

Arquitetura de Computadores / Programação Assembly

Custo



Simplicidade



Desempenho



Elegância



Espaço para crescimento



Tamanho de Programa



Isolamento de Arquitetura



Facilidade de programação

O Guia Prático RISC-V é uma concisa introdução e referência para programadores de sistemas embarcados, estudantes e aos curiosos sobre uma arquitetura moderna, popular e aberta. O RISC-V abrange desde o microcontrolador de 32 bits de baixo custo até o mais rápido computador na nuvem de 64 bits.

Ao utilizar os recursos visuais ilustrados acima, o texto mostra como o RISC-V incorporou as boas ideias de arquiteturas passadas, evitando os erros cometidos.

- Introduz o RISC-V em apenas 100 páginas, incluindo 75 figuras
- Cartão de Referência RISC-V de 2 páginas que resume todas as instruções
- Glossário de Instruções de 50 páginas que define todas as instruções detalhadamente
- 75 dicas de boas práticas de projeto de arquitetura utilizando os ícones acima
- 50 barras laterais com comentários interessantes e com o histórico do RISC-V
- 25 citações para transmitir o conhecimento de cientistas e engenheiros notáveis

Dez capítulos apresentam cada componente do conjunto de instruções modular RISC-V, muitas vezes, contrastando código compilado de C para RISC-V versus as arquiteturas ARM, Intel e MIPS, porém os leitores podem iniciar a programação após o Capítulo 2.

"I like RISC-V and this book as they are elegant—brief, to the point, and complete." — C. Gordon Bell



David Patterson (Google & UC Berkeley) e

Andrew Waterman (SiFive) são 2 dos 4 arquitetos RISC-V



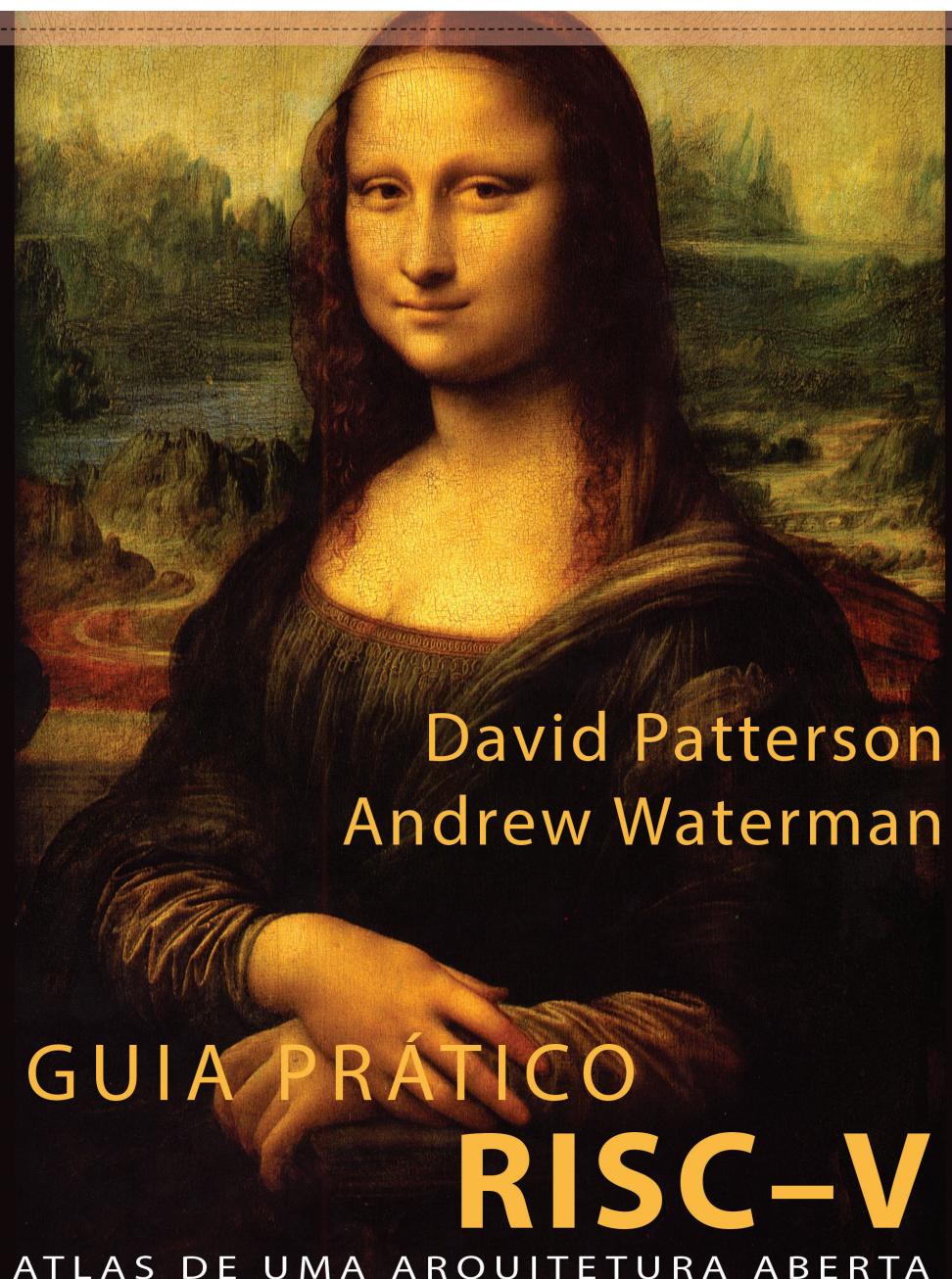
Strawberry Canyon LLC
San Francisco, CA, USA



