Capítulo 1

Introdução ou *O Mapa da Disciplina*

Quem mal lê, mal ouve, mal fala, mal vê.

Monteiro Lobato

Essa disciplina tem a função, um tanto ingrata, de servir de ponte entre as disciplinas do início do curso (Algoritmos I e II, Oficina, Circuitos, Projetos e Arquitetura) e as disciplinas mais avançadas (Sistemas Operacionais, Introdução à Computação Científica, Redes I e Compiladores). A Tabela 1.1 mostra versões bastante resumidas dos programas das disciplinas que precedem e sucedem *Software* Básico.

Na "linha de sistemas", as disciplinas de Circuitos e de Projetos introduzem abstrações para sinais elétricos – que são representados por 'bits' – e para inteiros – que são representados por "vetores de bits". Em Projetos é apresentada uma especificação de um processador, que é a sua linguagem *assembly*, e esta especificação é então implementada com os componentes estudados até então. Versões simplificadas de periféricos são apresentadas, juntamente com versões primitivas do tratamento destes periféricos em *software*.

A organização e a arquitetura de computadores são estudadas na disciplina de Arquitetura, desde o ponto de vista do *hardware*. Alguns dos conceitos estudados superficialmente em Circuitos e Projetos são revisitados em maior profundidade em Arquitetura. Simultaneamente, alguns daqueles conceitos são revistos, com ênfase no *software*, nesta disciplina.

As abstrações de *processos*, *concorrência* e *memória virtual* são retomadas e expandidas em Sistemas Operacionais. Estas abstrações são usadas extensivamente em Introdução à Computação Científica (ICC). Em ICC são investigadas técnicas para tirar proveito da hierarquia de memória que inclui memória virtual.

A *interface serial* é um modelo para praticamente todos os mecanismos de comunicação entre dois dispositivos computacionais. O trabalho prático desta disciplina é uma versão simplificada de um *driver* para uma interface serial. Este projeto é estendido em Redes I.

Voltemos a *Software* Básico.

Um importante objetivo é apresentar técnicas de programação e estruturas de dados mais complexas do que aquelas estudadas no primeiro ano do curso. O veículo para tal é a montagem e ligação de programas, e para tanto é necessário aprofundar a programação em *assembly* (6 aulas), e a compreensão de como uma implementação eficiente do processador interfere com a geração de código pelo compilador[1].

[1] Eficiente e atrapalha? Sim, porque não existe almoço grátis. Veja *engenheiros selvagens*, adiante.

Tabela 1.1: Inserção desta disciplina no curso.

DISCIPLINA	ASSUNTO	OBSERVAÇÕES
Circuitos	funções em $\mathbb{B}$	abstração para sinais elétricos
(1º per)	portas lógicas	implementação das funções em $\mathbb{B}$
	simplificação	implementações eficientes
	estado, FFs	sistemas interessantes têm memória
	contadores, MEs	circuitos sequenciais síncronos
	MICO, *sw* e *hw*	"meu primeiro processador"
Projetos	aritmética	abstração para inteiros
(2º per)	*assembly*	especificação do processador
	implementação do MIPS	implementação da especificação
	memória	primeira versão
	temporização	tempo é vital
	VHDL	*hardware* descrito em *software*
Arquitetura	implementação do MIPS	multi-ciclo e segmentação
(3º per)	memória, cache	*hashing* em *hardware*
	memória virtual	virtualização, versão *hardware*
	interrupções	mecanismos de *hardware*
	processamento paralelo	mecanismos de *hardware*
***Software* Básico**	*assembly*	versão para adultos
(4º per)	`gcc`	geração de código e otimização
	SO, sincronia	abstrações e complicações
	memória virtual	virtualização, versão *software*
	interrupções	mecanismos de *software*
	E/S	visão de *software*
	contador, interface serial	periféricos, visão de *software*
	ligação e carga	de programas a processos
Sistemas Operacionais	multiplexação	uso eficiente de recursos
(5º per)	sincronia	mais complicações, versão SO
	processos	abstração que provê paralelismo
	memória virtual	visão completa
	E/S	visão completa
Redes I	sincronia	mais complicações, versão Redes
(5º per)	processos comunicantes	implementação de paralelismo
Introd Comp. Científica	processamento paralelo	mecanismos de *software*
(4º per)	processos e *threads*	abstrações para paralelismo
	memória virtual, cache	influência no desempenho

Os conteúdos de Redes I, SO e ICC são um tanto mais extensos do que o indicado aqui.

Uma *interface de programação de aplicativos* (*application programming interface*, API) é um conjunto de funções, como definidas numa página de manual, que permitem ao programador fazer uso das funcionalidades de uma biblioteca para codificar um programa de aplicação. Por exemplo, a API da função `printf()` é descrita na sua página de manual, que é acessada com "`man 3 printf`".

Uma *interface binária de aplicação* (*application binary interface*, ABI) é um conjunto de definições de interfaces de programação para uso pelos programadores de sistemas operacionais e de seus componentes. ABIs impõem uma férrea disciplina aos projetistas para tentar minimizar os problemas criados pelos engenheiros de *hardware*[2]. Um exemplo do que é definido numa ABI é o protocolo de chamada e de retorno de funções do MIPS, que determina quais registradores transferem argumentos, o que é empilhado, como o valor da função é retornado, e o formato do registro de ativação, além inumeráveis detalhes sórdidos.

[2]Segundo Dijkstra, engenheiros de sistemas – como este que vos escreve – seriam uns 'selvagens' [**?**].

Do ponto de vista de programadoras, a principal interface entre o *hardware* e o *software* é o conjunto de instruções do processador, que é a API do *hardware*. Para o programador de sistemas, além do conjunto de instruções, a interface é ainda composta pelo sistema de interrupções e exceções e pela ABI. Estes documentos – ABI, API, manual de *hardware* do processador – definem uma série de convenções para a construção de *drivers* de dispositivos periféricos e das camadas do sistema operacional (SO) próximas ao *hardware*.

Um terço do semestre é empregado no estudo da interface entre o processador e o SO. Os conceitos de processo e concorrência são usados para a compreensão do paralelismo entre o processador e dois periféricos simples. O trabalho prático desta disciplina é escrever um *driver* para uma interface serial. A interface serial é um modelo para todos os dispositivos de comunicação entre um par de dispositivos computacionais, seja através de dois fios, um par de fibras ópticas ou duas antenas.

O subsistema de memória virtual é apresentado para justificar uma série de decisões de projeto dos ligadores e carregadores, bem como das bibliotecas para ligação estática e dinâmica. Na disciplina de Arquitetura, os programas magicamente 'aparecem' na memória; nesta disciplina vemos como se dá o processo de ligação e carga dos programas em memória. Este material é coberto em 1/4 do semestre.

Evidentemente, os exercícios propostos são uma parte do texto muito importante para o seu aprendizado. Ignore-os e retorne para outra edição desta disciplina no próximo semestre.

Caveat emptor.

Capítulo 2

Representações

No cinema, no teatro, ou na ópera, os atores *representam* um papel. O ator não é o guerreiro ou amante ou vilão, apenas *representa* sê-lo, com maior ou menor fidelidade ao comportamento do *representado*. Bons atores nos causam empatia com a criatura representada, enquanto que o que maus atores nos despertam é apenas uma certa dose de desprezo pelo que estão fingindo ao invés do que estão 'vivendo'. Atuações convincentes merecem prêmio, enquanto o fingimento nos causa repulsa e reprovação. Bons atores são custosos – como quaisquer indivíduos com um talento excepcionalmente desenvolvido – enquanto que os demais trabalham nas novelas de certas rede de televisão, com algumas honrosas porém ideologicamente inexplicáveis exceções.

Quando empregamos computadores digitais para representar o mundo em que vivemos, somos obrigados – independentemente de qualquer viés ideológico – a empregar as representações que são as "melhores possíveis". *Melhor* para o quê?

No julgamento de valor entra o "custo da representação" em contraposição à "qualidade da representação". Por exemplo, pode-se representar números reais com grande acurácia a um custo que é considerado excessivo para a maioria das aplicações. Como outro exemplo de representação considere os números inteiros, que são representados em campos finitos de 8, 16, 32, ou 64 bits, porque estas são as larguras 'razoáveis' para implementar unidades de lógica e aritmética com a tecnologia de circuitos integrados disponíveis na segunda década do século XXI.

"Inteiros de 8 bits" nos permitem distinguir até 256 objetos distintos, o que pode ser adequando para inúmeras aplicações, enquanto que "inteiros de 64 bits" podem ser necessários em aplicações que necessitem enumerar 2^{64} objetos distintos. Evidentemente, os custos destas duas representações são diferentes, e não apenas em termos de 'largura' dos circuitos, mas também em termos do tempo de propagação dos sinais através dos circuitos, que no caso geral, é mais do que proporcional à largura dos circuitos. Na grande maioria dos casos, circuitos complexos consomem mais energia do que os simples.

Vejamos então três representações da realidade que são empregadas com computadores digitais: números inteiros representados em campos com largura fixa, números reais representados com ponto flutuante, e texto representado com caracteres, dígitos, sinais de pontuação e caracteres de controle.

Espaço em branco proposital.

2.1 Representação para Números Inteiros

Considere uma representação para números naturais em 32 bits. Com esta representação deve ser possível diferenciar 2^{32} objetos distintos, que pode ser uma representação para 2^{32} números naturais, de 0 a $(2^{32} - 1)$. Empregando notação posicional, a representação para o número natural N é dada pela Equação 2.1.

$$N = \sum_{i=0}^{k-1} 2^i \cdot b_i \tag{2.1}$$

A representação mais popular para números inteiros em 32 bits é a *representação em complemento de dois*. Nesta representação, um inteiro x e seu negativo são relacionados pela Equação 2.2 – $\overline{x}$ é o complemento bit a bit de x. Para valores na faixa representável, as operações de soma, subtração, multiplicação e divisão dos inteiros representados têm as propriedades aritméticas esperadas, tais como elemento neutro, associatividade, comutatividade e distributividade.

$$\begin{aligned} x + \overline{x} &= -1 \\ x + \overline{x} + 1 &= 0 \\ \overline{x} + 1 &= -x \end{aligned} \tag{2.2}$$

Para inteiros de 32 bits, o conjunto de valores representáveis, que chamaremos de $\mathcal{I}_{32}$, pode ser mostrado como a união de um conjunto de números negativos cujo limite inferior é -2^{31}, com o conjunto de números positivos que exclui 2^{31}, mais o elemento inválido η:

$$\mathcal{I}_{32} = [-2^{31}, 0] \cup [0, 2^{31}) \cup \{\eta\}\,.$$

Por causa da finitude da representação, operações de soma e subtração podem produzir resultados inválidos, e para representar estes valores o elemento η (*eta*) deve ser acrescido ao conjunto. Quando o resultado r da soma de duas parcelas extrapola o conjunto representável, $r \notin [-2^{31}, 2^{31})$, o valor produzido pelo circuito que computa a soma não pode ser usado por ser incorreto; tal resultado incorreto é representado por η.

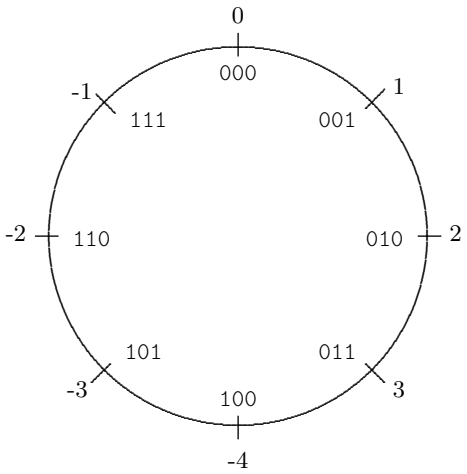


Figura 2.1: Representação em complemento de dois e círculo da representação.

A Figura 2.1 mostra o círculo da representação em complemento de dois para campos de três bits. Os números binários `000`, `001`, `010`, e `011` representam os inteiros 0, 1, 2 e 3, respectivamente, enquanto que os binários `111`, `110`, `101`, e `100` representam os inteiros -1, -2, -3, e -4, respectivamente.

Duas somas são mostradas na Figura 2.2. A primeira, no arco externo, é a soma de +2 com +3, que resulta em +5 e portanto não é representável em complemento de dois com três bits, porque o binário 101 representa o inteiro −3. Na segunda soma, no arco interno, +2 + −3 produz o resultado correto que é −1. Daqui se extrai uma regra: *a soma de dois inteiros positivos pode produzir resultado incorreto, enquanto que a soma de dois inteiros de sinais trocados produz resultados corretos.* A regra equivalente para a subtração é: *a diferença de números com sinais diferentes pode produzir resultados incorretos, enquanto que a diferença de números com sinais iguais produz resultados corretos.*

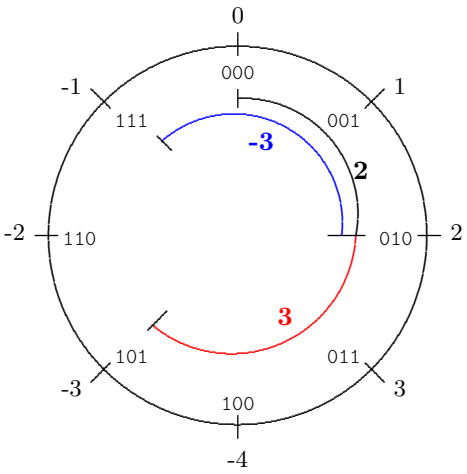


Figura 2.2: Representação em complemento de dois e *overflow*.

O termo em Inglês para este resultado incorreto em particular é *overflow*, porque o resultado da operação ultrapassa a capacidade de representação.

O teste para a detecção de *overflow* é simples: replique o bit mais significativo dos dois operandos e efetue a soma. Se o bit replicado e o bit mais significativo do resultado diferem, então o resultado da soma não pode ser representado corretamente. Considere a soma 4 + 7 numa representação em 4 bits: $0100_2 + 0111_2$. Replicando-se o bit mais significativo (mostrado em negrito) temos $\mathbf{0}0100_2 + \mathbf{0}0111_2 = \mathbf{0}1011_2$. O bit replicado (**0**) é diferente do bit 3 do resultado (1), e portanto ocorreu overflow e a representação para 4 + 7 em 4 bits é errada. Por outro lado, se somarmos $-1 + 1 = 1111_2 + 0001_2$, com o bit 3 replicado, obtemos $\mathbf{1}1111_2 + \mathbf{0}0001_2 = \mathbf{0}0000_2$. Os dois bits mais significativos do resultado são iguais e portanto não ocorre *overflow*.

Mesmo sendo – ligeiramente – assimétrica, a representação em complemento de dois possui duas características importantes, do ponto de vista da implementação dos circuitos para as operações aritméticas. Esta representação tem um único zero, ao invés de um "zero positivo" e de um "zero negativo", o que elimina testes para casos especiais tais como a soma de zeros com sinais opostos. Além, e por conta disso, os circuitos são simples e regulares. O teste para *overflow* é necessário em qualquer representação finita.

A Equação 2.3 mostra os pesos dos dígitos de um número representado em $\mathcal{I}_{32}$. O dígito na posição 31 é multiplicado por -2^{31}, e os demais dígitos são multiplicados por potências positivas de 2, decrescentes. Se o dígito mais significativo é zero, então o número representado é positivo, do contrário, é negativo.

$$\begin{aligned} (x_{31} \cdot -2^{31}) &+ x_{30} \cdot 2^{30} + x_{29} \cdot 2^{29} + \ldots + x_0 \cdot 2^0 && (2.3)\\ (0 \cdot -2^{31}) &+ 0\cdots01 && \mapsto +1\\ (1 \cdot -2^{31}) &+ 1\cdots11 && \mapsto -1 \end{aligned}$$

Exemplo 2.1 Alguns exemplos de números representados em complemento de dois, em 8 bits, são mostrados na Tabela 2.1. A posição mais significativa é multiplicada por 1 nos números negativos e portanto esta parcela contribui com -128 ao valor representado. Para representar um número maior do que -128, outras parcelas positivas devem ser adicionadas para 'descontar' a diferença do valor desejado para -128; como mostrado na segunda linha, $-1 = -128 + 127$. ◁

Tabela 2.1: Exemplos de representação em complemento de dois, em 8 bits.

		posição	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
valor		*peso*	-128	64	32	16	8	4	2	1
$+1$	$=$	$+0+1$	0	0	0	0	0	0	0	1
-1	$=$	$-128+127$	1	1	1	1	1	1	1	1
-125	$=$	$-128+3$	1	0	0	0	0	0	1	1
-3	$=$	$-128+125$	1	1	1	1	1	1	0	1

2.2 Representação para Números Reais – Ponto Fixo

As cinco primeiras potências negativas de 2 são

$$\begin{array}{lcll} 2^{-1} & = & 0,5 & = & 1/2 \\ 2^{-2} & = & 0,25 & = & 1/4 \\ 2^{-3} & = & 0,125 & = & 1/8 \\ 2^{-4} & = & 0,0625 & = & 1/16 \\ 2^{-5} & = & 0,03125 & = & 1/32. \end{array}$$

Para representar quantidades que não são inteiras, empregamos uma *representação posicional* que estende aquela usada para inteiros, e é exemplificada abaixo. As potências que multiplicam os dígitos da parte fracionária são negativas.

$$34,567_{10} = 3 \cdot 10^1 + 4 \cdot 10^0 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2} + 7 \cdot 10^{-3}$$

O número $3,3125_{10}$ é representado na base dois por $11,01010_2$

$$3,3125_{10} = 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-2} + 1 \times 2^{-4} = 2 + 1 + 0,25 + 0,0625.$$

Exemplo 2.2 Considerando uma representação em oito bits, com três bits para a parte inteira, e cinco bits para a fração, vejamos como representar números em *ponto fixo*. A Tabela 2.2 mostra a representação de cinco números fracionários, em complemento de dois.

A posição mais significativa é multiplicada por 1 nos números negativos e portanto esta parcela contribui com -4 ao valor representado. Para representar um número maior do que -4, parcelas positivas devem ser adicionadas para 'descontar' a diferença do valor desejado para -4.

O menor número representável é $-4_{10} = 100,00000_2$, e o maior número é $+3,96875 = 011,11111_2$. Esta é, em essência, a representação em complemento de dois em 8 bits, com todos os valores inteiros da faixa divididos por $32 = 2^5$. ◁

A precisão da representação depende do número de bits à direita do ponto. Para f bits de fração, a separação entre dois valores contíguos na representação é de 2^{-f}. Quanto maior o número de bits na fração, melhor a precisão dos valores representados.

Tabela 2.2: Exemplos de representação de frações em 3+5 bits.

valor		*posição* / *peso*	b_7 / -4	b_6 / 2	b_5 / 1	,	b_4 / 2^{-1}	b_3 / 2^{-2}	b_2 / 2^{-3}	b_1 / 2^{-4}	b_0 / 2^{-5}
$+0,03125$	$=$	$+0+0,03125$	0	0	0	,	0	0	0	0	1
$+0,0625$	$=$	$+0+0,0625$	0	0	0	,	0	0	0	1	0
-3	$=$	$-4+1$	1	0	1	,	0	0	0	0	0
$-1,78125$	$=$	$-4+2+2^{-3}+2^{-4}+2^{-5}$	1	1	0	,	0	0	1	1	1
$-0,03125$	$=$	$-4+2+1+2^{-1}+\ldots+2^{-5}$	1	1	1	,	1	1	1	1	1

2.3 Representação para Números Reais – Ponto Flutuante

A representação com o conjunto $\mathcal{I}_{32}$ nos permite trabalhar com relativo conforto no universo que pode ser modelado por números inteiros. Inúmeras situações necessitam de mais precisão, ou de uma faixa de valores mais larga, do que aquela possível com $\mathcal{I}_{32}$, ou mesmo com $\mathcal{I}_{64}$. Para tanto emprega-se uma representação para números reais chamada de *representação em ponto flutuante* ($\mathcal{F}_{32}$) porque a posição do ponto decimal não é fixa.

Números em ponto flutuante nos permitem representar tempo em picossegundos (10^{-12} s), o número de segundos num século ($3.1536 \cdot 10^9$ s), e aproximações para π e para e.

Para representar quantidades na faixa de valores num intervalo de $\approx \pm 2.0 \cdot 10^{\pm 38}$, empregamos a *representação em ponto flutuante*, que é similar ao tipo `float` da linguagem C. O conjunto $\mathcal{F}_{32}$ é representado em 32 bits e é dividido em três campos, um campo de `sinal`, com um bit de largura, um campo de `expoente` com 8 bits, e um campo de `fração` com 23 bits. O formato $\mathcal{F}_{32}$ é mostrado na Figura 2.3.

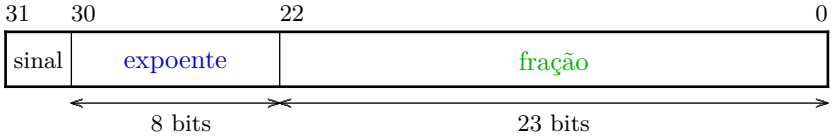


Figura 2.3: Representação em ponto flutuante, formato $\mathcal{F}_{32}$.

Para uma base β, expoente E, fração F e sinal s, o valor representado V é aquele definido pela Equação 2.4.

$$\begin{aligned} V &= (-1)^s \cdot F \cdot \beta^E \\ &= (-1)^s \cdot (f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + f_3 \cdot 2^{-3} + \ldots + f_{23} \cdot 2^{-23}) \cdot 2^E \end{aligned} \tag{2.4}$$

A largura do campo E determina a largura da faixa de representação, e quanto mais larga, maior a faixa de representação.

A largura do campo F determina a precisão da representação. A fração representa o intervalo $[0, 1)$, e este intervalo enorme é representado por 2^{23} padrões distintos. No limite inferior do intervalo, zero é representado por 23 zeros; o limite superior do intervalo é menor do que $1,0$ porque a parte fracionária é representada com 23 dígitos 1 e $1,0 > \sum_{i=1}^{23} 2^{-i}$.

Como $2^{23} \ll 2^{32}$, a precisão de $\mathcal{F}_{32}$ é algo menor do que a faixa da representação $\mathcal{I}_{32}$. O valor máximo da fração é $F_{\max} = 1 - ulp$, e no caso de $\mathcal{F}_{32}$, $ulp = 2^{-23}$. A abreviatura *ulp* significa *unit in the last position.*

Um número representado em $\mathcal{F}_{32}$ é dito *normalizado* se não há zeros à direita do ponto. Para *normalizar* um número desloca-se a fração para esquerda, aumentando precisão, enquanto decrementa-se o expoente. Para normalizar $0.00101 \cdot 2^3$ a fração é deslocada duas posições para a esquerda enquanto o expoente é reduzido de 2, resultando em $0.10100 \cdot 2^1$.

Exemplo 2.3 Vejamos um exemplo. Qual a representação de -1.5_{10} em $\mathcal{F}_{32}$?

$$-1.5_{10} = -3/2 = -3/2^1 = -11.0_2/2^1 = -11.0_2 \cdot 2^{-1} = -0.11 \cdot 2^1$$

Neste exemplo foi empregada uma mistura de conversões de base e de representação que, infelizmente, não há como ser mais elegante. ◁

Se F é a parte fracionária do resultado de operação aritmética e $F > F_{\max}$, então a fração deve ser reduzida, enquanto que o expoende é aumentado, $F \cdot 2^E \to (F/2) \cdot 2^{E+1}$, para que F seja representável em $\mathcal{F}_{32}$. A divisão é obtida pelo deslocamento de F para a direita.

2.3.1 Padrão IEEE 754

No início da década de 1980 imperava uma balbúrdia babeliana nas representações de ponto flutuante, com cada fabricante empregando o seu formato proprietário e incompatível com todos os demais. O IEEE publicou o padrão 754 para definir uma representação 'universal' para números em ponto flutuante. Com sua adoção, a entropia do universo diminuiu.

O formato é similar a $\mathcal{F}_{32}$, exceto que há um dígito implícito à esquerda do ponto. Isso significa que todos os números representados *devem* ser normalizados.

No formato `float` do padrão IEEE 754, para uma base β, expoente E, fração F e sinal s, o valor representado V é aquele na Equação 2.5.

$$\begin{aligned} V &= (-1)^s \cdot (1+F) \cdot \beta^E \\ &= (-1)^s \cdot (1 + f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + f_3 \cdot 2^{-3} + \ldots + f_{23} \cdot 2^{-23}) \cdot 2^E \end{aligned} \tag{2.5}$$

O termo $1 + F$ é chamado de *significando* e representa o intervalo $[1, 2)$. O expoente E tem 8 bits e a fração F tem 23 bits mais o dígito implícito à esquerda da vírgula.

O padrão define um formato `double` de 64 bits, que permite representar valores no intervalo $\approx \pm 2.0 \cdot 10^{\pm 308}$. Num `double`, o expoente E tem 11 bits e a fração F tem 52 bits.

A representação para zero é um caso especial: todos os bits do expoente e da fração devem ser zero.

No que segue, os `floats` são mostrados como "`s eeee eeee [1].ffff ··· fff`", com o bit `s` do sinal, 8 bits de expoente (`e`'s), 23 bits de fração (`f`'s), e o dígito implícito mostrado entre colchetes.

Qual a representação em `float` para os números 2^4 e 2^{-4}, supondo que o expoente seja representado em complemento de dois?

`0 0000 0100 [1].0000 ··· 000` $\mapsto 2^{+4}$
`0 1111 1100 [1].0000 ··· 000` $\mapsto 2^{-4}$

Considerando as duas representações como inteiros, qual delas representa o maior número?

No padrão IEEE 754 o expoente *não é representado em complemento de dois* para que se possa comparar `floats` como se fossem inteiros. Assim, comparações de magnitude podem empregar instruções que comparam inteiros; isso torna desnecessária uma nova instrução, específica para a comparação de magnitude de `floats`.

Finalmente, podemos enunciar a definição do formato `float`, como mostra a Equação 2.6. O valor V é representado por um significando de 23+1 bits $(1+F)$, um expoente deslocado $(E-d)$, e um bit de sinal(s). O deslocamento d é de 127 para `floats` e de 1023 para `doubles`.

$$\begin{aligned} V &= (-1)^s \cdot (1+F) \cdot 2^{(E-d)} \\ &= (-1)^s \cdot (1 + f_1 \cdot 2^{-1} + f_2 \cdot 2^{-2} + f_3 \cdot 2^{-3} + \ldots + f_{23} \cdot 2^{-23}) \cdot 2^{(E-d)} \end{aligned} \tag{2.6}$$

A Figura 2.4 mostra as faixas de valores para o expoente deslocado. No lado esquerdo do diagrama são mostrados o expoente E seguido de seu valor deslocado de 127 ($E : E - d$). O valor zero é usado para representar o caso especial que é o número zero e valores denormalizados. O menor expoente representável é o padrão de bits que representa -126, enquanto que o maior expoente é $+127$. O expoente 0 é representado pelo padrão de bits 127 em binário. O padrão de bits `0xff` é reservado para representar casos especias, descritos adiante. A Tabela 2.3 mostra os parâmetros do formato IEEE 754.

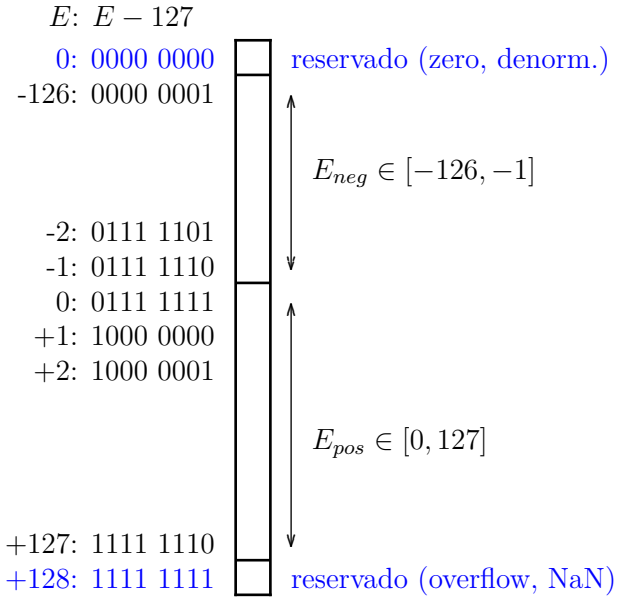


Figura 2.4: Expoente Deslocado em `float`.

Tabela 2.3: Parâmetros do Formato IEEE 754.

	`float`	`double`
bits de precisão	24	53
bits de expoente	8	11
Expoente máximo $E_{\max}$	127	1023
Expoente mínimo $E_{\min}$	-126	-1022
Deslocamento no exp. d	127	1023
unit in the last position	2^{-23}	2^{-52}
Fração máxima $F_{\max}$	$2 - 2^{-23}$	$2 - 2^{-52}$
Fração mínima $F_{\min}$	1	1

Voltemos aos números mostrados anteriormente. Com expoente deslocado, as representações em `float` para os números 2^4 e 2^{-4}, são

`0 1000 0011 [1].0000 ··· 000` $\mapsto 2^{+4}$
`0 0111 1011 [1].0000 ··· 000` $\mapsto 2^{-4}$

Nesta representação, o número menor (2^{-4}) tem expoente menor e portanto pode-se comparar `floats` e `doubles` com instruções para inteiros.

A faixa de valores representáveis com `float` é mostrada na Figura 2.5. As faixas de valores positivos e negativos tem a mesma largura, e a faixa dos `float` positivos é

$$F_{\min} \cdot 2^{E_{\min}} \leq V \leq F_{\max} \cdot 2^{E_{\max}}.$$

Ocorre *overflow* quando o expoente é muito grande para representação ($E > +127$), e ocorre *underflow* quando o expoente é muito pequeno para representação ($E < -126$).

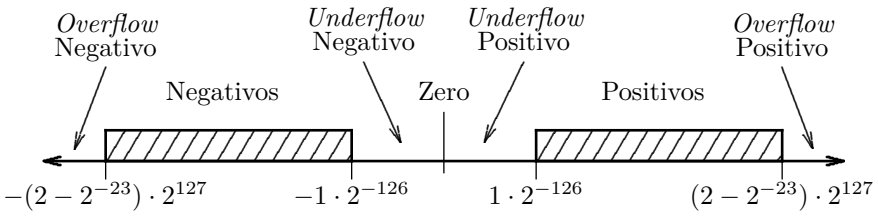


Figura 2.5: Faixa de valores representáveis em `float`.

A escolha dos deslocamentos tem uma consequência importante: as faixas de expoente e da fração permitem representar a recíproca de F_{min} sem *overflow*: $1/F_{min} < F_{max}$.

Vejamos alguns exemplos de conversão de números reais para `float`/`double` e vice-versa.

Exemplo 2.4 A conversão de -1.5_{10} para `float` tem um passo a mais do que a conversão para $\mathcal{F}_{32}$: o passo de normalização deve colocar um dígito à esquerda do ponto.

$$-1.5_{10} = -3/2 = -3/2^1 = -11.0_2/2^1 = -11.0_2 \cdot 2^{-1} \stackrel{\text{norm}}{=} -1.1 \cdot 2^0$$

A representação em `float` é
$(-1)^s \cdot (1+F) \cdot 2^{(E-127)}$
$(-1)^1 \cdot (1+0.1000\ldots0000) \cdot 2^{(127-127)}$
`1 0111 1111 1000 0000 0000 0000 0000 000`

A representação em `double` é
$(-1)^s \cdot (1+F) \cdot 2^{(E-1023)}$
$(-1)^1 \cdot (1+0.1000\ldots0000) \cdot 2^{(1023-1023)}$ ◁

Exemplo 2.5 A conversão de 0.5_{10} para `float` é: $0.5_{10} = 0.1_2 \stackrel{\text{norm}}{=} 1.0 \cdot 2^{-1}$
$(-1)^0 \cdot (1+0.0000\ldots0000) \cdot 2^{(126-127)}$
`0 0111 1110 0000 0000 0000 0000 0000 000` ◁

Exemplo 2.6 A conversão de 1.0_{10} para `float` é: $1.0_{10} = 1.0_2 \stackrel{\text{norm}}{=} 1.0 \cdot 2^0$
$(-1)^0 \cdot (1+0.0000\ldots0000) \cdot 2^{(127-127)}$
`0 0111 1111 0000 0000 0000 0000 0000 000` ◁

Exemplo 2.7 A conversão de 2.0_{10} para `float` é: $2.0_{10} = 10.0_2 \stackrel{\text{norm}}{=} 1.0 \cdot 2^1$
$(-1)^0 \cdot (1+0.0000\ldots0000) \cdot 2^{(128-127)}$
`0 1000 0000 0000 0000 0000 0000 0000 000` ◁

Exemplo 2.8 Dados, $s = 1$, $E = 129$, $F = 0100\ldots0000$, encontre o número representado.
`1 1000 0001 0100 0000 0000 0000 0000 000`
$(-1)^s \cdot (1+F) \cdot 2^{(E-127)}$
$(-1)^1 \cdot (1+0.0100\ldots0000) \cdot 2^{(129-127)} = -1 \cdot (1+0.25) \cdot 2^2 = -5.0_{10}$ ◁

O Padrão IEEE 754 reserva algumas codificações para valores especiais tais como zero, $\pm\infty$ e *NaN* ou *not a number*. Estas codificações são mostradas na Tabela 2.4. Operações com NaN resultam em NaN: $5 + NaN \mapsto NaN$, $0 \times \infty \mapsto NaN$; mas $\pm 1/0 \mapsto \pm\infty$.

Números com expoente menor que $E_{\min}$ são legais e possibilitam *underflow gradual*: para x, y pequenos, se $x \neq y$ então $x - y \neq 0$. Estes números são chamados de *de-normalizados*.

Tabela 2.4: Codificação dos valores especiais no padrão IEEE 754.

Expoente	Fração	representa
$E = E_{\min} - 1$	$F = 0$	± 0
$E = E_{\min} - 1$	$F \neq 0$	$0.F \times 2^{E_{min}}$ †
$E_{\min} \leq e \leq E_{\max}$	–	$1.F \times 2^e$
$E = E_{\max} + 1$	$F = 0$	$\pm\infty$
$E = E_{\max} + 1$	$F \neq 0$	NaN

† número de-normalizado: $2^{-149} < F < 2^{-126}$

2.3.2 Padrão IEEE 754 – Precisão

Seja x um número Real e $\mathcal{F}(x)$ sua representação em `float`.

O *erro absoluto* de representação é

$$\mathcal{F}(x) - x\,.$$

O *erro relativo* de representação é

$$\delta(x) = (\mathcal{F}(x) - x)/x\,.$$

Sejam F_1 e F_2 os números representáveis em $\mathcal{F}$ mais próximos de x tais que $F_1 \leqslant x \leqslant F_2$. A representação para x pode ser F_1 ou F_2. Se $F_1 = M \cdot 2^E$ então $F_2 = (M + ulp) \cdot 2^E$ e o *erro máximo* de representação é

$$[1/2|F_1 - F_2| = (ulp/2) \cdot 2^E\,, \tag{2.7}$$

se o arredondamento for "arredonda para o mais próximo". Isso significa que quanto maior o expoente, maior o erro absoluto de representação pois o expoente representa uma faixa maior enquanto que o campo da fração permanece com a mesma largura. A cada incremento no expoente, duplica o erro absoluto de representação. O diagrama na Figura 2.6 mostra a relação entre os expoentes e o erro de representação, considerando-se uma fração com somente 3 bits para simplificar o diagrama.

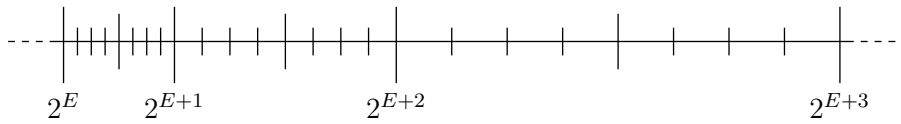


Figura 2.6: Precisão da representação `float`, para 3 bits de fração.

2.3.3 Representação de Ponto Flutuante em 8 Bits

Existem 256 valores diferentes que podem ser representados em ponto flutuante com 8 bits, e estes valores não são distribuídos uniformemente na reta dos Reais. A representação PF-8 é uma versão muito reduzida do Padrão IEEE 754.

Nesta representação, há seis intervalos (expoentes 001..110) com 16 pontos em cada intervalo (0000..1111). O intervalo com expoente 111 é usado para representar $\pm\infty$ e NaN. O intervalo com expoente 000 é dito de-normalizado. O deslocamento do expoente é 3.

s	exp	fração
1	1 0 0	1 0 0 1

A Figura 2.7 mostra os números reais que são representáveis em PF-8. Os diagramas mostram somente os Reais positivos, e o intervalo com expoente 000 é de-normalizado.

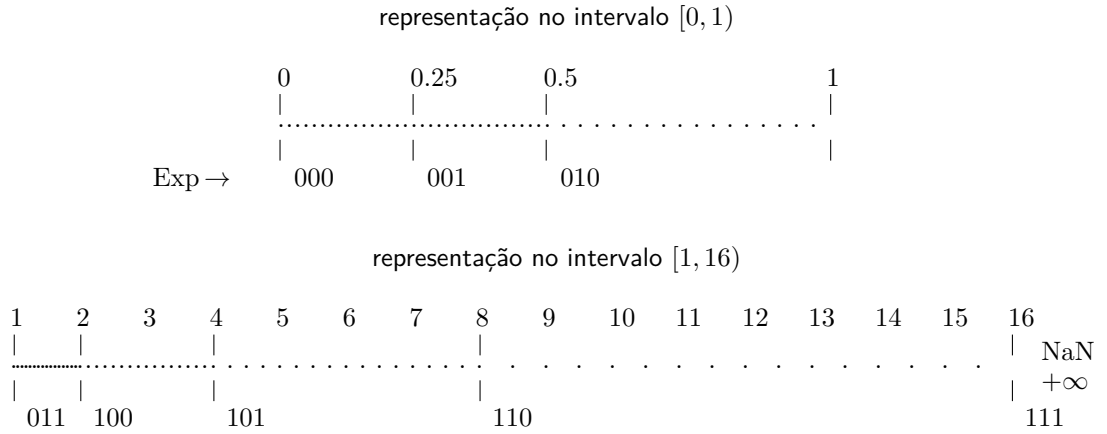


Figura 2.7: Representação de Reais em PF-8.

2.3.4 Exercícios

Ex. 1 Escreva as representações no Padrão IEEE 754 para os números 1/3, 1/5, 1/7, 1/9, π e e.

Ex. 2 Você foi encarregado de projetar um somador que satisfaz ao Padrão IEEE 754. Seu colega sugere que o somador das frações seja projetado com largura maior do que 23 bits. Isso é uma boa ideia? Justifique.

Ex. 3 Caso sua resposta ao Ex. 2 seja positiva, indique quais modos de arredondamento se tornam viáveis com (a) um bit adicional, (b) com dois bits adicionais, ou (c) com quatro bits adicionais.

Ex. 4 A Equação 2.7 indica o erro máximo de representação para a forma de arredondamento que "arredonda para o mais próximo". Qual é o erro máximo se o arredondamento for "arredonda para o menor", ou "truncamento"?

Ex. 5 Prove que as propriedades aritméticas abaixo são válidas, ou dê um contraexemplo. $\oplus$ e $\otimes$ são a soma e o produto de números representados em `float`. Os casos de *overflow* e *underflow* não podem ser usados como contraexemplos. *With thanks to Jeff Sanders.*

(a) monotonicidade c.r.a soma: $x \leqslant y \Rightarrow x \oplus z \leqslant y \oplus z$;

(b) associatividade c.r.a multiplicação: $x \otimes (y \otimes z) = (x \otimes y) \otimes z$; e

(c) distributividade: $x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$.

Ex. 6 Preencha a tabela abaixo com os números representáveis em PF-8. Não é necessário representar todas as 256 possibilidades, embora os casos limites devam todos ser preenchidos. O campo expoente deve conter o expoente deslocado. As magnitudes são os números representados, o '*gap*' é o intervalo irrepresentável (vazio) entre dois números vizinhos na representação. Não esqueça das representações para $\pm\infty$ e NaN.

Ex. 7 Quais são os piores erros de arredondamento, relativos e absolutos, quando se faz cálculos nos intervalos $[-8, 8]$ e $[-1, 1]$?

Ex. 8 Represente todas as potências de 2 entre 4 e 1024 no formato `float` IEEE 754.

sinal	expoente	significando	núm $\beta = 2$	núm $\beta = 10$	*gap*	comentário
0/1	000	0000	0.000	zero	–	caso especial

Empregue a estrutura definida no Prog. 2.1 para resolver os Exercícios 9, 10 e 11, que foram adaptados da Lista 01 de CI164 de 2015-1.

Programa 2.1: Estrutura para exibir `floats`.

```
#include <stdint.h>
#include <float.h>

typedef union {
  int32_t i;
  float f;
  struct {
    uint32_t frac : 23;
    uint32_t exp  : 8;
    uint32_t sign : 1;
  } parts;
} Float_t;
```

Ex. 9 Escreva uma função em C com um argumento do tipo `float` que imprime as representações do argumento como `float`, como `int`, e os campos de significando (1+fração), expoente e sinal.

Ex. 10 Escreva uma função em C com um argumento do tipo `float` que imprime o argumento mais os próximos cinco números representáveis, e o valor absoluto das diferenças entre cada par de números adjacentes na representação.

Ex. 11 Use a função do Ex. 10 para imprimir as sequências de números iniciadas por $0,01$, $1,0$, $10,0$ e $1000,0$. Explique o que ocorre com o valor absoluto das diferenças na medida em que o valor inicial de cada sexteto aumenta.

2.4 Representação para texto

Num computador, a informação é tipicamente representada por bits, bytes e palavras de 16, 32 ou 64 bits. Um *código* particular deve ser empregado para representar a informação

a ser armazenada e/ou transferida entre computadores, ou entre computadores e humanos. Cada código consiste de um conjunto de palavras compostas por elementos do *alfabeto* do código. Obviamente, todos os envolvidos devem estar de acordo quanto ao código e alfabeto a ser usado para evitar ambiguidade na troca de informações. Um alfabeto é um conjunto de *símbolos* e cada símbolo representa uma quantidade mínima de informação. Os símbolos do alfabeto Português são os caracteres 'a' a 'z' e 'A' a 'Z'. A representação hexadecimal dos caracteres 'a' e 'z', num dos códigos comumente empregados, é 0x61 e 0x7a, respectivamente.

O código ASCII (*American Standard Code for Information Interchange*) é um dos códigos mais empregados em 2021. No código ASCII cada caractere alfanumérico ou de pontuação é codificado em um número de 7 bits. Por exemplo, o caractere '0' (zero) equivale a 0x30, o caractere 'a' equivale a 0x61. O código ASCII contém ainda vários caracteres de controle usados em transmissão de dados.

A Tabela 2.5 mostra o mapeamento dos caracteres da versão estendida do alfabeto ASCII, que inclui a acentuação do Português, e que é chamada de Código Brasileiro de Intercâmbio de Informações (CBII). No código CBII, os caracteres são codificados em números de 8 bits. Este código também é conhecido como ISO-latin (ISO-8859-1). Os caracteres CBII cujo bit $b_7 = 0$ – na metade esquerda da Tabela 2.5 – são aqueles do alfabeto ASCII.

