double-precision floating-point numbers in successive memory doublewords.
`.option rvc`	Compress subsequent instructions (see Chapter 7).
`.option norvc`	Don't compress subsequent instructions.
`.option relax`	Allow linker relaxations for subsequent instructions.
`.option norelax`	Don't allow linker relaxations for subsequent instructions.
`.option pic`	Subsequent instructions are position-independent code.
`.option nopic`	Subsequent instructions are position-dependent code.
`.option push`	Push the current setting of all `.options` to a stack, so that a subsequent `.option pop` will restore their value.
`.option pop`	Pop the option stack, restoring all `.options` to their setting at the time of the last `.option push`.

Memory allocation in RV32I

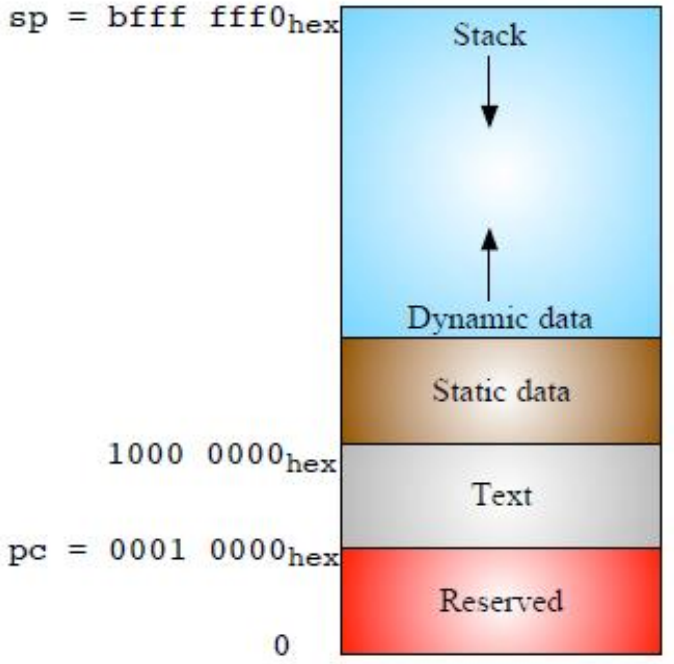
The high addresses are the top of the figure and the low addresses are the bottom.

The stack pointer (sp) starts at bfff fff0 and grows down toward the Static data.

The text (program code) starts at 0001 0000 and includes the statically-linked libraries.

The Static data starts immediately above the text region.

Dynamic data (heap), allocated in C by malloc(), grows upward toward the stack. It includes the dynamically-linked libraries.



The linker must adjust the program and data addresses of instructions in all the object files to match addresses in this figure

Position independent code (PIC).

PIC means that all the branches to instructions and references to data inside the file are correct wherever the code is placed.

The PC-relative branch of RV32I makes PIC much easier to fulfill.

Symbol table

In addition to the instructions, each object file contains a symbol table that includes all the labels in the program that must be given addresses as part of the linking process.

This list includes labels to data as well as to code.

```
  .text                    # Directive: enter text section
  .align 2                 # Directive: align code to 2^2 bytes
  .globl main              # Directive: declare global symbol main
main:                      # label for start of main
  addi sp,sp,-16           # allocate stack frame
  sw   ra,12(sp)           # save return address
  lui  a0,%hi(string1)     # compute address of
  addi a0,a0,%lo(string1)  #   string1
  lui  a1,%hi(string2)     # compute address of
  addi a1,a1,%lo(string2)  #   string2
  call printf              # call function printf
  lw   ra,12(sp)           # restore return address
  addi sp,sp,16            # deallocate stack frame
  li   a0,0                # load return value 0
  ret                      # return
  .section .rodata         # Directive: enter read-only data section
  .balign 4                # Directive: align data section to 4 bytes
string1:                   # label for first string
  .string "Hello, %s!\n"   # Directive: null-terminated string
string2:                   # label for second string
  .string "world"          # Directive: null-terminated string
```

- This code has two data labels to be set (string1 and string2) and two code labels to be assigned in (main and printf).
- It's hard to specify a 32-bit address within a single 32-bit instruction.
- Hence, the linker must adjust two instructions per label in the RV32I code: lui and addi for data addresses, and auipc and jalr for code addresses