Guia Prático RISC-V

1st

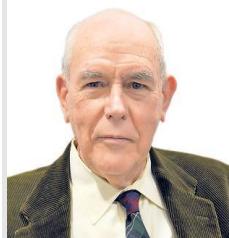
Patterson & Waterman



3

Linguagem Assembly do RISC-V

Ivan Sutherland (1938-) é reconhecido como o pai da computação gráfica pela invenção do Sketchpad—o precursor da interface gráfica de usuário nos computadores atuais, criado em 1962—que mais tarde levou-o a receber um Turing Award.



It's very satisfying to take a problem we thought difficult and find a simple solution. The best solutions are always simple.

—Ivan Sutherland

3.1 Introdução

A Figura 3.1 ilustra as quatro etapas clássicas de tradução de um programa escrito na linguagem C para um programa em linguagem de máquina pronto para ser executado pelo computador. Este capítulo aborda as últimas três etapas, mas iniciamos com o papel que o *assembler* desempenha na convenção de chamada do RISC-V.

3.2 Convenção de Chamada

Existem seis estágios que são executados em uma chamada de função [Patterson and Hennessy 2017]:

1. Colocar os argumentos onde a função possa acessá-los.
2. Saltar para a função (utilizando a instrução `jal` do RV32I's).
3. Adquirir recursos de armazenamento local que a função necessita, salvando registradores conforme necessário.
4. Executar a tarefa desejada da função.
5. Colocar o valor do resultado da função onde o programa de chamada pode acessá-lo, restaurar qualquer registrador, e liberar quaisquer recursos de armazenamento local.
6. Como uma função pode ser chamada de vários pontos em um programa, retornar o controle para seu respectivo ponto de origem (utilizando `ret`).

Para obter um bom desempenho, tente manter as variáveis nos registradores em vez da memória, mas por outro lado, evite também acessar a memória frequentemente para salvar e restaurar esses valores.

O RISC-V, felizmente, tem registradores suficientes para oferecer o melhor dos dois mundos: manter os operandos nos registradores e reduzir a necessidade de salvá-los e restaurá-los.