Tabela 2.5: Alfabeto CBII (ISO 8859-1).

	b_{7-4}															
b_{3-0}	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	NUL	DLE	SP	0	@	P	‘	p			NBSP	○	À	Đ	à	ð
1	SOH	DC1	!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
2	STX	DC2	"	2	B	R	b	r			¢	²	Â	Ò	â	ò
3	ETX	DC3	#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
4	EOT	DC4	$	4	D	T	d	t			◇	′	Ä	Ô	ä	ô
5	ENQ	NAK	%	5	E	U	e	u			ψ	µ	Å	Õ	å	õ
6	ACK	SYN	&	6	F	V	f	v			|	π	Æ	Ö	æ	ö
7	BEL	ETB	’	7	G	W	g	w			§	·	Ç		ç	
8	BS	CAN	(	8	H	X	h	x			¨	b	È	Ø	è	ø
9	HT	EM	)	9	I	Y	i	y			©	¹	É	Ù	é	ù
a	LF	SUB	*	:	J	Z	j	z				º	Ê	Ú	ê	ú
b	VT	ESC	+	;	K	[	k	{			≪	≫	Ë	Û	ë	û
c	FF	FS	,	<	L	\	l	|		¬	1/4	Ì	Ü	ì	ü
d	CR	GS	-	=	M	]	m	}			-	1/2	Í	Ý	í	ý
e	SO	RS	.	>	N	^	n	~			®	3/4	Î	Þ	î	þ
f	SI	US	/	?	O	_	o	DEL		‘	--	¿	Ï	β	ï	ÿ

2.5 Exercícios

Ex. 12 Escreva o número 133_{10} em binário, em hexadecimal, e na versão binária de ASCII.

Ex. 13 Escreva uma expressão em C que converte caracteres numéricos para os valores que os caracteres representam: `'5'` $\mapsto 5$.

Ex. 14 Escreva uma expressão em C que converte caracteres alfanuméricos para os valores na base 16 que os caracteres representam: `'5'` $\mapsto$ 5, `'f'` $\mapsto$ 15.

Ex. 15 Escreva uma função em C que converte um inteiro na base 10, representado por dígitos ASCII para o número representado: `"54321"` $\mapsto$ 54321. O `'\0'` não é parte do número.

Ex. 16 Escreva uma função em C que converte um inteiro na base 16, representado por caracteres ASCII para o número representado: `"54321abc"` $\mapsto$ 54321abc. O `'\0'` não é parte do número.

Ex. 17 Qual bit deve ser complementado para converter letras maiúsculas em minúsculas, e vice-versa, se o alfabeto da codificação é o ASCII?

Ex. 18 Qual bit deve ser complementado para converter letras maiúsculas em minúsculas, e vice-versa, se o alfabeto da codificação é o CBII?

Ex. 19 Escreva seu nome completo, abreviando o(s) nome(s) do meio, codificado na versão hexadecimal do alfabeto ASCII, com 7 bits, sem acentos. Não esqueça dos espaços entre os nomes e o ponto nas abreviaturas.

Ex. 20 Re-codifique a resposta ao Ex. 19, fazendo uso do bit 7 para conter a paridade ímpar de cada caractere.

Ex. 21 Leia a página de manual para a codificação ASCII, para a codificação ISO-8859-1, e para a codificação UTF-8: `man ascii`, `man iso-8859-1` e `man utf-8`. Quais as diferenças entre as versões do alfabeto latino ISO-8859-1 e UTF-8?

Ex. 22 Quando se trabalha com arquivos de texto, para que servem os programas `od` (*octal dump)*, `file` e `iconv` ?

Ex. 23 Codifique a sentença "`Uma arara arou de Araraquara ateh Jabaquara`" nos códigos CBII e Morse – definido na Figura 2.6. Qual o comprimento, em bits, das duas codificações? Suponha que o separador de caracteres no código Morse equivale a dois bits, e que o caractere 'espaço' equivale a quatro bits.

Tabela 2.6: Código Morse.

a .-	j .- - -	s ...	1 .- - - - -
b -...	k -.-	t -	2 ..- - -
c -.-.	l .-..	u ..-	3 ...- -
d -..	m - -	v ...-	4 ....-
e .	n -.	w .- -	5 .....
f ..-.	o - - -	x -..-	6 -....
g - -.	p .- -.	y -.- -	7 - -...
h	q - -.-	z - -..	8 - - -..
i ..	r .-.	0 - - - - -	9 - - - -.

Ex. 24 Por que a sequência de caracteres Morse "`sos`" é usada e reconhecida como um pedido de socorro? Por que a sequência "`ete`", bem mais curta, não é usada?

Capítulo 3

Programação em *Assembly*

Só passando para dizer:
muito obrigada por ter ensinado assembly do MIPS em vez do x86.
(sério mesmo). x86 = dor profunda
Estou tendo que aprender agora, e cara, é chato.
Aliás, se voce souber de algum manual bom por favor me avise...
Luiza Wille.

No relatório intitulado *First Draft of a Report on the EDVAC*, publicado em 1945, John von Neumann definiu o que se entende por *computador com programa armazenado*. Neste computador, a memória é nada mais, nada menos, do que um vetor de bits, e a interpretação dos bits é determinada pelo programador [**?**]. Este modelo tem algumas implicações interessantes, a saber:

(i) o código fonte que você escreve na linguagem C é "um programa" ou é "dados"?

(ii) o código *assembly*, produzido pelo montador a partir do seu código C é "um programa" ou é "dados"?

(iii) o arquivo `a.out` produzido pelo ligador, a partir do código *assembly* é "um programa" ou é "dados"?

A resposta, segundo o modelo de von Neumann, para as três perguntas é '*dados*'.

Como é que é?

É exatamente o que você acaba de ler. O código C é a entrada para o compilador, dados portanto. O código *assembly* é a entrada para o montador, portanto dados. O código binário contido no arquivo `a.out` está em formato de executável, mas é apenas "um arquivo cheio de bits". Este arquivo só se transforma em "um programa" no instante em que for carregado em memória e estiver prestes a ser executado pelo processador.

Uma vez que o programa esteja carregado em memória e a execução se inicia, quais seriam as fases de execução de uma instrução? A Figura 3.1 contém um diagrama de blocos de um computador com programa armazenado. Por razões que serão enunciadas adiante, a memória do nosso computador é dividida em duas partes, uma memória de instruções e uma memória de dados.

Entre as duas memórias está o processador. Um registrador chamado de `PC`, para *Program Counter*, mantém o endereço da próxima instrução que será executada. Este registrador é chamado de *instruction pointer* na documentação dos processadores da Intel, nome que melhor descreve sua função.

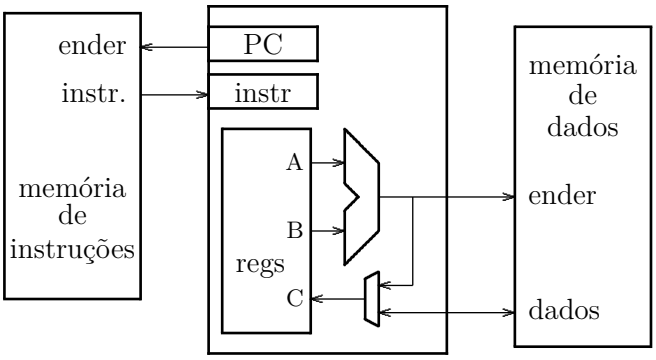


Figura 3.1: Modelo de computador com programa armazenado.

O registrador `instr` mantém a *instrução corrente*, que está sendo executada. O bloco de registradores (`regs`) contém 32 registradores que armazenam os valores temporários das variáveis do programa que está a executar. A unidade de lógica e aritmética efetua a operação determinada pela instrução corrente sobre os conteúdos de dois registradores e armazena o resultado num terceiro registrador.

O circuito que controla o funcionamento interno do processador sequencia a execução de cada uma instrução em quatro fases:

1. busca na memória a instrução apontada por PC, que se torna a instrução corrente;
2. decodifica a instrução corrente;
3. executa a operação definida pela instrução; e
4. armazena o resultado da operação, e retorna para 1.

O Capítulo 4 apresenta os detalhes sobre cada uma das fases. A Figura 3.2 indica três das as quatro fases da execução de uma instrução de adição. No topo da figura está a instrução **`add`** com seus três operandos, `r3` que é o registrador de resultado, `r1` e `r2` que são os registradores com as duas parcelas por somar. O caractere '#' é o indicador de comentário: ao registrador `r3` é atribuída a soma dos conteúdos de `r1` e `r2`.

```
add r3,r1,r2 # r3 ← r1+r2
```

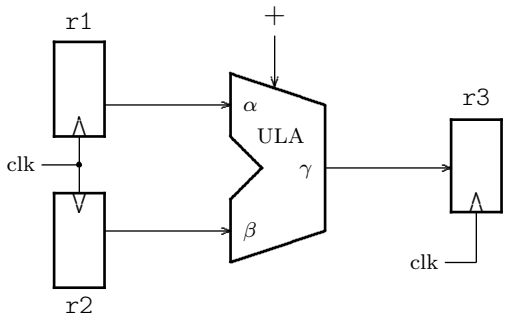


Figura 3.2: Acesso aos registradores, execução e resultado da instrução `add`.

Os registradores `r1`, `r2` e `r3` pertencem ao banco de registradores – `regs` na Figura 3.1 – e a cunha indica a ligação ao sinal de relógio. O trapézio chanfrado é a unidade de lógica e aritmética (ULA) do processador.

Durante a decodificação da instrução, o circuito de controle seleciona a operação da ULA, que é uma soma neste exemplo, e os conteúdos dos registradores `r1` e `r2` são apresentados

às suas entradas. Durante a execução os sinais se propagam através da ULA, e na fase de resultado, a soma é armazenada no registrador de destino, `r3`. Isso feito, o ciclo se repete e uma nova instrução é buscada.

O conjunto de instruções de um certo processador determina quais operações aquele processador é capaz de efetuar, bem como os tamanhos dos operandos das operações. Como um mínimo, as operações de aritmética e lógica são sempre suportadas – sobre operandos de 8, 16, 32 ou 64 bits – além de instruções para acesso à memória e para a tomada de decisões.

O processador é um circuito sequencial que implementa todas as operações do conjunto de instruções. Diz-se então que *o conjunto de intruções é a* especificação *do processador*, e ainda que *um determinado processador* é uma implementação *de seu conjunto de instruções.*

No que se segue o conjunto de instruções dos processadores MIPS de 32 bits é apresentado, ao mesmo tempo em que introduzimos a programação em *assembly.*

3.1 A Linguagem de Montagem MIPS32r2

Cada processador tem o seu conjunto de instruções e portanto sua linguagem de montagem lhe é exclusiva. As instruções do MIPS são diferentes das instruções dos processadores da família x86, tanto em termos de formato, quanto em termos da funcionalidade de cada instrução. Em geral, não há portabilidade entre linguagens de montagem, embora elas possam ser similares entre si. Nossa referência para o conjunto de instruções é o documento publicado pela própria MIPS [**?**].

Neste texto estamos interessados na linguagem *assembly* do MIPS. Esta linguagem é extremamente simples e um programa montador simplificado, porém completo, que traduz *assembly* para binário pode ser escrito em algo como 200 linhas de código C.

A notação empregada nas instruções de *assembly* é pré-fixada, tal como **`add`** `rc,ra,rb`, que significa "$c \leftarrow +(a\ b)$", enquanto que estamos habituados a usar uma forma infixada, tal como "$c \leftarrow (a + b)$". Nos comentários ao código usamos a forma infixada. As instruções são grafadas em negrito no que se segue.

Note que a "instrução **`add`**" tem três operandos, que são três registradores, enquanto que a operação de soma tem dois operandos e um resultado. Na descrição da instrução **`add`**, 'operandos' são os operandos da instrução **`add`**, enquanto que dois deles são operandos da operação *soma* que é efetuada por aquela instrução.

Cada instrução é identificada pelo seu *opcode*, ou seu *operation code*, que é um padrão de bits exclusivo àquela instrução. Para facilitar a vida de quem programa, a linguagem *asembly* é usada para representar os *opcodes* por mnemônicos que são mais facilmente memorizáveis. Por exemplo, ao invés de do padrão de 32 bits mostrado abaixo para representar a instrução de soma, empregamos o mnemônico **`add`** `r1, r2, r3`.

```
00000000010000110000100000100000
```

Uma instrução de lógica e aritmética como a instrução **`add`** é dividida em seis campos:

```
000000 00010 00011 00001 00000 100000
opcode   rs    rt    rd  shamt  func
```

O campo `opcode`, com 6 bits, identifica a instrução. Os três campos de 5 bits `rs` (*source*), `rt` (*target*), e `rd` (*destination*) são os três operandos da instrução. O campo `shamt` (*shift amount*) indica o número de posições por deslocar nas instruções de deslocamento. O campo `func` define a função da ULA, que neste caso é uma soma. A codificação das instruções é retomada na Seção 4.1.1.

Programas em linguagem de montagem do MIPS podem fazer uso de três tamanhos, ou tipos, de dados: byte, meia-palavra (*half word*, com 16 bits) e palavra (*word*, com 32 bits). As 'variáveis' dos programas podem ser armazenadas em dois conjuntos de variáveis: nos 32 registradores do processador, e num vetor de bytes que é a memória do computador. Estritamente falando, estes conjuntos disjuntos são dois *espaços de nomes*.

Um programa em *assembly* é editado num arquivo texto, com uma instrução por linha. Cada linha pode conter até três campos, todos opcionais: um *label*, uma instrução, e um comentário. O Programa 3.1 mostra um trecho com três linhas de código *assembly*. Um *label* é um nome simbólico que representa o endereço da instrução associada; o *label* `.L1:` representa o endereço da instrução **`add`** – o sufixo ':' é usado para indicar que o nome `.L1` é um *label* e não uma instrução. Um comentário inicia no caractere '`#`' e se estende até o final da linha. Nos comentários, a atribuição é denotada por '`<-`'.

Programa 3.1: Exemplo de código *assembly*.

```
.L1:    add r1, r2, r3    # r1 <- r2 + r3
        sub r5, r6, r7    # r5 <- r6 - r7
fim:    j   .L1           # salta para endereço com .L1
```

Na década de 1940, uma parte grande da tarefa "escrever um programa" incluía "montar os códigos de operação" – além de determinar a sequência de instruções do programa, a programadora devia traduzir as instruções para binário, que então eram gravadas diretamente na memória do computador.

O programa montador (*assembler*) traduz a "linguagem de montagem" (*assembly language*) para a "linguagem de máquina", que é o código binário interpretado pelo processador. Além da tradução, montadores sofisticados provêm uma série de comodidades ao programador, como veremos nas próximas páginas.

Vejamos três exemplos simples de tradução de código C para *assembly*. Nestes exemplos usaremos a convenção de chamar os operandos das instruções com os nomes das variáveis, prefixados de 'r' de registrador.

C	*assembly*
`a = b + c;`	`add ra, rb, rc`

Há uma correspondência direta entre o comando (simples) em C e sua tradução para *assembly*. Nosso segundo exemplo tira proveito da associatividade da soma. O resultado intermediário é acumulado no registrador `ra`.

C	*assembly*

`a = b + c + d + e;`	`add ra, rb, rc  # ra <- rb + rc`
	`add ra, ra, rd  # ra <- ra + rd`
	`add ra, ra, re  # ra <- ra + re`

No terceiro exemplo podemos tirar proveito da associatividade, ou então escrever código ligeiramente mais complexo para demonstrar o uso de dois registradores temporários (`t0` e `t1`) para a computação de expressões que não sejam triviais.

C	*assembly*

`f = (g+h) - (i+j);`	`add t0, rg, rh  # t0 <- rg + rh`
	`add t1, ri, rj  # t1 <- ri + rj`
	`sub rf, t0, t1  # rf <- t0 - t1`

Nos processadores MIPS de 32 bits, as instruções de lógica e aritmética sempre tem 3 operandos, e estes são: (i) três registradores, ou (ii) dois registradores e uma constante.

Esta escolha de projeto simplifica o circuito que decodifica as instruções, como veremos na Seção 4.2.1. A largura de todos os circuitos de dados no MIPS é 32 bits – os registradores a e unidade de lógica e aritmética, as interfaces com a memória, o PC e os circuitos de controle de execução.

O MIPS tem 32 *registradores visíveis*, que chamaremos de `r0` a `r31` por uma questão de gosto do autor, embora o programa montador os chame de `$0` a `$31`. O exemplo acima, reescrito com números de registradores ao invés de nomes de variáveis, mostra a linguagem aceita pelo montador. As variáveis `f..j` são alocadas nos registradores `$16..$20`.

C	*assembly*

`f = (g+h) - (i+j);`	`add $8, $17, $18  # $8=t1`
	`add $9, $19, $20  # $9=t2`
	`sub $16, $8, $9`

O registrador `$0` (`r0` ou `$zero`) retorna sempre o valor zero, e escritas neste registrador não tem nenhum efeito. Este registrador permite usar uma constante que é assaz popular. Por uma convenção da linguagem *assembly*, o registrador `r1` (ou `$1`) é usado como uma variável temporária para montador, e não deve ser usado nos programas em *assembly*.

3.1.1 Instruções de lógica e aritmética

A representação para números inteiros usada na linguagem de montagem do MIPS é o complemento de dois. Operações com inteiros podem ter operandos positivos e negativos, e talvez, dependendo da aplicação, um programa deva detectar a ocorrência de *overflow*, quando a soma de dois números de 32 bits produz um resultado que só pode ser representado corretamente em 33 bits.

A linguagem C ignora solenemente a ocorrência de *overflow*, enquanto que linguagens como C++ e Java permitem ao programador tratar o evento de acordo com as especifidades de cada aplicação. No caso de C, a programadora é responsável por detectar a ocorrência de *overflow* e tomar ação corretiva.

No MIPS as instruções de aritmética tem duas variedades, a variedade *signed* sinaliza a ocorrência de *overflow* com uma *exceção*, enquanto que a a variedade *unsigned* os ignora. Note que estes nomes são uma escolha infeliz porque ambíguos: *todas* as instruções de aritmética operam com números em complemento de dois, portanto números inteiros com sinal. O que *signed* e *unsigned* indicam é a forma de tratamento da ocorrência de eventuais resultados errados por causa de *overflow*. O mecanismo de exceções do MIPS é apresentado na Seção 7.3.

Instruções *unsigned* são empregadas no cálculo de endereços porque operações com endereços são sempre sem-sinal – todos os 32 bits compõem o endereço. Por exemplo, quando representa um endereço, `0x8000.0000` é um endereço válido e não o maior número negativo.

Vejamos mais algumas instruções de lógica e aritmética. Para descrever o efeito das instruções, ou sua *semântica*, empregaremos uma notação que é similar a VHDL: '&' denota a concatenação, `(x:y)` denota um vetor de bits com índices `x` e `y`.

```
add   r1, r2, r3        # r1 <- r2 + r3
addu  r1, r2, r3        # r1 <- r2 + r3
addi  r1, r2, const16   # r1 <- r2 + extSin(const16)
addiu r1, r2, const16   # r1 <- r2 + extSin(const16)
```

Como já vimos, as instruções **`add`** e **`addu`** somam o conteúdo de dois operandos e gravam o resultado no primeiro operando. As instruções **`addi`** e **`addiu`** somam o conteúdo de um

registrador ao conteúdo de uma constante de 16 bits, que é estendida para 32 bits, com a operação `extSin()`. As instruções *com* sufixo `u` são ditas *unsigned* e não sinalizam a ocorrência de *overflow*; as instruções *sem* o sufixo `u` são ditas *signed* e sinalizam a ocorrência de *overflow*.

Por que estender o sinal? É necessário estender o sinal para transformar um número de 16 bits, representado em complemento de dois, em número de 32 bits. A extensão é necessária para garantir que o número de 32 bits mantenha a mesma magnitude e sinal que o número de 16 bits. Considere as representações em binário, em 16 e em 32 bits para os números +4 e −4. Nas representações em binário, o bit de sinal do número em 16 bits está indicado em negrito, bem como o bit de sinal para a representação em 32 bits.

$+4 = 0\text{b}\mathbf{0}000.0000.\mathbf{0}000.0100 = 0\text{x}0004 \rightsquigarrow 0\text{x}0000.0004$
$-4 = 0\text{b}\mathbf{1}111.1111.\mathbf{1}111.1100 = 0\text{x}fffc \rightsquigarrow 0\text{x}ffff.fffc$

Vejamos algumas das instruções de lógica. As instruções **`and`**, **`or`**, **`nor`**, e **`xor`** efetuam a operação lógica Φ sobre pares de bits, um bit de cada registrador, e o resultado é atribuído ao bit correspondente do registrador de destino, como especifica a Equação 3.1.

$$\Phi\ r1, r2, r3 \quad \equiv \quad r1_i \leftarrow r2_i\ \Phi\ r3_i, \quad i \in [0, 31] \tag{3.1}$$

As instruções **`andi`**, **`ori`** e **`xori`** efetuam a operação lógica sobre o conteúdo de um registrador e da constante de 16 bits estendida com zeros (`extZero()`), e não com o sinal. Ao contrário das instruções de aritmética, quando se efetua operações lógicas, o que se deseja representar é uma constante lógica e não uma constante numérica positiva ou negativa.

A instrução **`not`** produz o complemento do seu operando `r2`.

A instrução **`sll`** (*shift left logical*) desloca seu segundo operando do número de posições indicadas no terceiro operando, que pode ser um registrador, ou uma constante de 5 bits; no primeiro caso, somente os 5 bits menos significativos são considerados.

As instruções **`srl`** (*shift right logical*) e **`sra`** (*shift right arithmetic*) deslocam seu segundo operando para a direita, de forma similar ao **`sll`**, exceto que a **`sra`** replica o sinal do número deslocado. A Tabela 3.1 define as instruções de lógica e aritmética.

Já sabemos como trabalhar com constantes de até 16 bits. Como se faz para obter constantes em 32 bits? São necessárias duas instruções: a instrução **`lui`** (*load upper immediate*) carrega uma constante de 16 bits na parte mais significativa de um registrador e preenche os 16 bits menos significativos com zeros.

```
lui r1, const16      # r1 <- const16 & 0x0000
```

Combinando-se **`lui`** com **`ori`**, é possível atribuir uma constante de 32 bits a um registrador:

```
lui r1, 0x0080       # r1 <- 0x0080 & 0x0000 = 0x0080.0000
ori r1, r1, 0x4000   # r1 <- 0x0080.0000 OR 0x0000.4000
                     #    =  0x0080.4000
```

Esta operação é usada frequentemente para efetuar o acesso à estruturas de dados em memória – o endereço da estrutura deve ser atribuído a um registrador que então aponta para seu endereço inicial. Por causa da popularidade, o montador nos oferece a *pseudoinstrução* **`la`** (*load address*) que é um apelido para o par **`lui`** seguido de **`ori`**.

```
la  r1, 0x0080.4000  # r1 <- 0x0080.4000
```

Quando o endereço é um *label*, ao invés de uma constante numérica, o montador faz uso dos operadores `%hi()` e `%lo()` para extrair as partes mais e menos significativas do seu operando. Suponha que o endereço associado ao *label* `.L1` seja `0x0420.3610`, então

```
la r1, .L1
```

Tabela 3.1: Instruções de lógica e aritmética.

```
add   r1, r2, r3        # r1 <- r2 + r3                    [1]
addi  r1, r2, const16   # r1 <- r2 + extSin(const16)       [1]
sub   r1, r2, r3        # r1 <- r2 - r3                    [1]

addu  r1, r2, r3        # r1 <- r2 + r3                    [2]
addiu r1, r2, const16   # r1 <- r2 + extSin(const16)       [2]
subu  r1, r2, r3        # r1 <- r2 - r3                    [2]

and   r1, r2, r3        # r1 <- r2 AND r3
or    r1, r2, r3        # r1 <- r2 OR r3
not   r1, r2            # r1 <- NOT(r2)
nor   r1, r2, r3        # r1 <- NOT(r2 OR r3)
xor   r1, r2, r3        # r1 <- r2 XOR r3

andi r1, r2, const16    # r1 <- r2 AND extZero(const16)
ori  r1, r2, const16    # r1 <- r2 OR  extZero(const16)
xori r1, r2, const16    # r1 <- r2 XOR extZero(const16)

sll   r1, r2, r3        # r1 <- (r2 << r3(4..0))
sll   r1, r2, const5    # r1 <- (r2 << const5)

srl   r1, r2, r3        # r1 <- (r2 >> r3(4..0))
srl   r1, r2, const5    # r1 <- (r2 >> const5)

sra   r1, r2, r3        # r1 <- (r2 >> r3(4..0))           [3]
sra   r1, r2, const5    # r1 <- (r2 >> const5)             [3]

lui   r1, const16       # r1 <- const16 & 0x0000

la    r1, const32       # r1 <- const32                    [4]
li    r1, const16       # r1 <- 0x0000 & const16           [4]
```

[1] sinaliza ocorrência de *overflow*, [2] ignora ocorrência de *overflow*,
[3] replica sinal, [4] pseudoinstrução.

é expandido para

```
lui r1, %hi(.L1)      # r1 <- 0x0420 & 0000
ori r1, r1, %lo(.L1) # r1 <- 0x0420.0000 OR 0x0000.3610
```

A pseudoinstrução **li** (*load immediate*) é usada para carregar constantes que não são endereços. Se a constante pode ser representada em 16 bits, **li** é expandida para **ori** ou **addi**, dependendo se a constante é um inteiro ou uma constante lógica. Se a constante é maior do que ±32K, então **li** é expandida com o par **lui;ori**.

3.1.2 Acesso a variáveis em memória

Até agora, empregamos somente registradores para manter os valores intermediários em nossas computações. Programas realistas usam um número de variáveis maior do que os 32 registradores – variáveis, vetores e estruturas de dados são alocados em memória. No MIPS, os operandos de todas as instruções que operam sobre dados são registradores, e por isso os operandos devem ser trazidos da memória para que sejam utilizados.

O modelo de memória do MIPS é um vetor com tipo `byte: M[`2^{32}`]`, com 4Gbytes de capacidade. Um *endereço em memória* é o índice **i** do vetor `M[`**i**`]`.

Bytes são armazenados em endereços consecutivos do vetor `M[ ]`, enquanto que palavras de 32 bits são armazenadas em endereços múltiplos de 4 – a memória acomoda 2^{30} palavras. Para simplificar a interface do processador com a memória, as referências devem ocorrer para *endereços alinhados*. Uma referência a um inteiro (uma palavra) deve empregar um endereço que é múltiplo de quatro, enquanto que uma referência a um `short` (*half word*) deve empregar um endereço par. Um `long long` (*double word*) deve ser referenciado num endereço múltiplo de 8. Referências a `char` são naturalmente alinhadas.

São de dois tipos as instruções para acessar a memória: *loads* e *stores*. Estas instruções existem em tamanho *word* (4 bytes), *half word* (2 bytes) e *byte*. Por enquanto, vejamos as de tamanho *word*. A instrução **`lw`** (*load word*) permite copiar o conteúdo de uma palavra em memória para um registrador. A instrução **`sw`** (*store word*) copia o conteúdo de um registrador para uma palavra em memória.

```
lw r1, desl16(r2)    # r1 <- M[ r2 + extSin(desl16) ]
sw r3, desl16(r4)    # M[ r4 + extSin(desl16) ] <- r3
```

O campo `desl16` é um inteiro representado em 16 bits. A soma do conteúdo do *registrador base* (`r2` e `r4`) com a constante estendida é chamada de *endereço efetivo*. O deslocamento pode ser negativo, quando o endereço efetivo é menor do que o apontado pelo registrador base, ou positivo, quando o endereço efetivo é maior do que o apontado pela base.

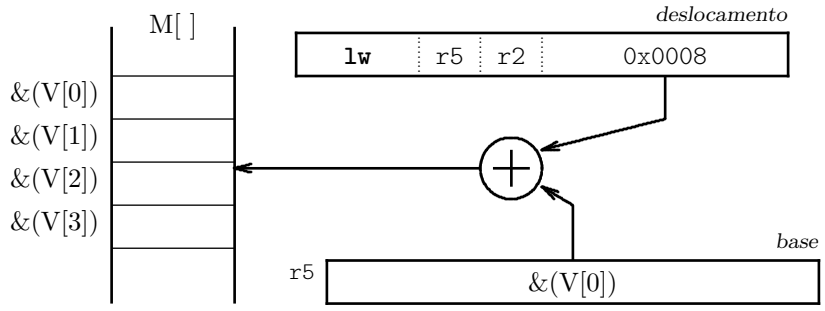


Figura 3.3: Cálculo do endereço efetivo para acessar `V[2]`.

A Figura 3.3 mostra a instrução que efetua um acesso de leitura no terceiro elemento de um vetor de inteiros, indicada abaixo em C e em *assembly*. O endereço de um vetor, na linguagem C ou em *assembly*, é representado pelo nome do vetor, que é `V` no nosso exemplo. Este mesmo endereço pode ser representado, verbosamente, por `&(V[0])`.

C	*assembly*
`i = V[2];`	`lw r2, 2*4(r5)`

O registrador `r5` aponta para o endereço inicial do vetor `V`. Ao conteúdo de `r5` é somado o deslocamento, que é $2 \times 4 = 8$, e o endereço efetivo é `&(V[0])+8`. Esta posição de memória é acessada e seu conteúdo é copiado para o registrador `r2`. O deslocamento com relação à base do vetor, apontado por `r5`, é de 8 bytes porque cada elemento do vetor ocupa quatro bytes consecutivos na memória.

3.1.3 Estruturas de dados em C – vetores e matrizes

Vejamos como acessar estruturas de dados em *assembly*. Antes de mais nada, recordemos os tamanhos das 'coisas' representáveis em C. 'Coisa' não chega a ser um termo técnico elegante, mas a palavra não é sobrecarregada como seria o caso da palavra 'objeto'. A função da linguagem C **`sizeof`**`(x)` retorna o número de bytes necessários para representar

a 'coisa' `x`. A Tabela 3.2 indica o tamanho das 'coisas' básicas da linguagem C – aqui, por 'coisa' entenda-se os tipos básicos das variáveis e constantes representáveis em C.

Tabela 3.2: Tamanho das 'coisas' representáveis em C, em 32 bits.

tipo de dado	`sizeof()`
`char`	1
`short`	2
`int`	4
`long long`	8
`float`	4
`double`	8
`char[12]`	12
`short[6]`	12
`int[3]`	12
`char *`	4
`short *`	4
`int *`	4
`void *`	4

Talvez o mais surpreendente seja a constatação de que ponteiros para caracteres e *strings*, para inteiros, para qualquer 'coisa' enfim, (**`char *`**, **`int *`**, **`void *`**) são *endereços* que *sempre* tem o mesmo tamanho, que é de 32 bits no MIPS.

A Figura 3.4 mostra como seria a alocação em memória de três vetores, de tipos **`char`**, **`short`**, e **`int`**, a partir do endereço 20. Elementos contíguos de vetores e estruturas de dados são alocados em endereços contíguos: `V[i+1]` é alocado no endereço seguinte a `V[i]`; o endereço do 'elemento seguinte' depende do tipo dos elementos do vetor `V`.

endereço	20	21	22	23	24	25	26	27
`char`	`c[0]`	`c[1]`	`c[2]`	`c[3]`	`c[4]`	`c[5]`	`c[6]`	`c[7]`
`short`	`s[0]`		`s[1]`		`s[2]`		`s[3]`	
`int`	`i[0]`				`i[1]`			

Figura 3.4: Endereços de vetores de tipo `char`, `short`, e `int` em C.

Alocação de espaço para variáveis O programador em *assembly* deve alocar espaço para as variáveis em memória. Para tanto o montador provê *diretivas*, que são comandos ao montador, e não instruções do processador. Tipicamente, uma diretiva é uma palavra reservada que inicia com um ponto. O programador, ou o compilador, sinaliza que o que segue é a alocação de espaço para variáveis com a diretiva **`.data`**, indicando que os *labels* que seguem são endereços de variáveis, e não de instruções. Uma *seção de dados* termina quando uma diretiva **`.text`** é encontrada, sinalizando que o que segue é uma seção que contém código.

A diretiva **`.byte`** é seguida de uma ou mais expressões, e cada expressão é montada no próximo byte. As expressões devem ser separadas por vírgulas.

```
label: .byte expr1,expr2,expr3, ...

cars:  .byte 'a','b','c','d'   # vetor de quatro caracteres
nums:  .byte 125,33,19,44,0,12 # vetor de seis números
```

A diretiva **.word** reserva espaço para uma palavra que é inicializada com uma constante. Se a diretiva tem mais de um argumento, são reservadas e inicializadas tantas palavras quantos são os argumentos.

A diretiva **.space** aloca `size` bytes e os preenche com o valor indicado em `fill`. Se `fill` é omitido, os bytes são preenchidos com zeros.

```
vet_4:      .word 1,2,3,4       # aloca e inicializa 4 palavras

label:      .space size, fill   # preeenche SIZE bytes com FILL

vetor_16:   .space 4*16, 0      # vetor de 16 inteiros
matriz_8x8: .space 4*8*8, 0xff # matriz de 8x8 inteiros
```

`vetor_16` é o endereço do primeiro de 16 inteiros (4 bytes), todos preenchidos com zeros, enquanto que `matriz_8x8` é o endereço do primeiro elemento da primeira linha de uma matriz de 8x8 inteiros, todos preenchidos com `0xff`. Mais detalhes sobre as seções emitidas pelo montador, e outras diretivas, são apresentados na Seção 3.4.2.

Exemplo 3.1 Considere a declaração da variável `var` e uma referência para incrementar seu conteúdo, mostrada no trecho de código abaixo, juntamente com sua tradução para *assembly*. A seção **.data** contém a alocação de espaço para as variáveis declaradas no código C, e a seção **.text** contém o código que incrementa a variável.

C

```
int var = 0;
...
var = var + 2;
...
```

assembly

```
.data           # seção de dados
var: .word 0
...
.text           # seção de código
...
la   r1, var    # r1 <- &var
lw   r4, 0(r1)  # r4 <- M[r1+0]
addi r4, r4, 2  # var += 2
sw   r4, 0(r1)  # M[r1+0] <- r4
```

A pseudoinstrução **la** (*load-address*) carrega o endereço da variável no registrador `r1`, e este endereço é usado para ler o conteúdo original (**lw**) e escrever o novo conteúdo (**sw**). ◁

Gerenciamento do acesso às variáveis Quem programa em *assembly* fica responsável por gerenciar o acesso a *todas* as estruturas de dados. A programadora é responsável por acessar palavras de 4 em 4 bytes, elementos de vetores alocados em endereços que dependem do tipo dos elementos, elementos de vetores de estruturas em endereços que dependem do `sizeof()` dos elementos, e assim por diante. Ao programador *assembly* não é dado o luxo de empregar as abstrações providas por linguagens de alto nível tais como C.

Na linguagem C, uma matriz é alocada em memória como um vetor de vetores. Para elementos de tipo τ, linhas com κ colunas, e λ linhas, o endereço do elemento de índices i, j é obtido com a Equação 3.2. A diagrama na Figura 3.5 indica a relação entre o endereço base da matriz ($M = \&(M[0][0])$), linhas e colunas de uma matriz de elementos do tipo τ.

$$\&(M[i][j]) = \&(M[0][0]) + |\tau|(\kappa \cdot i + j) \tag{3.2}$$

Espaço em branco proposital.

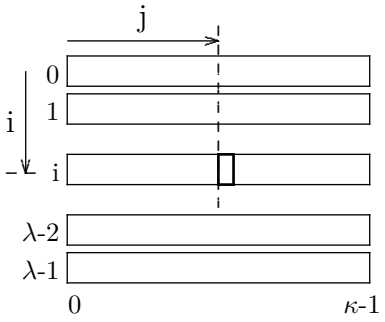


Figura 3.5: Endereço do elemento i, j, de tipo τ, numa matriz $\lambda \times \kappa$.

Exemplo 3.2 Considere o acesso ao vetor de inteiros `V`, mostrado no trecho de código abaixo e sua tradução para *assembly*. Ao registrador `r4` é atribuído o conteúdo de `V[1]` – note o deslocamento de 4 bytes para acessar o segundo inteiro – e ao registrador `r6` é atribuído o conteúdo de `V[2]` com deslocamento de 8 bytes. Ao registrador `r7` é atribuído o valor a ser armazenado no elemento zero do vetor `V`. Os comentários mostram o cálculo do *endereço efetivo*: à base do vetor (em `r1`) é adicionado o deslocamento de `i*4`, para índice `i`.

C

```
int V[256];
...
V[0] = V[1] + V[2]*8;
```

assembly

```
.data
V: .space 256*4

.text
la  r1, V      # r1 <- &(V[0])
lw  r4, 4(r1)  # r4 <- M[r1+1*4]
lw  r6, 8(r1)  # r6 <- M[r1+2*4]
sll r6, r6, 3  # r6*8 = r6<<3
add r7, r4, r6
sw  r7, 0(r1)  # M[r1+0*4] <- r4+r6
```

O código deste exemplo tem uma característica importante: em tempo de compilação – quando o compilador examina o código – é possível determinar sem ambiguidade os deslocamentos com relação à base do vetor, e é por isso que o código emprega deslocamentos fixos nas instruções **`lw`** e **`sw`**. A Figura 3.6 indica os deslocamentos com relação à base do vetor, que é `&(V[0])`. ◁

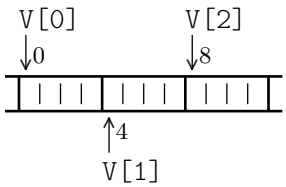


Figura 3.6: Deslocamento com relação à base de `V` dos elementos 0, 1, e 2.

Espaço em branco proposital.

Exemplo 3.3 O trecho de código abaixo é praticamente o mesmo do Exemplo 3.2, exceto que os índices são variáveis, e não constantes – no exemplo anterior, os deslocamentos estão fixados em tempo de compilação, e portanto a geração do código pode usar a informação quanto aos deslocamentos fixados no código. Neste caso, o código deve computar o endereço efetivo em função dos índices que variam ao longo da execução.

Para facilitar a leitura, as variáveis `i,j,k` são armazenadas nos registradores `ri,rj,rk`. A multiplicação por 16 é obtida pelo deslocamento de 4 posições para a esquerda, com a instrução **`sll`** `r6,r6,4`.

No código deste exemplo, os índices são variáveis e portanto os deslocamentos com relação à base do vetor devem ser computados explicitamente: ao endereço base do vetor é somado o índice do elemento multiplicado por 4, porque cada inteiro ocupa 4 bytes.

C

```
int V[256];
...
V[i] = V[j] + V[k]*16;
```

assembly

```
.data
V: .space 256*4

.text
la   r1, V       # r1 <- &(V[0])
sll  r2, rj, 2   # r2 <- j * 4
addu r3, r2, r1  # r3 <- V + j*4
lw   r4, 0(r3)   # r4 <- M[V + j*4]
sll  r2, rk, 2   # r2 <- k * 4
addu r3, r2, r1  # r3 <- V + k*4
lw   r6, 0(r3)   # r6 <- M[V + k*4]
sll  r6, r6, 4   # r6 <- r6*16
add  r7, r4, r6
sll  r2, ri, 2   # r2 <- i * 4
addu r3, r2, r1  # r3 <- V + i*4
sw   r7, 0(r3)   # M[V + i*4] <- r7
```

O próximo exemplo mostra o código para acessar uma estrutura de dados algo mais complexa do que um vetor. ◁

Exemplo 3.4 Considere a estrutura `aType`, com 4 elementos inteiros, `x,y,z,w`, e o vetor `V`, com 16 elementos do tipo `aType`. Como `sizeof(aType)`=16, o deslocamento de `V[3]` com relação à base do vetor é:

3 elementos x 4 palavras/elemento x 4 bytes/palavra = 48 bytes = 0x30 bytes.

Para simplificar o exemplo, suponha que o vetor `V` foi alocado no endereço `0x0080.0000`. Note que o código *assembly* está otimizado para o índice que é a constante 3.

C

```
typedef struct A {
   int x;
   int y;
   int z;
   int w;
} aType;
...
aType V[16];
...
   m = V[3].y;
   n = V[3].w;
   V[3].x = m+n;
```

assembly

```
# 48 = 3*sizeof(aType) = 3*16
la  r5, 0x00800000  # r5 <- &(V[0])
lw  r8, (48+4)(r5)  # r8 <- V[3].y
lw  r9, (48+12)(r5) # r9 <- V[3].w
add r5, r8, r9
sw  r5, (48+0)(r5)  # V[3].x <- m+n
```

A Figura 3.7 indica os deslocamentos com relação à base do vetor de estruturas `aType`, que é `&(V[0])`. Lembre que cada elemento ocupa 4 inteiros, ou 16 bytes. ◁

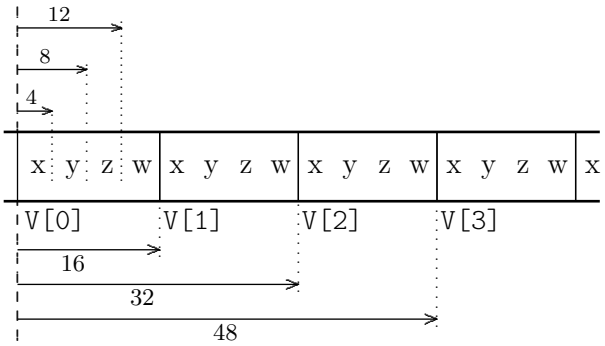


Figura 3.7: Deslocamento com relação à `V[0]` dos elementos 0, 1, e 2, do tipo `aType`.

Exemplo 3.5 Vejamos um exemplo com indexação indireta de um vetor. O conteúdo da i-ésima posição do vetor `X` é usado para indexar o vetor `Y`.

```
int a, i;
int X[2048], Y[256];
...
a = a + X[i] + Y[ (X[i] % 256) ]; // MOD
```

O resto da divisão inteira é obtido com uma divisão, que é uma operação custosa. Ao invés da divisão pode-se usar o seguinte truque: se $P = 2^k,\ k > 1$, então $n \,\%\, P = n \wedge (P-1)$. Para $P = 16$, temos que $n \,\%\, 16 \in [0, 15]$, e $16 - 1 = 15 = 1111_2$. A conjunção de qualquer número com 15 resulta num número que é, no máximo, 15.

```
la   rx,X             # rx <- &(X[0])
la   ry,Y             # ry <- &(Y[0])

sll  t1, ri, 2        # i*4
add  t2, t1, rx       # X + i*4
lw   t3, 0(t2)        # t3 <- X[i]
andi t4, t3, (256-1)  # t3 % 256 = t3 AND 0x0ff
sll  t4, t4, 2        # (t4 % 256)*4
add  t4, t4, ry       # Y + (t4 % 256)*4
lw   t5, 0(t4)        # t5 <- Y[ X[i] ]

add  t6, t5, t3       # X[i] + Y[ X[i] ]
add  ra, ra, t6       # a = a + X[i] + Y[ X[i] ]
```

O cálculo do índice módulo 256 é usado para garantir que, no acesso a `Y`, o índice não extrapola o espaço alocado àquele vetor. ◁

As instruções de movimentação de dados entre registradores e memória são mostradas na Tabela 3.3. As instruções **`lw`** e **`sw`** foram definidas nos parágrafos anteriores. Nesta tabela, por questões de espaço, a função `extSin()` é a função é indicada como `eS()`.

As instruções **`lh`** e **`lhu`** copiam para o registrador de destino os 16 bits apontados pelo endereço efetivo. A instrução **`lh`** (*load half*) estende o sinal do valor lido da memória, enquanto que a instrução **`lhu`** (*load half unsigned*) estende o valor lido com 16 zeros. O mesmo se aplica às instruções **`lb`** (*load byte*) e **`lbu`** (*load byte unsigned*), exceto que para acessos a 8 bits. Na tabela, um campo de `n` bits é mostrado como `((n-1)..0)`.

A instrução **`sh`** (*store half*) copia os 16 bits menos significativos do seu operando para os dois bytes apontados pelo endereço efetivo. A instrução **`sb`** (*store byte*) copia para a

memória o byte menos significativo do seu operando. Os deslocamentos nos acessos à palavra (**lw**, **sw**), e nos acessos à meia-palavra (**lh**, **sh**) devem ser alinhados de 4 e 2 bytes, respectivamente. Acessos à bytes são naturalmente alinhados.

3.1.4 Controle de fluxo de execução

Com o que vimos até o momento temos condições de escrever programas que acessam a memória e efetuam operações de lógica e aritmética com variáveis que foram copiadas para registradores. Isso é muito interessante, mas nos faltam as instruções que permitem decidir se uma determinada operação será ou não efetuada sobre uma dada variável. As instruções que nos permitem incluir a tomada de decisões no código são os *desvios*.

Vejamos um exemplo de decisão, no trecho de programa em C, mostrado abaixo. Dependendo dos valores em `a` e em `b`, ou ocorre uma atribuição à `c`, ou uma atribuição à `f`; qualquer das duas é seguida da atribuição à `j`.

```
if ( a == b ) then {
   c = d + e;
} else {
   f = g - h;
}
j = k * l;
```

As sequências podem ser

```
  c = d + e ; j = k * l;
```

ou

```
  f = g - h ; j = k * l;
```

O compilador gera o código tal que somente uma das cláusulas do `if()` seja executada. Ao programar em *assembly*, a sua tarefa é garantir que o fluxo de execução que é definido no código seja obedecido pela versão traduzida.

Desvios condicionais

O *fluxo sequencial de execução* é determinado pela próxima instrução, que é aquela em PC+4, com relação à instrução corrente. Tomar um desvio significa desviar do fluxo sequencial, e saltar sobre uma ou mais instruções. São duas as instruções que nos permitem avaliar uma condição e então desviar do fluxo sequencial: **beq** e **bne**, ou *branch if equal*, e *branch if not equal*.

Tabela 3.3: Instruções para acesso à memória.

```
lw  r1, des16(r2) # r1 <- M[r2 + eS(des16)]

lh  r1, des16(r2) # r1 <- eS(M[r2 + eS(des16)](15..0))
lhu r1, des16(r2) # r1 <- 0x0000 & M[r2 + eS(des16)](15..0)

lb  r1, des16(r2) # r1 <- eS(M[r2 + eS(des16)](7..0))
lbu r1, des16(r2) # r1 <- 0x000000 & M[r2 + eS(des16)](7..0)

sw  r1, des16(r2) # M[r2 + eS(des16)] <- r1
sh  r1, des16(r2) # M[r2 + eS(des16)](15..0) <- r1(15..0)
sb  r1, des16(r2) # M[r2 + eS(des16)](7..0)  <- r1(7..0)
```

Estas instruções nos permitem codificar os comandos de C `if()`, `while()` e `for(;;)`, por exemplo. Vejamos pois como se dão os desvios da conduta sequencial. O *endereço de destino* é o endereço da instrução para onde o fluxo de execução do programa deve desviar. Veremos adiante como o endereço de destino deve ser computado.

As instruções de desvio comparam o conteúdo de dois registradores e desviam para o endereço de destino se os conteúdos são iguais – **beq** – ou diferentes – **bne**. Se é para desviar, então PC ← `dest` e a execução prossegue daquela instrução; senão PC ← PC+4 e a instrução após o desvio é executada. Esta não é uma descrição exata das instruções de desvio – não tema que sanearemos esta imprecisão em breve.

```
beq r1, r2, dest # branch if equal: desvia se r1 == r2
bne r1, r2, dest # branch if not equal;: desvia se r1 != r2
```

O trecho de código abaixo mostra um desvio, e todas as instruções estão enumeradas para facilitar a descrição. Se os conteúdos de `r1` e `r2` são iguais, a próxima instrução é o **sw** em `L5`, do contrário, é o **add** em `L2`. A programadora identifica o destino do desvio, que é a instrução que deve ser executada caso a condição do teste seja verdadeira, e o montador traduz o símbolo `L5` para um endereço, e este endereço é o terceiro operando da instrução de desvio.

```
L1:  beq r1, r2, L5 # salta para L5 se r1==r2
L2:  add r5, r6, r7
L3:  sub r8, r9, r10
L4:  xor r11, r12, r13
L5:  sw  r14, 0(r15)
```

Os 'nomes' atribuídos às instruções no exemplo acima, seus *labels*, servem justamente para nomear uma instrução ou uma variável. O *label* associa um *nome simbólico* ao endereço em que aparece. O nome simbólico é algo de relevante ao programador, enquanto que seu endereço é necessário ao processo de tradução de *assembly* para binário. Para o montador, um *label* é um símbolo que inicia por '.', '_' ou uma letra, é seguido de letras, dígitos, '.' ou '_', e terminado por ':'. Os símbolos `_start:`, `.L23:`, e `a_b_c:` são *labels* válidos; `23x:` e `add` são inválidos.

Vejamos enfim a definição precisa do *destino de um desvio*. No MIPS, o destino de um desvio é a distância, ou o número de instruções por saltar, tomando por base a instrução seguinte ao desvio. O endereço de destino é determinado pelo *deslocamento* com relação ao PC+4, e é a distância, medida em instruções, do PC+4 até o endereço da instrução de destino. A razão para tal definição é discutida na Seção 4.3.

A definição completa da instrução **beq** `r1, r2, desloc` é: se desvio for tomado, o *endereço de destino* é (PC+4) + `extSin(desloc)`×4, ou (PC+4) se o desvio não for tomado. A constante `desloc` é o número de instruções por saltar com relação a PC+4. O deslocamento é representado em 16 bits, em complemento de dois, o que possibilita desvios com distância de até 32K instruções para adiante (`desloc` > 0) ou até de 32K instruções para trás (`desloc` < 0). A multiplicação por quatro ajusta o deslocamento para saltar instruções ao invés de bytes, e é obtida com um deslocamento de duas posições para a esquerda, ou a concatenação de dois zeros à direita.

Temos um conflito de versões aqui. O processador executa a instrução definida no parágrafo acima: o destino é (PC+4) + `extSin(desloc)`×4, ou o destino é (PC+4). O montador facilita sobremaneira a vida da programadora ao permitir que o código seja escrito usando *label*s como o destino, e transformar auto-magicamente o endereço do *label* para o número de instruções por saltar.

Agora que sabemos computar o endereço de destino dos desvios, qual é o efeito das sequên-

cias abaixo? A instrução **nop**, ou *no-operation*, não tem efeito algum, mas é uma instrução muito mais útil do que seu despretensioso nome possa indicar. Lembre que o destino é o deslocamento, contado em instruções de 4 bytes, com relação PC+4, que é instrução imediatamente após o desvio.

```
L1: nop
L2: beq r1, r1, -1
L3: nop

L4: nop
L5: beq r1, r1, 0
L6: nop

L7: nop
L8: beq r1, r1, +1
L9: nop
```

No desvio em `L2`, o deslocamento de -1 com relação a `L3` é `L2` ((PC+4)−4) e este é um laço infinito. O desvio em `L5` salta zero instruções com relação a `L6` e portanto se comporta como mais um **nop** ((PC+4)+0). O desvio em `L8` pula +1 instrução com relação a `L9`, e desvia portanto para a instrução após `L9` ((PC+4)+4).

O deslocamento é codificado numa constante de 16 bits, representada em complemento de dois, e portanto o conjunto dos endereços atingíveis fica no intervalo $[-2^{15}, +2^{15})$. Note que são $\pm 2^{15}$ instruções e não $\pm 2^{15}$ bytes, e que o intervalo é centrado em PC+4.

> *Folclore: na média, uma linha de código C é traduzida para 10 instruções de assembly.*
>
> *Evidentemente, a proporção varia em função do processador e da linguagem, mas esta é uma boa estimativa para o tamanho do código traduzido.*

Se o endereço de destino está mais longe do que $\pm 2^{15}$ instruções, então o compilador/montador deve alterar o código e adicionar uma instrução que alcançe o endereço longínquo – não se apoquente, tais instruções existem. Este é um caso relativamente raro porque, como reza a lenda, 32.000 instruções são 3.200 linhas de código C. Não são usuais laços com $\geqslant$ 3.200 linhas, ou cláusulas de **if**s com $\geqslant$ 3.200 linhas.

Comparação de magnitude

Comparações de igualdade são muito úteis mas comparar magnitude é igualmente necessário. A instrução **slt**, ou *set on less than*, compara a magnitude de seus segundo e terceiro operandos e atribui 1 ou 0 ao primeiro operando, dependendo do resultado da comparação. Aqui, como na linguagem C, 0 significa *falso* enquanto que 1 significa *verdadeiro*. O comando em C, que equivale à **slt** é a atribuição condicional:

```
rd = ( (r1 < r2) ? 1 : 0 );
```

A comparação de magnitude é efetuada com uma subtração; se o conteúdo de `r1` é menor do que o de `r2`, então o resultado deve ser negativo, e o bit de sinal do resultado é atribuído ao registrador `rd`.

São quatro as instruções de comparação de magnitude. **slt** pode gerar uma exceção de *overflow* caso a subtração dos operandos (`r1`−`r2`) produza resultado que não é representável em 32 bits. **sltu** não sinaliza a ocorrência de overflow. **slti** compara o conteúdo de um registrador com o valor de uma constante de 16 bits com o sinal estendido, e sinaliza a ocorrência de *overflow*. **sltiu** não sinaliza a ocorrência de *overflow*.

```
slt   rd, r1, r2       # rd <- (r1 < r2) ? 1 : 0
sltu  rd, r1, r2       # rd <- (r1 < r2) ? 1 : 0
slti  rd, r1, const16  # rd <- (r1 < extSin(const16)) ? 1 : 0
sltiu rd, r1, const16  # rd <- (r1 < extSin(const16)) ? 1 : 0
```

Combinando a comparação de magnitude com a comparação de igualdade, pode-se decidir em função da magnitude de operandos. São necessárias duas instruções, um **slt** e um desvio. O par de instruções abaixo equivale a **blt**, ou *branch on less than*. Lembre que r0 é sempre zero e representa *falso*.

```
slt r1, r2, r3   # r1 <- (r2 < r3)     TRUE=1, FALSE=0
bne r1, r0, dest # salta para dest se r1=TRUE != FALSE=r0
```

Esta sequência equivale a **bge**, ou *branch if greater or equal*.

```
slt r1, r2, r3   # r1 <- (r2 < r3)      TRUE=1, FALSE=0
beq r1, r0, dest # salta para dest se r1=FALSE == FALSE=r0
```

Saltos incondicionais

Os *saltos incondicionais* efetuam uma mudança no fluxo de execução incondicionalmente. O conjunto de instruções do MIPS nos oferece três saltos incondicionais: **j**, ou *jump*, **jr**, ou *jump register*, e **jal**, ou *jump and link*.

```
j   ender26      # PC  <- (PC+4)(31..28) & ender26 & 00
jr  rt           # PC  <- rt
jal ender26      # PC  <- (PC+4)(31..28) & ender26 & 00 ,
                 # r31 <- PC+4
```

Nos interessa agora a instrução **j**, que atribui ao PC o endereço de uma instrução, quer dizer, o argumento de 26 bits é concatenado com dois zeros, representando portanto um endereço alinhado na fronteira de uma palavra, com 28 bits. Infelizmente, isso não basta. O espaço de endereçamento é de 2^{30} instruções, e os quatro bits faltantes são tomados do PC+4 – os 4 bits mais significativos do PC incrementado são concatenados aos 26 bits da instrução e aos dois zeros. O endereço de destino resultante tem portanto 32 bits.

A instrução **jal** é uma das raras instruções que afetam mais de um elemento de estado. Além de atribuir o endereço de destino ao PC, esta instrução armazena o PC+4 no registrador r31. A vírgula indica execução simultânea: um valor é atribuído ao PC, e outro ao r31. Quando estudarmos a implementação de funções, veremos a razão para armazenar o PC+4.

A Tabela 3.4 mostra as instruções de controle de fluxo. Nesta tabela, por questões de espaço, a função extSin() é a função é indicada como eS(). A pseudoinstrução *branch always* (**b** dest16) é uma abreviatura para **beq** r0, r0, dst16.

Isso tudo posto, vejamos como traduzir código com saltos e desvios. Nestes exemplos os registradores são identificados com os nomes das variáveis.

Exemplo 3.6 Vejamos como traduzir um **if**() simples.

C	*assembly*
`if (a == b)`	`bne ra, rb, L1`
`  c = d + e;`	`add rc, rd, re`
`f = g - h;`	`L1: sub rf, rg, rh`

Se os conteúdos de a e b são iguais, então a soma e a subtração são efetuadas. Se a e b são diferentes, então somente a subtração é efetuada. Note a inversão no teste: se a é *diferente* de b, enão salta a instrução da soma. ◁

Tabela 3.4: Instruções para saltos e desvios.

```
slt   rd, r1, r2      # rd <- (r1 < r2 ? 1 : 0)           [1]
slti  rd, r1, cnst16  # rd <- (r2 < eS(cnst16) ? 1 : 0)   [1]

sltu  rd, r1, r2      # rd <- (r1 < r2 ? 1 : 0)           [2]
sltiu rd, r1, cnst16  # rd <- (r2 < eS(cnst16) ? 1 : 0)   [2]

beq r1, r2, dst16     # PC <- PC+4+(r1==r2 ? eS(dst16)*4 : 0)
bne r1, r2, dst16     # PC <- PC+4+(r1!=r2 ? eS(dst16)*4 : 0)
b   dst16             # PC <- PC+4+eS(dst16)*4            [3]

j   ender26           # PC <- (PC+4)(31..28) & ender26 & 00
jr  rt                # PC <- rt
jal ender26           # PC <- (PC+4)(31..28) & ender26 & 00 ,
                      # r31 <- PC+4
```

[1] sinaliza ocorrência de *overflow*, [2] ignora ocorrência de *overflow*, [3] pseudoinstrução.

Exemplo 3.7 Vejamos a tradução de um `if()` com cláusulas mutuamente exclusivas: *ou* a cláusula do `if` é executada; *ou* a cláusula do `else` é executada, mas nunca as duas em sequência.

C

```
if (a == b)
   c = d + e;
else
   f = g - h;
```

assembly

```
      bne ra, rb, Else
If:   add rc, rd, re
      j Exit          # salta else
Else: sub rf, rg, rh
Exit: nop
```

A função do *jump* é saltar sobre as instruções da cláusula `else`. ◁

Exemplo 3.8 Vejamos como traduzir um `if()` cuja condição tem dois testes.

C

```
if ( (a == b) && (c == d) ) {
   e = f + g;
}
h = i - j;
```

assembly

```
      bne ra, rb, L2
      nop
      bne rc, rd, L2
      add re, rf, rg
L2:   sub rh, ri, rj
```

Se os conteúdos de `a` e `b` são iguais, e mais ainda, se os conteúdos de `c` e `d` são iguais, então a soma é efetuada. A razão para o `nop` entre os dois desvios é apresentada na Seção 4.3.

Este trecho de código exemplifica a "avaliação preguiçosa" das condições em C. No caso de uma conjunção (`&&`), se o primeiro teste resulta falso, o segundo é desnecessário porque ($0 \wedge * = 0$), emquanto que para uma disjunção (`||`), se o primeiro teste resulta verdadeiro, o segundo é desnecessário porque ($1 \vee * = 1$). ◁

Exemplo 3.9 Este exemplo é um tanto mais interessante. O laço procura o índice do primeiro elemento de um vetor que é diferente de `c`. Supondo que os elementos do vetor sejam inteiros, o índice é multiplicado por quatro (`r9 <- ri*4`) e então adicionado à base do vetor `vet`. Recorde que `vet = &(vet[0])`.

C

```
while (vet[i] == c)
   i = i + j;
```

assembly

```
   la  r7, vet      # r7 <- vet
L: sll r9, ri, 2    # r9 <- i*4
   add r9, r7, r9   # r9 <- i*4+vet
   lw  r8, 0(r9)    # r8 <- M[ r9 ]
   bne r8, rc, End  # vet[i] != c ?
   add ri, ri, rj   # i <- i + j
   j   L            # repete
End: nop
```

O valor lido da memória é comparado com `c`, o que causa a terminação do laço com o **`bne`**, ou sua continuação (**`j`** `L`). ◁

Exemplo 3.10 O endereço de destino da instrução *jump* é obtido da concatenação dos quatro bits mais significativos do PC incrementado, com os 26 bits da própria instrução, mais dois zeros:

`j` ender26 *#PC <- (PC+4)(31..28) & ender26 & 00*.

A instrução aponta para um endereço chamado de *pseudo-absoluto* porque a faixa de endereços alcançáveis é de 2^{28} bytes ou $2^{26} = 64$M instruções. Os quatro bits mais significativos do PC incrementado dividem o espeço de endereçamento em 16 faixas de 64M instruções cada faixa.

Se o montador detecta um endereço de destino que extrapola a faixa de 64M instruções, o próprio montador altera o código e constrói um endereço *absoluto* num registrador, e este registrador é usado como destino de uma instrução **`jr`**. O trecho de código abaixo mostra as duas versões.

código original

```
       j   ender
       nop
       ...
       # mais de 64M instr
       # de distância
ender: add ra, rb, rc
```

modificado pelo montador

```
       lui  at, %hi(ender)
       ori  at, at, %lo(ender)
       jr   at
       ...
       # mais de 64M instr
       # de distância
ender: add ra, rb, rc
```

O registrador `r1`, apelidado de `at` (*assembler temporary*), é reservado para uso pelo montador para armazenar valores temporários, como o endereço de destino no código modificado.

Como indicado no Folclore da página 31, 64M instruções correspondem a aproximadamente 6,4 milhões de linhas de código C, o que é uma distância considerável para um salto, e essa manipulação raramente é necessária. ◁

Espaço em branco proposital.

Exemplo 3.11 Vejamos um laço que inclui o código do Exemplo 3.5.

```
int x[2048], y[256];
int a = 0; int i = 0;
...
while (i < 1024) {
   a = a + x[i] + y[ (x[i] % 256) ]; // MOD
   i = i + 1;
}
```

O teste do laço usa uma instrução `slti` para fazer a comparação de magnitude, que resulta em verdadeiro=1 ou falso=0, e o registrador `$zero` é usado na comparação com falso.

```
        la   rx, x            # rx <- &(x[0])
        la   ry, y            # ry <- &(y[0])
        li   ri, 0            # i <- 0
        li   ra, 0            # a <- 0

while:  slti t0, ri, 1024     # t0 <- (ri < 1024)
        beq  t0, $zero, fim   # t0 == FALSE -> fim
        sll  t1, ri, 2        # i*4
        add  t2, t1, rx       # x + i*4
        lw   t3, 0(t2)        # t3 <- x[i]
        andi t4, t3, (256-1)  # t3 % 256 = t3 AND 255
        sll  t4, t4, 2        # (t4 % 256)*4
        add  t4, t4, ry       # y + (t4 % 256)*4
        lw   t5, 0(t4)        # t5 <- y[ x[i] ]
        add  t6, t5, t3       # x[i] + y[ x[i] ]
        add  ra, ra, t6       # a = a + x[i] + y[ x[i] ]
        addi ri, ri, 1
        j    while

fim:    nop
```

Como o limite do laço é uma constante, o teste poderia estar no final do laço, o que eliminaria a instrução `j`, e tornaria o código mais eficiente. Se o limite fosse uma variável, então o teste deveria estar, necessariamente, no topo do laço porque não é possível prever, durante a compilação, quantas voltas seriam executadas. ◁

3.1.5 Estruturas de dados em C: cadeias de caracteres

Na linguagem C, cadeias de caracteres, ou *strings*, são vetores de caracteres terminados por `'\0'`. Uma *string* é um vetor do tipo **`char`**, de tamanho não definido, sendo o final da *string* sinalizado pelo caractere `'\0'`, que é `0x00`.

No código fonte, *strings* são representadas entre aspas duplas, enquanto que caracteres são representados entre aspas simples. Quando lemos código C, na *string* `"palavra"` não vemos o caractere o `'\0'`, mas ele ocupa o espaço necessário para sinalizar o fim da cadeia. Supondo que esta *string* seja alocada em memória a partir do endereço 0x400, o que é armazenado é o vetor mostrado na Figura 3.8. Quando se computa o tamanho de uma *string*, o `'\0'` deve ser contado porque ele ocupa espaço, embora seja invisível. Veja a Seção 2.4 para a codificação de texto e o alfabeto ASCII e suas extensões.

Exemplo 3.12 O trecho de código no Programa 3.2 copia uma cadeia de caracteres, do vetor `fte`, para o vetor de caracteres `dst`, e no Programa 3.3 está a sua tradução para *assembly*.

A condição do laço contém uma leitura da memória (`fte[i]`), e no corpo do laço uma leitura em `fte` e uma escrita em `dst`. A segunda leitura é desnecessária porque o valor que é usado para testar a condição é o mesmo a ser usado na atribuição. A caractere `'\0'` não é atribuído ao destino no corpo do laço, e por isso é atribuído após o seu final.

Na tradução para *assembly* é necessário lembrar que cada elemento dos vetores ocupa um byte e portanto as instruções para acessar `fte` e `dst` devem ser **`lbu`** e **`sb`**. Os elementos dos vetores são caracteres representados em 8 bits – não são inteiros de 8 bits – e por isso a instrução *load byte unsigned* (**`lbu`**) é usada: quando o byte é carregado para o registrador de 32 bits, o valor é estendido com 24 zeros e não com o sinal (bit 7) do valor lido. Lembre que para representar a concatenação usamos o '&', do VHDL.

O registrador `r5` recebe o caractere de `fte[i]` e este é estendido com 24 zeros na esquerda. Quando o caractere `'\0'` é lido, ao registrador `r5` são atribuídos 32 zeros (24 & 8) e é por isso que o `'\0'` é comparado com `r0` no **`beq`** `r5,r0`. A instrução *store byte* (**`sb`**) escreve somente o byte menos significativo na memória e portanto não é necessário nenhum tipo de extensão.

O compilador, ou o montador, aloca o espaço necessário em memória para acomodar os vetores fonte e destino, e os endereços destas variáveis podem ser referenciados pela programadora, ao usar os nomes `fte` e `dst`.

No Programa 3.3, o índice `i` é incrementado dentro do laço, mas o endereço `&(dst[i])` *não* é computado explicitamente após o teste `fte[i] != '\0'`. O deslocamento de 1 no **`sb`** da última instrução tem o mesmo efeito que adicionar `r4` a `r19` após a saída do laço. ◁

Programa 3.2: Laço que copia uma *string* para um vetor de `char`.

```
char fte[16]="abcd-efgh-ijkl-"; // sizeof(fte)=16, com '\0'
char dst[32];
int i;

i = 0;
while ( fte[i] != '\0' ) {       // terminou?
   dst[i] = fte[i];
   i = i + 1;
}
dst[i] = '\0';                   // atribui o '\0'
```

endereço	400	401	402	403	404	405	406	407
caractere	'p'	'a'	'l'	'a'	'v'	'r'	'a'	'\0'

Figura 3.8: Leiaute de uma *string* em memória.

Programa 3.3: Versão em *assembly* do laço que copia uma *string*.

```
        la   r8, fte        # r8 <- fte
        la   r9, dst        # r9 <- dst
        add  r4, r0, r0     # i = 0;
                            # while ( fte[i] != '\0' ) {
lasso:  add  r18, r8, r4    #    r18 <- fte+i
        lbu  r5, 0(r18)     #    r5 <- 0x0000.00 & fte[i]
        beq  r5, r0, fim    #    (r5 == '\0') ? -> terminou
        add  r19, r9, r4    #    r19 <- dst+i
        sb   r5, 0(r19);    #    dst[i] <- (char)fte[i]
        addi r4, r4, 1      #    i = i + 1;
        j    lasso          # }
fim:    sb   r0, 1(r19)     # dst[i] = '\0';
```

Espaço em branco proposital.

Exemplo 3.13 O trecho de código no Programa 3.4 percorre uma lista encadeada, cujos elementos são do tipo `elemType`. O primeiro componente da estrutura é um apontador para o próximo elemento da lista, e o segundo componente é um vetor de seis inteiros.

Antes do laço, o apontador é inicializado com o endereço da estrutura de dados. Se a lista é não vazia então o apontador não é nulo, e os elementos do vetor de inteiros são inicializados. Isso feito, o teste é repetido para o próximo elemento da lista.

Na versão em assembly, no Programa 3.5, o registrador `rp` é carregado com o endereço do primeiro elemento do vetor, e as constantes são carregadas em seis registradores. Do ponto de vista de eficiência da execução do código, estas constantes *devem* ser carregadas *fora* do laço para evitar a repetição destas operações, cujo resultado é constante, no corpo do laço.

O teste compara o valor do apontador para o próximo elemento (`rn = p->next`) com `NULL` e o laço termina se `rn == r0`. Os elementos do vetor são inicializados com deslocamentos de 4 (apontador) mais o índice multiplicado por 4.

No final do laço, o apontador é de-referenciado para que `rn` aponte para o próximo elemento da lista, e o teste é então repetido. Note que o conteúdo de um apontador é um endereço, que pode ser usado diretamente como tal. ◁

Programa 3.4: Laço que percorre uma lista encadeada com *pointers*.

```
typedef struct elem {
   elem *next;
   int   vet[4];
} elemType;

elemType *p;
elemType estrut[256];
   ...
   p = estrut;      // p <- &(estrut[0])
   while (p->next != NULL) {
      p->vet[0] = 1;
      p->vet[1] = 2;
      p->vet[2] = 4;
      p->vet[3] = 8;
      p = p->next;
   }
```

Programa 3.5: Versão em *assembly* do laço que percorre uma lista encadeada com *pointer*.

```
       la rp, estrut    # rp <- &(estrut[0])
       li r1, 1         # estes são inicializados FORA do laço
       li r2, 2         #   porque são valores constantes
       li r3, 4
       li r4, 8

lasso: lw   rn, 0(rp)   # rn <- p->next
       beq  rn, r0, fim # (p->next == NULL) ? terminou
       sw   r1, 4(rp)   # vet[0] <- 1; deslocamento = 4*(i+1)
       sw   r2, 8(rp)   # vet[1] <- 2
       sw   r3, 12(rp)  # vet[2] <- 4
       sw   r4, 16(rp)  # vet[3] <- 8
       move rp, rn      # próximo: p <- p->next
       j    lasso

fim:   nop
```

3.1.6 Exercícios

Ex. 25 Traduza para *assembly* do MIPS os trechos de programa em C abaixo. Em C, o valor de um comando de atribuição é o valor atribuído. As constantes são todas 1024.

```
// (a) ----------------------------------------
int P[NN];  int Q[MM];  int k;

k = P[4] - P[9];
P[ P[k] ] = Q[i] + Q[k*4] + Q[i+j];

// (b) ----------------------------------------
int i, sum, v[NN];
...
for (sum=0, i=0; i < NN; i+=2)
   sum += v[i];

// (c) ----------------------------------------
char *fte, *dst;
...
while ( ( *dst = *fte ) != '\0') {
   dst++; fte++; i++;
}

// (d) ----------------------------------------
char fte[NN]; char dst[NN];
int i, num;
...
i = 0;
while (fte[i] != '\0') {
   i = i + 1;
}
num = i;
for (i=0; num > 0; num--, i++) {
   dst[i] = fte[num - 1];
}

// (e) ----------------------------------------
typedef struct A {
   int   x;
   short z[4];
   char  s[8];
} aType;
aType V[ SZ ]; // compilador aloca V em 0x0040.0000
int i,a,b,c;
...
a = b = c = 0;
for (i=0; i < SZ; i+=4) {
   a = a + V[i].x + (int)V[i].z[1];
   b = b + (int)(V[i].s[1] + V[i].s[7]);
   c = c + V[i].x - (int)V[i].z[3];
}

p = q = r = 0;
for (i=0; i < SZ; i+=16) {
   p = V[i].x;
   q = q + (int)(V[i].s[(p % 8)] + V[i].s[(p % 8)]);
   r = r + V[i].x - (int)V[i].z[(q % 4)];
}
```

3.2 Implementação de Funções em *Assembly*

Nesta seção examinamos com algum detalhe a implementação em *assembly* de funções escritas na linguagem C.

3.2.1 Definição e declaração de funções em C

A *definição* de uma função declara o tipo do valor a ser retornado, declara os tipos dos parâmetros, e contém o corpo da função que computa o valor da função a partir dos parâmetros. Além dos comandos, o corpo da função pode conter declarações das variáveis locais à função. O Programa 3.6 mostra o esqueleto da definição de uma função na linguagem C [**?**, **?**].

Programa 3.6: Definição de uma função em C.

```
tipo nome_da_função( parâmetros formais ) { // cabeçalho
   declarações                               // corpo da função
   comandos
}
```

Para que uma função que é definida em outro arquivo com código fonte possa ser usada, é necessário que ela seja *declarada* antes da primeira invocação, porque sem a declaração, o compilador não tem como gerar o código para invocar a função. O mesmo vale para funções definidas em bibliotecas. A declaração contém somente os tipos dos parâmetros e do valor retornado pela função. Tipicamente, as declarações de funções são agrupadas num arquivo de cabeçalho que é incluído nos arquivos que necessitam daquelas funções. O Programa 3.7 mostra uma declaração de função.

Programa 3.7: Declaração de uma função em C.

```
tipo nome_da_função( lista de tipos dos parâmetros );
```

O Programa 3.8 mostra algumas declarações de funções. Uma função que não retorna um valor tem o 'tipo' `void`. Uma função sem parâmetros tem *um* parâmetro de 'tipo' `void`. A palavra 'tipo' aparece entre aspas porque `void` não é exatamente um tipo, no sentido estrito do termo, mas sim um *placeholder* para o tipo ou argumento nulo. A função `j()` é similar à função `printf()`, que recebe um argumento que, ao ser interpretado em tempo de execução, determina quantos e quais os tipos dos demais parâmetros.

Programa 3.8: Exemplos de declarações de funções em C.

```
void f(void);        // sem argumentos e não retorna valor
int  g(void);        // sem argumentos e retorna inteiro
int  h(int, char);   // dois argumentos, retorna inteiro
int  j(const char *, ...); // número variável de argumentos
```

3.2.2 Avaliação de expressões e de funções

Como são avaliados os comandos e expressões em C? Da esquerda para a direita, em avaliação preguiçosa, e com efeitos colaterais. *Avaliação preguiçosa* consiste em avaliar uma expressão somente até que seu valor seja determinado. Por exemplo, a avaliação preguiçosa da expressão $0 \wedge X$ ignora o valor de X porque a conjunção de 0 com qualquer coisa é 0; da mesma forma, a avaliação de $1 \vee Y$ ignora o valor de Y porque a disjunção de 1 com qualquer coisa é 1.

Uma linguagem com *efeitos colaterais* permite que os efeitos da avaliação de uma subexpressão alterem a avaliação de outras subexpressões de uma mesma expressão. Por exemplo, este comando é válido em C e tem um efeito colateral da avaliação de `E`:

```
a = E + (E = a*b) + z*E + w/E;
```

O lado direito da atribuição é avaliado da esquerda para a direita e o "valor de uma atribuição" é o "valor atribuído". Na primeira parcela da soma, `E` tem o valor que lhe fora atribuído anteriormente, enquanto que nas demais parcelas `E` vale `a*b`. Programas com efeitos colaterais podem ser extremamente difíceis de depurar porque os valores das subexpressões mudam durante a avaliação da expressão que as contêm. Facilmente, o código pode se tornar ininteligível.

Pior ainda, o manual que define a linguagem C informa que a ordem de avaliação pode ser escolhida arbitrariamente pelo compilador e que o resultado não é portável[1]. Por *código portável* entende-se o código fonte que, compilado em qualquer máquina e com qualquer compilador, produz resultados idênticos[2].

Os argumentos de uma função também são avaliados da esquerda para a direita, com avaliação preguiçosa e efeitos colaterais, com este algoritmo:

1. Cada expressão na lista de argumentos é avaliada, da esquerda para a direita;
2. se necessário, os valores das expressões são convertidos para o tipo do parâmetro formal, e o valor é atribuído ao argumento correspondente no corpo da função;
3. o corpo da função é executado;
4. se um comando **`return`** é executado, o controle é devolvido à função que chamou;
5. se o **`return`** inclui uma expressão, seu valor é computado e o tipo convertido para o tipo do valor de retorno da função. Se o **`return`** não contém uma expressão, nenhum valor útil é retornado. Se o corpo da função não inclui um **`return`**, então o controle é devolvido quando a execução do corpo da função chegar ao seu último comando;
6. todos os argumentos são passados "por valor" (*call by value*), mesmo que o 'valor' seja um endereço (*pointer*).

Como é avaliado o comando `a = f(p*2, q, g(r,s,t), q/2, x+4, y*z);` ?

1. o valor de `p*2` é atribuído ao primeiro argumento;
2. o conteúdo da variável `q` é atribuído ao segundo argumento;
3. a função `g()` é avaliada com argumentos `r`, `s` e `t`, e seu valor atribuído ao terceiro argumento;
4. `q/2` é avaliado e atribuído ao quarto argumento;
5. `x+4` é avaliado e atribuído ao quinto argumento;
6. `y*z` é avaliado e atribuído ao sexto argumento; e
7. a função é invocada e seu valor de retorno atribuído à variável `a`.

Regras de escopo

O valor de identificadores, ou os seus conteúdos, só pode ser acessado nos blocos em que são declarados.

[1] Conheço um exemplo de código que produz resultados distintos para duas versões do *mesmo* compilador.

[2] De acordo com `stackoverflow.com`, o manual de C99, informa que *"the order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call"*.

Classes de armazenagem (*storage classes*)

As *classes de armazenagem*, que determinam o local em que uma determinada variável é armazenada, são:

`auto` variáveis declaradas dentro de um bloco (variáveis locais), armazenadas na pilha;

`extern` variáveis declaradas fora do corpo de uma função; seu escopo se estende a todas as funções que aparecem após sua declaração. Funções podem ser declaradas como **`extern`**;

`register` indica ao compilador que variável deve, se possível, ser alocada num registrador físico (raramente implementado nos geradores de código/compiladores);

`static` variáveis declaradas como **`static`** num bloco retém seus valores entre execuções do bloco;

`static (external)` variáveis declaradas fora de um bloco mas com escopo restrito ao arquivo em que são declaradas. Funções declaradas como **`static`** são visíveis apenas no arquivo em que são declaradas;

`volatile` variáveis são declaradas 'voláteis' porque podem corresponder ao registrador de um periférico, ou a uma variável compartilhada entre o programa e um tratador de interrupção. O conteúdo destas variáveis pode mudar sem que código do programa as altere e deve-se notificar o compilador para não otimizar referências a uma variável que é volátil;

`const` se uma variável é declarada como sendo 'constante', então o compilador emite um erro se houver uma tentativa de atribuição a esta variável.

Variáveis das classes **`extern`** e **`static`**, se não forem inicializadas pelo programador, são inicializadas em 0 pelo compilador.

3.2.3 Implementação de funções no MIPS32

O conjunto de instruções MIPS32r2 provê duas instruções para o suporte a funções, *viz*:

`jal` `end` *jump and link*, com dois efeitos: salta para o endereço indicado no argumento e salva o endereço de retorno em `r31=ra` (*return address*), que é o *link*:

```
jal ender #  PC <- ender , ra <- PC+4
```

`jr` `reg` *jump register*, que salta para o endereço de ligação/retorno, que foi armazenado em `ra` por **`jal`**:

```
jr ra     #  PC <- ra
```

A Tabela 3.5 mostra a convenção de uso dos registradores definida na *Application Binary Interface* (ABI) do MIPS32 [**?**]. Somente os registradores `r0` e `r31` tem usos determinados pelo *hardware*; a utilização de todos os demais é fruto de convenção de *software*.

Espaço em branco proposital.

Tabela 3.5: Convenção de uso de registradores para chamadas de função.

REG	FUNÇÃO	NÚMERO
`$zero`	sempre zero (em *hardware*)	`r0`
`at`	temporário para montador (*assembly temporary*)	`r1`
`v0-v1`	dois registradores para retornar valores (*value*)	`r2,r3`
`a0-a3`	quatro regs. para passar argumentos	`r4-r7`
`s0..s7`	regs. 'salvos' são preservados (*saved*)	`r16-r23`
`t0..t9`	regs 'temporários' não são preservados	`r8-r15,r24,r25`
`k0,k1`	temporários para o SO (*kernel*)	`r26,r27`
`gp`	*global pointer* (dados estáticos 'pequenos')	`r28`
`sp`	apontador de pilha (*stack pointer*)	`r29`
`fp`	apontador do registro de ativação (*frame pointer*)	`r30`
`ra`	endereço de retorno (*return address*, em *hardware*)	`r31`

A cada chamada de função encontrada num programa, o compilador deve gerar instruções para efetuar os 6 passos listados abaixo. O Programa 3.9 mostra o código *assembly* para a implementação do comando `z = f(x);`. As variáveis são inteiros e `f()` retorna um inteiro. Os números das linhas indicadas referem-se ao Programa 3.9.

1. Alocar os argumentos onde o corpo da função possa encontrá-los (linha 1);
2. transferir controle para a função e armazenar *link* no registrador `ra` (linha 2);
3. o corpo da função deve alocar o espaço necessário na pilha para computar seu resultado (linha 5);
4. executar as instruções do corpo da função (linha 6);
5. colocar o valor computado onde a função que chamou possa encontrá-lo (linha 7);
6. devolver o espaço alocado em pilha (linha 8); e
7. retornar controle ao ponto de invocação da função (linha 9).

Programa 3.9: Protocolo de invocação de função.

```
1      move a0,rx        # prepara argumento
2      jal  f            # salta para a função e salva link
3      move rz,v0        # valor da função, end. de retorno=3
4      ...
5  f:  addi sp, sp, -32  # aloca espaço na pilha, 32 bytes
6      ...               # computa valor
7      move v0, t0       # prepara valor por retornar
8      addi sp, sp, 32   # devolve espaço alocado na pilha
9      jr   ra           # retorna, usando o link
```

3.2.4 Registro de ativação

Qual é a estrutura de dados necessária para suportar funções? Por que?

Uma *função folha* é uma função que não invoca outra(s) função(ões). Um *registro de ativação* (*stack frame*) é alocado para cada função não-folha e para cada função folha que necessita alocar espaço para variáveis locais. A pilha cresce de endereços mais altos para endereços mais baixos. A Figura 3.9 mostra o leiaute de um registro de ativação completo.

	registro da função que chamou
	argumentos a mais que 4 (5,6,7...)
fp →	registradores com argumentos 1 a 4, se existem (a0..a3)
	endereço de retorno (ra)
	registradores salvos, se alterados pela função (s0..s7)
	variáveis locais, se existem
sp →	área para construir argumentos a mais que 4, se existem
	registro da próxima função a ser chamada

Figura 3.9: Registro de ativação no MIPS-32.

Um registro de ativação deve conter espaço para:

variáveis locais e temporárias declaradas no escopo da função;

registradores salvos espaço só é alocado para aqueles registradores que devem ser preservados. Uma função não-folha deve salvar `ra`. Se qualquer dentre `s0-s7` (`r16-r23`) e `sp,fp,ra` (`r29-r31`) são alterados no corpo da função, estes devem ser preservados na pilha e restaurados antes do retorno da função. Registradores são empilhados na ordem de número, com registradores de números maiores armazenados em endereços mais altos. A área de salvamento de registradores deve ser alinhada como *doubleword* (8 bytes);

área para argumentos de chamada de função numa função não-folha, o espaço necessário para todos os argumentos que podem ser usados para invocar outras funções deve ser reservado na pilha. No mínimo, quatro palavras devem ser sempre reservadas, mesmo que o número de argumentos passados a qualquer função seja menor que quatro palavras; e

alinhamento a convenção (válida para o SO, inclusive) exige que um registro de ativação seja alinhado como *doubleword*. O alinhamento é em *doubleword* porque este é o maior tamanho de palavra que pode ser empilhado, que é um valor de ponto flutuante do tipo `double`, no caso do MIPS32r2.

Uma função aloca seu registro de ativação ao subtrair do *stack pointer* (`sp`) o tamanho de seu registro, no início de seu código. O ajuste no `sp` deve ocorrer antes que aquele registrador seja usado na função, e antes de qualquer instrução de salto ou desvio. A desalocação do registro deve ocorrer no último bloco básico da função, que inclui todas as instruções após o último salto ou desvio do código até a instrução de retorno (*jump-register*). Este protocolo não é observado pelo gerador de código do `gcc`.

As posições relativas em que os componentes no registro de ativação são armazenados devem ser respeitadas, mesmo que o código de uma função não os utilize todos. O leiaute do registro de ativação é importante e deve ser respeitado.

A Tabela 3.6 mostra quais recursos devem ser preservados pelo código de uma função. Do ponto de vista da função que chama, nenhum dos recursos do lado esquerdo da tabela é alterado pela função chamada. Os detalhes sórdidos estão em [**?**], páginas 3.11 a 3.21.

O programador em *assembly* é responsável por definir o leiaute do registro de ativação de cada função em função de quantos e quais são os argumentos da função, quais as variáveis locais, e quais os registradores que são alterados no corpo da função. As variáveis locais e registradores salvos devem ser referenciados usando o apontador da pilha como registrador base.

Tabela 3.6: Preservação de conteúdos entre chamadas de funções.

PRESERVADOS	NÃO PRESERVADOS
`s0-s7` (regs. salvos)	`t0-t9` (temporários)
`fp` (*frame pointer*)	`a0-a3` (argumentos)
`sp` (*stack pointer*)	`v0,v1` (valores de retorno)
`ra` (*return address*)	`at, k0, k1` (*assembler temporary, kernel*)
pilha acima do `sp`	pilha abaixo do `sp`

Exemplo 3.14 O código da função `int g(int x, int y, int z);` é mostrado no Programa 3.10. A função `g()` declara três variáveis locais em seu corpo, e é uma função não-folha. Seu registro de ativação, mostrado na Figura 3.10, deve acomodar 3 argumentos, o registrador com o endereço de retorno, e as três variáveis locais, perfazendo 28 bytes, que alinhado como *doubleword*, resulta em 32 bytes.