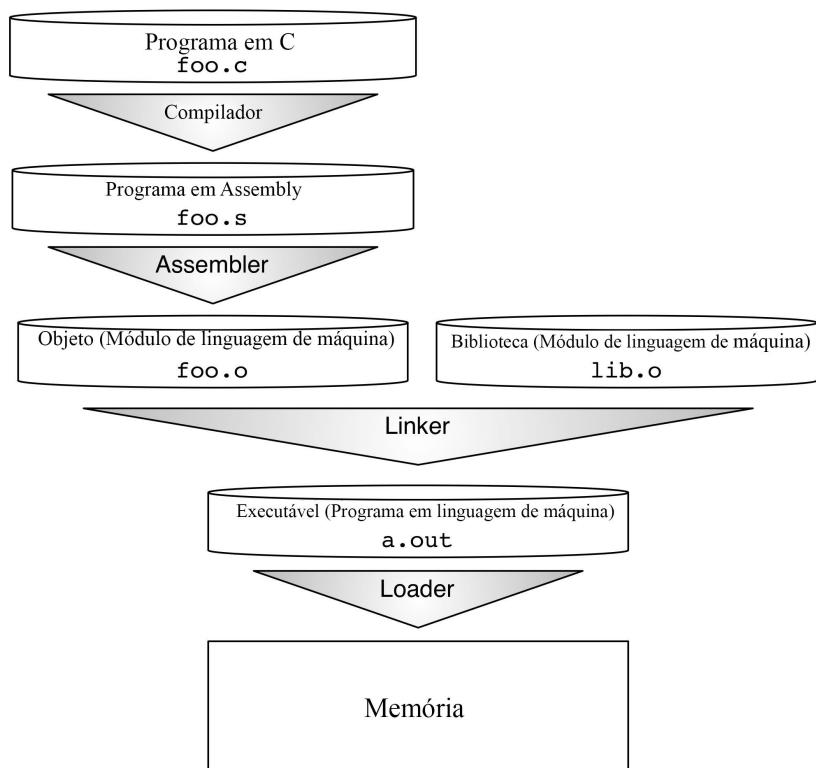


Figura 3.1: Passos de tradução de um código fonte C para um programa em execução. Estas são as etapas lógicas, embora algumas sejam combinadas para acelerar a tradução. Usamos a convenção de nomes de sufixos de arquivos Unix para cada tipo de arquivo. Os sufixos equivalentes no MS-DOS são .C, .ASM, .OBJ, .LIB, e .EXE.

Registrador	Nome ABI	Descrição	Preservado em toda a chamada?
x0	zero	Hard-wired zero	—
x1	ra	Endereço de retorno	Não
x2	sp	Ponteiro de pilha	Sim
x3	gp	Ponteiro global	—
x4	tp	Ponteiro de Thread	—
x5	t0	Registrador de link temporário/alternativo	Não
x6–7	t1–2	Temporários	Não
x8	s0/fp	Registrador salvo/Ponteiro de quadro	Sim
x9	s1	Registrador salvo	Sim
x10–11	a0–1	Argumentos de função / valores de retorno	Não
x12–17	a2–7	Argumentos de função	Não
x18–27	s2–11	Registradores salvos	Sim
x28–31	t3–6	Temporários	Não
f0–7	ft0–7	Temporários FP	Não
f8–9	fs0–1	Registradores salvos FP	Sim
f10–11	fa0–1	Argumentos e valores de retorno FP	Não
f12–17	fa2–7	Argumentos FP	Não
f18–27	fs2–11	Registradores salvos FP	Sim
f28–31	ft8–11	Temporários FP	Não

Figura 3.2: Mnemônicos do Assembler para registradores inteiros e de ponto flutuante do RISC-V. O RISC-V tem um número suficiente de registradores para que a ABI possa alocar registradores que procedimentos e métodos podem utilizar livremente sem a necessidade de salvar e restaurar seus valores nos casos em que outros procedimentos não são chamados. Os registradores preservados após uma chamada de procedimento também são denominados *caller saved*, e *callee saved* aqueles que não são. O Capítulo 5 descreve os registradores f de ponto flutuante. (Tabela 20.1 de [Waterman and Asanović 2017] é a base desta figura.)



A ideia é ter alguns registradores que não têm a garantia de serem preservados durante de uma chamada de função, chamados de *registradores temporários*(temporary registers), e alguns que são, os *registradores salvos*(saved registers). Funções que não realizam chamadas a outras funções são chamadas de funções *leaf*. Quando uma função leaf possui apenas alguns argumentos e variáveis locais, podemos manter tudo nos registradores sem “derramar” nada para a memória principal. Se essas condições persistirem, o programa não necessitará salvar os valores dos registradores na memória, e uma fração significativa de chamadas de função encontram-se nessa categoria.

Outros registradores dentro de uma chamada de função devem ser considerados da mesma classe que os registradores salvos, que são preservados através de uma chamada de função, ou na mesma classe que os registradores temporários, que não são preservados. Uma função alterará o(s) registrador(es) contendo o(s) valor(es) de retorno, como os mesmos sendo registradores temporários. Não há necessidade de preservar o endereço de retorno ou os argumentos da função, portanto, esses registradores são como temporários. O chamador pode confiar no último registradores, responsável pelo *stack pointer*, para permanecer inalterado após uma chamada de função. A Figura 3.2 lista os nomes de registradores do *RISC-V application binary interface*(ABI) e a convenção sobre os mesmos de serem ou não preservados nas chamadas de função.

A partir das convenções da ABI, é possível observar o código RV32I padrão para entrada

e saída de funções. A porção inicial *prologue* da função é dada da seguinte forma:

```
entry_label:
    addi sp,sp,-framesize      # Aloca espaço para stack frame
                                # ajustando stack pointer (registrador sp)
    sw   ra,framesize-4(sp)    # Salva endereço de retorno (registrador ra)
    # salva outros registradores na pilha, caso necessário
    ... # corpo da função
```

Se houverem muitos argumentos e variáveis de função para serem alocados nos registradores, o *prologue* da função aloca espaço na pilha para seu *frame*. Depois que a tarefa da função estiver completa, a posição final (*epilogue*) desfaz o *stack frame* e retorna ao ponto de origem da chamada de função.

```
# restaura estado dos registradores da pilha, caso necessário
lw   ra,framesize-4(sp)    # Restaura registrador ra
addi sp,sp, framesize     # Desaloca espaço do stack frame
ret                      # Retorna ao ponto de chamada
```

Veremos um exemplo que segue a ABI em breve, mas primeiro é necessário explicar as tarefas restantes do *assembly*, que encontram-se além da transformação dos nomes de registradores ABI em números de registradores.