A ordem em que os registradores são escritos, e depois lidos, não é importante – o que importa é que o endereço em que cada registrador é salvo no início da função seja exatamente o mesmo de onde seu conteúdo é recuperado logo antes do retorno. ◁

Programa 3.10: Parte do código da função `g(x,y,z)`.

```
1     # w = g(x,y,z);
2     move  a0,rx       # prepara 3 argumentos
3     move  a1,ry
4     move  a2,rz
5     jal   g           # salta e armazena link em ra
6     move  rw,v0       # recebe valor de retorno
7     ...
8  g: addiu sp,sp,-32   # espaço para 3 args + ra + 3 vars
9     sw    a0,20(sp)   # empilha a0
10    sw    a1,24(sp)   # empilha a1
11    sw    a2,28(sp)   # empilha a2
12    sw    ra,16(sp)   # empilha endereço de retorno
13    ...   # corpo de g()
14    move  v0,t5       # valor de retorno em t5 -> v0
15    lw    ra,16(sp)   # recompõe endereço de retorno
16    lw    a0,20(sp)   # recompõe a0
17    lw    a1,24(sp)   # recompõe a1
18    lw    a2,28(sp)   # recompõe a2
19    addiu sp,sp,32    # desaloca espaço na pilha
20    jr    ra          # retorna
```

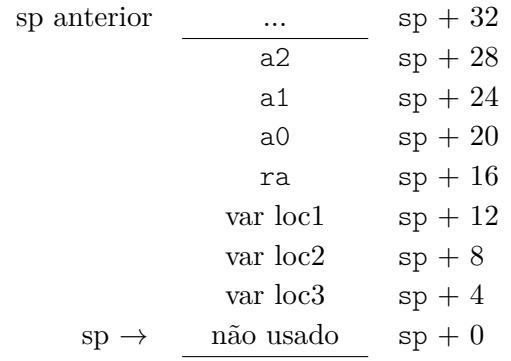


Figura 3.10: Registro de ativação do Programa 3.10.

Exemplo 3.15 Vejamos a tradução para *assembly* de uma função simples, empregando as convenções de programação do MIPS.

```
int fun(int g, int h, int i, int j) {
   int f = 0;

   f = (g+h)-(i+j);
   return (f*4);
}
```

Os quatro argumentos são armazenados, da esquerda para a direita nos registradores `a0` a `a3`. O valor da função é retornado em `v0`.

A função `f` é uma função folha e portanto é desnecessário empilhar o endereço de retorno, assim como os registradores com os argumentos.

O trecho inicial de código mostra a preparação dos argumentos – as variáveis são copiadas para os registradores, e o valor da função é copiado para a variável `k` após o retorno.

Espaço na pilha é alocado para a variável local `f`, num registro de ativação alinhado como *doubleword*. O registro de ativação contém somente a variável local, que é alocada no endereço apontado por `sp`.

As operações intermediárias salvam seus resultados em registradores temporários (`t0` a `t3`). O valor intermediário é salvado na variável local, recuperado e então multiplicado por quatro com um deslocamento para a esquerda.

O espaço na pilha é desalocado antes do retorno da função.

```
main: ...
      move a0, rg         # quatro argumentos
      move a1, rh
      move a2, ri
      move a3, rj
      jal  fun            # salta para fun()
      move rk, v0         # valor de retorno
      ...
fun: addiu sp, sp, -8     # aloca f na pilha, alinhado
     sw    r0, 0(sp)      # f <- 0
     add   t0, a0, a1     # t0 <- g + h
     add   t1, a2, a3     # t1 <- i + j
     sub   t2, t0, t1
     sw    t2, 0(sp)      # f <- (g+h)-(i+j);
     lw    t3, 0(sp)
     sll   v0, t3, 2      # v0 <- f*4
     addiu sp, sp, 8      # desaloca espaço na pilha
     jr    ra             # retorna
```

Esta função está codificada em 10 instruções, sendo três delas acessos à memória, que são operações deveras custosas. Este estilo de código é o produzido por um compilador, ao compilar sem nenhuma otimização, tal como com `gcc -O0`. ◁

Espaço em branco proposital.

Exemplo 3.16 Vejamos algumas otimizações para reduzir o tamanho e complexidade no código do Exemplo 3.15.

O corpo da função é tão simples, que a variável local pode ser mantida num registrador e portanto não é necessário salvar nada na pilha e o registro de ativação da função é vazio. Economia: duas instruções para manipular a pilha, inicialização de `f`, salvamento e leitura desta variável – todos os acessos à pilha foram eliminados porque desnecessários.

```
fun: add   a0, a0, a1     # a0 <- g + h
     add   a2, a2, a3     # a2 <- i + j
     sub   a2, a0, a2     # a2 <- (g+h)-(i+j);
     sll   v0, a2, 2      # v0 <- f*4
     jr    ra             # retorna
```

Os registradores temporários também são desnecessários porque o corpo da função é simples – tão simples que uma macro seria suficiente. Os valores intermediários são computados nos registradores de argumentos.

A versão otimizada tem cinco instruções e nenhum acesso à memória, além dos acessos inevitáveis para buscar as instruções da função. Esta economia é obtida quando o compilador otimiza o código, por exemplo, com `gcc -O2`. ◁

Espaço em branco proposital.

3.2.5 Exercícios

Ex. 26 Traduza para *assembly* as funções abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS. Todas as variáveis estão declaradas e tem os tipos e tamanhos adequados.

```
// (a) ------------------------------------------------
int fun(int a, int b, int c, int d, int e, int f);
   ...
   int a, p, q, z, w, v[N];
   ...
   x = fun(16*a, z*w, gun(p,q,r,s), v[3], v[z], z-2);
   ...

// (b) ------------------------------------------------
int fati(int n) {
   int i,j;
   j=1;
   if(n > 1)
      for(i = 1; i <= n; i++)
        j = j*i;
   return(j);
}

// (c) ------------------------------------------------
int fatr(int n) {
   if(n < 1)
      return (0);
   else
      return (n * fatr(n-1));
}

// (d) ------------------------------------------------
int log2(int n) {
   if (n < 2) then
      return 0;
   else
      return (1 + log2(n/2));
}
   ...
   x = log2(96000);   // maior do que |32.767|
   ...
```

Espaço em branco proposital.

Ex. 27 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS. Não escreva o código para `print()`; somente prepare os argumentos para sua invocação.

```
void print(char *, int); // não escreva o código desta função

int fib(int n) {
  if ( n == 0 )
     return 0;
  else
     if ( n == 1 )
        return 1;
  else
    return ( fib(n-1) + fib(n-2) );
}

void main() {
  int c;

  for (c = 1 ; c < 6 ; c++)
     print("%d\n", fib(c));
}
```

Ex. 28 Traduza para *assembly* a função abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
typedef elem {
   elemType *next;
   int  vet[3];
} elemType;

elemType *x;
elemType strut[256];
   ...
   x = insert( strut, strut, j) );
   x->vet[2] = 512;
   ...

elemType* insert(elemType *p, elemType *s, int i) {
  while (p != NULL) {
     p = p->next;
  }
  p->next = &(s[i]);
  (p->next)->next = NULL;
  return p->next;
}
```

Espaço em branco proposital.

3.3 Modos de Endereçamento

Agora que já conhecemos o conjunto de instruções, podemos definir os *modos de endereçamento* da arquitetura MIPS. Um "modo de endereçamento" é a forma que o processador usa para acessar os operandos e o resultado de cada instrução.

O conjunto de instruções do MIPS foi projetado sob a perspectiva de um "conjunto de instruções simples" (*Reduced Instruction Set Computer* ou RISC). O conjunto não é simples nem reduzido, mas cada instrução é simples e efetua o mínimo possível de trabalho útil, e portanto altera minimamente o estado da computação. Operações complexas podem ser facilmente sintetizadas com sequências de operações simples.

Esta filosofia de projeto contrasta com os "conjuntos de instruções complexas" (*Complex Instruction Set Computer* ou CISC), nos quais algumas instruções efetuam operações complexas, e que alteram mais do que um único elemento do estado da computação. O grande problema dessas instruções complexas é que raramente elas 'casam' com o programa fonte, e portanto o compilador não gera código empregando estas instruções. Um processador CISC é projetado para executar instruções complexas que raramente são incluídas nos executáveis pelo compilador. Esse é um caso em que "esperteza demais matou o gato".

Os cinco modos de endereçamento do MIPS32 são definidos abaixo.

Registrador Os operandos estão em registradores, e a própria instrução determina quais registradores contêm quais operandos.

Exemplos: `add s0, t3, a2` , `jr ra`

Imediato Um dos operandos, que é uma constante, é parte da instrução; o outro operando e o resultado são registradores.

Exemplos: `addi s3, t0, 1024` , `ori v0, v0, 0xfff0` , `lui a0, 0x3c00`

Base-deslocamento Este modo é usado nas instruções de acesso à memória. Um dos operandos, a *base*, é um registrador; o *deslocamento* é uma constante e é parte da instrução. O endereço efetivo é obtido somando-se o conteúdo do registrador base ao deslocamento, depois que este é estendido para 32 bis, com sinal. Frequentemente, o registrador base é um apontador (*pointer*).

Exemplos: `lw s5, 0x100(a0)` , `sb v1, 3(v0)` , `sw t0, -48(sp)`

Relativo ao PC Um dos operandos é o PC e o outro uma constante que é uma parte da instrução. Esse modo é usado nas instruções de desvio condicional, e o endereço de destino está a uma distância de até ±32K instruções da instrução corrente.

Exemplos: `beq v0, t5, +200` , `b -32`

Pseudo-absoluto Este modo de endereçamento é usado em saltos incondicionais e o endereço de destino é formado pelos 26 bits do operando (`ender`), concatenados com dois 0s e com os quatro bits mais significativos de (PC+4). O endereço de destino do salto é portanto `(PC+4) & ender & 00`. O endereço é chamado de *pseudo*-absoluto porque um endereço absoluto é completamente definido por uma constante de 32 bits.

Exemplos: `j 0x0003.fc30` , `jal 0x0000.c080`

Arquiteturas CISC, tais como a Intel x86 e a Digital Vax-11/780 [**?**], definem modos de endereçamento um tanto mais complexos do que aqueles do MIPS. Três exemplos são descritos no que segue.

Pós-incremento e pré-decremento Um dos operandos é um registrador, e o outro é lido da, ou escrito na, memória. Estas instruções são usadas nas operações de acesso à pilha. Há versões em que o argumento é uma lista de registradores – vetor de bits, um bit para cada registrador – e vários registradores são empilhados ou desempilhados com uma única instrução.

A instrução com pós-incremento é usualmente chamada de **pop**, porque o topo da pilha é removido (*popped-out*) da pilha: `pop r5  # r5 ← M[sp] , sp ← sp+1.`

A instrução com pré-decremento é chamada de **push**, porque um novo elemento é inserido (*pushed-in*) no topo da pilha: `push r2  # sp ← sp-1 , M[sp] ← r2.`

Note que tanto a memória quanto um registrador (`sp`) são atualizados num **push**, e que dois registradores (`sp` e `r2`) são atualizados num **pop**.

Indireto a registrador O endereço do operando é obtido de um registrador e é dado por `M[r]`. A instrução **addr** é executada em duas fases:
`addr r5, r6, (r7)  # tmp ← M[r7] ; r5 ← r6 + tmp.`
O conjunto de instruções do Vax-11/780 é *ortogonal* e o que é possível para um argumento é possível para todos os casos que façam sentido:
`addr (r5), r6, r7  # tmp ← r6 + r7 ; M[r5] ← tmp.`
Levando ortogonalidade ao extremo temos: `addr (r5), (r6), (r7).`
O detalhamento da execução desta instrução fica como um exercício para o leitor interessado.

Indireto a memória O endereço do operando é obtido de uma posição de memória – para acessar o operando de uma instrução, deve-se fazer um acesso à memória para de lá obter o endereço do operando. O operando é obtido de `M[M[r]]`. O parêntese duplicado denota a indireção através da memória. A instrução **addm** é executada em três fases:
`addm r5, r6, ((r7))  # tmp1 ← M[r7] ; tmp2 ← M[tmp1] ; r5 ← r6 + tmp2.`
Uma versão ligeiramente mais interessante é
`addm ((r5)), r6, r7  # tmp1 ← M[r5] ; tmp2 ← r6 + r7 ; M[tmp1] ← tmp2.`
A versão *überhaupt* interessante é:
`addm ((r5)), ((r6)), ((r7)).`
O detalhamento da execução desta instrução fica como um exercício para a dedicada e atenta leitora. Ela também deve dispender um par de minutos a considerar o circuito que implementa tal monstruosidade.

Foi mencionado que o conjunto de instruções do Vax-11/780 é *ortogonal*. Os projetistas da Digital levaram o conceito de ortogonalidade ao extremo, como indicado para os dois modos de endereçamento apresentados acima. Mas não só. Se uma operação aritmética Φ faz sentido para operandos de tipo **char**, então o processador suporta Φ para operandos de tipo **short** e **int**. Da mesma forma, se a operação Ψ é semelhante à Φ, então tudo o que é permitido para Φ, e faça sentido, é também permitido para Ψ – isso inclui todos os tamanhos de operandos, e qualquer dos modos de endereçamento em quaisquer dos três operandos da instrução.

A implementação era tão complexa quanto brilhante, do ponto de vista de engenharia. A razão para a semântica complexa era 'facilitar' a programação em *assembly* ao prover um poderoso conjunto de instruções, para que – em tese – programadores humanos produzissem rapidamente código compacto e dotado de grande funcionalidade. Lembre que nas décadas de 1970 e 80 memórias eram pequenas ($\ll$ 1Mbytes) e custosas – código compacto era de especial importância então.

Os compiladores desenvolvidos no início da década de 1980 geravam código de qualidade comparável ao gerado por humanos. Contudo, era virtualmente impossível aos geradores de código empregar as instruções complexas. O código gerado era uma longa sequência de instruções simples, que eram usadas para sintetizar o comportamento das instruções complexas. Na medida em que programar em C se tornou uma proposição viável, a programação em *assembly* foi rapidamente abandonada. Os processadores CISC, tais como o Vax-11/780, tornaram-se obsoletos porque eram caros, complexos, e uma boa parte do conjunto de instruções não era utilizado. Ao mesmo tempo que que alguns dos processadores CISC, como os Vax, se encaminhavam para a extinção, os microprocessadores RISC mais simples, mais baratos e com melhor desempenho executando código compilado, ganhavam mercado.

3.4 Montadores

Computadores não executam diretamente programas escritos em linguagens de alto nível tais como C ou Pascal. O formato dos programas que eles executam diretamente chama-se de *executável*, e sob o modelo de VonNeuman, tal código é nada mais que uma sequência de bits, cujo significado é atribuído pelo projetista do *conjunto de instruções* (CdI), ou pelo *arquiteto* do sistema/computador.

Cada *instrução* ao computador consiste de um certo número de bits, 32 no caso do MIPS, e a cada padrão de 32 bits corresponde a uma ação distinta do computador. Felizmente, computadores podem ser muito úteis mesmo sem executar as 2^{32} ações distintas – 4 bilhões é um número deveras grande.

O *montador* (*assembler*) é um programa que 'monta' o código em *linguagem de montagem* (*assembly language*), gerando o código binário que é interpretado pelo computador. Como veremos adiante, instruções em *assembly* são algo mais compreensíveis do que um número binário de 32 dígitos. No caso do MIPS, a instrução que soma o conteúdo dos registradores `$5` e `$3`, e armazena o resultado no registrador `$3` é:

```
addu $3,$5,$3  00000000101000110001100000100000     0x00a31820
               000000 00101 00011 00011 00000 100000
               opcode  rs    rt    rd  shamt  fun
```

Por *compilador*[3] *nativo*, ou *montador nativo*, entende-se o programa que traduz código para ser executado no mesmo computador em que o tradutor executa. Um *cross-compilador* é um compilador que produz código para ser executado num processador distinto. No nosso caso, usaremos `mips-gcc` e `mips-as` para traduzir código C e *assembly* para ser executado num processador que executa as instruções do MIPS, e não para o processador nativo, que provavelmente é algum membro da família estendida dos x86.

3.4.1 O processo de compilação

Considere uma aplicação cujo código fonte está separado em dois arquivos, `x.c` que contém a função `main()`, e `y.c` que contém a função `fun()`. A Figura 3.11 mostra um diagrama com as etapas da compilação dos dois arquivos para produzir o executável `a.out`. Os círculos contém os programas que traduzem "o programa", de código C para código executável; os nomes sobre as setas indicam os arquivos gerados em cada etapa do processo de compilação.

[3] O nome 'completo' do montador disponível em nosso sistema é `gas`, abreviatura para *Gnu ASsembler*. O nome do "montador nativo" em sistemas Unix é `as`, enquanto que o nome do "compilador nativo" para C é `cc`, ou *C Compiler*. O nome `gcc` é uma abreviatura para *Gnu C Compiler* ou *Gnu Compiler Colection*.

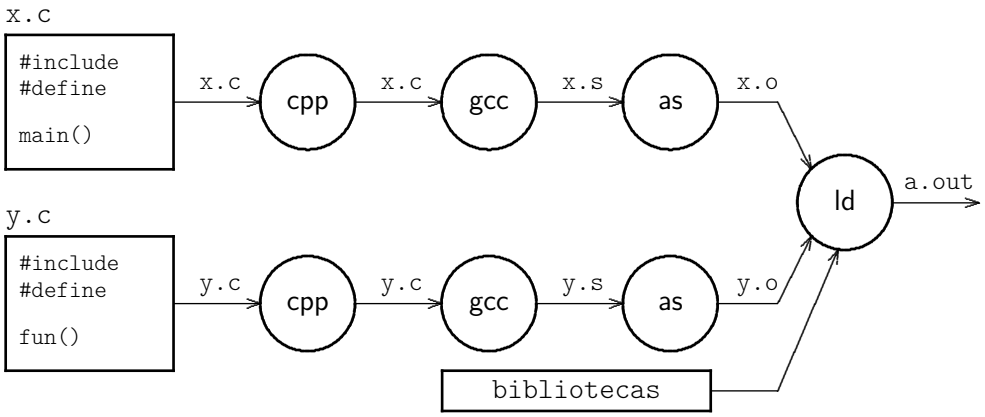


Figura 3.11: Etapas do processo de compilação.

Os arquivos fonte contém diretivas do tipo `#include` e `#define` que são processadas pelo preprocessador cpp, que faz a expansão das macros, a inclusão dos arquivos com cabeçalhos e remove os comentários. A saída do cpp é entregue ao compilador gcc, que faz a tradução de C para linguagem de montagem. Ao contrário do cpp, que só manipula texto, o compilador traduz comandos da linguagem C para instruções em *assembly*.

O código em *assembly* é processado pelo montador as, que produz um *arquivo objeto*, já com as instruções traduzidas para seus equivalentes em binário, além da tabela de símbolos – veja a Seção 3.4.3 para uma descrição do processo de montagem.

Os arquivos objeto, e se necessário, o código mantido em bibliotecas, são agregados pelo *ligador* (ld), que finalmente produz o arquivo executável `a.out`. O processo de ligação é descrito no Capítulo 10 e seguintes.

Felizmente, tudo que uma criatura deve fazer é invocar um único comando, tal como

```
gcc -Wall x.c y.c -lm && ./a.out
```

para que todo o processo de compilação seja efetuado por um comando chamado gcc. O uso do gcc é detalhado no Capítulo 5.

3.4.2 Anatomia e fisiologia de um montador

Examinemos então a entrada e a saída de um montador, bem como alguns de seus componentes internos mais importantes.

Entrada e saída do montador

A entrada para o programa montador (*assembler*) é um ou mais arquivos texto com código em linguagem de montagem (*assembly*), que foi gerado por uma pessoa ou pelo compilador. A saída do montador é um *arquivo objeto* formatado para ser usado como entrada por um programa ligador. Em "sistemas pequenos" o montador pode produzir um executável diretamente – este não é o caso de sistemas Unix/Linux.

Os arquivos de entrada do montador tem sufixo `.s` (*aSsembly*) e o arquivo de saída tem sufixo `.o`, para "arquivo Objeto".

Espaço em branco proposital.

Além das instruções da linguagem de montagem, montadores aceitam *diretivas*, que são instruções para o montador, e não para o processador. Diretivas permitem reservar espaço para variáveis, determinam se determinado trecho do código fonte corresponde a instruções ou a dados, além de várias outras tarefas administrativas. Geralmente, diretivas iniciam com um ponto, para diferenciá-las das instruções do *assembly*, como por exemplo '`.space`'.

Seções geradas pelo montador

O arquivo com o código objeto gerado pelo mips-as é armazenado em *seções* denominadas `.text`, `.data`, `.bss`, `.absolute`, `.undefined`. Uma lista reduzida dos conteúdos de cada seção é mostrada abaixo. A lista completa inclui ainda vários outras seções um tanto obscuras – detalhes na Seção 11.3. As siglas "RO, RW, EX" significam "Read Only", "Readable and Writable", e "EXecutable".

`.text` instruções do programa e constantes, geralmente RO EX;

`.data` as variáveis de um programa, geralmente RW;

`.bss` variáveis não-inicializadas e *common*s, variáveis inicializadas com zero (*block started by symbol*), geralmente RW;

`.absolute` símbolos com endereço absoluto, em seção que será relocada pelo ligador para endereços absolutos em tempo de execução; e

`.undefined` lista de símbolos com endereço indefinido; durante a ligação, estes endereços devem ser preenchidos pelo ligador.

Location counter

O *location counter* é um contador mantido pelo montador que é incrementado a cada byte emitido (traduzido), e que aponta para o endereço em que algo está sendo gerado/montado.

Um ponto (ou *dot*) é o símbolo que contém o endereço do *location counter*. O trecho de código abaixo mostra um exemplo de uso do *dot*. No exemplo, o símbolo '`aqui`' contém seu próprio endereço, que é o endereço de uma palavra. A diretiva **`.word`** reserva espaço para uma palavra em memória e este espaço é identificado pelo seu *label* '`aqui:`'. O conteúdo da palavra apontada por '`aqui:`' é o próprio valor do símbolo, que é o valor corrente do *location counter*, ou o *dot*.

```
aqui: .word .      # define símbolo para o endereço corrente
```

A expressão `.=.+4` ('aqui' recebe 'aqui' mais quatro) equivale a reservar o espaço correspondente a quatro bytes. O mesmo efeito pode ser obtido com a diretiva **`.space`**:

```
meuint: .space 4   # define símbolo para um inteiro
```

Símbolos

Símbolos são usados para nomear endereços, na montagem, ligação e depuração. Símbolos iniciam por um caractere dentre '$', '.', '_', minúsculas ou maiúsculas e dígitos.

Comentários iniciam com '#' ou ';' e se estendem até o fim da linha.

Um *label* é um símbolo terminado por ':' que representa o valor do *location counter* naquele ponto da montagem. Um *label* corresponde a um endereço, que pode ser de uma instrução – função, destino de `goto` – ou variável, ou estrutura de dados.

Um símbolo pode representar um valor, que lhe é atribuído por uma diretiva tal como `.set`, `.size` ou `.equ`.

Diretivas

Diretivas são comandos ao montador e não fazem parte do conjunto de instruções de nenhum processador, mas permitem a alocação de espaço para variáveis (`.word`, `.byte`), definição de escopo de visibilidade de nomes (`.global`), além de outras funções de caráter administrativo.

De novo e paradoxalmente: diretivas fazem parte da linguagem *assembly* de um certo montador, mas não pertencem ao conjunto de instruções de nenhum processador.

As diretivas do montador mips-as estão documentadas no espartano manual (`man as`) e fartamente documentadas na página HTML do montador. Algumas diretivas são brevemente descritas abaixo – elas serão empregadas nas próximas aulas, quando então sua utilidade ficará mais evidente.

`.text` o que segue deve ser alocado na seção `.text` – código;
`.data` o que segue deve ser alocado na seção `.data` – dados;
`.section` texto que segue é montado na seção nomeada;
`.previous` troca esta seção pela que foi referenciada mais recentemente;
`.ent` endereço de "entrada" no código – primeira instrução da função;
`.end` marca o final de código, não monta nada até a próxima diretiva `.ent`;
`.align expr` ajusta contador de locação para valor múltiplo de 2^{expr};
`.global` torna símbolo visível ao ligador, aumentando seu escopo;
`.comm` declara símbolo (*common*) na seção BSS (*block started by symbol*);
`.ascii` aloca espaço para cadeias, sem `'\0'`;
`.asciiz` aloca espaço para cadeias, com `'\0'`;
`.byte` resolve expressões e as aloca em bytes consecutivos;
`.set symb expr` resolve expressão e atribui valor ao símbolo;
`.size symb expr` resolve expressão e atribui tamanho [bytes] ao símbolo;
`.equ symb, expr` resolve expressão a atribui valor ao símbolo;
`.type symb tipo` atribui tipo ao símbolo, que pode ser função ou variável;
`.mask .fmask` máscaras dos registradores usados no código.

Os valores de `.mask` e `.fmask` indicam quais registradores são usados no código montado: se o bit 19 está em 1 então o registrador `r19` é usado. Estes valores são usados pelo ligador para gerar a seção `.reginfo` do arquivo objeto de processadores MIPS. A disjunção dos valores dos `.mask` indica quais registradores de inteiros são usados no código; os valores em `.fmask` indicam quais registradores de ponto flutuante são usados – a justificativa para tal é apresentada na Seção 7.3.4.

Exemplo 3.17 O trecho de código C no Programa 3.11 foi compilado para gerar código *assembly* do MIPS, o que nos permite observar a saída do compilador, que é a entrada para o montador.

O código nos Programas 3.12 e 3.13 foi gerado com o comando `mips-gcc -S -O1 strcopy.c strcopy.s`. Os números à esquerda não fazem parte da saída, e servem apenas para identificar as linhas do programa.

Espaço em branco proposital.

Programa 3.11: Código fonte de `strcopy()`.

```
int strcopy(char x[], char y[]) {
  int i=0;
  while ( (x[i] = y[i]) != '\0' ) // copia e testa fim da cadeia
    i = i+1;
  return(i);
}

char fnte[] = "abcdefgh";    // variável global
char dest[] = "ABCDEFGH";    // variável global

int main (int argc, char** argv) {
  int num=0;
  num = strcopy(fnte, dest);
  return(num);
}
```

As linhas 1–8 são o cabeçalho do arquivo (prólogo) e descrevem seu conteúdo e características. O símbolo `strcopy` é declarado como um símbolo global (linha 6) e portanto sua visibilidade se estende para além deste arquivo. A linha 4 indica que o que segue deve ser gerado numa seção de código (`.text`).

Programa 3.12: Início da versão *assembly* de `strcopy()`.

```
 1          .file   1 "strcpy.c"
 2          .section .mdebug.abi32
 3          .previous
 4          .text
 5          .align  2
 6          .global strcopy
 7          .set    nomips16
 8          .ent    strcopy
 9
10  strcopy:
11          .frame  sp,0,ra   # vars=0, regs=0/0, args=0
12          .mask   0x00000000,0
13          .fmask  0x00000000,0
14          .set    noreorder
15          .set    nomacro
16
17          move    a2,zero
18          lbu     v0,0(a1)
19          beq     v0,zero,.L6
20          sb      v0,0(a0)
21  .L4:
22          addiu   a2,a2,1
23          addu    v0,a0,a2
24          addu    v1,a1,a2
25          lbu     v1,0(v1)
26          bne     v1,zero,.L4
27          sb      v1,0(v0)
28  .L6:
29          jr      31
30          move    v0,a2
```

As linhas 10–15 não avançam o *location counter*; portanto o endereço de `strcopy` é o da instrução `move` na linha 17.

Programa 3.13: Final da versão *assembly* de `strcopy()`.

```
31          .set    macro
32          .set    reorder
33          .end    strcopy
34          .size   strcopy, .-strcopy
35          .globl  fnte
36          .data
37          .align  2
38          .type   fnte, @object
39          .size   fnte, 9
40  fnte:
41          .ascii  "abcdefgh\000"
42          .global dest
43          .align  2
44          .type   dest, @object
45          .size   dest, 9
46  dest:
47          .ascii  "ABCDEFGH\000"
48          .text
49          .align  2
50          .global main
51          .set    nomips16
52          .ent    main
53  main:
54          .frame  sp,24,ra # vars=0, regs=1/0, args=16
55          .mask   0x80000000,-8
56          .fmask  0x00000000,0
57          .set    noreorder
58          .set    nomacro
59
60          addiu   sp,sp,-24
61          sw      ra,16(sp)
62          lui     a0,%hi(fnte)
63          addiu   a0,a0,%lo(fnte)
64          lui     a1,%hi(dest)
65          jal     strcopy
66          addiu   a1,a1,%lo(dest)
67
68          lw      ra,16(sp)
69          jr      ra
70          addiu   sp,sp,24
71
72          .set    reorder
73          .end    main
74          .size   main, .-main
75          .ident  "GCC: (GNU) 3.4.4 mipssde-6.06.01-20070420"
```

As linhas 38-47 indicam que as constantes inicializadas são geradas na seção `.data` (linha 36), e que os nomes `fnte` e `dest` são símbolos globais e visíveis fora deste arquivo (linhas 35 e 42). `main()` é gerada na seção `.text` (linha 48) e as linhas 68-70 contém o epílogo do programa. ◁

Os *labels* gerados pelo compilador iniciam com '`.`' (`.L1:`) porque a linguagem C proíbe símbolos que iniciam com '`.`'. Isso garante que símbolos como `main:` sejam facilmente diferenciados dos símbolos que representam fim/início de laços, tais como `.L4` e `.L6` .

3.4.3 Algoritmo e estruturas de dados

Para entender a operação do montador, é necessário uma rapidíssima descrição do processo de compilação. Considere o processo de compilação dos arquivos `x.c` e `y.c`, mostrado na Figura 3.11. `y.c` contém a definição de várias funções que são empregadas em `x.c`. Os dois arquivos fonte são compilados e montados separadamente; o programa *ligador* edita o arquivo que resulta da compilação de `x.c` e ajusta os endereços das funções referenciadas naquele arquivo, mas definidas em `y.c`, para que estas referências reflitam os endereços no código gerado para `y.c`. Não se preocupe se, no momento, parece um tanto confuso porque várias páginas são dedicadas aos copiosos detalhes que faltam nesta explanação.

A estrutura de dados principal de um montador é sua *tabela de símbolos* (TS), que contém os símbolos declarados no programa e seus valores. Durante a montagem, alguns dos valores podem estar momentaneamente indefinidos, ou permanecer indefinidos até o final da execução do montador. A implementação mais simples de um montador consiste de "duas passadas" sobre o código *assembly*: (i) a tradução das instruções; e (ii) o ajuste dos endereços, que são indicados na Figura 3.12 e detalhados no que se segue.

primeira passada	lê código fonte
	traduz todas instruções sem endereços
	insere símbolos na tabela de símbolos
	gera arquivo intermediário (saída parcial)
segunda passada	lê arquivo intermediário
	pesquisa na tab. de símbolos e ajusta endereços
	gera saída completa

Figura 3.12: Algoritmo de montagem em dois passos.

Na *primeira passada*, o montador lê o arquivo com o código fonte e traduz cada instrução que esteja completamente definida, tal como uma adição. Se um operando de uma instrução é um endereço que ainda está indefinido, o montador insere o símbolo correspondente àquele endereço na tabela de símbolos, e marca a instrução como "incompletamente traduzida".

A cada instrução traduzida, o montador avança o *location counter*. O mesmo vale para a seção de dados – a cada variável declarada no programa, o espaço necessário é reservado na seção apropriada, e o *location counter* daquela seção é avançado de tantos bytes quanto necessário para acomodar a variável ou estrutura de dados.

Ao final da primeira passada, todos os símbolos do programa foram armazenados na tabela de símbolos, e seus valores (endereços) podem ser determinados pelo montador. Se o endereço de um símbolo não pode ser determinado em tempo de montagem, esta informação é repassada para o ligador, que então resolverá o valor do símbolo. Se isso não é possível então ocorreu um erro de compilação, ou de ligação, e o programa não pode ser executado porque uma função, ou uma variável, está com um endereço indeterminado.

Na *segunda passada*, o montador percorre novamente o arquivo com o código, e para cada instrução incompletamente traduzida, a tabela de símbolos é consultada para resolver o símbolo que não foi resolvido no primeiro passo. Ao consultar a tabela, o valor do símbolo é usado para alterar o arquivo de saída que então reflete a informação atualizada.

Ao final da segunda passada, o arquivo de saída é gerado, e possivelmente, todas as instruções estão completamente traduzidas, com as informações de endereço completas. Caso algum símbolo não tenha sido resolvido, como uma invocação de `printf()`, por exemplo, esta informação é armazenada no arquivo de saída para que o ligador se encarregue de alterar a instrução que invoca a função `printf()`, fazendo a ligação entre o símbolo `printf:` no

arquivo recém-montado e o endereço correspondente ao símbolo, que é a primeira instrução daquela função na biblioteca `libc.a`.

De volta ao exemplo desta seção: se o arquivo `x.o` contém símbolos indefinidos, tal como a invocação de `fun()`, o ligador cria uma nova tabela de símbolos, com os símbolos de `x.o` e com os símbolos definidos em `y.o`. O arquivo de saída com a junção de `x.o` e `y.o` é o executável `a.out`, *se e somente se*, todos símbolos indefinidos em `x.o` foram resolvidos por símbolos definidos em `y.o`, ou na biblioteca `libc.a` é ligada a estes dois arquivos objeto.

Vários detalhes importantes foram omitidos nesta descrição, tal como os detalhes da ligação com funções de biblioteca; estes serão investigados no Capítulo 10 e seguintes.

Num processador como o MIPS, no qual todas as instruções tem o mesmo tamanho, o processamento em duas passadas pode parecer exagero porque basta contar as instruções para gerar todos os endereços no arquivo objeto. Na montagem de código para processadores que usam instruções de tamanho variável, como é o caso do x86, cujas instruções tem de um a 17 bytes, é necessário traduzir as instruções para binário no primeiro passo, porque só então é possível determinar todos os endereços de todas as instruções.

3.4.4 Exercícios

Ex. 29 É possível efetuar a montagem com uma única passagem sobre o código? Se sim, indique as estruturas de dados e o algoritmo para fazê-lo.

Ex. 30 Talvez seja mais eficiente efetuar a montagem em três passadas sobre o código. Se sim, indique as estruturas de dados e o algoritmo para fazê-lo, e explique em quais condições três passadas é mais eficiente do que duas.

Ex. 31 (a) Dê dois exemplos de diretivas do montador que *não* causam a inclusão de bits adicionais no arquivo objeto e explique suas funções; (b) dê dois exemplos de diretivas do montador que produzem saída no arquivo objeto e explique suas funções; (c) qual a função da diretiva `.align` ? Esta diretiva não pode estar incluída nas suas respostas anteriores.

Ex. 32 (a) Escreva uma função em C que computa a redução por *ou-exclusivo* de um vetor de inteiros ($x = \bigoplus_{i=0}^{n-1} V_i$), que é apontado pelo primeiro argumento, conforme o protótipo abaixo; (b) traduza sua função para o *assembly* do MIPS; (c) escreva em *assembly* o trecho de código em que `reduz()` é invocada, e mostre o leiaute do registro de ativação. Seu código *assembly* deve respeitar as convenções para a codificação de funções.

```
int reduz(int *v, int n);
```

Espaço em branco proposital.

Ex. 33 Traduza para *assembly* do MIPS os trechos de programa abaixo. Seu código *assembly* deve empregar as convenções de programação do MIPS.

```
// (a)
int a, x[2048], y[64];
   ...
   a = fun(x, 2048, y, 64);
   ...
int fun(int *p, int np, int *q, int nq) {
   int i=1;
   int s=0;
   while (i < np) {
      s = s + p[i] + p[ q[i%nq] % np ]; // MOD, MOD
      i = i * 2;
   }
   return s;
}


// (b) ---------------------------------------------
#define SIZE 1024
typedef struct x {
   int a;
   int b;
   int c;
   short x;
   short y;
} xType;

xType V[SIZE], Z[SIZE];

void reduz(int lim, xType *v, xType *z, int pot) {
   int i=0;
   while (i < lim) {
      v[i].a = z[i].b + z[i].c;
      v[i].x = z[i].x <<pot;
      i = i + 1;
   }
}
   ...
   reduz(SIZE/4, V,  Z, 4);
   ...


// (c) ---------------------------------------------
#define SZ 1024
int A[SZ], B[SZ];
...
int escalar(int tam, int *a, int *b, int const) {
   int i,j, soma;
   for (i=1, j=0, soma=0; i < tam; i=i*2, j=j+1) {
      b[j] = a[i]*const;
      soma += a[j];
   }
   return soma;
}
   ...
   escalar(SZ, A, B, 16);
   ...
```

Capítulo 4

Duas Implementações para MIPS32r2

Esta seção descreve duas implementações do conjunto de instruções MIPS32, uma que é a mais simples possível, e outra que tira proveito de paralelismo na execução e portanto tem melhor desempenho. A implementação com paralelismo introduz dois artefatos que podem reduzir a eficiência almejada. Para mais detalhes quanto ao *hardware*, veja [**?**]; para detalhes quanto ao *software*, veja [**?**].

Evidentemente, a implementação do conjunto completo de instruções não é discutida por ser demasiado complexa; o que se deseja é mostrar os princípios de funcionamento do processador. A implementação de algumas poucas instruções representativas é discutida; outras são indicadas nos exercícios. O código VHDL de uma implementação completa do conjunto de instruções MIPS32r2 pode ser encontrado em [**?**].

4.1 Um subconjunto das instruções MIPS32r2

O conjunto de instruções a ser estudado aqui é mostrado na Tabela 4.1. Cada uma destas representa uma classe de instruções e a implementação de uma classe completa depende de pequenas variações nos circuitos que são discutidos no que se segue.

Convenção: o caractere ‘ ; ’ significa execução sequencial; o caractere ‘ , ’ significa execução em paralelo dos eventos à esquerda e à direita da vírgula; ‘R(s)’ é a sintaxe de VHDL para indexar um vetor, seja um vetor de bits seja um vetor de bytes ou palavras; ‘&’ é o operador de concatenação de VHDL, e ‘←’ representa atribuição.

Tabela 4.1: Subconjunto das instruções do MIPS

INSTRUÇÃO	DESCRIÇÃO	
`addu rd, rs, rt`	R(rd) ← R(rs) + R(rt) ,	PC←PC+4
`ori rt, rs, im16`	R(rt) ← R(rs) OR extZero(im16) ,	PC←PC+4
`lw rt, des16(rs)`	R(rt) ← M(R(rs) + extSinal(des16)) ,	PC←PC+4
`sw rt, des16(rs)`	M(R(rs) + extSinal(des16)) ← R(rt) ,	PC←PC+4
`beq rs, rt, des16`	if (R(rs) = R(rt)) PC ← PC+4 + extSinal(des16) & 00	
	else PC ← PC+4	
`j ender26`	PC ← PC(31..28) & ender26 & 00	

Quais são os recursos necessários para implementar estas instruções?

Para quase todas elas é necessário um somador para incrementar o PC de 4.

Para quase todas elas é necessário um bloco de registradores que permita a leitura de um ou dois registradores e a escrita em um registrador.

Para as instruções de lógica e aritmética é necessária uma unidade de lógica e aritmética que implemente as operações indicadas na Tabela 4.1.

Para a instrução `ori` é necessário um circuito que efetue a disjunção bit a bit – que existe na ULA – e um circuito que estenda com zeros um número de 16 bits para um número com 32 bits.

Para as instruções `lw` e `sw` é necessária uma memória que permita escritas e leituras, um somador – da ULA – e um circuito que estenda o sinal de uma constante de 16 bits, representada em complemento de dois, para 32 bits.

Para `beq` é necessário um comparador de igualdade, para comparar dois números de 32 bits.

Embora não esteja indicado na Tabela 4.1, todas as instruções devem ser buscadas da memória de instruções, para que possam ser decodificadas e então executadas. A memória de instruções é indexada pelo PC. A cada instrução que não seja um desvio ou salto, o PC é incrementado de 4; em desvios e saltos, o valor carregado no PC depende da instrução.

A Figura 4.1 mostra os circuitos combinacionais necessários para a implementação do conjunto de instruções. A *Unidade de Lógica e Aritmética* (ULA) tem como entradas dois operandos de 32 bits e produz um resultado de 32 bits. O valor na saída depende da entrada func, que define qual a função a ser aplicada aos operandos, que é uma dentre soma, subtração, conjunção, disjunção, ou-exclusivo ou *nor*. O *somador* efetua a soma em 32 bits de seus operandos. O *multiplexador* (*mux*) apresenta em sua saída uma dentre as suas entradas, selecionadas pela entrada sel.

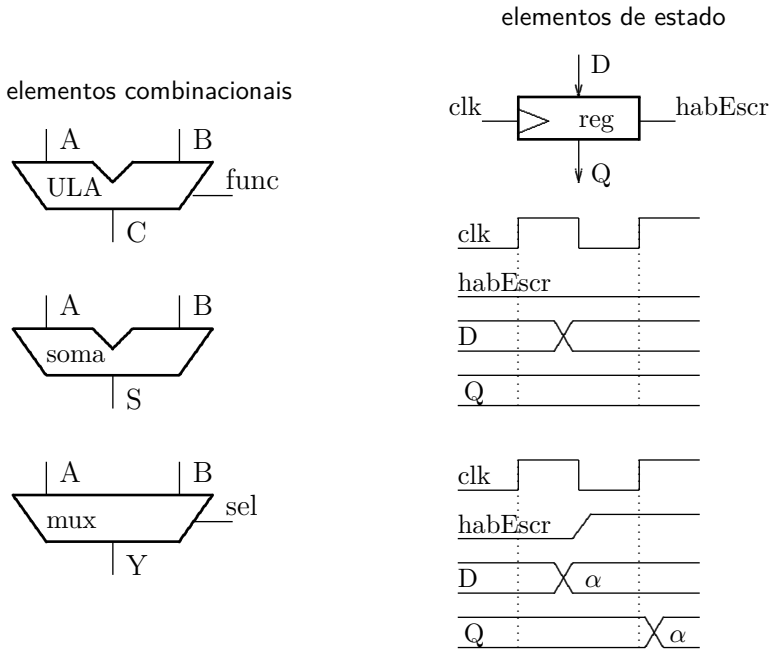


Figura 4.1: Elementos combinacionais e elementos de estado.

A Figura 4.1 também mostra os elementos de estado necessários. Os mais simples são *registradores*, que nada mais são do que vários *flip-flops* tipo D que compartilham os sinais de relógio (clk) e habilitação (habEscr). Quando habEscr=1, na borda ascendente do relógio o valor em D é capturado e mantido em Q. Se habEscr=0, então o valor memorizado anteriormente não se altera. Estas duas situações são mostradas nos diagramas de tempo.

O PC é um registrador cujo sinal de habilitação está sempre ativo.

O *bloco de registradores* (*register bank*) contém 32 registradores de 32 bits cada. Dois registradores podem ser acessados simultaneamente para leitura – são duas portas de leitura, A e B – e um registrador pode ser atualizado na borda do relógio, se o sinal de habilitação está ativo – que é a porta de escrita C. A Figura 4.2 mostra a interface do banco de registradores e das memórias. O registrador 0 é *sempre* zero, e escritas neste registrador são ignoradas.

Os registradores cujos conteúdos são mostrados nas portas de leitura são selecionados por dois endereços de 5 bits. As portas de leitura se comportam como circuitos combinacionais e podem ser consideradas como dois vetores de inteiros independentes:

```
x <= A(a);
y <= B(b);
```

sendo a e b dois números representados em 5 bits ($\log(32) = 5$). A porta de escrita se comporta como um registrador e o novo valor só é atualizado na borda do relógio se o sinal de habilitação (hab=1) está ativo:

```
if (rising_edge(clk) and hab=1) then C(c) <= z end if;
```

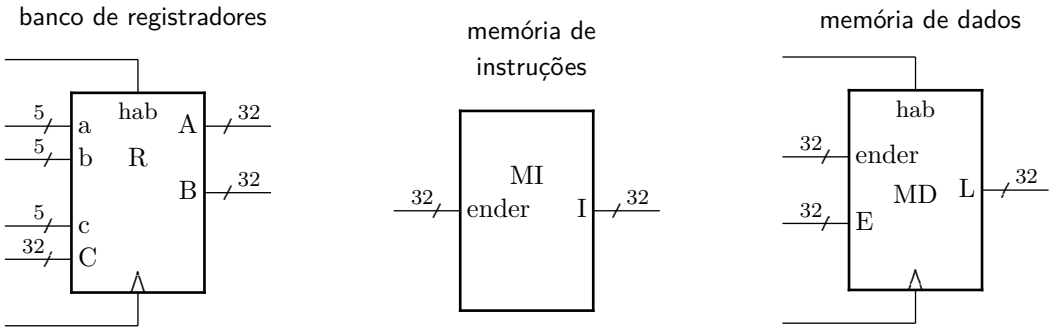


Figura 4.2: Banco de registradores, memória de instruções e de dados.

A *memória de instruções* (MI) se comporta como um circuito combinacional – uma vez que o endereço estabiliza, passado o tempo de acesso à memória, a instrução fica estável na porta I:

```
instr <= MI(ender);
```

A *memória de dados* (MD) tem um comportamento similar ao banco de registradores. A porta de leitura L se comporta como um vetor e a leitura do conteúdo endereçado é combinacional – uma vez que o endereço estabiliza e decorrido o tempo de acesso, o dado fica disponível na porta L:

```
dado <= MD(ender);
```

A atualização da palavra indexada pelo endereço só é efetivada na borda do relógio, se a escrita estiver habilitada (hab=1):

```
if (rising_edge(clk) and hab=1) then MD(ender) <= dado end if;
```

4.1.1 Codificação das instruções

Antes que possamos falar da implementação do processador é necessário examinarmos a codificação das instruções do MIPS32. Todas as instruções tem 32 bits e são três os seus formatos: Tipo R, Tipo I e Tipo J, mostrados na Figura 4.3.

Nos três formatos, o campo *opcode* (`opc`), abreviatura para *operation code*, é o 'nome' da instrução, e em todas elas este campo tem 6 bits. Os campos `rs` (*register source*), `rt` (*register target*) e `rd` (*register destination*) endereçam os registradores com os operandos

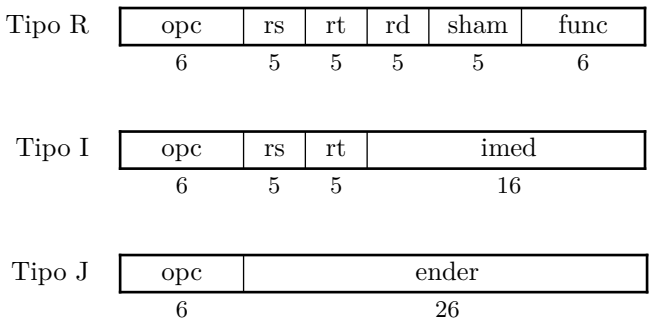


Figura 4.3: Formatos das instruções.

da instrução. O campo `sham` (*shift amount*) é um campo de 5 bits que indica o número de posições por deslocar nas instruções de deslocamento. O campo `func` (*function*) determina a operação da ULA nas instruções de lógica e aritmética quando `opc`=0. O campo `imed` (*immediate*) é uma constante de 16 bits, que é representada em complemento de dois sempre que isso faça sentido. Finalmente, `ender` é um endereço de 26 bits.

Na Tabela 4.1, a instrução **`addu`** é do Tipo R porque a instrução tem seus três operandos em registradores; as instruções **`ori`**, **`lw`**, **`sw`** e **`beq`** são do tipo I porque todas tem dois registradores como operandos e o terceiro operando é uma constante que faz parte da instrução. As instruções do tipo J são as instruções de saltos, **`j`** e **`jal`**.

Esta codificação tem uma característica muito importante que é a sua *regularidade*: todas as instruções tem o mesmo tamanho; todas as instruções tem o *opcode* no mesmo lugar e com o mesmo tamanho (6 bits), a maioria das instruções tem seus operandos identificados nos campos `rs` e `rt`; a maioria das constantes tem 16 bits (`imed`), e pode ser interpretada como uma constante lógica ou como um inteiro.

Os projetistas do conjunto de instruções reservaram 6 bits para $2^6 = 64$ *opcodes*. Este número é suficiente para codificar todas as instruções do processador? A resposta é *não*, e para aumentar o número de *opcodes* foi usado um truque: quando `opc`=0, a função da ULA é determinada pelo campo `func`, também com 6 bits. Assim, o total de *opcodes* disponíveis é $(2^6 - 1) + 2^6 = 127$.

Como a atenta leitora pode verificar em [**?**], a realidade é ligeiramente mais complexa do que o exposto aqui. Por ora, nos interessa mais estudar os princípios de projeto do que as poucas idiossincrasias de um conjunto de instruções real e bem projetado.

A decodificação das instruções é extremamente simples, graças à codificação simples: basta indexar uma tabela de 64 elementos que contém todos os sinais de controle do processador. O *opcode* indexa a tabela e os sinais de controle de todos os componentes do processador são definidos pelos campos dessa tabela – logo veremos alguns exemplos.

4.2 Implementação com relógio de ciclo longo

Vamos pois à implementação das instruções da Tabela 4.1. No que segue, os diagramas mostrados estão incompletos para simplificar a explanação. O circuito completo do processador é apresentado na Seção 4.2.8.

4.2.1 Busca e decodificação de instruções

A memória de instruções (MI) mantém o código binário do programa que está sendo executado. No momento não nos interessa como o programa foi gravado nesta memória; mais adiante estudaremos a carga de programas para execução. A MI é indexada pelo *program counter* (PC) e este registrador é incrementado de 4 a cada ciclo do relógio. A Figura 4.4 mostra um diagrama com o circuito que efetua a busca e a decodificação das instruções. O PC é incrementado de 4 em 4 porque a memória é indexada byte a byte mas as instruções ocupam uma palavra de 4 bytes.

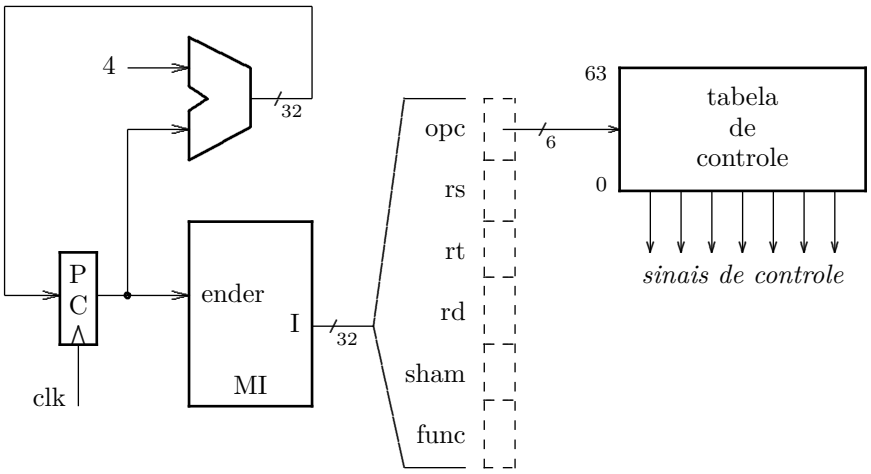


Figura 4.4: Circuito de busca e decodificação de instruções.

Uma vez que a saída do PC estabilize após a borda do relógio, e decorrido o tempo de acesso da memória, uma nova instrução fica disponível para ser decodificada e executada. A *decodificação* da instrução ocorre com a indexação da tabela de controle com o *opcode*, o que ativa todos os sinais de controle para a instrução recém buscada.

A largura dos sinais é indicada no diagrama na Figura 4.4 com um traço diagonal e pelo número de bits daquele sinal. O *opcode* tem 6 bits de largura, a instrução I e o próximo PC tem 32 bits. As setas indicam a 'direção' dos sinais – de saídas para entradas.

O circuito de busca não é mostrado nos próximos diagramas, que descrevem os caminhos de dados das instruções.

Espaço em branco proposital.

4.2.2 Operações de lógica e aritmética

As instruções de lógica e aritmética com formato R usam dois registradores como fonte dos operandos e um terceiro registrador como destino para o resultado. A Figura 4.5 mostra uma instrução **addu** no topo da figura e as ligações necessárias. Os campos da instrução `rs` e `rt` indexam os registradores e seus conteúdos são apresentados nas portas A e B do banco de registradores, e de lá para as entradas α e β da ULA. O *opcode* desta instrução é zero e portanto o campo `func` é usado para determinar a operação que a ULA efetua em seus operandos, que neste caso é uma soma.

O resultado da soma é levado da saída γ da ULA para a porta de escrita (C) do banco de registradores. O registrador que deve ser atualizado é indicado pelo campo `rd`, e o sinal escReg é ativado. Na borda ascendente do relógio a soma de `R(rs)` com `R(rt)` é armazenada em `R(rd)`.

```
addu rd, rs, rt     # R(rd) ← R(rs) + R(rt)
```

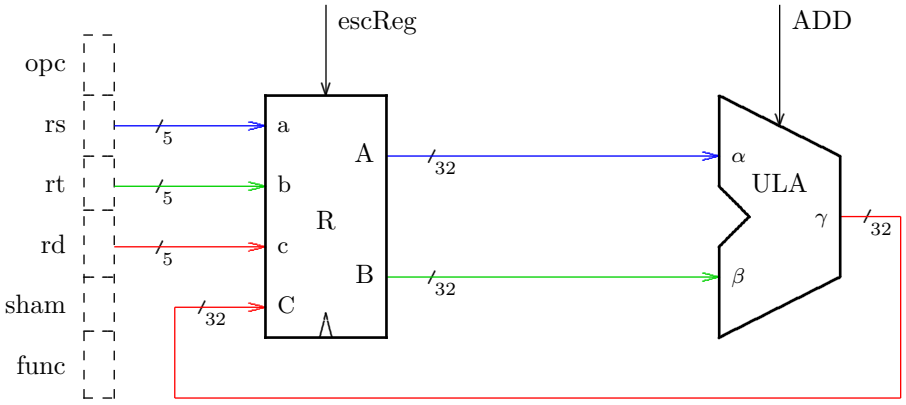


Figura 4.5: Operações de lógica e aritmética – `addu`.

4.2.3 Operações de lógica e aritmética com imediato

As instruções de lógica e aritmética com um operando imediato usam a constante que é parte da instrução como um dos operandos, e o outro operando é o conteúdo de um registrador, apontado por `rs`. O registrador de destino é apontado por `rt`. O operando da porta B do banco de registradores não é utilizado nesta instrução e portanto as ligações de b e B não são mostradas.

A constante é codificada em 16 bits, no campo `im16`, e ela deve ser estendida para 32 bits para que possa ser aplicada à entrada β da ULA. Se a operação é aritmética (**`addi`** ou **`addiu`**), então a constante é estendida para 32 bis pela replicação do seu bit de sinal. Se a operação é lógica (**`ori`** ou **`andi`**), a constante é estendida com 16 zeros.

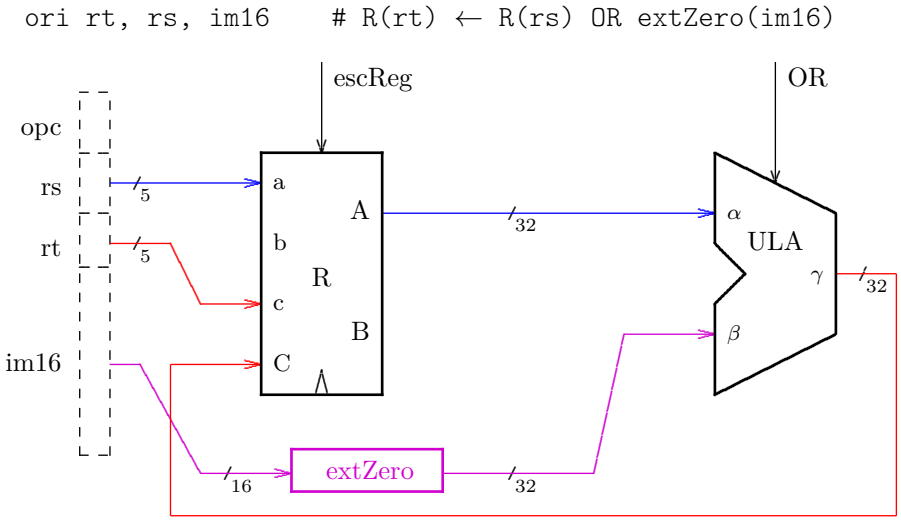


Figura 4.6: Operações de lógica e aritmética com imediato – `ori`.

O operando apontado por `rs` é apresentado à entrada α da ULA; o segundo operando (`im16`) é estendido para 32 bits e apresentado à entrada β da ULA. A operação da ULA é definida como OR e o resultado é armazenado no registrador apontado por `rt`, na borda de subida do relógio, e escReg deve ser 1.

As operações aritméticas são similares, exceto que a constante é estendida com sinal ao invés de com zeros.

Espaço em branco proposital.

4.2.4 Operação de acesso à memória – leitura

A instrução **`lw`** (*load-word*) copia, para o registrador apontado por `rt`, o conteúdo da palavra indexada pela soma de um deslocamento de 16 bits (`des16`) com o conteúdo do registrador apontado por `rs`. A soma do deslocamento com o registrador base é chamada de *endereço efetivo*.

A constante é estendida com sinal para permitir deslocamentos positivos e negativos com relação ao endereço apontado por `R(rs)`. A saída da ULA, que é o endereço efetivo, é aplicada à entrada de endereços da memória de dados. Decorrido o tempo de acesso à memória, a porta de saída `L` contem a cópia do valor lido da memória. Na borda do relógio este valor é armazenado no registrador apontado por `rt`.

A Figura 4.7 mostra o diagrama do processador com as ligações para a instrução **`lw`**.

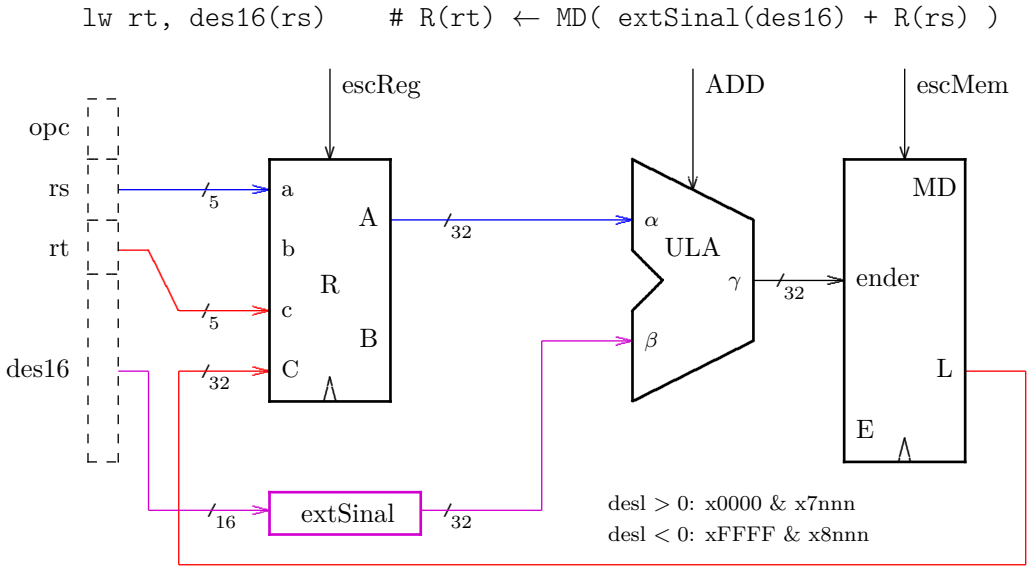


Figura 4.7: Operação de leitura da memória – `lw`.

Espaço em branco proposital.

4.2.5 Operação de acesso à memória – escrita

A instrução **`sw`** (*store-word*) copia o conteúdo do registrador apontado por `rt` para a palavra indexada pela soma de um deslocamento de 16 bits (`des16`) com o conteúdo do registrador base, apontado por `rs`.

O endereço efetivo é aplicado à entrada de endereços da memória de dados e o conteúdo de `R(rt)` é aplicado à porta de escrita E da memória de dados. O sinal escMem é ativado e na borda ascendente do relógio a posição indexada pelo endereço efetivo é atualizada.

A Figura 4.8 mostra o diagrama do processador com as ligações para a instrução **`sw`**.

```
sw rt, des16(rs)     # MD( extSinal(des16) + R(rs) ) ← R(rt)
```

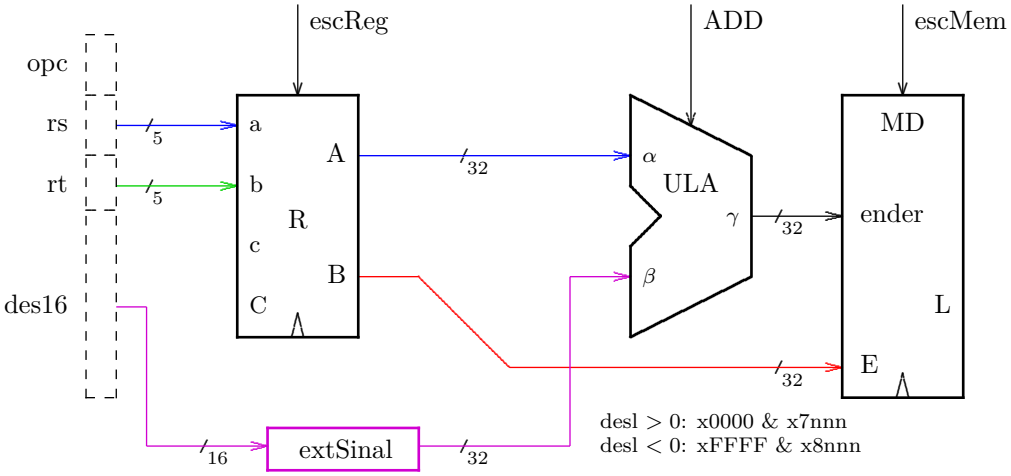


Figura 4.8: Operação de escrita em memória – `sw`.

Espaço em branco proposital.

4.2.6 Desvio condicional

Num desvio condicional (**beq**), o endereço da próxima instrução a ser buscada depende de uma comparação de igualdade. Para tanto, os dois valores por comparar são subtraídos e se o resultado for zero, então eles são iguais. A Figura 4.9 mostra o diagrama do processador com as ligações para a instrução **beq**. A operação na ULA é definida como uma subtração e o sinal iguais indica o resultado da comparação.

O *endereço de destino* de um desvio é obtido pela soma da constante `desl16`, estendida para 32 bits com sinal, mais o PC incrementado de quatro. A constante estendida é multiplicada por quatro antes de ser somada ao PC+4. Isso significa que o deslocamento nos desvios é o número de instruções por pular, e não o número de bytes. A multiplicação por quatro é obtida pelo deslocamento de duas posições para a esquerda.

Se iguais=1 então o endereço de destino é selecionado e aplicado à entrada do PC, e a instrução apontada por aquele endereço será buscada no próximo ciclo. Do contrário, iguais=0, PC+4 é aplicado à entrada do PC e a instrução após o **beq** será buscada.

```
beq rs, rt, des16    # if (R(rs)=R(rt)) PC <- PC+4 + extSinal(des16)*4
```

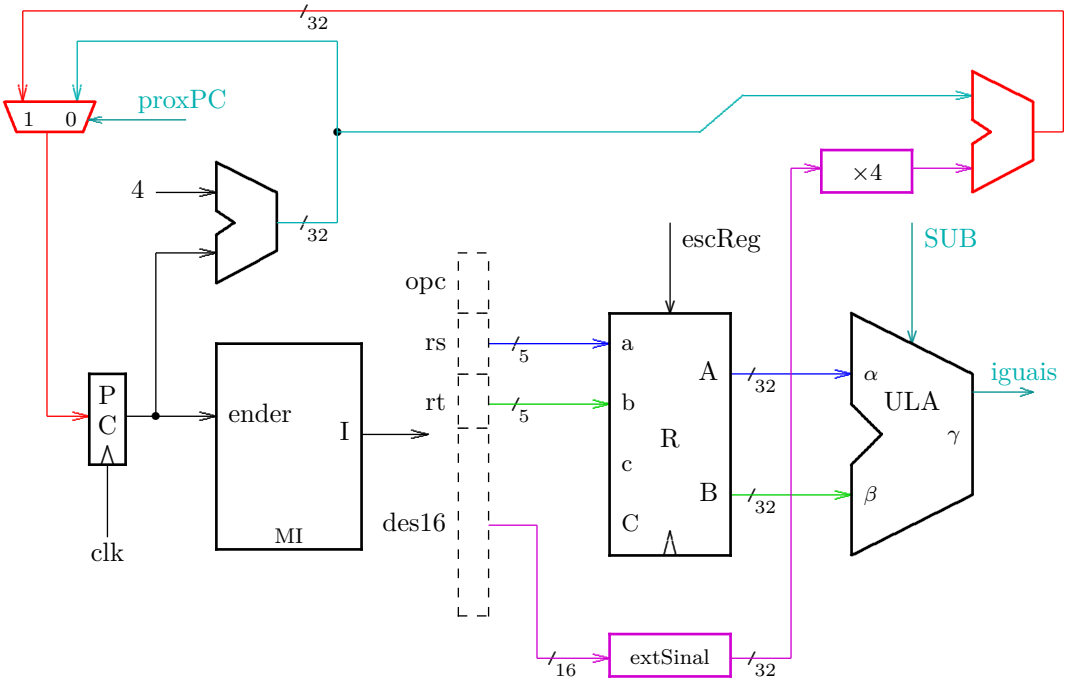


Figura 4.9: Desvio condicional – `beq`.

4.2.7 Salto incondicional

Num salto incondicional (`j` para *jump*), o endereço da próxima instrução a ser buscada está codificado na própria instrução. Esta é do Tipo J, com 6 bits para o *opcode* e 26 bits para o endereço de destino.

A especificação do conjunto de instruções determina que as instruções estejam em endereços alinhados, o que significa que os dois bits menos significativos do endereço de destino sejam *sempre* 00_2. Assim, os 26 bits da instrução concatenados com os dois 0s compõem um endereço de 28 bits. A faixa de endereços alcançáveis com 28 bits é $2^{28} = 256$ Mbytes, ou 64 M instruções. O folclore nos diz que, na média, um comando em C é implementado com 10 instruções, e portanto uma instrução de salto pode cobrir até 6,4 milhões de linhas de código fonte. Convenhamos que uma 'perna' de `if()` com 6 milhões linhas é assaz incomum, mas não é proibida pela definição da linguagem C e portanto o projetista do *hardware* deve garantir que um salto mais longo que 64 M instruções possa ser usado por programadores néscios, ou programas em Java.

Para permitir saltos para (quase) qualquer distância, os projetistas do MIPS decidiram por usar os 4 bits mais significativos do PC incrementado (PCinc) para completar os 32 bits do endereço. Assim, o endereço de destino de um salto incondicional é

$$\mathsf{PCinc(31..28)} \;\&\; \mathsf{ender26} \;\&\; \mathsf{00}$$

com a concatenação representada por &. Em *hardware*, a concatenação é a mera justaposição dos sinais, e no circuito da Figura 4.10, isso é representado pela "integral quadrada" ($\int$), sendo que a parte 'cortada' representa o lado mais significativo da justaposição.

```
j ender26     # PC <- PC+4(31..28) & ender26 & 00
```

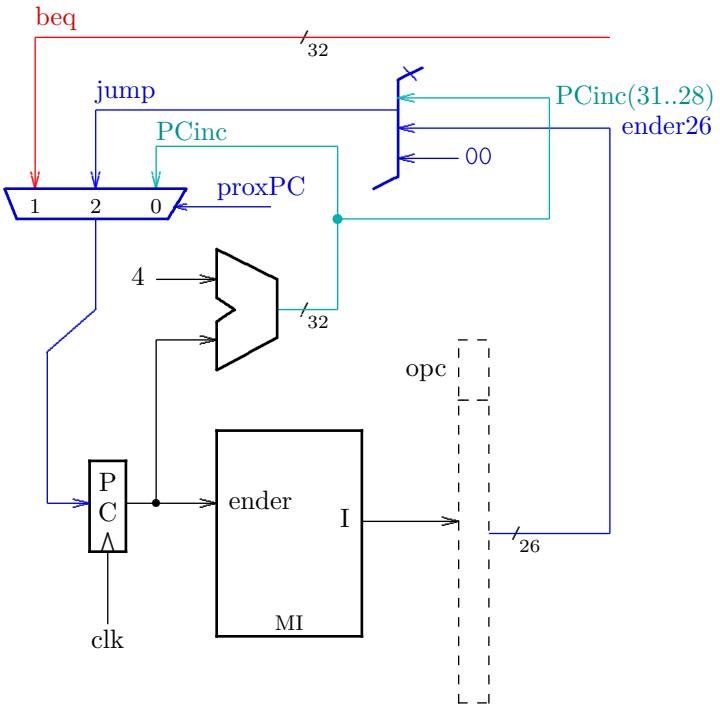


Figura 4.10: Salto incondicional – `j`.

O Exemplo 3.10, na página 34, mostra o código necessário para que qualquer das 2^{30} instruções possa ser alcançada como o destino de um salto incondicional.

4.2.8 Circuito de dados completo

O circuito de dados completo é a 'união' dos circuitos necessários para cada classe de instruções e é mostrado na Figura 4.11. Para simplificar o diagrama, somente uma parte do circuito para a instrução `j` é mostrada.

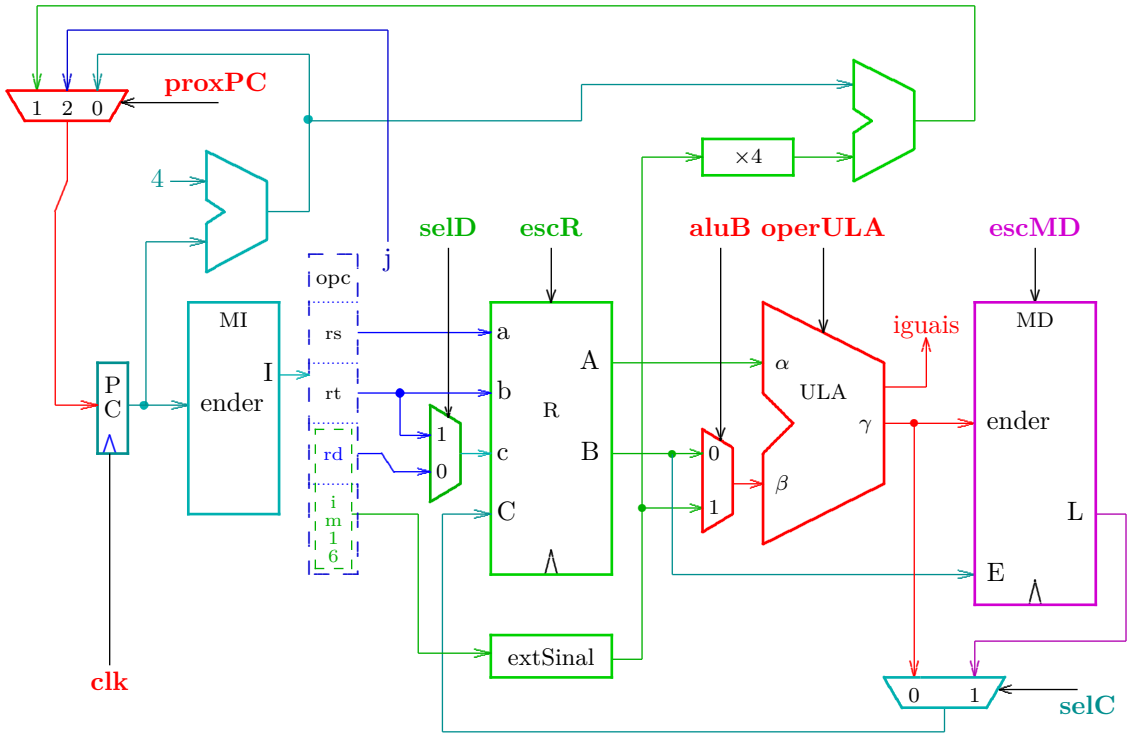


Figura 4.11: Circuito completo do processador.

Nas instruções com imediato, o registrador de destino não pode ser determinado pelo campo `rd` porque este é uma parte da constante de 16 bits, e por isso o campo `rt` é usado para determinar o registrador de destino.

O registrador de destino, cujo endereço é determinado ora por `rd` ora por `rt`, é escolhido por um multiplexador controlado pelo sinal selD, cuja saída é ligada à porta c do banco de registradores.

O valor que será gravado no registrador de destino pode ser aquele na saída da ULA, ou aquele lido da memória. Um multiplexador, controlado pelo sinal selC, define qual dos dois valores é apresentado à porta C do bloco de registradores para ser gravado no registrador de destino.

Um dos operandos da ULA é sempre o registrador apontado por `rs`, que é a porta A do banco de registradores. O outro operando pode ser a porta B do banco de registradores ou a constante estendida. O multiplexador controlado pelo sinal aluB determina o operando na porta β da ULA.

4.2.9 Tabela de controle

O projetista do processador deve elaborar uma *tabela de controle* com todos os sinais que controlam os componentes do processador. A Figura 4.12 mostra o circuito de busca e decodificação com os sinais de controle das unidades funcionais do processador – os nomes estão abreviados no diagrama por causa do espaço. A tabela de controle é indexada pelo

opcode da instrução recém buscada e o conteúdo do elemento indexado pelo *opcode* ativa somente os sinais de controle apropriados à instrução por executar.

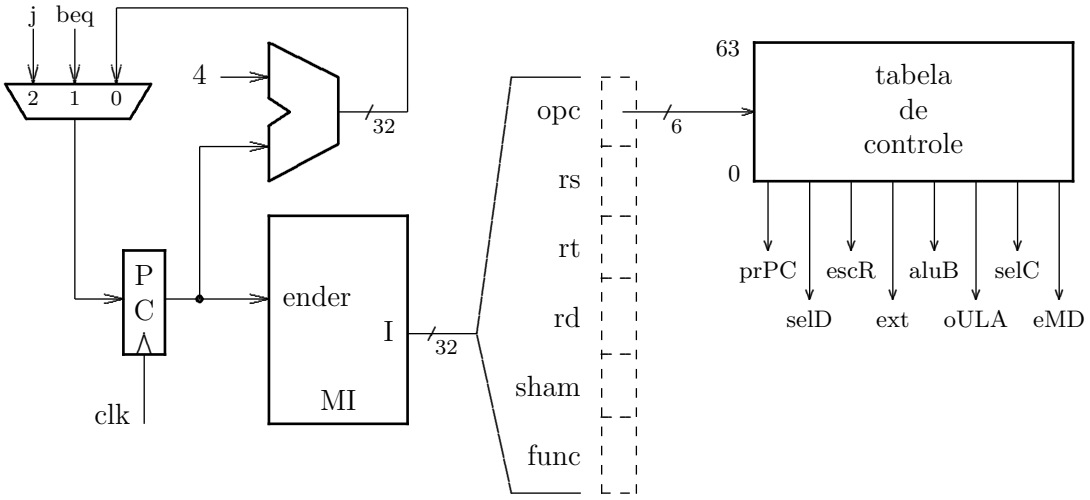


Figura 4.12: Circuito de busca e decodificação com tabela de controle.

A Tabela 4.2 mostra os sinais de controle que devem ser ativados para as instruções vistas nas seções anteriores. Sinais que prescindem de um valor determinado são indicados com um 'X' (*don't care*).

Os sinais de controle que alteram o estado da computação, proxPC, escR e escMD, devem estar *sempre* determinados porque não há nenhuma instrução que *talvez* altere a memória, ou que *talvez* altere os registradores.

Tabela 4.2: Tabela de controle para as instruções.

INSTRUÇÃO	proxPC	selD	escR	ext	aluB	operULA	selC	escMD
addu	0	0	1	X	0	func	0	0
ori	0	1	1	zero	1	OR	0	0
lw	0	1	1	sinal	1	ADD	1	0
sw	0	X	0	sinal	1	ADD	X	1
beq	iguais	X	0	sinal	0	SUB	X	0
j	2	X	0	X	X	X	X	0

4.2.10 Metodologia de sincronização

A metodologia de sincronização para esta implementação do conjunto de instruções MIPS32 é chamada de *ciclo longo* porque o período do relógio é longo o bastante para executar uma instrução durante um único ciclo.

A Figura 4.13 mostra o diagrama de tempo da execução de um **addu**. Um diagrama simplificado do processador é mostrado na base da figura, com com o bloco de registradores dividido em duas partes: no centro do diagrama está a parte na qual ocorre a leitura dos operandos, e na direita a parte em que ocorre a gravação do resultado.

Espaço em branco proposital.

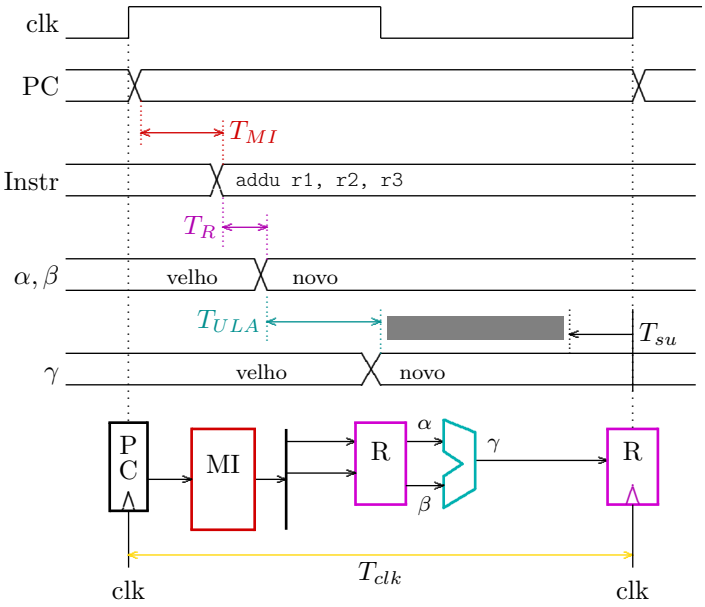


Figura 4.13: Diagrama de tempo de uma operação de ULA.

Decorrido o tempo de acesso à memória de instruções (T_{MI}), a nova instrução é decodificada paralelamente à leitura do banco de registradores (T_R). Com os operandos disponíveis, decorre o tempo de propagação através da ULA (T_{ULA}) até que o resultado esteja disponível no sinal γ. Há folga para respeitar a limitação de *setup* do banco de registradores (T_{su}), como mostra o intervalo entre a disponibilidade do novo valor em γ e a indicação do tempo de estabilização dos sinais (*setup*) – a folga é indicada pela barra de cor cinza.

A Figura 4.14 mostra o diagrama de tempo da execução de um `lw`. O período do relógio deve ser longo o bastante para acomodar o tempo de acesso à memória de instruções (T_{MI}), a leitura do banco de registradores (T_R), o tempo de propagação através da ULA (T_{ULA}), o tempo de acesso para leitura da memória de dados (T_{MD}), e respeitar a limitação de *setup* do banco de registradores (T_{su}), e a folga é indicada pela barra de cor cinza.

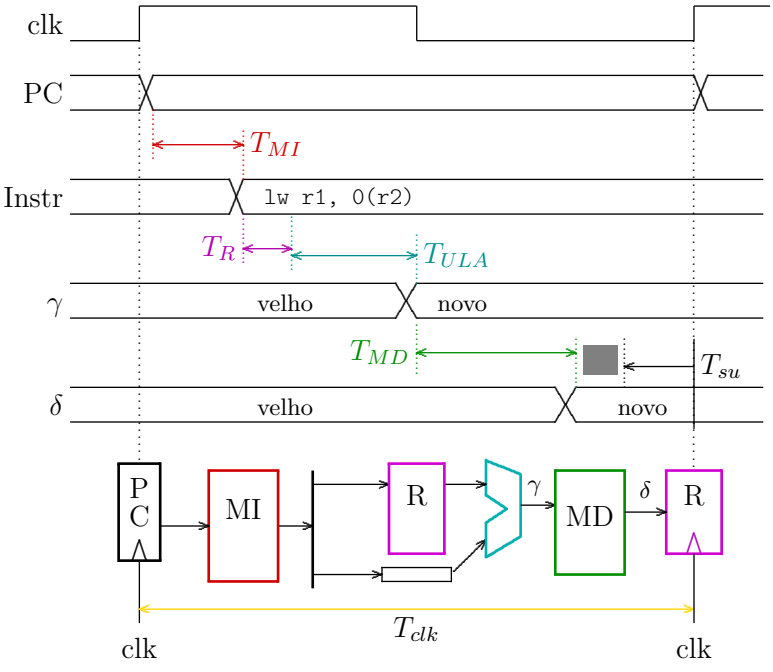


Figura 4.14: Diagrama de tempo de uma leitura em memória.

A instrução **lw** é a instrução mais demorada porque todas as unidades funcionais do processador são usadas em série, MI $\rightsquigarrow$ R $\rightsquigarrow$ ULA $\rightsquigarrow$ MD $\rightsquigarrow$ R, e portanto o ciclo do processador é limitado pela sua temporização.

As outras instruções não usam todas as unidades funcionais e há algum desperdício de tempo porque o ciclo do relógio deve ser fixado para atender ao pior caso, que é o **lw**. Por exemplo, na instrução **addu**, as unidades funcionais utilizadas são MI $\rightsquigarrow$ R $\rightsquigarrow$ ULA $\rightsquigarrow$ R, não há acesso à memória de dados, e o ciclo desta instrução poderia ser encurtado de T_{MD}.

Alterar o ciclo do relógio em função da instrução não é realizável e por isso o período do relógio deve ser tal que acomode a temporização da instrução mais demorada. Na próxima seção veremos uma implementação mais eficiente do que a do ciclo longo.

Exemplo 4.1 Vejamos como deve ser modificado o processador para acrescentar o circuito de dados para a instrução BRANCH-AND-LINK definida abaixo. A instrução **bal** tem formato I.

```
bal desl  # R[31] ← PC+8 , PC ← (PC+4)+(ext(desl)≪2)
```

Esta instrução é uma mistura de uma instrução **jal** com um "desvio incondicional": o endereço de ligação é armazenado no registrador `ra` como num **jal**: `R[31] ← PC+8`, e o endereço de destino é computado como numa instrução **beq**: `PC ← (PC+4)+(ext(desl)≪2)`.

Quais são as modificações necessárias ao circuito da Figura 4.11?

(a) é necessário acrescentar uma entrada ao multiplexador controlado pelo sinal selD, fixada em 31 para garantir que ao registrador 31 seja atribuído o valor de PC+8;

(b) deve-se acrescentar um somador para somar 4 ao valor PC+4;

(c) a saída deste somador deve ser ligada a uma entrada adicional no multiplexador controlado pelo sinal selC, para que PC+8 possa ser atribuído ao registrador 31; e

(d) o sinal `(PC+4)+(ext(desl)≪2)` é computado para os desvios, e este sinal já está ligado à entrada 1 do multiplexador controlado por proxPC.

A Figura 4.15 mostra o diagrama de tempos da execução de **bal**. Assim que o valor no PC estabiliza, decorrido T_{PC}, o incremento de 4 fica disponível após o tempo de propagação do somador da esquerda (T_{add}). O segundo incremento de 4 fica estável após $T_{PC} + 2 \times T_{add}$. A faixa cinza indica que a limitação de *setup* do banco de registradores é respeitada com folga. Decorrido o tempo de acesso à memória de instruções, o endereço de destino fica estável após $T_{PC} + T_{MI} + T_{add}$.

A Tabela 4.3 mostra o estado dos sinais de controle durante a execução de **bal**. Para fins de comparação, a tabela também mostra os sinais da instrução **beq**. Os sinais que controlam multiplexadores que foram alargados necessitam de dois bits de controle ao invés de um. ◁

Tabela 4.3: Tabela de controle para a instrução `bal`.

INSTRUÇÃO	proxPC	selD	escR	ext	aluB	operULA	selC	escMD
beq	iguais	X	0	sinal	0	SUB	X	0
bal	1	2	1	sinal	X	X	2	0

4.2.11 Exercícios

Ex. 34 Mostre como implementar as instruções abaixo. Para tanto, indique quaisquer modificações que sejam necessárias no processador da Figura 4.11 e mostre a tabela de

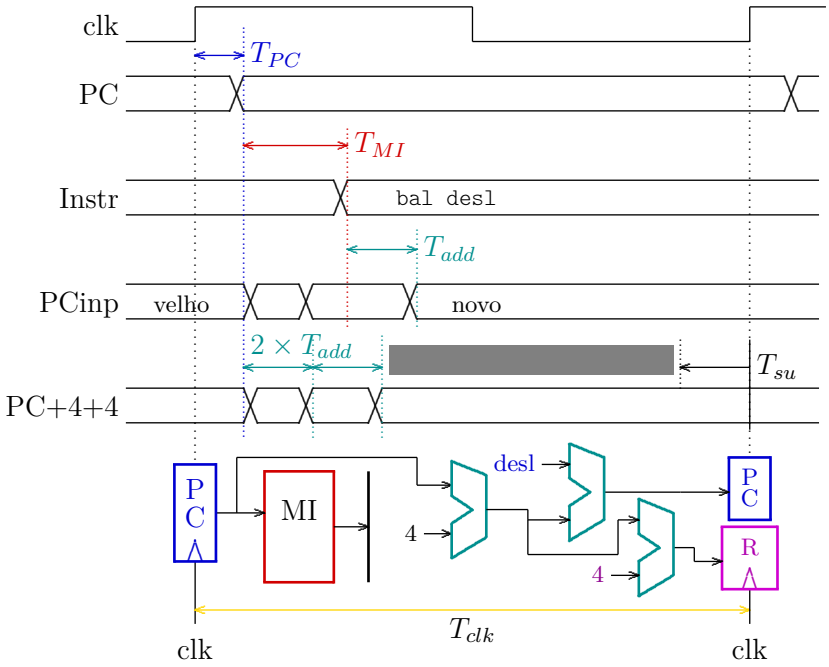


Figura 4.15: Diagrama de tempo da instrução `bal`.

sinais de controle ativos durante a execução da instrução. Nos comentários, a vírgula significa "execução simultânea".

```
                # jump-register
jr rs           # PC <- R(rs)

                # jump-and-link
jal ender26     # R(31) <- PC+8 ,
                # PC <- PCinc(31..28) & ender26 & 00

                # jump-and-link-register
jalr rt,rs      # R(rt) <- PC+8 , PC <- R(rs)
```

4.3 Implementação segmentada

A técnica de *segmentação* (*pipelining*) é usada para aumentar a taxa com que as instruções completam. Suponha que um determinado circuito combinacional tenha tempo de propagação T_p – veja a Figura 4.16. A entrada do circuito é alimentada por um registrador e sua saída é capturada por outro registrador.

O período mínimo do relógio para este circuito é dado por

$$T_{\min} \geq T_p + T_r + T_{su},$$

que é a soma do tempo de propagação do circuito ($T_p = n$), do tempo de propagação do registrador de entrada (T_r), mais o *setup* do registrador de saída (T_{su}). Para este circuito, a latência é $n + T_r + T_{su}$ segundos, e a vazão é $1/(n + T_r + T_{su})$ resultados por segundo, e pode ser aproximada por $1/n$, porque, no geral, $n \gg (T_r + T_{su})$.

Se for possível particionar o circuito combinacional em duas metades iguais, o período do relógio pode ser reduzido para a metade se um registrador for inserido entre as duas partes, e neste caso

$$T_{\min,2} \geq n/2 + T_r + T_{su},$$

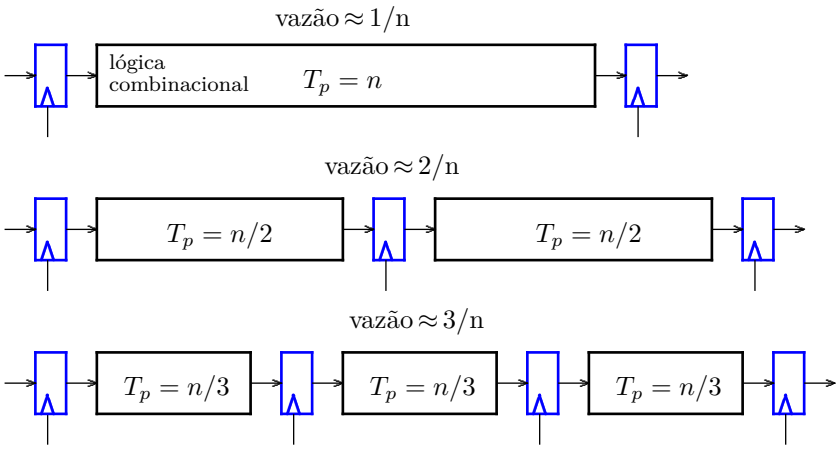


Figura 4.16: Circuito combinacional segmentado.

a vazão é de $\approx 2/n$ resultados/s, e a latência permanece em $2n/2 + T_r + T_{su}$ s. Se o circuito pode ser particionado em três segmentos,

$$T_{\min,3} \geq n/3 + T_r + T_{su},$$

a vazão sobe para $\approx 3/n$ resultados/s e a latência permanece em $3n/3 + T_r + T_{su}$ s.

Por "a latência permanece a mesma" entenda-se que o tempo para produzir um resultado não aumenta com a divisão do circuito em dois ou em três segmentos; o que aumenta é a taxa com que novos resultados são produzidos pelo circuito segmentado – a vazão aumenta de forma quase proporcional ao número de segmentos. O 'quase' decorre de que dificilmente os circuitos combinacionais possam ser particionados de forma a que o tempo de propagação dos dois segmentos seja exatamente igual, e esse desbalanceamento é a principal causa para que a vazão não seja exatamente proporcional ao número de segmentos.

A analogia mais próxima com a segmentação de um circuito é uma linha de montagem de automóveis, na qual o longo processo de montagem de um carro é subdividido em muitas tarefas simples. Um carro demora algo como 48 horas para ser produzido, mas uma linha de montagem moderna produz um veículo pronto e acabado a cada 15 minutos, dependendo do modelo e da fábrica.

4.3.1 Processador segmentado

A Figura 4.17 mostra o processador de ciclo longo já segmentado em cinco estágios, com registradores de segmento introduzidos nos locais apropriados. A execução de uma instrução é dividida em cinco estágios: (i) busca na memória de instrução e incremento do PC; (ii) decodificação e leitura dos operandos; (iii) execução na ULA; (iv) acesso à memória nos *loads* e *stores*; e (v) gravação do resultado.

Note que o banco de registradores aparece dividido no diagrama: no estágio de decodificação (**decod**), os registradores são lidos, enquanto que no estágio de resultado (**result**) o registrador de destino é atualizado.

O período do relógio do processador é determinado pelo estágio mais demorado, e este é o estágio de acesso à memória. A latência de uma única instrução é aproximadamente a mesma daquela do processador de ciclo longo, enquanto que a vazão, ou a taxa em que uma nova instrução é completada, é aproximadamente cinco vezes maior. A Figura 4.18 mostra dois diagramas de tempo, um do processador de ciclo longo executando duas instruções, e outro do processador segmentado executando três instruções.

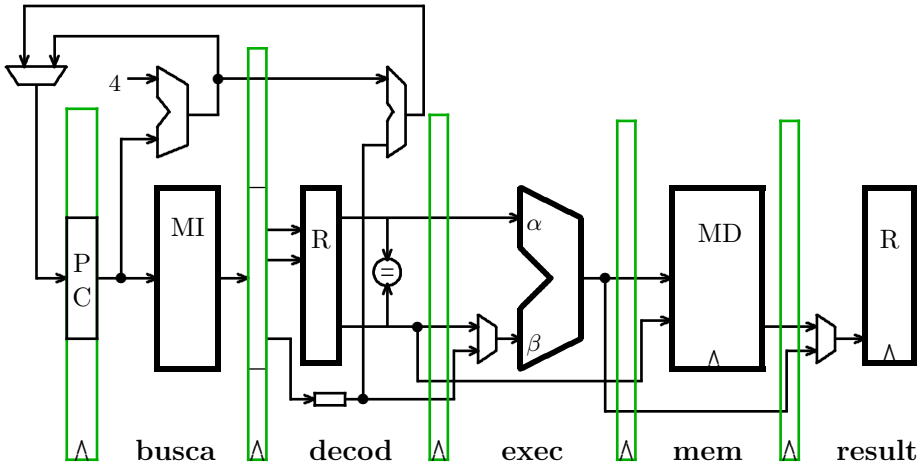


Figura 4.17: Processador segmentado em cinco estágios.

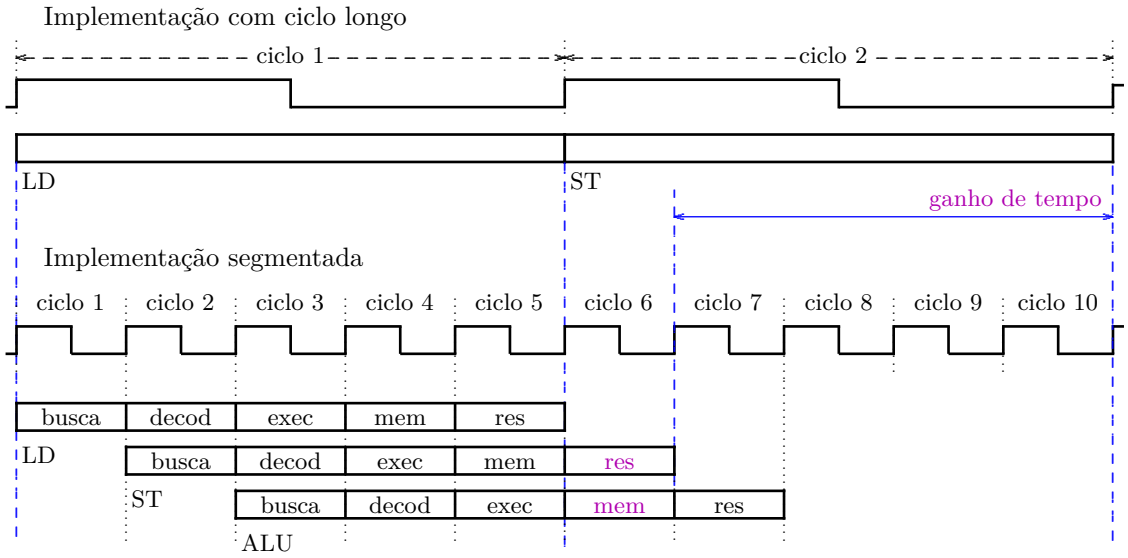


Figura 4.18: Diagrama de tempo do processador de ciclo longo e segmentado.

Enquanto o processador de ciclo longo emprega 2 ciclos longos para executar um *load* seguido de um *store* – estes dois ciclos longos equivalem a 10 ciclos do processador segmentado, o processador segmentado executaria *seis* instruções neste mesmo intervalo. A primeira instrução completa no ciclo número 5, a segunda no ciclo 6, e a sexta no ciclo 10. Passado o intervalo para encher os segmentos com instruções, uma nova instrução completa a cada ciclo curto. O ganho potencial de desempenho, medido pela taxa de instruções executadas, é de um fator de cinco, que é o número de segmentos.

A Figura 4.19 mostra o quinto ciclo do diagrama de tempos da Figura 4.18: o valor lido da memória pelo **`lw`** é gravado no registrador de destino; o **`sw`** escreve na memória de dados; o **`add`** é computado na ULA; o **`xor`** é decodificado, `r12` e `r13` são obtidos do bloco de registradores; o **`sub`** é buscado da MI e o PC é incrementado de 4.

O ganho de desempenho advém do paralelismo no nível de instrução: cinco instruções executam ao mesmo tempo, cada uma delas num estágio distinto de execução. Os registradores de segmentação efetivamente separam cada instrução da instrução que a precede, e da instrução que a sucede. Como na linha de montagem de carros, a cada borda do relógio as instruções ficam mais perto de completar.

Espaço em branco proposital.

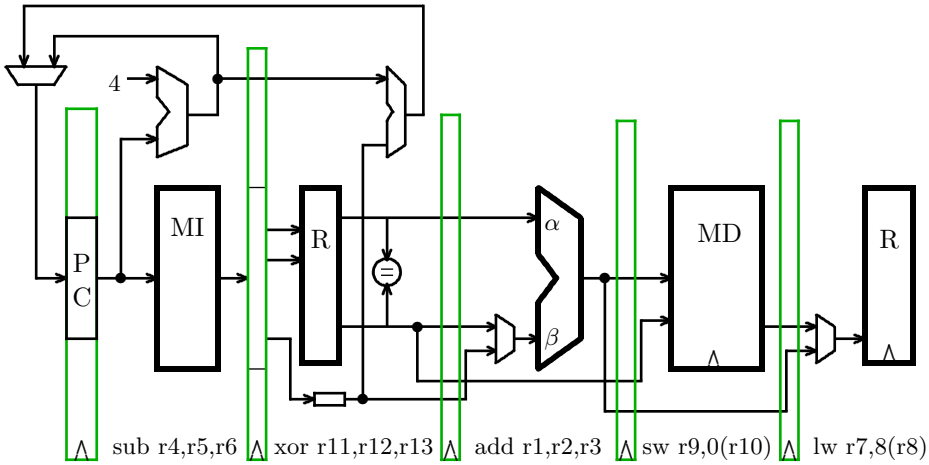


Figura 4.19: Cinco instruções em cinco estágios de execução.

Compare o diagrama do processador na Figura 4.19 com aquele da Figura 4.11. No processador segmentado, o circuito necessário para as instruções de desvio está todo concentrado em **decod**. Isso é possível com a adição de um comparador de igualdade entre as portas de leitura do banco de registradores para gerar o sinal iguais, ao invés de empregar-se a ULA na comparação. Em breve veremos o motivo para resolver os desvios em **decod**.

Os registradores de segmento são *invisíveis* ao programador, porque estes registradores não fazem parte do estado da computação como definido pelo conjunto de instruções MIPS32r2. Os *registradores visíveis* são aqueles explicitamente usados como operandos das instruções.

Espaço em branco proposital.

Exemplo 4.2 Vejamos como se dá a execução de um `addu` no processador segmentado. A Figura 4.20 mostra a sequência de execução. O tempo decorre de cima para baixo, ao longo dos cinco ciclos de relógio necessários para completar uma instrução (C_0 a C_4), um ciclo em cada estágio. O registrador "de saída" dos estágios é mostrado em azul, e na próxima borda do relógio, o registrador de saída é carregado com os valores produzidos pelos circuitos combinacionais, alimentados pelo "registrador de entrada" do estágio.

Busca (C_0): o conteúdo do PC indexa a memória de instruções (MI), e no final do ciclo a instrução `addu` é carregada no registrador de saída; em paralelo ao acesso à memória de instruções, o PC é incrementado de 4.

Decod (C_1): a instrução é decodificada – a tabela de controle é indexada com o *opcode*, e o banco de registradores é acessado: `A` $\leftarrow$ `R(rs)` e `B` $\leftarrow$ `R(rt)`. O endereço do registrador de destino (`rd`=c) é transferido para o próximo estágio.

Exec (C_2): às entradas α e β da ULA são atribuídos os conteúdos dos registradores `A = R(rs)` e `B = R(rt)`, e o resultado é apresentado em γ. O endereço do registrador de destino (c$'$ $\leftarrow$ c) é transferido para o próximo estágio.

Mem (C_3): o resultado é transferido sem alteração para o estágio de resultado $\gamma' \leftarrow \gamma$. O endereço do registrador de destino (c$''$ $\leftarrow$ c$'$) é transferido para o próximo estágio.

Result (C_4): o resultado da operação da ULA ($\gamma'' \leftarrow \gamma'$) é gravado no registrador apontado pelo registrador de destino `rd` (c$'''$ $\leftarrow$ c$''$): `R(`c$'''$`)` $\leftarrow \gamma''$.

O resultado da operação na ULA (γ) é copiado, sem alteração, do registrador de entrada para o registrador de saída no estágio de memória ($\gamma' \leftarrow \gamma$), e este mesmo valor ($\gamma'' \leftarrow \gamma'$) é selecionado pelo multiplexador no estágio de resultado, e atribuído ao registrador de destino, como definido para a instrução `addu`: `R(rd)` $\leftarrow$ `(R(rs) + R(rt)`. $\triangleleft$

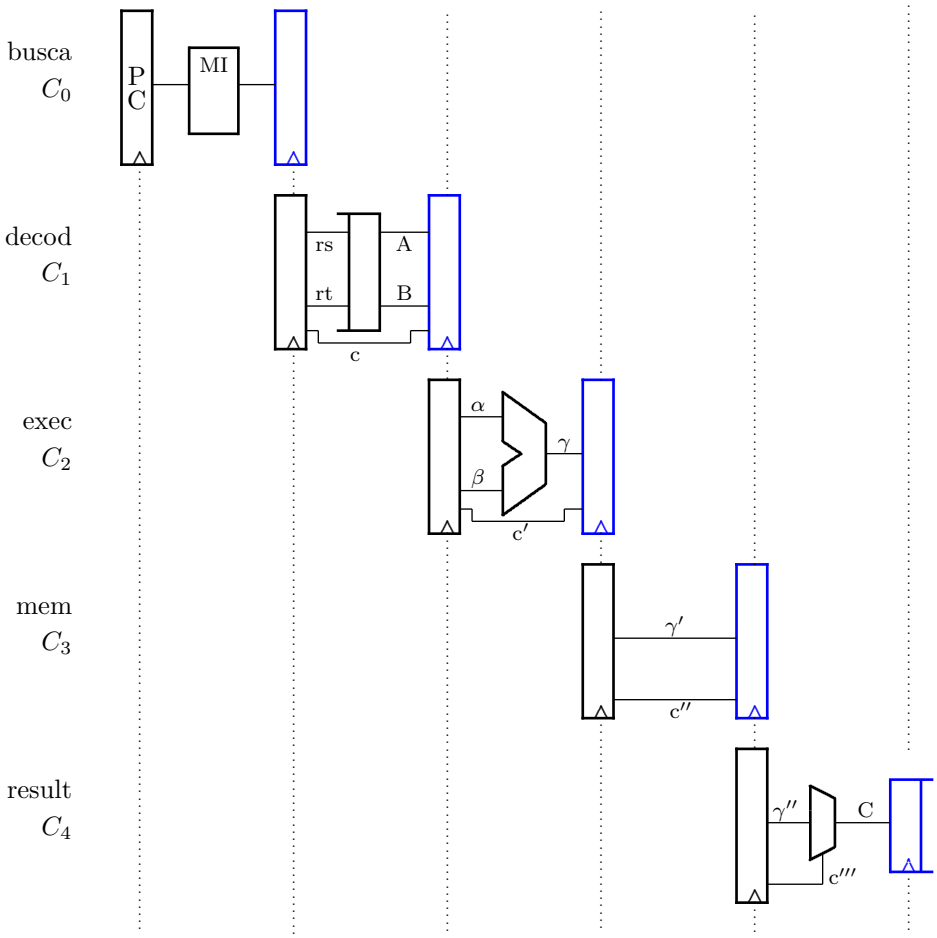


Figura 4.20: Progresso de um `addu` pelos segmentos.

Exemplo 4.3 Como um segundo exemplo, sigamos a execução de um `lw`. A Figura 4.21 mostra a sequência de execução.

Busca (C_0): o conteúdo do PC indexa a memória de instruções (MI), e no final do ciclo a instrução `lw` é carregada no registrador de saída; em paralelo ao acesso à memória de instruções, o PC é incrementado de 4.

Decod (C_1): a instrução é decodificada – a tabela de controle é indexada com o *opcode*, o banco de registradores é acessado (`A` $\leftarrow$ `R(rs)`), e o deslocamento é estendido com o sinal (`D` $\leftarrow$ `extS(desl)`. O endereço do registrador de destino (`rt`=c) é transferido para o próximo estágio.

Exec (C_2): às entradas α e β da ULA são atribuídos os conteúdos dos registradores ($\alpha \leftarrow$ `A`) e do deslocamento ($\beta \leftarrow$`D`), e o endereço efetivo é apresentado em γ. O endereço do registrador de destino (c′ $\leftarrow$ c) é transferido para o próximo estágio.

Mem (C_3): o endereço efetivo γ' é aplicado à entrada de endereço da memória de dados (MD), e o conteúdo indexado é atribuído para o registrador de saída ($\delta \leftarrow \mathrm{MD}(\gamma')$). O endereço do registrador de destino (c″ $\leftarrow$ c′) é transferido para o próximo estágio.

Result (C_4): o valor obtido da memória ($\delta' \leftarrow \delta$) é gravado no registrador apontado pelo registrador de destino (c‴ $\leftarrow$ c″), como definido para um `lw`: `R(rt) ← MD( R(rs) + extS(desl) ).` ◁

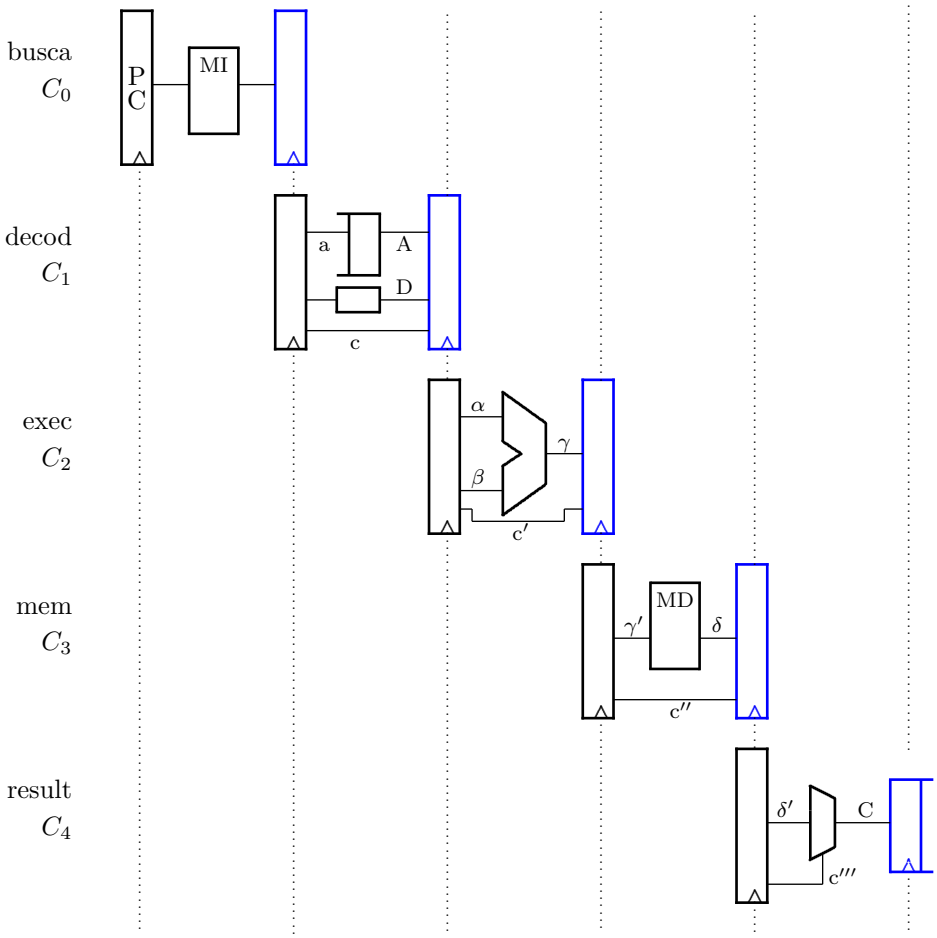


Figura 4.21: Progresso de um `lw` pelos segmentos.

Exemplo 4.4 A instrução BRANCH-AND-LINK é similar a um JUMP-AND-LINK, exceto que o modo de endereçamento é o mesmo daquele de um desvio condicional: o endereço de destino é um deslocamento com relação ao PC. A especificação da instrução `bal` é:

```
bal desl  # R(31) ← PC+8 , PC ← (PC+4)+(ext(des)≪ 2)
```

Esta instrução tem formato do Tipo I. O circuito para executar a instrução BRANCH-AND-LINK é mostrado na Figura 4.22 – neste diagrama o banco de registradores *não* está separado em uma seção de leitura e outra de escrita.

O endereço de destino independe de qualquer comparação, e o circuito para computá-lo é o mesmo que para as instruções `beq`: a constante é estendida para 32 bits, multiplicada por 4 e então adicionada ao PC+4.

O endereço de retorno pode ser computado no estágio de decodificação ((PC+4)+4), e deve ser levado até o estágio de resultado, para então ser armazenado no registrador 31. Para tanto, o multiplexador no estágio de resultado ganha uma nova entrada (PC+8), assim como multiplexador do registrador de destino ganha uma entrada fixa em 31.

O endereço de retorno deve ser PC+8 porque este é o endereço da primeira instrução que está além do *branch delay slot* – este assunto deve esperar até a Seção 4.3.2. ◁

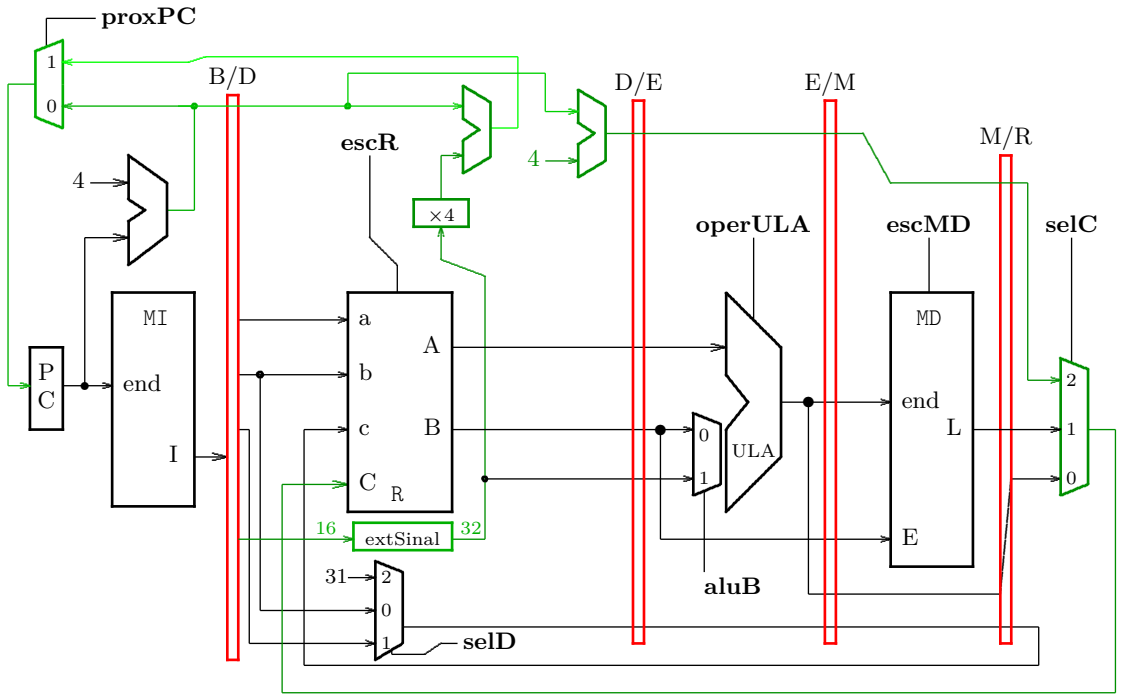


Figura 4.22: Processador com suporte à instrução `bal`.

4.3.2 O ganho é menor que o número de estágios

Ao contrário dos automóveis numa linha de montagem, as instruções de um programa dependem de outras instruções que se encontram adiante na "linha de instruções", e isso tem impactos significativos no desempenho de processadores segmentados.

Nos interessam três situações em particular: dependências de dados, desvios e *loads*.

Usaremos um diagrama de blocos simplificado do processador para examinarmos a execução das instruções através dos segmentos. A Figura 4.23 mostra o modelo simplificado do processador.

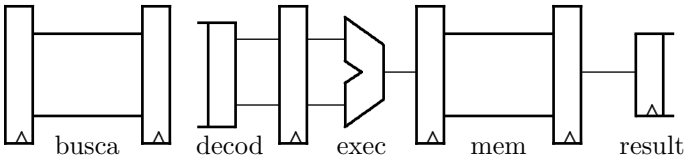


Figura 4.23: Modelo simplificado do processador.

Dependências de dados

Uma das formas de dependência entre instruções é chamada de *dependência de dados* e o nome decorre do fluxo de dados que existe entre duas ou mais instruções. Os dados fluem de uma instrução produtora para instruções consumidoras através dos registradores visíveis ao programador. As instruções consumidoras são chamadas de *instruções dependentes*.

Considere o trecho de programa abaixo, no qual o valor computado pelo **`add`** em `r1` é usado como operando pelo **`sub`**. O **`sub`** *depende* do **`add`** para computar a diferença entre `r2` e `r1`.

```
add r1, r6,r7
sub r3, r2, r1
```

O diagrama do processador na Figura 4.24 mostra o ciclo em que o **add** está no estágio de execução e o **sub** está obtendo seus operandos do banco de registradores. O valor contido em r1, no banco de registradores, é um valor caduco que foi produzido por alguma instrução anterior, e não a soma de r6 com r7, que é o desejado pelo programador.

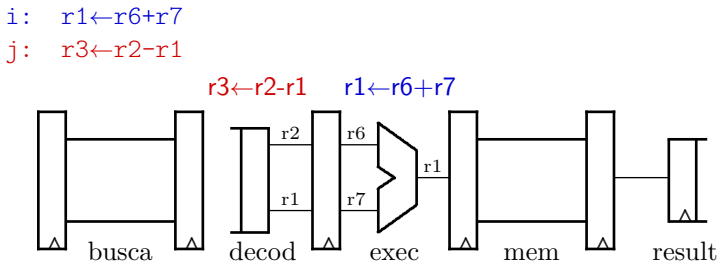


Figura 4.24: Dependência de dados.

A implementação do processador segmentado cria um *risco*: se esta sequência for executada, existe o risco – neste caso uma certeza – de o programa produzir resultados errados. Esta distinção é importante: as dependências são introduzidas no programa pelo programador; os riscos são introduzidos pelo projetista do processador. Se a implementação pode violar as dependências entre instruções, então os projetistas do *hardware* e do *software* devem prover os meios para a execução correta dos programas.

No caso acima a solução é simples porém ineficiente: basta afastar a instrução dependente da instrução produtora para garantir que seu resultado seja depositado no banco de registradores *antes* que este valor seja lido pela instrução dependente. No diagrama da Figura 4.25, a instrução dependente está separada da produtora por dois **nop**s, o que garante que o valor usado pelo **sub** será aquele produzido pelo **add**.

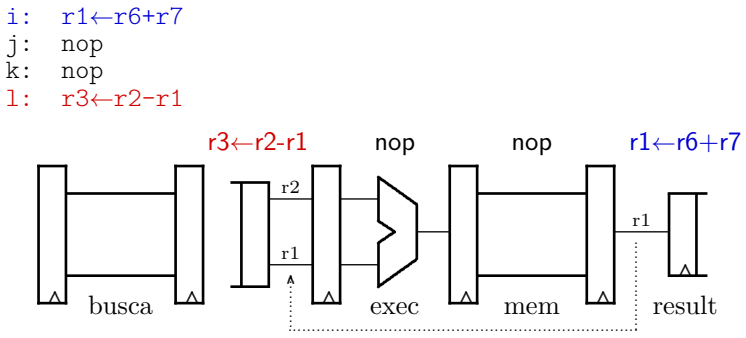


Figura 4.25: Dependência de dados resolvida com dois nops.

A corretude é obtida ao custo de dois ciclos desperdiçados por causa da dependência de dados. Se o compilador reordenar o código, é possível substituir um, ou os dois, **nop**s por instruções que efetuem trabalho útil e assim minorar a queda de desempenho causada pela dependência.

Exemplo 4.5 Abaixo estão dois trechos de código que computam o mesmo resultado. No lado esquerdo, o código ignora as dependências de dados entre as instruções. No lado direito está a versão que executaria corretamente no processador da Figura 4.23. Verifique se todas as dependências de dados estão resolvidas com os `nop`s que foram adicionados para garantir execução correta.

```
1  # riscos por resolver
2  add  r5, r6, r7
3  sub  r8, r9, r5 # r5
4  addi r2, r8, 9  # r8
5  xor  r10, r8, r5 # r8,r5
6  add  r5, r10, r2 # r10,r2
```

```
7  # riscos resolvidos
8  add  r5, r6, r7
9  nop
10 nop
11 sub  r8, r9, r5
12 nop
13 nop
14 addi r2, r8, 9
15 xor  r10, r8, r5
16 nop
17 nop
18 add  r5, r10, r2
```

Os `nop`s das linhas 9 e 10 separam o produtor de `r5` (`add`) do consumidor daquele valor (`sub`). Os `nop`s das linhas 12 e 13 separam o produtor de `r8` (`sub`) do consumidor daquele valor (`addi`) e também resolvem a dependência do (`xor`) em `r8`. Os `nop`s das linhas 16 e 17 resolvem as dependências em `r10` e `r2`. A resolução dos riscos neste trecho de código custa seis ciclos adicionais, o que reduz a vazão em mais de 50% neste trecho de código. Felizmente, é possível minimizar as perdas causadas por dependências de dados. ◁

Existem duas técnicas de *hardware* para minimizar o risco causado pela dependência de dados. Na mais simples, o controle do processador detecta a dependência, ao comparar os números dos registradores de resultado nos estágios **exec** e **mem** com os operandos das instruções em **decod**. Uma vez detectada a dependência, a instrução que está em **decod** fica bloqueada até que a instrução produtora chegue a **result**, quando então o valor atualizado é obtido do banco de registradores. Há desperdício de tempo, mas o compilador não é obrigado a inflar o código com a inserção dos **nop**s. Os ciclos perdidos por causa dos bloqueios são chamados de *bolhas*.

Exemplo 4.6 Vejamos uma parte do circuito que computa as dependências de dados entre as instruções. Este circuito é geralmente implementado no estágio de decodificação, e é usado para decidir se a instrução que está sendo decodificada deve esperar até que seus operandos estejam disponíveis no banco de registradores.

O número do registrador de destino deve ser transportado com a instrução, na medida em que esta progride pelos estágios, para que seja possível 'casar' o resultado com o registrador de destino. Chamemos este registrador de a_c, e cada um dos registradores de segmento possui um campo a_c, *viz.* EXE.a_c, MEM.a_c, e RES.a_c.

O circuito que detecta uma dependência no primeiro operando de uma instrução, `DEC.rs` pode ser modelado em VHDL como:

```
if (DEC.rs = EXE.a_c) and
   (EXE.a_c != 0)     and
   (EXE.regWR = true)
then
  segura_em_decod <= true;
end if;
```

Se a instrução no estágio de execução escreve num registrador (`regWR`) e este registrador não é o registrador `r0`, e o registrador de destino daquela instrução é o operando da instrução que está sendo decodificada, então esta instrução deve permanecer no estágio **decod** (`segura_em_decod`), para que a instrução produtora avance até o estágio **result**, quando esta gravará o seu resultado no registrador de destino.

Um teste similar deve ser efetuado para a instrução que está em **mem**:

```
if ( ( (DEC.rs = EXE.a_c) and
       (EXE.a_c != 0)     and
       (EXE.regWR = true) )
     or
     ( (DEC.rs = MEM.a_c) and
       (MEM.a_c != 0)     and
       (MEM.regWR = true) ) )
then
  segura_em_decod <= true;
end if;
```

Se a instrução no estágio de memória produz um operando para a instrução em **decod**, então esta deve esperar até que aquela chegue em **result**.

Estes testes, com as adaptações necessários, devem ser efetuados para o segundo operando das instruções, que é apontado pelo campo `rt`. ◁

O que é necessário para transformar uma instrução numa bolha? Para cancelar o efeito de uma instrução basta impedir que ela altere o estado da computação – deve-se anular os sinais que habilitam a escrita no PC, no banco de registradores e na memória de dados.

A técnica mais eficiente, e com maior complexidade, consiste em adiantar os resultados das instruções produtoras para a instrução dependente. O *adiantamento* de resultados pode eliminar todas as dependências de dados, com exceção do uso do *load*.

Um multiplexador é adicionado a cada entrada da ULA e ligações adiantam os valores que estão disponíveis nos registradores de segmentos nos estágios adiante da instrução que está em **exec**. O circuito de detecção de dependências é usado para acionar as entradas dos multiplexadores de forma a adiantar os valores dos registradores de segmento diretamente às entradas da ULA. A Figura 4.26 mostra o circuito de adiantamento.

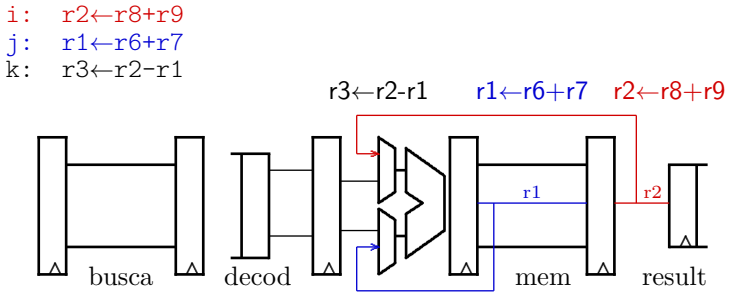


Figura 4.26: Dependência de dados resolvida com adiantamento.

Com adiantamento evitam-se as bolhas e o processador pode executar instruções na máxima taxa possível, completando uma instrução por ciclo. Infelizmente, *loads* e desvios são problemáticos, como mostra a discussão que se segue.

Uso do *load*

A Figura 4.27 mostra um diagrama do processador quando executa um `lw` seguido de um `add`. O resultado do `lw r1,100(r6)` é gravado no registrador de segmento do estágio **mem** *somente* no final do ciclo em que a leitura da memória é concluída, porque o circuito de memória é o componente mais lento do processador.

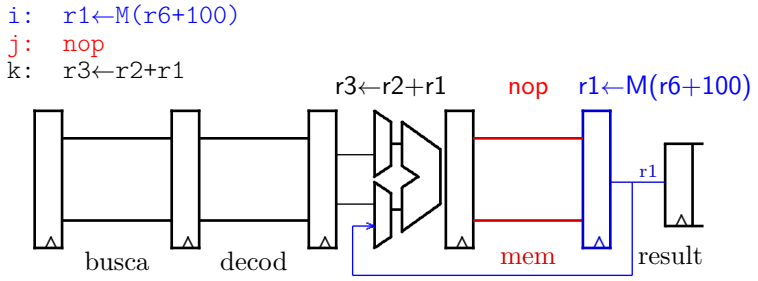


Figura 4.27: Bolha introduzida pelo *load delay-slot*.

Neste mesmo ciclo, a instrução `add r3,r2,r1`, que segue o `lw`, *não pode usar* o valor adiantado do registrador de estágio **exec** porque este valor não é o resultado obtido da leitura da memória. Para evitar o erro decorrente da leitura do valor caduco que está no registrador do segmento **exec**, o `add` deve ser mantido em **exec** por um ciclo, para garantir que o valor no registrador **mem** seja adiantado para a entrada da ULA, como indicado pela seta que adianta `r1`, na Figura 4.27.

O compilador deve inserir um `nop` entre todos os pares de instruções com um *uso do load* para garantir que a instrução dependente do *load* obtenha o valor correto para seu operando. Esta instrução `nop`, ou uma bolha, é necessária para garantir corretude mas é um desperdício de tempo. Se o compilador consegue reordenar o código de forma a preencher o *load delay-slot* com uma instrução útil, então não há desperdício de um ciclo a cada *load*. Evidentemente, a instrução que preenche o *delay-slot* não pode alterar a corretude do programa.

Exemplo 4.7 Os trechos de código abaixo não computam o mesmo resultado. No lado esquerdo, o código não considera o uso do *load*. No lado direito está a versão que executaria corretamente no processador da Figura 4.27.

```
# riscos por resolver
      li   r1, 0
      li   r2, 1024
      la   r3, A
      li   r4, 0
for:  lw   r5, 0(r3)
      add  r4, r4, r5
      addi r3, r3, 4
      addi r1, r1, 4
      bne  r1, r2. for
fim:
```

```
# riscos resolvidos
      li   r1, 0
      li   r2, 1024
      la   r3, A
      li   r4, 0
for:  lw   r5, 0(r3)
      nop
      add  r4, r4, r5
      addi r3, r3, 4
      addi r1, r1, 4
      nop
      nop
      bne  r1, r2. for
fim:
```

Há uma dependência no registrador `r1`, entre o segundo `addi` e o desvio `beq`. Esta dependência é resolvida com a introdução de dois `nop`s.

A resolução dos riscos neste trecho de código custa três ciclos adicionais a cada volta do laço, o que reduz a vazão em 9/6=1,5. É possível reordenar as instruções para eliminar um ou mais `nops`? Se sim, quantos? A reordenação deve manter a corretude do código. ◁

Adiantamento

O circuito completo de adiantamento é mostrado na Figura 4.28. O multiplexador na entrada α da ULA tem suas entradas ligadas aos registradores de saída dos segmentos **decod** – quando não há adiantamento, **exec** – adiantamento com distância um, e **mem** – adiantamento com distância dois. Este multiplexador é controlado pelo sinal fwdA, que é gerado por lógica similar àquela indicada no Exemplo 4.6.

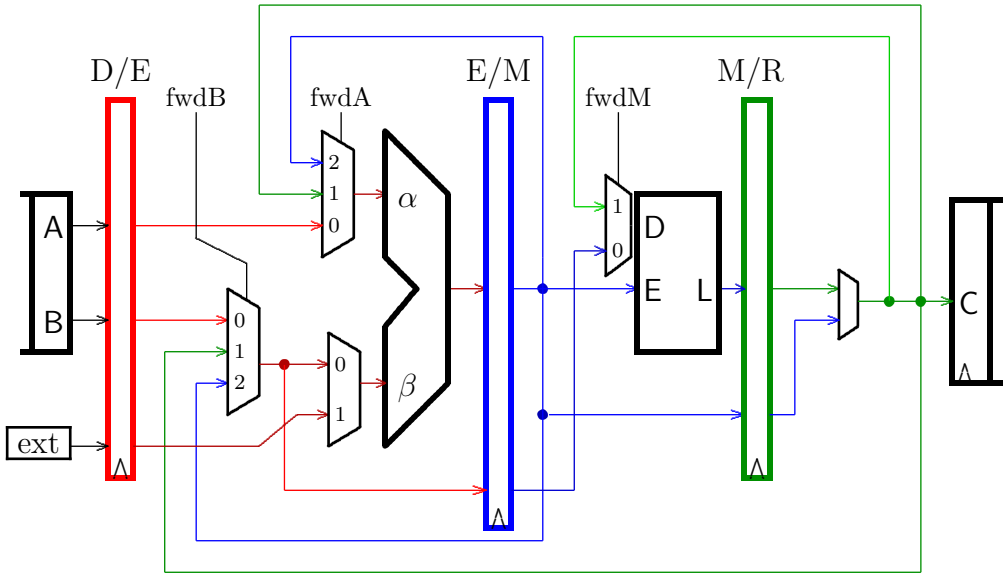


Figura 4.28: Circuito de adiantamento completo.

O multiplexador na entrada β da ULA tem suas entradas ligadas aos registradores de saída dos segmentos **decod** – quando não há adiantamento, **exec** – adiantamento com distância um, e **mem** – adiantamento com distância dois. Este multiplexador é controlado por fwdB.

O multiplexador da entrada de escrita da memória de dados tem uma entrada ligada à saída do registrador do estágio **exec** – quando não há adiantamento, e ao registrador do estágio **result** para adiantar o resultado de um *load* para um *store*. Este multiplexador é controlado por fwdM.

Cada sinal de dados no circuito de adiantamento é de 32 bits, o que significa que o espaço ocupado pela lógica de adiantamento na superfície do circuito integrado é enorme. Os circuitos dos multiplexadores são diminutos quando comparados com a área ocupada pelos barramentos de dados. A lógica para gerar os sinais de controle também é relativamente simples embora empregue sinais provenientes de quatro estágios do processador, o que significa que os sinais de controle podem ser gerados com atrasos significativos. O projeto lógico deve garantir que os ganhos advindos do adiantamento não sejam cancelados por problemas de temporização.

Dependências de controle

As instruções de saltos e desvios causam o seu próprio *delay-slot*. A instrução de desvio só é decodificada, e o destino do desvio determinado, ao final do estágio de decodificação.

Isso significa que a instrução que será executada após o desvio, caso este seja tomado, só será conhecida um ciclo após a determinação do destino. No processador segmentado, a instrução que segue o desvio é buscada antes que o destino do desvio seja conhecido. A Figura 4.29 mostra uma instrução de desvio no estágio de decodificação e a instrução que a segue, um **nop**.

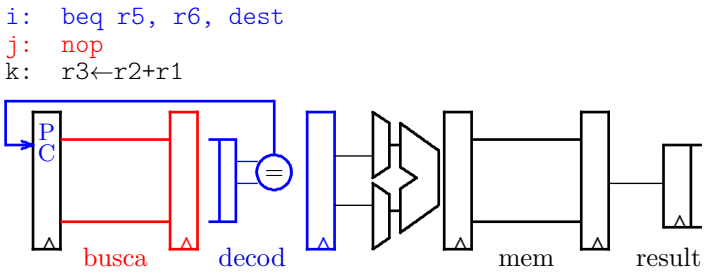


Figura 4.29: Bolha introduzida pelo *branch delay-slot*.

Os projetistas do MIPS transformaram o limão em limonada e decretaram que a instrução que segue um desvio é sempre executada. A tarefa de preencher o *branch delay-slot* com uma instrução útil recai sobre o compilador. Se não há instrução que possa ser movida para o *delay-slot*, então este deve ser preenchido com um **nop**. Retornaremos ao tópico de reordenamento de código na Seção 5.5.2.

Exemplo 4.8 O trecho de código abaixo é uma versão reordenada daquele do Exemplo 4.7, que resolve a dependência de uso do *load* mas não elimina a bolha causada pela dependência de controle.