■ Elaboração: Os registradores salvos e temporários não são contíguos

para fornecer suporte ao RV32E, uma versão embarcada do RISC-V que possui apenas 16 registradores (consulte o Capítulo 11). São utilizados números de registrador x0 a x15, portanto, alguns registradores salvos e temporários estão neste intervalo, enquanto os restantes estão nos últimos 16. O RV32E é menor, mas ainda não possui suporte ao compilador, já que ele não corresponde ao RV32I.

3.3 Assembly

A entrada para esta etapa no Unix é um arquivo com o sufixo .s, como foo.s; para MS-DOS é da forma .ASM.

O trabalho da etapa *assembler* da Figura 3.1 não é simplesmente produzir código objeto a partir das instruções que são compreendidas pelo processador, mas estende-las para incluir operações úteis para o programador *assembly* ou para o desenvolvedor do compilador. Esta categoria, baseada em configurações inteligentes de instruções regulares, é chamada de *pseudoinstruções*. As Figuras 3.3 e 3.4 listam as pseudoinstruções RISC-V, com as da primeira figura considerando o registrador x0 o valor zero, enquanto as da segunda lista não consideram esse contexto. Por exemplo, *ret* mencionado acima é na verdade uma pseudoinstrução em que o *assembler* substitui por *jalr x0, x1, 0* (veja a Figura 3.3). A maioria das pseudoinstruções RISC-V dependem de x0. Como você pode ver, reservar um dos 32 registradores para assumir o valor estático de zero simplifica bastante o conjunto de instruções do RISC-V, ao fornecer muitas operações populares —como *jump*, *return*, e *branch on equal to zero*— como pseudoinstruções.

A Figura 3.5 mostra o clássico programa “Hello world” escrito em C. A saída em *assembly* produzida pelo compilador é apresentada na Figura 3.6, usando a convenção de chamada da Figura 3.2 e as pseudoinstruções das Figuras 3.3 e 3.4.



Simplicidade

Tipicamente o programa “Hello world” é o primeiro programa executado em um processador recém projetado.

Arquitetos de sistema tradicionalmente consideram a execução do sistema operacional bem o suficiente para imprimir “Hello world” como um forte sinal de que seu novo chip funciona. Eles enviam essa saída por e-mail para seus gerentes e colegas, e então comemoram.

Pseudo-Instrução	Instrução (ões) Base	Significado
nop	addi x0, x0, 0	Operação No
neg rd, rs	sub rd, x0, rs	Complemento de dois
negw rd, rs	subw rd, x0, rs	Palavra em complemento de dois
snez rd, rs	sltu rd, x0, rs	"Seta" se \neq zero
sltz rd, rs	slt rd, rs, x0	"Seta" se $<$ zero
sgtz rd, rs	slt rd, x0, rs	"Seta" se $>$ zero
beqz rs, offset	beq rs, x0, offset	Desvia se = zero
bnez rs, offset	bne rs, x0, offset	Desvia se \neq zero
blez rs, offset	bge x0, rs, offset	Desvia se \leq zero
bgez rs, offset	bge rs, x0, offset	Desvia se \geq zero
bltz rs, offset	blt rs, x0, offset	Desvia se < zero
bgtz rs, offset	blt x0, rs, offset	Desvia se > zero
j offset	jal x0, offset	Pula
jr rs	jalr x0, rs, 0	Registrador de pulo
ret	jalr x0, x1, 0	Retorna da sub-rotina
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	Chamada de cauda sub-rotina far-away
rdinstret[h] rd	csrrs rd, instret[h], x0	Lê o contador de instruções retiradas
rdcycle[h] rd	csrrs rd, cycle[h], x0	Lê o contador de ciclos
rdtime[h] rd	csrrs rd, time[h], x0	Lê o relógio em tempo real
csrr rd, csr	csrrs rd, csr, x0	Lê o CSR
csrwr csr, rs	csrrw x0, csr, rs	Escreve o CSR
csrs csr, rs	csrrs x0, csr, rs	"Seta" bits em CSR
csrc csr, rs	csrrc x0, csr, rs	Limpa bits em CSR
csrwi csr, imm	csrrwi x0, csr, imm	Escreve CSR, imediato
csrsi csr, imm	csrrsi x0, csr, imm	"Seta" bits em CSR, imediato
csrci csr, imm	csrrci x0, csr, imm	Limpa bits em CSR, imediato
frcsr rd	csrrs rd, fcsrc, x0	Lê o controle de FP/registrador de status
fscsr rs	csrrw x0, fcsrc, rs	Escreve controle de FP/registrador de status
frrm rd	csrrs rd, frm, x0	Lê o modo de arredondamento do FP
fsrm rs	csrrw x0, frm, rs	Escreve o modo de arredondamento do
frflags rd	csrrs rd, fflags, x0	Lê flags de exceção de FP
fsflags rs	csrrw x0, fflags, rs	Escreve flags de exceção de FP