```
      li   r1, 0
      li   r2, 1024
      la   r3, A
      li   r4, 0

for:  lw   r5, 0(r3)
      addi r1, r1, 4      # resolve uso do load
      add  r4, r4, r5
      addi r3, r3, 4      # resolve dependência em r1
      bne  r1, r2. for
      nop                 # branch delay-slot por preencher
fim:
```

Esta versão reordenada executa com a máxima eficiência possível no processador da Figura 4.29. Este é um caso de "cobertor curto" porque falta uma instrução para preencher o *branch delay-slot*: se a instrução **add** `r4,r4,r5` for movida para após o **beq**, a dependência em `r1` fica exposta, e deve ser resolvida com um **nop**. ◁

Exemplo 4.9 Vejamos uma implementação da instrução BRANCH-EQUAL , que tira proveito do adiantamento dos valores de seus operandos, Os acréscimos ao circuito são mostrados na Figura 4.30 – neste diagrama o banco de registradores *não* está separado em uma seção de leitura e outra de escrita.

A escolha do endereço de destino depende da comparação de dois registradores, e tipicamente, um dos operandos foi computado por uma instrução que precede o `beq`, como mostrado abaixo:

```
addi r1, r1, 4
sub  r2, r4, r5
bne  r1, r2. dest   # depende de r2 e de r1
nop                 # branch delay slot
```

Ao decodificar o `beq`, as dependências são detectadas, e o `beq` é mantido na decodificação até que o `sub` avance para o estágio de acesso à memória, quando então o valor de `r2` é adiantado da saída do registrador de segmento E/M até a entrada do multiplexador controlado por fwdD. O valor de `r1` é obtido da porta A do bloco de registradores.

Além do *branch delay slot*, a dependência nos operandos causa uma bolha. Esta combinação de instruções é comum, e nem sempre o compilador consegue preencher as *duas* bolhas que circundam o desvio, uma da dependência de dados, e uma do *delay slot*. ◁

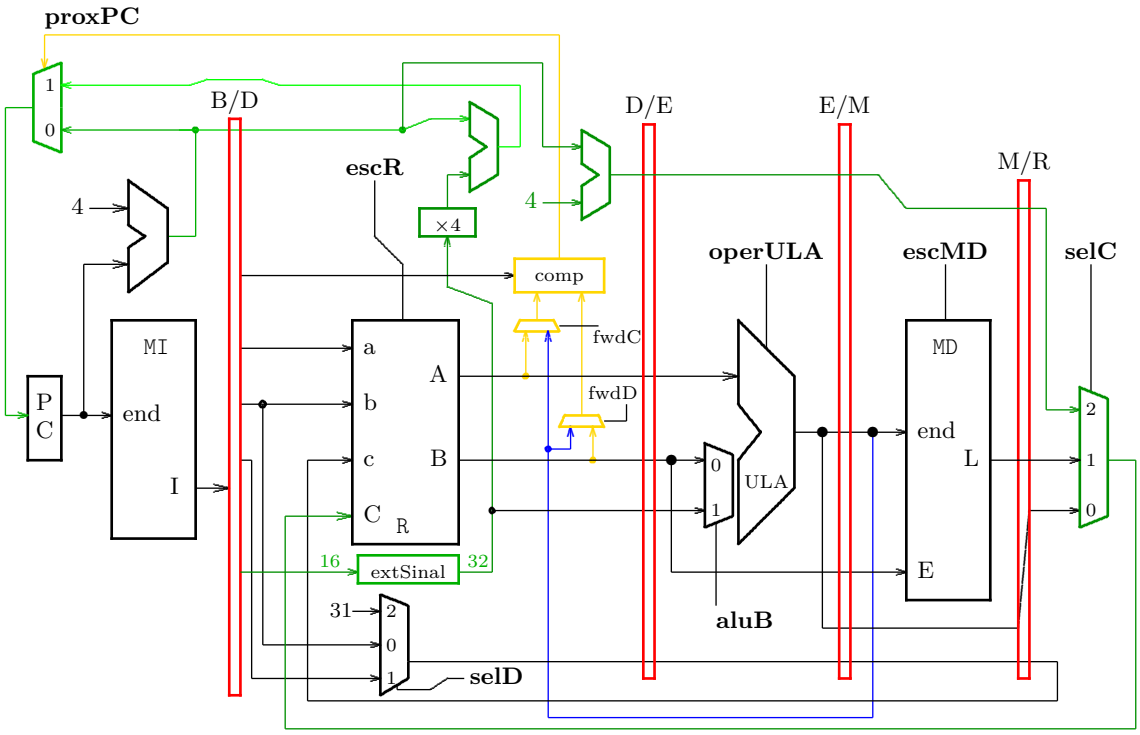


Figura 4.30: Suporte a desvios condicionais com adiantamento.

4.3.3 Exercícios

Ex. 35 Mostre como implementar a instrução JUMP-REGISTER no processador segmentado. Sua resposta deve conter: (i) uma indicação clara de quais modificações são necessárias no circuito de dados do processador; (ii) quais são os sinais de controle ativos em cada estágio; (iii) uma indicação de quais são os caminhos de adiantamento para esta instrução. O formato é R. `jr rs #PC <- R(rs)`

Ex. 36 Repita para a instrução JUMP-AND-LINK-REGISTER, de formato R.

```
jalr rd, rs # R(rd) <- PC+8 , PC <- R(rs)
```

Ex. 37 Com base na sequência abaixo, que é deveras popular, acrescente quaisquer circuitos de adiantamento que possam ser úteis ao processador da Figura 4.30.

```
sub  r2, r4, r5
lw   r1, 0(r5)
bne  r1, r2. dest  # depende de r2 e de r1
nop                # branch delay slot
```

Ex. 38 Escreva as equações para o controle de bloqueios, necessárias para eliminar os riscos de dados que você descobriu ao resolver o Exercício. 37. Veja o Exemplo 4.6.

Ex. 39 Para simplificar a exposição, todos os exemplos de código em *assembly* do Capítulo 3 ignoram os *branch delay-slots* e os *load delay-slots*. Revise todos os exemplos, e reordene as instruções para eliminar quaisquer ciclos desperdiçados.

Ex. 40 Para responder a esta pergunta, explicite quaisquer suposições necessárias. Considere o programa ao lado, para ser executado no processador segmentado mais simples, sem nenhuma forma de adiantamento e nem bloqueios, mas com *branch delay-slot* e *load delay-slot*. (i) Acrescente ao código o que for necessário para garantir a execução correta deste programa. Qual o número total de ciclos necessários para armazenar a redução contida em r5? (ii) Otimize o código para reduzir o número de ciclos. Qual o novo número total de ciclos? Qual o ganho com relação ao item anterior?

```
    la   r20, 0x40000000
    li   r21, 800
    move r5, r0
L:  lw   r10, 0(r20)
    add  r5, r5, r10
    addiu r20, r20, 4
    addiu r21, r21, -4
    bne  r21, r0, L
    nop
    sw   r5, -808(r20)
```

Espaço em branco proposital.

Ex. 41 Indique, com setas da instrução produtora para a(s) instrução(ões) dependente(s), as dependências entre as instruções do trecho de programa abaixo.

```
; int a, *b, i, j, x[N], y[M], z[P];
; a = 16 / y[ -33000 + i ];
la   ry, Y
lui  at, %hi(-33000)     ; -33.000 > 2**15
ori  rc, at, %lo(-33000)
add  r5, ri, rc
sll  r5, r5, 2           ; índice * 4
add  r5, r5, ry          ; r5 <= Y+4*(i-33000)
lw   r5, 0(r5)
li   r16, 16
div  r16, r5
mflo r6                  ; r6 <= quoc
sw   r6, 0(ra)           ; *a <= y[i-33000]

; b = &(x[ y[ z[j]+2 ] ]);
la   rz, Z
la   ry, Y
la   rx, X
sll  r5, rj, 2           ; índice*4
add  r5, r5, rz          ; r5 <= Z+4*j
lw   r6, 0(r5)           ; r6 <= z[j]
addi r6, r6, 2
sll  r6, r6, 2           ; índice*4
add  r6, r6, ry          ; r6 <= Y+4*([z[j]+2])
lw   r7, 0(r6)           ; r7 <= y[ z[j]+2 ]
sll  r7, r7, 2           ; índice*4
add  r7, r7, rx          ; r6 <= X+4*(y[z[j]+2])
sw   r7, 0(rb)           ; *b <= &( x[y[z[j]+2]] )
```

Capítulo 5

Compilação com gcc e Otimização

Instruction tables will have to be made up by mathematicians with computing experience and perhaps a certain puzzle-solving ability. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.

Alan Turing

O *compilador* é o programa que traduz código escrito em uma linguagem de alto nível, tal como C ou Pascal, para um programa que é o seu equivalente lógico na linguagem *assembly* do processador que executará o programa.

Para simplificar a discussão, no que se segue consideramos somente um compilador da linguagem C e que produz código para o processador MIPS. A discussão pode ser facilmente generalizada para outras linguagens imperativas (Pascal, Fortran) e outros processadores. Em particular, o compilador que é discutido aqui é o *GCC*, ou *GNU Compiler Collection* – seu nome original era *GNU C Compiler*.

Estritamente falando, o `gcc` não é um único programa mas uma coleção de ferramentas sofisticadas que efetuam a compilação, montagem e ligação de programas escritos em C. O `gcc` é um programa que executa as ferramentas na ordem apropriada para produzir um executável a partir de um ou mais arquivos com código fonte C, *assembly* e bibliotecas.

Do ponto de vista de funcionalidade, o que se deseja de um compilador pode ser resumido em quatro requisitos: (i) todos os programas corretos com relação à definição de C executam corretamente; (ii) a maioria dos programas compilados executa rapidamente; (iii) a compilação é rápida; e (iv) o compilador oferece suporte a depuração dos programas na linguagem de alto nível. O `gcc` provê essas funcionalidades e outras.

5.1 Tempo de Execução

São duas as razões para otimizar o código gerado pelo compilador: melhorar o tempo de execução e/ou reduzir o tamanho do executável. A primeira razão é em geral a mais popular enquanto que a segunda pode ser um objetivo importante em sistemas embarcados. Um executável menor pode ter melhor desempenho porque faz melhor uso de memória cache e da memória virtual. Mais detalhes sobre estes tópicos são estudados na disciplina de Arquitetura de Computadores, e mais adiante nesta disciplina. O material nesta seção é baseado nos textos clássicos de Hennessy &Patterson [**?**, **?**, **?**, **?**].

O tempo de execução é reduzido quando o programa executa menos instruções – este é um

ponto importante: *o tempo de execução é determinado pelas instruções que são executadas e não pelo tamanho do código fonte/assembly de um programa.* As instruções que são executadas são chamadas de *instruções dinâmicas*, enquanto que as instruções no arquivo executável são *instruções estáticas.*

O *tempo de execução* de um programa é o tempo medido com um cronômetro, desde o instante em que a tecla ENTER é pressionada, até que o resultado (completo) seja produzido. Para um programa P, executando num processador com período de relógio τ, o tempo de execução T é dado pela Equação 5.1.

$$T_P = \tau \times I \times \gamma \tag{5.1}$$

sendo I o número de instruções executadas. Supondo que o processador executa cada instrução em um ciclo de relógio ($\gamma = 1$), então o tempo de execução é o produto do número de instruções dinâmicas pela duração de cada instrução.

T_P é 'tempo' e portanto uma grandeza com unidade de segundos [s], a duração de um ciclo do relógio τ tem unidade segundos por ciclo [s/c], o número de instruções I tem unidade 'instruções' [i], e o número médio de ciclos para executar uma instrução γ tem unidade "ciclos por instrução" [c/i].

A análise dimensional da Equação 5.1, que é equivalente à verificação dos tipos das variáveis da equação, mostra que a dimensão de T_P, no lado esquerdo da igualdade, tem a mesma dimensão que o lado direito da igualdade:

$$s = \frac{s}{\cancel{c}} \times i \times \frac{\cancel{c}}{i} = s \times \cancel{i} \times \frac{1}{\cancel{i}} = s.$$

Como vimos na Seção 4.3, dependências de dados e de controle podem fazer com que a execução de uma instrução demore mais do que um ciclo. Por isso, emprega-se o *número médio de ciclos por instrução*,, ou o *CPI*.

O número de ciclos por instrução depende (i) do programa – algoritmo e sua implementação; (ii) do compilador – mais ou menos otimização; (iii) e do processador – implementação de ciclo longo, ou segmentada, ou superescalar. Para um programa P_1, o CPI pode variar em função do processador em que é executado, do compilador, e do nível de otimização do código. Dois programas, P_1 e P_2, executando num mesmo processador e compilados com o mesmo compilador, podem ter CPIs distintos em função da frequência e distribuição das instruções e das dependências de dados e de controle.

Num programa 'típico', de todas as instruções dinâmicas, as de lógica e aritmética perfazem 25-30% do total, leituras da memória (`lw`) contribuem com 20-25% do total de instruções, escritas em memória (`sw`) contribuem com 10-15%, desvios condicionais (`beq`) com 20-25%, desvios incondicionais (`j`, `jr`) e chamadas de funções (`jal`) contribuem com 1-5%.

Considerando-se o CPI (γ na Equação 5.1), o tempo de execução de um programa é dado pela Equação 5.2.

$$T_P = \tau \times I \times \text{CPI}\,. \tag{5.2}$$

Para minimizar o tempo de execução, é necessário reduzir o número de instruções dinâmicas, e também o número de bolhas causadas por dependências de dados e de controle. Este é o objetivo das transformações no código fonte e no código *assembly*, discutidas no que segue.

Exemplo 5.1 Considere o programa P_1 que é executado num processador MICO-1, com período de relógio de 1ns. A execução completa após $2,2 \times 10^9$ instruções. A distribuição de

instruções é mostrada na Tabela 5.1. O número médio de ciclos por instrução é

$$\text{CPI}_{\text{MICO-1}} = 0,30 \times 1,10 + 0,25 \times 1,45 + 0,15 \times 1,00 + 0,25 \times 1,60 + 0,05 \times 1,40 = 1,3125$$

O tempo de execução de P_1 no MICO-1 é

$$T(P_1, \text{MICO-1}) = 1 \cdot 10^{-9} \times 2,2 \cdot 10^9 \times 1,3125 = 2,89s\,.$$

O modelo MICO-2 emprega otimizações no circuito para aumentar a frequência do relógio para 750ps, ao custo da eliminação de alguns dos circuitos de adiantamento, o piora seu CPI:

$$\text{CPI}_{\text{MICO-2}} = 0,30 \times 1,15 + 0,25 \times 1,50 + 0,15 \times 1,05 + 0,25 \times 1,75 + 0,05 \times 1,45 = 1,3875$$

O tempo de execução de P_1 no MICO-2 é

$$T(P_1, \text{MICO-2}) = 0,750 \cdot 10^{-9} \times 2,2 \cdot 10^9 \times 1,3875 = 2,29s\,.$$

No MICO-2, o tempo de execução de P_1 é $2,29/2,89 = 0,79$ do tempo no MICO-1, um ganho de aproximadamente 20% obtido com uma redução no período do relógio de 25%. Para usuários do programa P_1, a troca é vantajosa se o MICO-2 não custar 20% a mais que o MICO-1. ◁

Tabela 5.1: Distribuição de instruções e CPI do programa P_1.

CLASSE	ALU	*loads*	*stores*	desvios	saltos
FREQUÊNCIA	0,30	0,25	0,15	0,25	0,05
MICO-1	1,10	1,45	1,00	1,60	1,40
MICO-2	1,15	1,50	1,05	1,75	1,45
MICO-2 `-O4`	1,15	1,45	1,05	1,50	1,35

Exemplo 5.2 O time de compiladores da MICO introduziu uma série de melhorias no compilador C, com um nível de otimização batizado de `-O4`.

Com estas otimizações, o número de desvios e de saltos executados aumenta em 10%, mas se obtém ganhos no CPI. O esforço dispendido compensa?

$$\text{CPI}_{\texttt{-O4}} = 0,30 \times 1,15 + 0,25 \times 1,45 + 0,15 \times 1,05 + 0,25 \times 1,50 + 0,05 \times 1,35 = 1,3075$$

O tempo de execução de P_1, compilado com `-O4`, é

$$T(P_1, \text{MICO-2 } \texttt{-O4}) = 0,750 \cdot 10^{-9} \times (2,2 \times 0.7 + 2.2 \times 1,1 \times (0,25 + 0,05)) \cdot 10^9 \times 1,3075 = 2,22s\,.$$

Com `-O4`, o tempo de execução de P_1 é $2,22/2,29 = 0,97$ do tempo sem a otimização. Para P_1, possivelmente o ganho de 3% não compensa o esforço do time de compiladores. ◁

5.2 Blocos Básicos

Uma vez que circuitos de adiantamento resolvem a grande maioria das dependências de dados, são as dependências de controle que causam as maiores perdas de desempenho porque não há um modo eficiente de resolvê-las. Os piores CPIs são associados aos desvios condicionais porque, geralmente estes são dependentes da leitura de uma variável em memória, ou então do cômputo do valor de uma variável de controle de laço.

Para simplificar a discussão, nesta seção, usaremos o termo 'desvio' para indicar todas as instruções que provocam mudança no fluxo de execução, que são as instruções de desvio (`beq`) e as de salto (`j`, `jal`).

Um *bloco básico* é uma sequência de instruções que não é interrompida por um desvio. O bloco inicia na primeira instrução *após* um desvio, ou na primeira instrução da sequência que é o destino de um desvio. O bloco termina numa instrução de desvio, ou no destino de um desvio. Blocos básicos são curtos e tipicamente, têm de 3 a 6 instruções.

Exemplo 5.3 Vejamos os blocos básicos na versão *assembly* do `strcopy`, no Programa 3.12, da página 56. As instruções no Programa 5.1 foram reordenadas para eliminar todas as otimizações, para explicitar os limites dos blocos básicos. ◁

Programa 5.1: Blocos básicos na versão *assembly* de `strcopy()`.

```
1  strcopy:
2          move    a2,zero       # bloco 1, destino de JAL
3          lbu     v0,0(a1)      #
4          sb      v0,0(a0)      # fim do bloco 1
5          beq     v0,zero,.L6       ; dependência em v0
6          nop
7
8  .L4:
9          addiu   a2,a2,1       # bloco 2, após BEQ, destino de BNE
10         addu    v0,a0,a2      #
11         addu    v1,a1,a2      #
12         lbu     v1,0(v1)      #
13         sb      v1,0(v0)      # fim do bloco 2
14         bne     v1,zero,.L4       ; dependência em v1
15         nop
16
17 .L6:
18         move    v0,a2         # bloco 3, após BNE, antes do JR
19         jr      ra
20         nop
```

Como veremos em breve, há pouco o que fazer em blocos básicos pequenos, do ponto de vista de otimização do código para eliminar bolhas. O fluxo da computação, como definido pelo programador, geralmente envolve dependências de dados que não podem ser eliminadas sem comprometer a corretude do programa. No Programa 5.1, as bolhas causadas pelas dependências nos *loads* nas linhas 3 a 5, e 12 a 14, não podem ser resolvidas facilmente.

5.3 Interface de entrada – linguagem C

O *código fonte* em C por ser compilado deve estar num arquivo texto com sufixo `.c`. O arquivo pode incluir um ou mais arquivos com cabeçalhos de funções (*header files*) e definições de constantes e/ou macros. Arquivos de cabeçalho tem sufixo `.h`.

Além de código fonte, programas empregam funções de biblioteca de sistema tais como `printf()` e `open()`. O compilador examina o código fonte e gera uma lista com as funções de biblioteca que devem ser ligadas ao programa executável. As bibliotecas consistem de código previamente compilado e no formato apropriado para ligação com a versão objeto do programa que delas faz uso. Detalhes sobre bibliotecas e o processo de ligação serão estudados no Capítulo 10 e seguintes.

5.4 Argumentos para gcc

Esta seção é baseada em [**?**] e a leitura daquele texto é enfaticamente recomendada.

Suponha que você editou seu programa em um arquivo `fonte.c`. O comando abaixo produz um executável `a.out`:

```
prompt: gcc -Wall fonte.c
prompt: ./a.out              # seu programa é executado
```

A opção `-Wall` força o compilador a emitir todos os avisos sobre o código que está sendo compilado. Esta opção é excepcionalmente útil para a resolução de problemas durante um ciclo de edição–compilação.

Se desejamos salvar o executável em um arquivo chamado `prog`, a opção `-o` nos permite fazê-lo. Esta opção é normalmente a última das opções. Sem ela a saída é sempre gravada em `a.out`.

```
prompt: gcc -Wall fonte.c -o prog
prompt: ./prog               # seu programa é executado
```

Se desejamos traduzir o programa em C para um *arquivo objeto*, e salvar o resultado em `prog.o`, a opção `-c` deve ser usada. O sufixo `.o` denota arquivo com código objeto, que é um arquivo binário mas não é um executável porque, possivelmente, deve ser ligado a outros arquivos objeto e a uma ou mais bibliotecas. Se a opção `-o` não for usada, o objeto será gravado em `fonte.o`.

```
prompt: gcc -Wall -c fonte.c -o prog.o
```

Se desejamos ligar os arquivos objeto `prog.o`, `grog.o` e `clog.o` para produzir o executável `prog`, o comando abaixo invoca o ligador para produzir o executável. Note que a opção `-Wall` não é necessária neste caso porque os arquivos objeto foram compilados anteriormente.

```
prompt: gcc prog.o grog.o clog.o -o prog
```

A ordem dos objetos é importante! `prog.o` pode usar funções definidas em `grog.o`, e `clog.o` pode usar funções definidas em `grog.o`. Se, ao percorrer os arquivos objeto, o uso das funções não for encontrado antes de suas definições, a compilação resultará em erro.

Se nosso programa emprega funções da biblioteca de matemática `libm.a`, então esta biblioteca deve ser indicada na linha de comando.

```
prompt: gcc prog.o -lm -o prog
```

Note a forma abreviada para nominar a biblioteca: `-lm` indica que a biblioteca `libm.a` deve ser usada, e esta pode ser encontrada num dos diretórios buscados normalmente pelo compilador (`/lib` e `/usr/lib`). O caminho completo para a biblioteca também pode ser empregado. Na minha instalação o caminho completo para `libm.a` é apontado pelo comando `locate`.

```
prompt: locate libm.a
/usr/lib/x86_64-linux-gnu/libm.a
prompt: gcc prog.o /usr/lib/x86_64-linux-gnu/libm.a -o prog
```

O caminho completo é útil se você necessita ligar seu programa a uma biblioteca que não é uma das bibliotecas oficiais.

Os arquivos com cabeçalhos de funções das "bibliotecas oficiais" são buscados num conjunto de "diretórios oficiais" (`/usr/include` e `/usr/local/include`). Suponha que você

trabalha num projeto grande e que os arquivos de cabeçalho sejam armazenados todos num só diretório. O caminho para este diretório pode ser indicado ao compilador com a opção `-I`. O primeiro caminho a ser procurado é aquele apontado pelo `-I`, e então os caminhos dos diretórios oficiais são percorridos. Mais de uma opção `-I` pode ser usada.

```
prompt: gcc -Wall -c prog.c -I /path/to/my/headers
```

A opção `-L` pode ser usada para indicar um diretório com "bibliotecas não oficiais", de forma similar à da opção `-I`.

Suponha que você instalou bibliotecas opcionais no seu sistema em `/my`. O comando abaixo informa ao compilador os locais dos cabeçalhos e da biblioteca `libprecious.a`.

```
prompt: gcc -Wall -c prog.c -I/my/include -L/my/lib -lprecious -o prog
```

A opção `-v` (*verbose*) pode ser usada para mostrar na tela os passos intermediários da compilação. Além de informativa, esta opção ajuda a detectar problemas com a instalação do compilador ou bibliotecas.

A opção `-D` (*define*) pode ser usada para definir uma macro com o valor 1, ou para atribuir um valor a uma macro. `-D` é processada antes da inclusão de arquivos com `#include`. Dos dois comandos abaixo, o primeiro comando equivale a incluir "`#define cMIPS 1`" no topo do arquivo com código fonte, enquanto que o segundo comando atribui o valor `YES` ao símbolo `SIMUL`.

```
prompt: gcc -Wall -c prog.c -DcMIPS
```

```
prompt: gcc -Wall -c prog.c -DSIMUL=YES
```

A definição de uma macro pode ser cancelada com a opção `-U` (*undefine*). No nosso exemplo, seu resultado equivale a "`#undef cMIPS`".

```
prompt: gcc -Wall -c prog.c -UcMIPS
```

A Tabela 5.2 mostra as opções mais usadas com `gcc`. Detalhes sobre a opção `-g` e otimização são apresentados na Seção 5.5.3.

5.5 Otimização

Esta seção é baseada em [**?**] e a leitura daquele texto é enfaticamente recomendada.

5.5.1 Otimização do código fonte

Vejamos algumas das técnicas para otimizar um programa, através de alterações no código fonte, para possibilitar ao compilador gerar código mais eficiente – por eficiente entenda-se aqui *que executa em menos tempo*, ou *que executa menos instruções* e portanto "quase certamente" executa em menos tempo.

Eliminação de subexpressões

Frequentemente escrevemos código com repetições, que aumentam a legibilidade, mas pioram o desempenho porque instruções desnecessárias são executadas. Considere o trecho de código abaixo, que computa a Fórmula de Bhaskara.

```
x1 = (-b + sqrt(b*b - 4*a*c)) / 2*a;
x2 = (-b - sqrt(b*b - 4*a*c)) / 2*a;
```

Tabela 5.2: Opções comuns do `gcc`.

OPÇÃO	FUNÇÃO
`-Wall`	avisa sobre possíveis erros ou situações marginais
`-o arq`	salva resultado em `arq`
`-c`	somente compila código fonte, produz `.o`
`-S`	somente traduz para *assembly*, produz `.s`
`-lX`	liga com biblioteca `libX.a`
`-I path`	acrescenta `path` aos caminhos para arquivos com cabeçalhos
`-L path`	acrescenta `path` aos caminhos para arquivos com bibliotecas
`-v`	produz saída verbosa
`-Dmacro`	define `macro=1`
`-Dmacro=VAL`	define `macro=VAL`
`-Umacro`	cancela a definição de `macro`
`-g`	acrescenta informação de depuração ao objeto/executável
`-O0`	compila sem otimizar
`-O1`	compila com algumas otimizações
`-O2`	compila com otimizações de `-O1` e mais algumas
`-O3`	compila com otimizações de `-O2` e mais algumas
`-Os`	compila com otimizações para reduzir tamanho do executável
`-E`	mostra a expansão de todas as macros e os arquivos incluídos

A repetição dos termos `(2*a)` e `sqrt(b*b - 4*a*c)` pode melhorar a legibilidade do código, mas é um desperdício que o compilador pode facilmente identificar e eliminar[1]. O código fonte otimizado pelo compilador seria algo como:

```
float t1 = sqrt(b*b - 4*a*c);
float t2 = 2*a;
x1 = (-b + t1) / t2;
x2 = (-b - t1) / t2;
```

As variáveis temporárias são instanciadas pelo compilador, e possivelmente são alocadas em registradores quando o código é gerado e portanto não aumentam o tamanho do executável e nem o espaço alocado para a seção de dados.

Outra fonte de repetições é no acesso à estruturas de dados complexas. Frequentemente o compilador elimina cálculos de endereços repetidos, como no trecho de código abaixo:

```
x[i+1] += x[i]/2.0 + x[i-1]/2.0;
```

Supondo que o vetor seja de `float`s, os três elementos são alocados em endereços que distam entre si de 4 bytes e portanto o endereço de um deles deve ser calculado e os outros dois estão a uma distância conhecida e constante daquele endereço.

Expansão de funções

Se o corpo de uma função consiste de umas poucas linhas de código e esta função não invoca nenhuma outra função – é portanto uma função folha – então o número de instruções dinâmicas é menor se o código da função for expandido diretamente no local da chamada[2]. Considere o trecho abaixo.

[1] *Common subexpression elimination.*
[2] *Function inlining.*

```
int square(int i) {
  return(i*i);
}
...
for(i=0; i<N; i++)
  x[i] = square(i);
```

O código para a invocação desta função implica em armazenar o argumento no registrador `a0` e na execução das instruções de chamada e retorno de função, **`jal`** e **`jr`**. Sem otimização, `3*N` instruções adicionais são executadas para invocar a função.

Se o código da função for expandido no local da sua chamada, estas instruções dinâmicas não são executadas. Além disso, o mesmo registrador que armazena `i` pode ser usado como argumento para a instrução de multiplicação.

```
for(i=0; i<N; i++)
  x[i] = (i*i);
```

Evidentemente há um compromisso entre o custo de invocar a função e o custo de expandi-la no corpo do programa, e este equilíbrio depende do tamanho do código gerado e do ganho de desempenho que se obtém pela redução no número de instruções dinâmicas. A expansão de funções geralmente aumenta o tamanho de código se a expansão ocorrer em mais de um local no código fonte.

Como um exercício, expanda a função no trecho de código abaixo. Note que um dos argumentos na chamada é uma constante, e que variáveis temporárias são necessárias.

```
int m3(int a, int b, int c) {
  int p,q;
  p = 23*a - b*b;
  q = 23*a - c*c;
  return( p + 9*b - 2*q );
}

int main(...) {
   ...
   x = m3(r, s, 12);
  ...
}
```

Desenrolar laços

Outra otimização importante é desenrolar laços[3]. O corpo do laço é replicado de forma a que o número de repetições seja reduzido pela metade. O tamanho do código aumenta mas o número de repetições diminui. O ganho de desempenho decorre da redução do número de instruções de controle que são executadas porque o teste ao final do laço é executado menos vezes. Esta otimização troca tempo de execução por mais espaço na memória de instruções.

Considere o laço abaixo, que executa um número fixo de voltas:

```
for (i=0; i<1024; i+=1)        // original
   A[i] = 35.0 * B[i] + C[i];
```

Na versão otimizada, o corpo do laço é replicado, os índices são ajustados no comando replicado e o incremento passa a ser de dois em dois. O número de voltas cai de 1024 para

[3] *Loop unrolling.*

512 porque o corpo do laço efetua o dobro de trabalho do que a versão original.

```
for (i=0; i < 1024; i+=2) {   // desenrolado
   A[i] = 35.0 * B[i] + C[i];
   A[i+1] = 35.0 * B[i+1] + C[i+1];
}
```

Além da redução no número de voltas, também há ganho no cálculo dos endereços dos vetores porque os elementos consecutivos podem ser acessados sem que os endereços sejam recalculados.

Se o limite do laço for uma variável, ou o número de voltas for ímpar, então deve-se acrescentar código para garantir que o número correto de iterações seja efetuado. Como um exercício, desenrole o laço original considerando que o limite é a variável `lim`, ao invés de um número fixo e par.

Desenrolar laços tem outro efeito importante que é aumentar as oportunidades para reordenar o código e reduzir os efeitos das dependências entre instruções. Voltaremos ao assunto na Seção 5.5.2.

Propagação de constantes

Outra otimização que pode ser efetuada em laços é a propagação do valor de variáveis cujos conteúdos não alterados no corpo do laço[4]. No laço abaixo, a raiz quadrada é recomputada a cada volta do laço, quando esta constante pode ser computada em tempo de compilação, ao invés de em tempo de execução.

```
x = 35.0;
for (i=0; i<1024; i++)
   A[i] = sqrt(x) * B[i] + C[i];
```

O código pode ser reescrito, removendo-se a invocação de `sqrt()` do corpo do laço, e propagando o valor constante através da variável que não varia `t_SQRT`.

```
x = 35.0;
t_SQRT = sqrt(35.0);              // constante
for (i=0; i<1024; i++)
   A[i] = t_SQRT * B[i] + C[i];
```

Considere o exemplo abaixo, que pode parecer simples demais para a aplicação da técnica mas que todavia é interessante. Note que tanto `a` quanto `b` são constantes e portanto seus valores podem ser mantidos em registradores pelo compilador.

```
a = CONST;
if (k == 0)
   b = 1;
else
   b = 2;
for (s=1, i=0; i<lim; i++)
  s = s * a+b;
```

Espaço em branco proposital.

[4] *Constant propagation.*

Suponha que o compilador efetua a otimização mostrada abaixo. Há ganho de desempenho na replicação dos laços nas duas cláusulas do **if**? O código gerado evidentemente fica maior; pode o desempenho ser melhor? Os laços podem ser desenrolados?

```
a = CONST;
if (k == 0) {
   b = 1;
   for (s=1,i=0; i<lim; i++)
      s = s * CONST+1;
} else {
   b = 2;
   for (s=1,i=0; i<lim; i++)
      s = s * CONST+2;
}
```

Exemplo 5.4 Considere o Programa 5.2. O que se pode fazer para otimizá-lo?

(i) A constante '`16*a`' pode ser re-escrita como '`16*32=512`'; (ii) o laço pode ser desenrolado porque não há dependências entre duas iterações consecutivas. Isso reduz o número de voltas – de 1024 para 512 – e portanto o número de instruções de controle executadas, bem como o número de cálculos de endereços. O código fonte re-escrito é mostrado no Programa 5.3. ◁

Programa 5.2: Laço por otimizar.

```
1 #define N 1024
2 int a, X[N], A[N], B[N];
3 ...
4 a = 32;
5 for(i=0; i < N; i++)
6    X[i] = (int)(A[i] * B[i] + 16*a);
```

Programa 5.3: Laço com o código fonte otimizado.

```
1 for(i=0; i < N; i = i + 2) {
2    X[i] = (int)(A[i] * B[i] + 512);
3    X[i+1] = (int)(A[i+1] * B[i+1] + 512);
4 }
```

5.5.2 Otimização do código *assembly*

O conjunto de instruções que estudamos até agora é o MIPS32r2, definido em [**?**]. A implementação deste conjunto de instruções como um *pipeline* de cinco estágios introduz dois artefatos no conjunto de instruções, que são *load delay slots* e o *branch delay slots*, descritos na Seção 4.3.2.

Uso do *load*

No primeiro doa artefatos da segmentação, *load delay slot*, o uso de um valor lido da memória só pode ocorrer *um ciclo* após o acesso à memória. Isso ocorre porque o valor retornado pela memória só fica estável imediatamente antes da borda do relógio. Este valor não pode ser adiantado para a entrada da ULA porque não há tempo hábil para usá-lo com a segurança de que os sinais na saída da ULA ficariam estáveis e corretos antes da borda do relógio – para tanto, o ciclo teria que ser elongado em algo como 100% para

garantir que a duração de um ciclo supere o tempo de acesso à memória mais o tempo de propagação da ULA. Péssima ideia!

O exemplo abaixo mostra, à esquerda, um trecho de código com um uso do *load* e o **nop** que separa o *load* e a instrução dependente. No lado direito, o **xor** foi movido para o *delay slot*, tomando o lugar do **nop**. Isso só é possível porque o resultado do **xor** não afeta, e nem é afetado, pelas instruções **lw** e **add**.

```
# sem reordenar                    # reordenado
xor r5, r6, r7                     lw  r1, 100(r6)
lw  r1, 100(r6)                    xor r5, r6, r7 # delay slot
nop             # delay slot       add r3, r2, r1 # depende de r1
add r3, r2, r1 # depende de r1     ...
```

Instruções que leem da memória perfazem algo como 15-25% de todas as instruções executadas, e em geral, o valor lido da memória é usado em seguida para computar algum outro valor. Portanto, *load delay slots* podem ter um efeito nefasto no desempenho – uma bolha a cada 4 a 6 instruções. Felizmente, compiladores conseguem preencher cerca de metade destes *delay slots* com instruções úteis.

Dependências de controle

Uma instrução de salto ou desvio só é decodificada, e o destino do desvio determinado, ao final do estágio de decodificação. A instrução que será executada após o desvio, se tomado, somente será conhecida no ciclo após a determinação do destino. Independentemente de o desvio ser tomado, a instrução imediatamente após o desvio é buscada, decodificada e executada.

O reordenamento de código também ajuda aqui. Se o *branch delay slot* for preenchido com uma instrução que efetua trabalho útil, então não há prejuízo. A reordenação do código é um pouco mais complicada no caso de desvios: o caso simples é aquele em que a instrução de antes do desvio pode ser movida para o *delay slot*, como mostra o exemplo abaixo. Neste caso, não há dependências entre o **xor** e o **beq** e portanto o **xor** pode ser deslocado para o *delay slot*, ganhando-se um ciclo a cada vez que este trecho de código for executado.

```
# sem reordenar                    # reordenado
xor r5, r6, r7                     beq r1, r2, dest
beq r1, r2, dest                   xor r5, r6, r7  # delay slot
nop             # delay slot       add r8, r8, r9  # nao-tomado
add r8, r8, r9  # nao-tomado       ...
...                                dest:
dest:                              sub r8, r8, r9  # tomado
sub r8, r8, r9  # tomado          ...
```

Os casos menos simples são aqueles em que não há uma instrução independente *antes* do desvio, o que é o caso se o **xor** produz o valor que é testado no desvio. Se possível, tenta-se mover a instrução *após* o **beq** para o *delay slot*. A complicação decorre da dependência de controle: a instrução movida não pode alterar a corretude do programa. Os próximos trechos de código mostram os dois casos aplicados ao trecho acima, um no qual a instrução que segue o desvio (desvio não tomado) é movida para o *delay slot* (esq.), e o outro em que a instrução do destino do desvio é movida (dir.).

```
# move instrução seguinte
xor r2, r6, r7
beq r1, r2, dest # dep em r2
add r8, r8, r9   # delay slot
...
...
dest:
sub r8, r8, r9   # tomado
```

```
# move instrução do destino
xor r2, r6, r7
beq r1, r2, dest # dep em r2
sub r8, r8, r9   # delay slot
add r8, r8, r9   # nao-tomado
...
dest:
...              # tomado
```

A reordenação deste desvio é capciosa porque a instrução no *delay slot* será executada *sempre* e isso pode alterar o resultado do programa. Nestes exemplos, pode ser necessário acrescentar, no outro lado do desvio, uma instrução que desfaça o efeito daquela que foi movida para o *delay slot*. Neste caso, ganha-se tempo somente na execução da sequência em que não é necessário desfazer o efeito da reordenação.

Se a direção do desvio é previsível em tempo de compilação, em tese seria possível fazer a reordenação mais proveitosa. Desvios previsíveis são aqueles que testam a condição de final de laço, por exemplo.

Desvios são mais frequentes do que *loads* e portanto seu efeito no desempenho é um tanto mais deletério. Desvios são de 20 a 30% de todas as instruções executadas, o que pode significar uma bolha a cada 3-5 instruções. Como não é possível ao compilador preencher todas os *branch delay slots*, foram desenvolvidas técnicas sofisticadas de *hardware* para prever os desvios – se tomado/não tomado – bem como para lembrar do endereço de destino na última execução de um desvio. Mais detalhes em Arquitetura de Computadores.

Escalonamento de instruções

A reordenação de instruções é uma técnica bastante eficaz para minimizar os prejuízos causados pelos *delay slots*. A reordenação também é chamada de *escalonamento de instruções*[5]. Considere o laço abaixo, já desenrolado, e sua tradução parcial para *assembly*.

```
for (i=0; i<1024; i+=2) {   // desenrolado
   A[i]   = B[i]   + C[i];
   A[i+1] = B[i+1] + C[i+1];
}
```

As instruções da "segunda volta" são mostradas em maiúsculas e os números dos registradores da segunda volta foram alterados para evitar o reuso incorreto. No lado esquerdo está o código sem escalonamento, e no lado direito está a versão escalonada. A reordenação afasta as instruções dependentes (`add`) das instruções produtoras (`lw`) de forma a preencher os *load delay slots* com instruções "da outra volta do laço". O *branch delay slot* causado pelo salto seria preenchido com o incremento em `rC` com ou sem desenrolar o laço.

Espaço em branco proposital.

[5] *Instruction scheduling*.

```
# sem reordenar
li  ri,0
lasso:
slt t0, ri, 1024
beq t0, r0, fim
nop
lw  t0, 0(rB)
lw  t1, 0(rC)
nop             # delay slot t1
add t1, t1, t0
sw  t1, 0(rA)
LW  t2, 4(rB)
LW  t3, 4(rC)
NOP             # delay slot t3
ADD t3, t3, t2
SW  t3, 4(rA)
addi ri, ri, 2
addi rA, rA, 4
addi rB, rB, 4
j   lasso
addi rC, rC, 4 # delay slot jump
```

```
# reordenado
li  ri,0
lasso:
slt t0, ri, 1024
beq t0, r0, fim
lw  t0, 0(rB)  # sem bolha
lw  t1, 0(rC)
LW  t2, 4(rB)
LW  t3, 4(rC)
add t1, t1, t0 # sem bolha
ADD t3, t3, t2 # sem bolha
sw  t1, 0(rA)
SW  t3, 4(rA)
addi ri, ri, 2
addi rA, rA, 8
addi rB, rB, 8
j   lasso
addi rC, rC, 8 # sem bolha
```

Se este laço não fosse desenrolado, é possível que um ou mais `nop`s fossem necessários para garantir a corretude. Ao desenrolar, aumenta o número de instruções independentes no corpo do laço que podem ser reordenadas – as instruções de cada volta desenrolada são independentes daquelas das outras voltas somente se o compilador alocar registradores distintos para cada volta. O ganho em desempenho é potencialmente grande: (i) o *overhead* de controle diminui, (ii) o reescalonamento fica mais fácil porque há mais instruções independentes, e (iii) o cálculo dos endereços pode ser otimizado.

No exemplo acima, uma volta do laço sem desenrolar consome 13 ciclos, se o *jump delay slot* for preenchido com o incremento do `rC`, perfazendo $13 \times 1024 = 13.312$ ciclos. No laço desenrolado sem reescalonar (esq.), são $18 \times 512 = 9.216$ ciclos – o laço sem desenrolar usa $13.312/9.216 = 1,44$ mais ciclos do que o laço desenrolado e o ganho advém das instruções de controle e cálculo de endereços que não são executadas. O laço desenrolado e reescalonado consome $15 \times 512 = 7.680$ ciclos, e o laço original usa $13.212/7.680 = 1,72$ vezes mais ciclos do que a versão otimizada. Os ganhos de desempenho com estas otimizações podem ser consideráveis.

Em processadores mais sofisticados que o nosso *pipeline* de cinco estágios, empregam-se técnicas de escalonamento em *hardware* para tirar proveito do paralelismo que existe entre as instruções. Este paralelismo é limitado pelo reduzido tamanho dos *bloco básicos*, que são trechos de código entre desvios. Um bloco básico é uma sequência (curta) de instruções entre o destino de um desvio e o desvio no seu final. Tipicamente, blocos básicos tem de 3 a 6 instruções – lembre que desvios são algo como 20–30% de todas as instruções executadas. Em geral, poucas são as instruções sem dependências num bloco básico.

Exemplo 5.5 Reveja os Programas 5.2 e 5.3. Os Programas 5.4 e 5.5 são as traduções para *assembly* otimizado daqueles dois programas. Alguns dos registradores são indicados como as variáveis que representam: `ra` aponta para o vetor `A[]`.

O laço sem otimizar executa em $5 + 1024 \times 15 = 15.365$ ciclos, supondo que `X[]`, `A[]` e `B[]` são alocados em endereços contíguos – um `lui` e três `ori` para computar os endereços dos vetores, que os acessos à memória completam em 1 ciclo, e que o processador possui todos os circuitos de adiantamento e *branch delay slots*. Note que a constante foi movida para fora do laço.

O laço otimizado executa em $5 + 512 \times 21 = 10.757$ ciclos. O laço otimizado é 1,43 vezes mais

rápido do que o original.

Se o conteúdo do registrador `t1` não é relevante após o laço, o **`nop`** após o **`beq`** pode ser eliminado, o que reduz o número de ciclos para executar os dois laços. ◁

Programa 5.4: Tradução para *assembly* do Programa 5.2.