Figura 3.3: 32 pseudoinstruções RISC-V que utilizam o x0, o registrador zero. O Apêndice A inclui as pseudoinstruções RISC-V, bem como as instruções reais. Aquelas que lêem os contadores de 64 bits podem ler 32 bits superiores em RV32I utilizando a versão “h” das instruções, e os 32 bits inferiores usando a versão padrão. (Tabelas 20.2 e 20.3 of [Waterman and Asanović 2017] são a base desta figura.).

Pseudo-Instrução	Instrução (ões) Base	Significado
lla rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Carrega endereço local
la rd, symbol	<i>PIC</i> : auipc rd, GOT[symbol] [31:12] l{w d} rd, rd, GOT[symbol] [11:0]	Carrega endereço
	<i>Non-PIC</i> : Same as lla rd, symbol	
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
f{l w d} rd, symbol, rt	auipc rt, symbol[31:12] f{l w d} rd, symbol[11:0](rt)	Load ponto-flutuante global
fs{w d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0](rt)	Store ponto-flutuante global
li rd, immediate	<i>Miríades de sequências</i>	Load valor imediato
mv rd, rs	addi rd, rs, 0	Copia registrador
not rd, rs	xori rd, rs, -1	Complemento de um
sext.w rd, rs	addiw rd, rs, 0	Estende o sinal da palavra
seqz rd, rs	sltiu rd, rs, 1	"Seta" se = zero
fmv.s rd, rs	fsgnj.s rd, rs, rs	Copia registrador de precisão simples
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Valor absoluto de precisão simples
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Negação de precisão única
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copia registrador de precisão dupla
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Valor absoluto de precisão dupla
fneg.d rd, rs	fsgnjn.d rd, rs, rs	Negação de precisão dupla
bgt rs, rt, offset	blt rt, rs, offset	Desvia se >
ble rs, rt, offset	bge rt, rs, offset	Desvia se ≤
bgtu rs, rt, offset	bltu rt, rs, offset	Desvia se >, sem sinal
bleu rs, rt, offset	bgeu rt, rs, offset	Desvia se ≤, sem sinal
jal offset	jal x1, offset	Pula e linka
jalr rs	jalr x1, rs, 0	Jump e linka o registrador
call offset	auipc x1, offset[31:12] jalr x1, x1, offset[11:0]	Chame a sub-rotina far-away
fence	fence iorw, iorw	Cerca em toda a memória e I/O
fcsr rd, rs	csrrw rd, fcsr, rs	Troca controle de FP/registrador de status
fsrm rd, rs	csrrw rd, frm, rs	Troca o modo de arredondamento do FP
fsflags rd, rs	csrrw rd, fflags, rs	Troca sinalizadores de exceção de FP

Figura 3.4: 28 pseudoinstruções RISC-V que são independentes de x0, o registrador zero. Para la, GOT significa *Global Offset Table*, a tabela que contém o endereço em tempo de execução de símbolos nas bibliotecas "linkadas" dinamicamente. O Apêndice A descreve estas pseudoinstruções RISC-V, bem como as instruções reais. (Tabelas 20.2 e 20.3 of [Waterman and Asanović 2017] são a base para esta figura.)

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

Figura 3.5: Programa Hello World escrito em C (hello.c).

```
.text          # Diretiva: insere a seção de texto
.align 2      # Diretiva: alinha o código a 2 ^ 2 bytes
.globl main   # Diretiva: declara o símbolo global principal
main:         # rótulo para início da função main:
    addi sp,sp,-16   # aloca quadro de pilha
    sw ra,12(sp)     # salva o endereço de retorno
    lui a0,%hi(string1) # endereço de computação de
    addi a0,a0,%lo(string1) # string1
    lui a1,%hi(string2) # endereço de computação de
    addi a1,a1,%lo(string2) # string2
    call printf       # chama a função printf
    lw ra,12(sp)      # restaura o endereço de retorno
    addi sp,sp,16      # desaloca o quadro de pilha
    li a0,0            # dá load no valor de retorno 0
    ret                # retorna
.section .rodata # Diretiva: insira a seção de dados somente leitura
.balign 4        # Diretiva: alinha a seção de dados a 4 bytes
string1:        # rótulo para primeira string
    .string "Hello, %s!\n" # Diretiva: string terminada com nulo
string2:        # rótulo para segunda string
    .string "world"      # Diretiva: string terminada com nulo
```

Figura 3.6: Programa Hello World na linguagem assembly do RISC-V (hello.s).

```

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

```

Figura 3.7: Programa Hello World na linguagem de máquina do RISC-V (hello.o). As seis instruções que são posteriormente corrigidas pelo *linker* (locais 8 a 1c) têm zero em seus campos de endereço. A tabela de símbolos registra os rótulos e endereços de todas as instruções que necessitam ser editadas pelo *linker*.

```

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

```

Figura 3.8: Programa Hello World na linguagem de máquina do RISC-V após linkagem. Em sistemas Unix, o arquivo seria nomeado a.out.

Os comandos que iniciam com um ponto são as *diretivas assembler*. Esses são comandos específicos para o *assembler*, e não código para ser traduzido por ele. Estas diretivas informam ao *assembler* onde colocar código e dados, especificam constantes de código e dados para uso no programa, e assim por diante. A Figura 3.9 apresenta as diretivas *assembler* do RISC-V. Para o exemplo da Figura 3.6, as diretivas são:

- `.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.

O *assembler* produz o arquivo de objeto (Figura 3.7) seguindo o padrão *Executable and Linkable Format*(ELF) [TIS Committee 1995].

3.4 Linker

Em vez de compilar todo o código-fonte toda vez que um arquivo é alterado, o *Linker*, ou ligador, permite que arquivos individuais sejam compilados e montados separadamente. Ele “costura” o novo código objeto junto aos módulos de linguagem de máquina existentes, como bibliotecas. Seu nome é derivado de uma de suas tarefas, que é editar todos os links das instruções de jump and link no arquivo objeto. Na verdade, o *linker* é a abreviação para editor de links, que foi historicamente o nome dessa etapa na Figura 3.1. Em sistemas Unix, a entrada para o linker são arquivos com o sufixo `.o` (E.x., `foo.o`, `libc.o`), e sua saída é um arquivo `a.out`. Para MS-DOS, as entradas são arquivos com o sufixo `.OBJ` ou `.LIB` e a saída é um arquivo `.EXE`.

A Figura 3.10 apresenta os endereços das regiões de memória alocadas para código e dados em um típico programa RISC-V. O linker deve ajustar o programa e os endereços de dados das instruções em todos os arquivos de objeto para corresponder aos endereços nessa figura. É menos trabalho para o linker se os arquivos de entrada conterem *código independente de posição* (*PIC*). O que significa que todas os desvios a instruções e referências a dados dentro do arquivo estarão corretas onde quer que o código seja executado. Como mencionado no Capítulo 2, o desvio relativo ao PC do RV32I torna o PIC muito mais fácil de ser garantido.

Além das instruções, cada arquivo de objeto contém uma tabela de símbolos que inclui todos os rótulos no programa que devem receber endereços como parte do processo de vinculação realizado pelo linker. Essa lista inclui rótulos para dados, bem como para código. A Figura 3.6 tem dois rótulos de dados a serem “setados” (`string1` e `string2`) e dois rótulos de código a serem designados (`main` e `printf`). Um endereço de 32 bits é difícil de encaixar em uma instrução de 32 bits, portanto o linker deve ajustar duas instruções por rótulo no código RV32I, como mostra a Figura 3.6: `lui` e `addi` para endereços de dados, e `auipc` e `jalr` para endereços de código. A Figura 3.8 é a versão final `a.out`, correspondendo ao arquivo objeto na Figura 3.7.

Directive	Description
.text	Itens subsequentes são armazenados na seção de texto (código de máquina).
.data	Itens subsequentes são armazenados na seção de dados (variáveis globais).
.bss	Itens subsequentes são armazenados na seção bss (variáveis globais inicializadas em 0).
.section .foo	Itens subsequentes são armazenados na seção denominada .foo.
.align n	Alinha o próximo dado em um limite de 2^n -byte. Por exemplo, .align 2 alinha o próximo valor em um limite de palavra.
.balign n	Alinha o próximo dado em um limite de n-byte. Por exemplo, .balign 4 alinha o próximo valor em um limite de palavra.
.globl sym	Declara que o label sym é global e pode ser referenciado de outros arquivos.
.string "str"	Armazena a string str na memória e termina-a através de NULL.
.byte b1, ..., bn	Armazena as quantidades n de 8 bits em bytes sucessivos de memória.
.half w1, ..., wn	Armazena as n quantidades de 16 bits em meias palavras de memória sucessivas.
.word w1, ..., wn	Armazena as n quantidades de 32 bits em palavras de memória sucessivas.
.dword w1, ..., wn	Armazena as n quantidades de 64 bits em palavras duplas de memória sucessivas.
.float f1, ..., fn	Armazena os n números de ponto flutuante de precisão única em palavras de memória sucessivas.
.double d1, ..., dn	Armazene os n números de ponto flutuante de precisão dupla em palavras duplas de memória sucessivas.
.option rvc	Comprime as instruções subsequentes (veja o Capítulo 7).
.option norvc	Não comprime as instruções subsequentes.
.option relax	Permite relaxamentos do linker para instruções subsequentes.
.option norelax	Não permite relaxamentos do linker para instruções subsequentes.
.option pic	As instruções subsequentes são códigos independentes de posição.
.option nopic	Instruções subsequentes são códigos dependentes da posição.
.option push	Empurra a configuração atual de todos .options para uma pilha, de modo que um .option pop subsequente irá restaurar seu valor.
.option pop	Aplica um Pop na pilha de opções, restaurando todos .options a sua configuração no tempo do último .option push.

Figura 3.9: Diretivas assembler comuns do RISC-V.

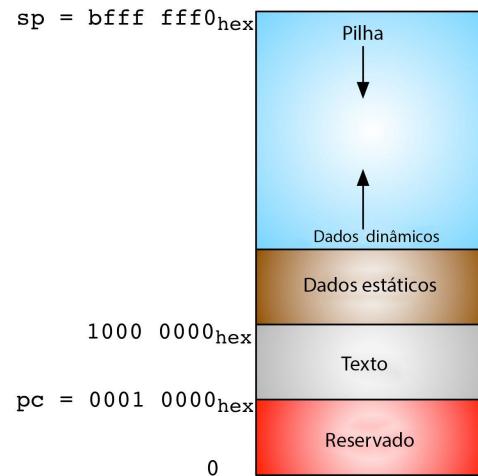


Figura 3.10: Alocação de memória para programa e dados do RV32I. Os endereços superiores estão no topo da figura, enquanto os inferiores estão na parte inferior. Nesta convenção de software do RISC-V, o ponteiro de pilha (sp) inicia em bfff fff0_{hex} e cresce para baixo em direção aos dados estáticos. O *text* (código do programa) inicia em 0001 0000_{hex} e inclui todas bibliotecas linkadas dinamicamente. Os dados estáticos iniciam imediatamente acima da região do *text*; Neste exemplo, consideramos que o endereço é 1000 0000_{hex}. Dados dinâmicos, alocados em C por `malloc()`, estão logo acima dos dados estáticos. Chamado de *heap*, este conjunto cresce em direção à pilha, e inclui todas bibliotecas linkadas dinamicamente.

Os compiladores RISC-V suportam várias ABIs, dependendo se as extensões F e D estão presentes. Para o RV32, as ABIs são denominadas ilp32, ilp32f e ilp32d. ilp32 significa que os tipos de dados de linguagem C int, long e pointer são todos de 32 bits; o sufixo opcional indica como os argumentos de ponto flutuante são passados. No ilp32, argumentos de ponto flutuante são passados em registradores de inteiros. No ilp32f, argumentos de ponto flutuante de precisão simples são passados em registradores de ponto flutuante. No ilp32d, argumentos de ponto flutuante de precisão dupla também são passados em registradores de ponto flutuante.

Naturalmente, para passar um argumento de ponto flutuante em um registrador de ponto flutuante, é necessário a extensão de ponto flutuante F ou D da ISA (consulte o Capítulo 5). Para compilar o código para o RV32I (GCC flag ‘-march=rv32i’), você deve usar o ABI ilp32 (GCC flag ‘-mabi=ilp32’). Por outro lado, ter instruções de ponto flutuante não significa que a convenção de chamada deva utilizá-las; Assim, por exemplo, o RV32IFD é compatível com todas as três ABIs: ilp32, ilp32f e ilp32d.

O linker verifica se a ABI do programa corresponde a todas as suas bibliotecas. Embora o compilador tenha suporte muitas combinações de extensões ISA e ABIs, apenas alguns conjuntos de bibliotecas podem ser instalados. Assim, um erro comum é vincular um programa sem ter as bibliotecas compatíveis instaladas. O linker não produzirá uma mensagem de diagnóstico útil nesse caso; ele tentará simplesmente vincular-se a uma biblioteca incompatível e, em seguida, informará a incompatibilidade. Esse erro geralmente ocorre apenas quando compila-se em um computador para um computador diferente (*cross compiling*).

■ Elaboração: Relaxamento do Linker

A instrução de *jump and link* tem um campo de endereço relativo ao PC de 20 bits, portanto com uma única instrução é possível realizar um salto mais longo. Enquanto o compilador gera duas instruções para cada função externa, muitas vezes apenas uma instrução é necessária. Como essa otimização economiza tempo e espaço, o linker fará passagens sobre o código para substituir sempre que puder duas instruções por uma. Como um passe pode reduzir a distância entre uma chamada e a função, de modo que ela se encaixa em uma única instrução, o linker continua otimizando o código até que não haja mais alterações. Este processo é chamado de *Linker relaxation*, com o nome referindo-se a técnicas de relaxamento para resolver sistemas de equações. Além das chamadas de procedimento, o "linkador" RISC-V relaxa o endereçamento de dados para usar o ponteiro global quando o dado está dentro de ± 2 KiB de gp, removendo um lui ou auipc. Ele também relaxa o endereçamento de armazenamento local quando o dado está dentro de ± 2 KiB de tp.

3.5 Linkagem Estática vs. Linkagem Dinâmica

A seção anterior descreve a *linkagem estática*, onde todo o código de biblioteca potencial é vinculado e, em seguida, carregado antes da execução. Essas bibliotecas podem ser relativamente grandes, portanto, vincular uma biblioteca popular a vários programas desperdiça memória. Além disso, as bibliotecas são legadas quando vinculadas—mesmo quando são atualizadas posteriormente para correção de bugs—forçando o código vinculado estaticamente a usar a versão legado com bugs.

Para evitar ambos problemas, a maioria dos sistemas dependem de *linkagem dinâmica*, onde uma função externa desejada é carregada e vinculada ao programa somente após ser chamada pela primeira vez; se nunca for chamada, nunca é carregada e vinculada. Cada chamada após a primeira usa um link de acesso rápido, portanto, a sobrecarga dinâmica acontece apenas uma vez. Quando um programa é iniciado, o mesmo vincula-se à versão atual das funções da biblioteca de que esse necessita, que é como ele obtém a versão mais recente. Além disso, se vários programas usam a mesma biblioteca vinculada dinamicamente, o código da biblioteca aparece apenas uma vez na memória.

O código que o compilador gera se assemelha àquele gerado para linkagem estática. Em vez de saltar para uma função real, ele salta para funções de *stub* (funções curtas de aproximadamente três instruções). A função stub carrega o endereço da função real de uma tabela na memória e, em seguida, salta para ela. No entanto, na primeira chamada a tabela não possui o endereço da função real, mas contém o endereço da rotina de linkagem dinâmica. Quando chamado, o linker dinâmico usa a tabela de símbolos para localizar a função real, copia-a na memória e atualiza a tabela para apontar para a função real. Cada chamada subsequente paga apenas a sobrecarga de três instruções da função stub.

Arquitetos normalmente medem o desempenho do processador usando benchmarks static linked apesar da maioria dos programas reais serem *linkados dinamicamente*. A desculpa é que os usuários interessados em desempenho devem linkar estaticamente, mas isso é uma justificativa fraca. Faz mais sentido acelerar o desempenho de programas reais, e não de benchmarks.

3.6 Loader

Um programa como o apresentado na Figura 3.8 é um arquivo executável armazenado no computador. Quando esse deve ser executado, o trabalho do *Loader* é de carregá-lo na memória e pular para o endereço inicial. O “loader” hoje é o sistema operacional; dito de outra maneira, a carga de a.out é uma das muitas tarefas de um sistema operacional.

A carga é uma tarefa um pouco mais complicada para programas linkados dinamicamente. Em vez de simplesmente iniciar o programa, o sistema operacional deve iniciar o linkador dinâmico, que por sua vez inicia o programa desejado e, em seguida, lida com todas as chamadas externas, copia as funções na memória e edita o programa após cada chamada para apontá-lo para a função correta.

3.7 Considerações Finais

Keep it simple, stupid.

—Kelly Johnson, engenheiro aeronáutico que criou o “KISS Principle,” 1960

OOC
Facilidade de Programação



Custo



Desempenho



Elegância

O assembler aprimora uma simples ISA RISC-V com 60 pseudo-instruções que tornam o código RISC-V mais fácil de ler e escrever sem aumentar os custos de hardware. Simplesmente dedicar um registrador RISC-V para assumir o valor zero habilita muitas operações úteis. As instruções *Load Upper Immediate* (lui) e *Add Upper Immediate to PC* (auipc) tornam mais fácil para o compilador e o linker ajustar endereços de dados e funções externas, e os desvios relativas ao PC facilitam o trabalho do linker com código independente de posição. Possuir vários registradores permite uma convenção de chamada que torna a chamada e retorno de função mais rápidas, reduzindo o número de derramamentos e restaurações de registradores.

O RISC-V oferece uma excelente seleção de mecanismos simples e impactantes, que reduzem custos, melhoram o desempenho e facilitam a programação.

3.8 Para Saber Mais

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. *TIS Committee*, 1995.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL <https://riscv.org/specifications/>.