```
   li   ri, 0
   la   rx, X
   la   ra, A
   la   rb, B
f: slti t0, ri, N   # i < N
   beq  t0, r0, e
   nop
   sll  t1, ri, 2    # i*4
   add  t2, t1, ra   # t2 <- A + i*4
   add  t3, t1, rb   # t3 <- B + i*4
   lw   t4, 0(t2)    # t4 <- M[ A + i*4 ]
   lw   t5, 0(t3)    # t5 <- M[ B + i*4 ]
   add  t7, t1, rx   # t7 <- X + i*4
   mult t4, t5
   mflo t6,          # t6 <- t4 * t5
   addi t6, t6, 512  # t6 <- t4*t5 + 512
   addi ri, ri, 1    # i += 1
   j f
   sw t6, 0(t7)      # M[ X + i*4 ] <- t6
e: nop
```

Programa 5.5: Tradução para *assembly* do Programa 5.3.

```
   li   ri, 0
   la   rx, X
   la   ra, A
   la   rb, B
f: slti t0, ri, N    # i < N
   beq  t0, r0, e
   nop
   sll  t1, ri, 2    # i*4
   add  t2, t1, ra   # t2 <- A + i*4
   add  t3, t1, rb   # t3 <- B + i*4
   lw   t4, 0(t2)    # t4 <- M[ A + i*4 ]
   lw   t5, 0(t3)    # t5 <- M[ B + i*4 ]
   lw   s0, 4(t2)    # s0 <- M[ A + (i+1)*4 ]
   lw   s1, 4(t3)    # s1 <- M[ B + (i+1)*4 ]
   mult t4, t5
   mflo t6,          # t6 <- t4 * t5
   mult s0, s1
   mflo s2,          # s2 <- s0 * s1
   addi t6, t6, 512  # t6 <- t4*t5 + 512
   addi s2, s2, 512  # s2 <- s0*s1 + 512
   add  t7, t1, rx   # t7 <- X + i*4
   addi ri, ri, 2    # i += 2
   sw   t6, 0(t7)    # M[ X + i*4 ] <- t6
   j    f
   sw   s2, 4(t7)    # M[ X + (i+1)*4 ] <- s2
e: nop
```

5.5.3 Níveis de otimização

O que segue é uma tradução resumida da Seção 6.4 de [**?**].

O `gcc` permite escolher o nível de otimização por aplicar ao processo de compilação. Quanto mais alto o nível de otimização mais longo é o tempo de compilação e melhor o desempenho do executável. Quanto mais baixo o nível de otimização, mais rápida é a compilação. Além do nível de otimização, há opções para ativar ou desativar tipos específicos de otimização.

O nível de otimização pode ser escolhido com o argumento '`-O`*nível*' (letra 'o' maiúscula), com *nível* variando de 0 a 3.

`-O0` nenhuma otimização – o código fonte é traduzido diretamente para *assembly*, sem nenhuma reordenação. Esta é a melhor opção para usar com código que está sendo depurado. A opção `-O0` equivale a nenhuma opção de otimização;

`-O1` efetua formas simples de otimização que não envolvem escolhas entre tempo e espaço. Executáveis ficam menores e mais rápidos do que com `-O0`;

`-O2` efetua todas as otimizações de `-O1` e outras tais como o escalonamento de instruções. Estas otimizações também não envolvem escolhas entre tempo e espaço e os executáveis não são maiores do que com `-O1`;

`-O3` efetua todas as otimizações de `-O2` e otimizações mais custosas como expansão de funções. Com `-O3` os executáveis ficam mais rápidos e podem ficar maiores;

`-Os` efetua otimizações para reduzir o tamanho do executável. Em alguns casos, o executável menor pode ser mais rápido por utilizar melhor a cache de instruções.

`-funroll-loops` efetua o desenrolamento de laços de forma independente das outras opções. Os executáveis são, em geral, maiores. Não há garantia de que o desempenho melhore com esta opção;

`-g` a opção `-g` não é uma opção de otimização mas sim de depuração. A depuração de programas otimizados pode ser dificultada pelas otimizações – variáveis que são eliminadas, laços desenrolados, funções expandidas – porque as linhas do código fonte podem não corresponder aos endereços no executável. Para depurar um programa o recomendável é usar `-g -O0`.

Otimização e *warnings*

Quando se emprega níveis mais altos de otimização, o compilador pode emitir *warnings* sobre variáveis não inicializadas que não ocorrem com níveis mais baixos de otimização. Isso ocorre porque, com as otimizações mais sofisticadas, o compilador efetua análise de fluxo de dados (*data-flow analysis*) e então detecta variáveis não inicializadas.

Exemplo 5.6 Este exemplo é aquele da Seção 6.5 de [**?**]. Copie o trecho de código abaixo para um arquivo chamado `teste.c` e verifique o resultado da compilação com diferentes níveis de otimização. Este exemplo é interessante porque a função `powern()` pode ser expandida, ou o seu laço desenrolado. O tempo de execução deve ser medido com o comando `time`.

```
prompt: gcc -Wall -O0 teste.c -lm
prompt: time ./a.out
```

```
#include <stdio.h>

double powern (double d, unsigned n) {
   double x = 1.0;
   unsigned j;

   for (j = 1; j <= n; j++)
      x *= d;

   return x;
}

int main (void) {
   double sum = 0.0;
   unsigned i;

  for (i = 1; i <= 100000000; i++) {
     sum += powern (i, i % 5);
  }

  printf ("sum = %g\n", sum);
  return 0;
}
```

Recompile com `-O1`, `-O2`, `-O3`, `-Os`, `-funroll-loops`, e `-O3 -funroll-loops`. Observe o resultado para `user` porque a medição do tempo total (`real`) pode incluir eventos externos à execução do programa. ◁

5.6 Exercícios

Ex. 42 Desenrole o laço abaixo quatro vezes. Cuidado com o limite, que não é uma constante e pode não ser um múltiplo de quatro.

```
for (i=0; i < num; i+=1) { A[i] = B[i] + C[i]; }
```

Ex. 43 Acrescente ao seu caminho o diretório com o cross-compilador para MIPS:

prompt: `export PATH=${PATH}:/home/soft/linux/mips/cross/bin`

Compile o código da Seção 5.5.3 com `mips-gcc`, com os mesmos níveis de otimização lá indicados, e verifique as diferenças no código *assembly* gerado.

Ex. 44 Otimize os programas abaixo, alterando somente o código fonte.

```
// (a)
#define N 1024
x=0;
for (i=0; i < N; i++)
  x += A[i] * B[i];
```

```
// (b)
var = ...;
x=0;
for (i=0; i < var; i++)
  x += A[i] * B[i];

// (c)
typedef struct S {
  int x;
  int y;
  short z[12];
} Stype;

Stype s[256];

x=0; y=1; z=1;
for (i=0; i < 256; i++) {
  acum_x += s[i].x * 321/17;
  acum_y *= s[i].y;
  for (t=1, j=0; j < 12; j++)
    t *= s[i].z[j];
  acum_z *= t;
}
```

Ex. 45 Traduza as versões otimizadas dos três programas do exercício anterior para *assembly* e efetue o reescalonamento das instruções. Todas as bolhas foram eliminadas? Se não, por que?

Ex. 46 Compare o código que você escreveu para responder ao Ex. 45 com aquele gerado pelo `gcc` com `-O0`, `-O1`, `-O2`, `-O3` e `-funroll-loops`. Quais as diferenças? Seu código é melhor do que o gerado automaticamente? É possível que o compilador elimine o código porque os trechos indicados "não fazem nada de útil". Se isso ocorrer, escreva os trechos como se fossem funções.

Capítulo 6

Uma Breve Introdução aos Sistemas Operacionais

Os *Sistemas Operacionais* (SO) estão entre os objetos mais complexos construídos pelos humanos e são compostos de vários subsistemas e de um núcleo, ou *kernel*, que os controla e gerencia o uso dos recursos do computador.

A função principal do SO é isolar o programador dos componentes de *hardware*, provendo um conjunto de abstrações que simplificam o uso e a programação do computador. Este capítulo contém uma breve introdução a um vasto campo e pretende-se abordar somente uns poucos tópicos, necessários para o projeto e a implementação de dois programas para controlar periféricos (*drivers*) simples. Nos interessa estudar dois tópicos importantes que são os *processos* e o *subsistema de Entrada e Saída* (E/S). Adiante estudaremos uma parte do subsistema de gerenciamento de memória, que é memória virtual com paginação.

A nomenclatura não é das mais felizes, porque tanto o "sistema operacional" quanto seus subsistemas (de memória, de arquivos, de E/S) são todos 'sistemas'. Neste capítulo, as palavras 'usuário' e 'programadora' devem ser consideradas sinônimos.

Se uma parte do texto parece superficial, isso é inevitável porque o que se pretende é apresentar uma breve introdução a alguns poucos tópicos de uma ampla e complexa área. Estes e outros tópicos serão retomados com profundidade na disciplina de Sistemas Operacionais.

6.1 Modelo do computador

O computador em que este material foi editado, e que é projetado para ser usado sobre uma mesa, contém um processador, 4Gbytes de memória DRAM, 0.5Tbytes de espaço em disco magnético, uma interface de rede, *mouse*, teclado, interface de vídeo, e interfaces USB. Um diagrama de blocos com um modelo para este computador é mostrado na Figura 6.1.

O processador é ligado à *memória principal* através de um *barramento de memória*, que é uma via de transporte de bits projetada para operar a uma elevada taxa de transferência de bytes entre o processador e a memória DRAM – a taxa é da ordem de 10^9 B/s. A este barramento também é ligada uma *ponte* que faz a ligação elétrica e lógica entre o barramento de memória e o *barramento de Entrada e Saída* (E/S). A este segundo barramento estão ligados os *dispositivos periféricos* rápidos tais como disco magnético e interface de rede, bem como os dispositivos lentos, tais como mouse, teclado e interfaces USB. Por 'rápidos' entenda-se taxas de transferência maiores do que 10^6 B/s, e por 'lentos', taxas menores que 10^3B/s. Dispositivos USB podem operar em qualquer dessas faixas, mas os enquadraremos na classe de "dispositivos lentos".

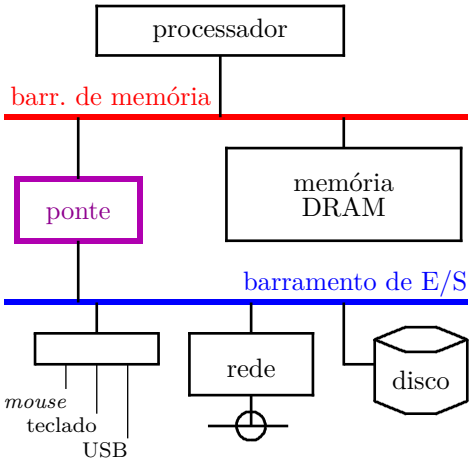


Figura 6.1: Diagrama de blocos de um computador de mesa.

A interface de vídeo é ligada diretamente à ponte porque a taxa de transferência para a interface de vídeo é maior que 10^7 B/s. Em sistemas convencionais, o armazenador que mantém uma cópia do que é 'desenhado' na tela é acessado concorrentemente pela interface de vídeo – acessos só para leitura – e pelo processador para alterar a imagem 'desenhada' na tela – estes acessos são de leitura e de escrita. A interface de vídeo é um dispositivo muito peculiar, e seu tratamento está fora do escopo deste texto.

A memória DRAM é chamada de "memória principal" porque o processador somente executa instruções e modifica dados que estejam presentes nesta memória. A capacidade da memória principal não é infinita, e é estendida pela *memória secundária*, tipicamente implementada num disco magnético. O tempo de um acesso ao disco magnético corresponde à execução de algo como 10^5 a 10^6 instruções pelo processador, enquanto que sua capacidade é da ordem de 10^3 vezes a capacidade da memória DRAM. Por conta disso, alguns mecanismos são usados para transferir para o disco porções da memória principal, de forma a aumentar – ao menos temporariamente – sua capacidade. Estas transferências são invisíveis ao usuário, que pode assim fazer uso de uma memória virtualmente 'infinita'. O Capítulo 9 descreve as interações entre as memórias principal e secundária.

Além de atuar como memória secundária, o disco magnético é o repositório de arquivos com os dados de usuário e os programas que tornam o computador uma ferramenta útil. Nesta função, de repositório de arquivos, o disco se comporta como um periférico, da mesma forma que a interface de rede. Estes dois dispositivos são ditos 'gulosos' porque consomem e produzem grandes quantidades de informação ao longo do tempo, algo da ordem de 10^4 a 10^5 B/s cada um. Estas taxas de transferência exigem tratamento especial, tanto do *hardware* quanto do *software* do sistema. Do contrário, o processador passaria mais tempo efetuando transferências de/para estes periféricos, do que executando programas.

Os dispositivos que permitem ao humano interagir com o computador são o conjunto teclado-tela e o *mouse*. A interface de vídeo é também uma interface gulosa porque são necessários algo entre 10 a 30MB/s para desenhar a tela, dependendo de suas dimensões e resolução. Este dispositivo, embora guloso, é passivo porque somente consome dados e portanto seu tratamento é efetuado quase todo em *hardware*. O processamento das imagens consome enormes recursos computacionais – rotações, deslocamentos, *rendering* – mas o tratamento da interface de vídeo como um periférico é relativamente simples porque as operações se limitam à acessos sequenciais ao armazenador em DRAM.

Teclado e *mouse*, por outro lado, são dispositivos lentos, que produzem dezenas de bytes/segundo e portanto o tratamento dos eventos gerados por estes dispositivos é relativamente

simples e demanda pouco tempo e/ou memória. As interfaces do tipo USB tem uma taxa de produção/consumo de dados que é algo menor do que disco e rede e seu tratamento também é relativamente simples.

Em uso normal, todos os periféricos operam ao mesmo tempo. A interface de rede produz dados, que são temporariamente armazenados em DRAM, para serem definitivamente armazenados em disco. Estas transferências se dão nas duas direções, evidentemente. Quando da leitura de um arquivo, o disco magnético produz dados que são copiados para a memória principal, ao mesmo tempo em que os dados recebidos da interface de rede são copiados da DRAM para um outro arquivo. Nesse meio tempo, o humano escreve ao teclado, movimenta o *mouse*, e escuta música que é obtida de um *pendrive*.

O componente do computador que gerencia e coordena as atividades do processador, periféricos é o *sistema operacional*. A Figura 6.2 mostra as fronteiras entre os programas de aplicação, o SO e o *hardware* do computador. O SO provê uma série de 'serviços' a programas de aplicação tais como navegadores, e coordena a operação concomitante e concorrente de todos os periféricos do sistema.

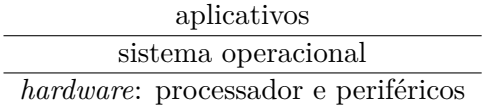


Figura 6.2: Fronteiras entre aplicativos, SO e *hardware*.

No que se segue são apresentados alguns dos conceitos fundamentais no projeto e implementação de sistemas operacionais. O Capítulo 7 descreve, em mais detalhe, o subsistema de entrada e saída de um SO, enquanto que o Capítulo 8 discute em algum detalhe uma interface serial, que é um modelo básico para todos os mecanismos de comunicação que transferem um bit de cada vez. O Capítulo 9 trata do subsistema de memória, do ponto de vista do SO.

6.2 Abstração, Interfaces e Virtualização

Vejamos os conceitos de 'abstração', 'interfaces' e 'virtualização' e como estes se relacionam.

Abstração

Abstração é uma forma de gerenciar complexidade, empregada para esconder detalhes irrelevantes e apresentar somente a informação que é relevante num dado contexto.

Em se tratando de sistemas operacionais, abstrações são construídas sobre modelos estratificados: um determinado nível de abstração oferece serviços através de uma *interface* bem definida, e estes serviços, por sua vez, são implementados usando serviços providos através das interfaces dos níveis inferiores.

O nível mais baixo é o próprio *hardware* do computador, e o serviço provido pelo processador é a execução das instruções dos programas. A interface para acesso aos serviços providos pelo processador é o seu conjunto de instruções, seu mecanismo de interrupções, e os registradores de controle e de status dos periféricos. Além do conjunto de instruções e dos periféricos, uma *application binary interface* (ABI) é usada para definir convenções de uso dos recursos do processador, tal como a implementação de funções em C no MIPS.

Acima deste nível são implementados um conjunto grande de serviços, necessários para que a programadora possa interagir com o sistema empregando funções de mais alto nível de abstração, tais como `open()` e `write()`, ao invés de programar diretamente os registradores de controle da interface dos dispositivo de armazenamento.

Interfaces

Uma *interface* é a 'superfície' que une, e ao mesmo tempo separa, duas camadas. No nosso caso, as camadas são aquela acima de uma abstração, e aquela que implementa a abstração. Em geral, a interface para uma abstração é definida através de uma *application programming interface* (API), que tipicamente consiste da documentação, e portanto especificação, e de páginas de manual dos serviços providos pela interface.

Como um exemplo de abstração, considere o sistema de arquivos e sua implementação com discos rígidos. O lado esquerdo da Figura 6.3 mostra um dispositivo com duas superfícies, um braço que suporta duas cabeças de leitura e gravação. O diagrama da direita mostra que uma *trilha* é dividida em *setores*. Um *cilindro* é a posição de uma mesma trilha nas duas superfícies. O braço gira sobre um pivô, para que a cabeça possa acessar todas as trilhas ao longo do raio.

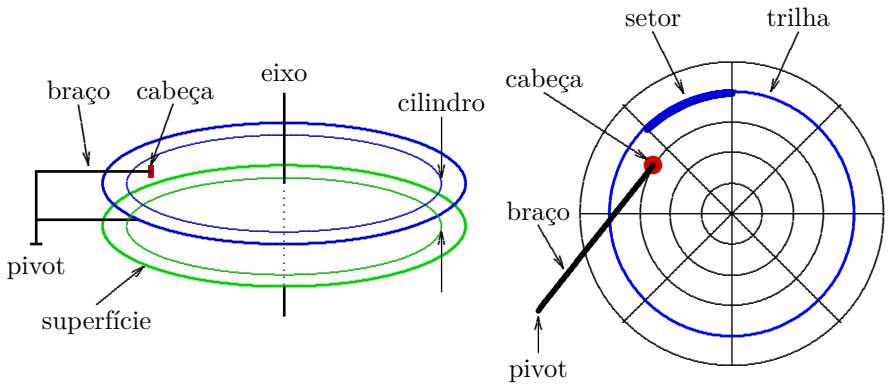


Figura 6.3: Modelo simplificado para um disco rígido.

No *dispositivo físico*, que é a unidade de disco rígido, as informações são armazenadas em *setores* de 1, 2 ou 4 Kbytes. Uma *trilha* contém centenas de setores, e uma *superfície* do disco contém milhares de trilhas.

O sistema de arquivos é projetado com base num *dispositivo lógico* que se comporta como um *vetor de blocos*, indexado por um inteiro – tipicamente, um bloco lógico tem o mesmo tamanho que um setor, ou bloco físico. O dispositivo lógico é uma abstração baseada numa estrutura de dados, que é o vetor de blocos lógicos, mais uma função – e sua inversa – que mapeia blocos físicos em blocos lógicos. O dispositivo lógico também esconde todos os detalhes relativos à temporização do dispositivo físico, que é deveras complexa.

O mapeamento das três dimensões do dispositivo físico para uma dimensão no dispositivo lógico – `disco[superfície,trilha,setor]` $\rightsquigarrow$ `bloco[i]` – é uma das tarefas de "baixo nível" de abstração efetuadas pelo SO e é concentrada 'próxima' do dispositivo físico.

O mapeamento de um certo número de blocos num *arquivo* é uma das tarefas de "alto nível" efetuadas pelo SO, que mapeia os blocos do dispositivo lógico em diretórios e arquivos, e este mapeamento é efetuado 'longe' do dispositivo físico. A programadora pensa em termos de operações sobre arquivos e diretórios, tais como `open()` e `read()`, sem se preocupar com os detalhes de operação do dispositivo físico, que pode ser um disco rígido, ou um *pendrive*, ou um DVD, dispositivos físicos que são organizados de formas radicalmente distintas.

Neste exemplo, o SO provê uma abstração para "dispositivo de armazenamento" que esconde os detalhes de programação do dispositivo físico. Mais ainda, a abstração permite que vários usuários acessem, simultaneamente, seus arquivos no mesmo dispositivo físico. O "dispositivo abstrato" que é acessado pelo programador pode estar implementado sobre um disco rígido ou sobre um *pendrive*, e isso é ocultado pela abstração.

De uma forma muito simplificada, a abstração "sistema de arquivos" é baseada em quatro níveis, e quanto mais alto o nível, maior a simplicidade conceitual do dispositivo. Quanto mais baixo o nível de abstração, mais expostos ficam os complexos detalhes do *hardware* e da temporização do dispositivos.

Os quatro níveis são: (i) a interface de programação de aplicativos através das funções de biblioteca `open()`, `read()`, `write()`; (ii) a interface do sistema de arquivos, que gerencia arquivos, diretórios, blocos, *inodes*; (iii) uma interface que esconde os detalhes dos dispositivos físicos e os 'transforma' em dispositivos lógicos acessados através de um vetor de blocos; e (iv) uma interface de programação dos registradores de controle/*status* para cada dispositivo físico, que é chamada de "*driver* do dispositivo físico" – esta interface, por sua vez, é dividida em dois níveis, que são discutidos adiante.

Virtualização

A *virtualização* é uma técnica que transforma um dispositivo físico numa versão 'virtual' daquele dispositivo, e esta transformação pode aumentar ou reduzir a funcionalidade do dispositivo virtual com relação ao dispositivo físico.

Considere um computador que suporta um ou mais usuários. Este computador executa vários programas ao "mesmo tempo", ou *concorrentemente*, e cada usuário executa dezenas de programas num único processador – um usuário trabalha num ambiente que inclui gerenciador de janelas, navegador, uma ou mais janelas, cada uma delas executando uma *shell* e um programa que reproduz música, como um mínimo.

O processador é virtualizado de forma a executar todos estes programas 'simultaneamente', ou como se o computador fosse capaz de executar cada um dos programas num processador dedicado exclusivamente àquele programa. O que ocorre é que *um* processador é multiplexado no tempo entre os vários programas em execução, cada programa executando repetidamente no processador por um intervalo relativamente curto, de uns poucos milissegundos.

Um programa executa, e sua computação progride, durante um intervalo curto, mas os intervalos de execução são tão frequentes que o usuário fica com a impressão de que cada programa executa em um processador dedicado para si. Tais sistemas são chamados de *multiprogramados* porque, aparentemente, executam muitos programas simultaneamente.

O Capítulo 9 descreve a *virtualização da memória* do computador, de modo que cada programa executa numa memória que é virtualmente infinita e isolada dos demais programas.

6.3 Modelo estratificado

Um sistema operacional pode ser projetado segundo um modelo estratificado, no qual cada camada oferece, à camada superior, serviços de mais alto nível, que são abstrações mais fáceis de usar, do que os serviços que recebe da camada inferior, que são abstrações menos fáceis de usar. A Figura 6.4 mostra o esquema de um SO estratificado idealizado; num sistema real, as interfaces são um tanto mais complexas do que esta pilha bem organizada.

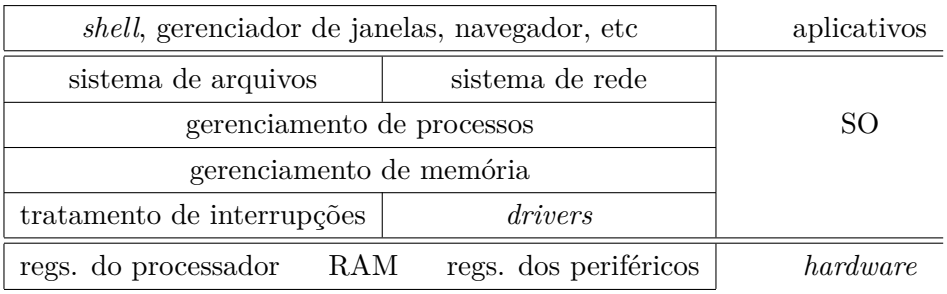


Figura 6.4: Modelo estratificado de um SO.

No nível mais baixo estão as interfaces de *hardware*, tais como os registradores de controle dos periféricos, o registrador de *status* do processador, o PC, apontador de pilha e registradores de uso geral. Estes não fazem parte do SO, mas são recursos do processador gerenciados pelo compilador – no caso dos registradores do processador – ou pelo SO – no caso dos periféricos e da memória física.

Logo acima do *hardware* executam os *drivers* dos periféricos. Tipicamente, um *driver* consiste de um conjunto de funções que incluem (i) uma função para inicializar o dispositivo; (ii) uma função para alterar/programar seu modo de operação; (iii) uma função para efetuar operações de entrada – o processador lê dados do periférico; e (iv) uma função para efetuar operações de saída – o processador escreve dados no periférico. Os *drivers* são frequentemente fornecidos pelo fabricante do periférico e sua programação envolve conhecimento detalhado da função do periférico e dos seus registradores de controle e de *status*. O mecanismo de interrupção provido pelo processador influencia no projeto e implementação dos *drivers*, como veremos em breve.

Acima da camada de interface com o *hardware* fica o sistema de gerenciamento de memória, que controla as operações de alocação de memória pelos programas, e implementa uma parte importante dos mecanismos de proteção de acesso. Mais detalhes no Capítulo 9.

Sobre a camada de gerenciamento de memória está o sistema de gerenciamento de processos, que determina como o principal recurso do sistema – o processador – é utilizado pelos programas. Esta é uma delimitação controversa, mas a estes três níveis se dá o nome de *núcleo* ou *kernel* do sistema operacional.

Acima do gerenciamento de processos geralmente são implementados o sistema de arquivos e o sistema de redes. Esta camada provê os serviços mais comumente utilizados pelos programadores, tais como as funções `open()`, `write()`, e `send()`. Estas funções são a "parte visível" do SO ao programador, e portanto são a API do sistema operacional.

Finalmente, os programas de usuário e que não fazem parte do SO, tais como Bash, gerenciadores de janelas, navegadores e terminais, se utilizam dos serviços providos pelas camadas inferiores, acessados através de funções de biblioteca tais como `open()` e `send()`.

6.4 Processos

Uma das abstrações mais importantes providas por um SO é a noção de processo: *um processo é um programa que está em execução.* Note que o arquivo executável de um programa não é um processo, a menos que o código contido no arquivo seja carregado em memória e seja executado pelo processador.

O *contexto de execução* de um processo consiste dos valores dos registradores do processador, do PC, do apontador de pilha, do registrador de *status*, das variáveis em memória, do seu mapa de memória, e do estado de quaisquer dispositivos de E/S que o processo utilize.

O contexto de todos os processos é mantido pelo SO na *tabela de processos*. Num sistema Unix, a tabela de processos detém, para cada processo, informações que incluem aquelas mostradas abaixo. O PID é usado para indexar a tabela de processos e portanto não é um campo da tabela, apesar de ser mostrado como se um campo fosse. Cada elemento da tabela de processos é chamado de um *descritor de processo*. A função de alguns destes campos é explicitada nas próximas seções.

PID o *Process IDentifier* é o número pelo qual o processo é identificado no sistema;

PPID o *Parent PID* é o PID do processo que é o pai deste processo;

estado o estado de execução do processo, que pode ser um dentre *executando*, *suspenso*, *pronto*, *morto*;

UID *User IDentifier* do seu *owner* ou *user*;

GID *Group IDentifier* do grupo do seu *owner*;

ambiente o ambiente de execução (variáveis da *shell*), herdado do processo pai;

diretório de trabalho que tipicamente é o diretório em que o processo filho foi criado (*working directory* ou WD);

área de salvamento de registradores uma região da memória DRAM na qual são salvados todos os registradores do processo quando este não está executando;

mapa de memória um apontador para o mapa de memória deste processo, que tipicamente é sua *Tabela de Páginas* (detalhes no Capítulo 9);

arquivos abertos lista com os descritores de todos os arquivos que foram abertos pelo processo, incluindo `stdin`, `stdout`, `stderr`, e é herdada do pai; e

eventos de E/S agendados lista de todos os eventos de E/S (leituras do disco ou da interface de rede) agendados por este processo, e que ainda não completaram.

6.4.1 Ciclo de vida de um processo

O processador é um dos recursos mais importantes dentre os gerenciados pelo SO. Quando as instruções de um processo estão sendo executadas, este processo está *executando*; do contrário, quando as instruções do processo não estão sendo executadas, o processo pode estar *suspenso* enquanto espera por uma operação de E/S demorada, ou pode estar *pronto para executar*, enquanto espera por uma nova chance de usar o processador. Por exemplo, acessos a disco custam de 10^5 a 10^6 ciclos do processador e se pode fazer um uso mais eficiente desse tempo, executando outro processo até que o acesso ao disco complete.

A Figura 6.5 mostra o diagrama de estados de um processo. O processo que está executando no processador pode devolver o processador para o SO quando termina e "morre", ou quando solicita uma operação de E/S demorada, quando então o processo suspende sua execução voluntariamente. Um processo suspenso torna-se pronto depois que o dispositivo de E/S sinalizar ao SO que a operação solicitada completou. Um processo que inicia é inserido na fila de prontos para executar. Vejamos as definições dos estados.

Um processo cujas instruções estão sendo executadas pelo processador está no estado *executando*.

Um processo está *morto* depois de executar a chamada de sistema `exit()`. Todos os recursos utilizados pelo processo são devolvidos, todas as operações de E/S são completadas, o mapa de memória do processo é apagado, e seu descritor é marcado 'livre' para reuso.

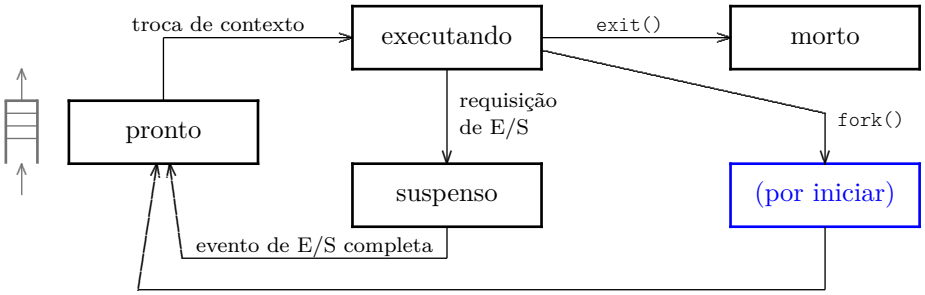


Figura 6.5: Diagrama de estados de um processo.

Um processo fica *suspenso* quando, ao solicitar uma operação de E/S demorada, devolve o processador ao sistema, para que algum outro processo execute enquanto a operação de E/S não completar.

Um processo está *pronto* para executar quando espera pela chance de usar o processador. Processos prontos aguardam numa fila, e o processo na cabeça da fila é o próximo a executar. Esta fila é chamada *fila de (processos) pronto(s)*.

Considere o que ocorre quando você executa o comando ls no Bash. O processo ls inicia sua execução ao ser criado por Bash, que é seu "processo pai"; o processo filho é criado com uma chamada de sistema (*system call*) `fork()`, e esta função cria uma réplica do processo pai em memória. Quando o filho executa a chamada de sistema `execve()`, a imagem executável do ls é copiada do disco e sobrescreve a cópia do processo pai gerada pelo `fork()`.

Um diagrama de tempos com a execução concorrente de dois processos, pai e filho, é mostrado na Figura 6.6. Os eventos relevantes são marcados pelos números em itálico. Em (1), quando o processo pai executa `fork()`, este é suspenso para permitir que sua cópia seja preparada para o processo filho – um elemento da tabela de processos é inicializado e uma região de memória é alocada para a pilha do novo processo. Quando o processo filho executa `execve()`, em (2), o carregador é invocado para copiar do disco o executável do filho; como esta cópia é demorada, o processo filho também é suspenso. Em (3) o processo pai volta a executar porque chegou à cabeça da fila de processos prontos. Em (4), quando a carga do executável completa, o processo filho passa a executar o código recém carregado. Em algum momento o filho completa sua tarefa – em (5) – e executa `exit()`; o final de sua execução é sinalizado ao processo pai, que estava suspenso a esperar por este evento desde a execução do `wait()`. A retomada da execução do pai ocorre em (6).

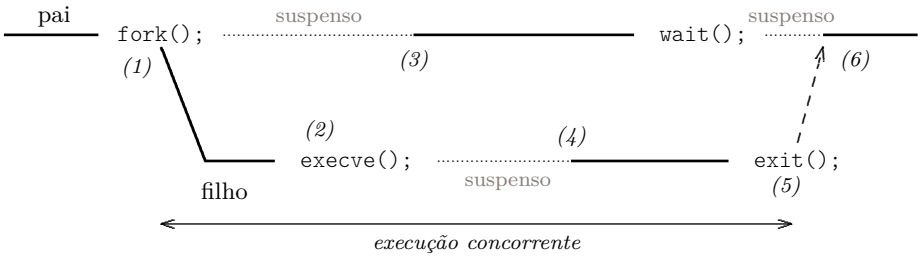


Figura 6.6: Diagrama de tempos de processos pai e filho.

O Programa 6.1 contém um modelo simplificado do código de criação de um processo [**?**]. O programa do processo pai deve inicializar as variáveis apropriadas para entregar ao filho os argumentos de linha de comando, como nas linhas 5 a 7. O `fork()` cria uma réplica do processo pai em memória e o SO preenche o descritor do novo processo. O filho herda os descritores de arquivos do pai, seu GID e UID.

A chamada a `execve()` (linha 15) provoca a substituição dos segmentos `TEXT` (código) e `DATA` do processo pai pelo conteúdo do arquivo executável apontado por `prog`. A pilha do processo filho é inicializada com os argumentos em `argv` e o ambiente apontado por `envp`. Ao executar o `execve()`, o processo filho passa a executar as instruções do programa `xxx`. O processo pai executa um `wait()` para esperar que o processo filho termine.

Programa 6.1: Modelo de código que cria um processo.

```
1   int child, status;
2   char *prog, *argv, *envp;
3   ...
4   // processo pai prepara os argumentos para criar o filho
5   prog="/usr/bin/xxx";
6   argv = "arg1␣arg2␣arg3";
7   envp = "VAR_1=valor1␣VAR_2=valor2";
8
9   child = fork();  // processo pai cria cópia para o filho
10
11  // processo pai recebe o PID do filho em 'child'
12  // processo filho recebe 0
13  if (child == 0) {
14     // se cheguei aqui sou o filho
15     execve(prog, argv, envp); // sobrescreve cópia do pai
16  } else {
17     // se cheguei aqui sou o pai
18     atividadesPaternais();
19     ...
20     wait(&status);           // proc pai espera morte do filho
21  }
22  ...
```

O efeito da execução do trecho de código no Programa 6.1 seria o de digitar, na linha de comando o seguinte:

```
(VAR_1=valor1 VAR_2=valor2 xxx arg1 arg2 arg3)
```

O ambiente de execução de `xxx` é uma cópia do ambiente do processo pai, aumentado com as variáveis de ambiente `VAR_1` e `VAR_2`, e o programa `xxx` é invocado com três argumentos, `arg1`, `arg2` e `arg3`.

A chamada de sistema `fork()` tem um comportamento distinto daquele da invocação de uma função em C. Quando invoca uma função, tal como `printf()`, o processo segue executando até o final do corpo da função, quando o fluxo de controle retorna ao ponto do programa onde ocorreu o salto para o código do `printf()`. No caso do `fork()`, o controle retorna para o local de sua invocação *em dois processos distintos*, e esta função retorna um valor diferente para cada processo. A diferença mais importante de `fork()` para funções comuns é que os processos pai e filho passam a executar concorrentemente no processador.

Exemplo 6.1 O que ocorre quando a programadora invoca o programa `ls` na sua *shell*?

Reveja o diagrama de tempos da Figura 6.6. O processo pai é a *shell*, que prepara os argumentos para a execução do `ls` – seus argumentos de linha de comando – e executa a chamada de sistema `fork()`. O processo pai (*shell*) e o processo filho (`ls`) passam a executar 'simultaneamente'.

O filho, que ainda é uma cópia do pai, executa `execve()`, fazendo com que o conteúdo do arquivo executável em `/bin/ls` seja gravado sobre a cópia do processo pai. O processo `ls`, que obteve o controle sobre `stdout` e `stdin` da *shell*, exibe o conteúdo do diretório indicado na linha de comando e termina com um `exit()`, avisando ao pai que terminou.

Ao receber o sinal do término do filho, o processo pai recupera o controle sobre `stdin` e `stdout`, e espera por um novo comando através de `stdin`. ◁

Exemplo 6.2 O que ocorre se o usuário, a partir da *shell* Bash, executar um *script* que deve ser interpretado por Bash? A interpretação do *script* criará uma nova instância de Bash – um processo filho com o mesmo código, mas com argumentos de linha de comando e ambiente de execução distintos, além de um conjunto de variáveis em memória que é disjunto das variáveis do processo pai. Um mesmo programa – código estático – pode ser executado em vários processos – código em execução – e cada um destes processos executa num contexto separado, com seu próprio PID e elemento da tabela de processos, com um conjunto disjunto de variáveis em memória, pilha, e de valores do PC, *status*, arquivos abertos, e eventuais operações de E/S. ◁

Exemplo 6.3 Considere o *pipeline* `ls | sort`. O programa `ls` efetua uma ou mais leituras do disco para obter os conteúdos do diretório corrente e sua saída padrão é entregue para `sort`, que ordena os dados recebidos da entrada padrão.

Do ponto de vista do SO, tudo o que atravessa o '*pipe*' são bytes, sem nenhum significado especial. A `stdout` de `ls` é ligada à `stdin` de `sort`, e o significado dos bytes transferidos através do pipe é atribuído pelos programadores que escreveram aqueles dois programas.

A Figura 6.7 mostra uma fila de bytes entre `ls` e `sort`, que é uma área de memória compartilhada pelos dois processos, e esta área, ou *buffer*, tem tamanho finito. A finitude significa que `ls` não pode pode escrever se o *buffer* está cheio, bem como `sort` não pode ler se o *buffer* estiver vazio.

Uma variável de controle, chamada de *semáforo*, é usada para suspender a execução de `ls` se o *buffer* estiver cheio, e um outro semáforo é usado para suspender a execução de `sort` se o *buffer* estiver vazio. Estes semáforos *'sincronizam'* as execuções dos dois processos. Quando o `ls` termina, o `sort` é sinalizado para que este execute `exit()`.

Os dois processos executam *concorrentemente* porque eles competem, ou concorrem, pelo uso dos recursos 'processador' e '*buffer*', e a competição é organizada por código que emprega semáforos. Quem organiza a concorrência? O programador dos aplicativos, que faz uso dos serviços do SO tais como a sincronização com *semáforos*, entrada e saída com `stdout` e `stdin`, e o *compartilhamento de memória*, implícito no uso do *pipeline*. ◁

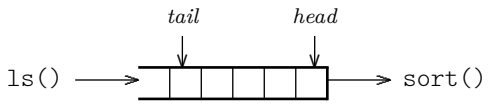


Figura 6.7: *Pipeline* com processos `ls` e `sort`.

6.4.2 Escalonamento de processos

Existem, pelo menos, duas formas de se escolher quando o processo que está executando devolverá o processador. A mais simples é chamada de *não-preemptiva*: o próprio processo decide quando devolverá o processador, e normalmente isso ocorre a cada operação de E/S. Em sistemas não-preemptivos, os processos devem cooperar para que um deles não monopolize o processador em detrimento dos outros.

Em sistemas *preemptivos* existe um "relógio do sistema" que provoca uma troca de processos periodicamente, e este período é chamado de *quantum*. Um processo executa até que o seu *quantum* de tempo seja excedido e então é removido do processador por uma ação do SO que é descrita na Seção 6.4.3. Nestes sistemas não existe a possibilidade de

um processo monopolizar o processador, mas o tempo necessário para completar uma certa tarefa se torna imprevisível por causa das interrupções no processamento provocadas pelo relógio do sistema.

Em sistemas preemptivos, o diagrama de estados contém um evento adicional na transição de *executando* para *pronto*, que ocorre ao final do seu *quantum*. A Figura 6.8 mostra o diagrama de estados com esta transição.

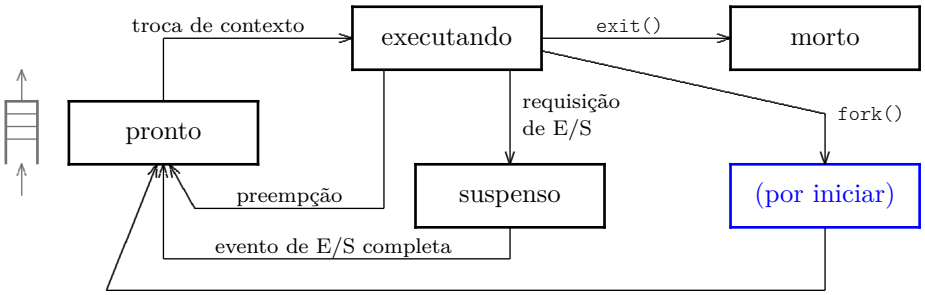


Figura 6.8: Diagrama de estados de um processo, com preempção.

Tipicamente, o relógio do sistema provoca uma troca de processos, ou o *reescalonamento* dos processos, 100 vezes num segundo, o que corresponde a uma interrupção a cada 10ms. A cada 10ms, o processador é forçado a passar a executar o processo que está na cabeça da fila de prontos. É este o mecanismo que virtualiza o processador – o processador é multiplexado no tempo, e a cada 10ms um novo processo pode executar até que seu *quantum* se esgote, quando então um novo processo terá a oportunidade de executar.

Existe a possibilidade de que um processo volte a executar, para logo em seguida se suspender enquanto espera por uma operação de E/S. A não ser que o sistema esteja sobrecarregado, este processo retornará ao processador em breve, porque os demais processos também poderão se suspender voluntariamente, à espera por operações de E/S.

A Figura 6.9 mostra dois diagramas de tempo com a execução dos processos P1, P2, e P3, com escalonamento não-preemptivo e escalonamento preemptivo. No exemplo de escalonamento não-preemptivo, o processo P1 executa até que efetua uma operação de E/S, e então devolve o processador ao escalonador que escolhe o processo P2, que está na cabeça da fila de prontos. P2 executa até terminar com uma chamada à `exit()`. O escalonador escolhe P3 para executar porque aquele está na cabeça da fila, e este executa até terminar com `exit()`. P1 executa até terminar.

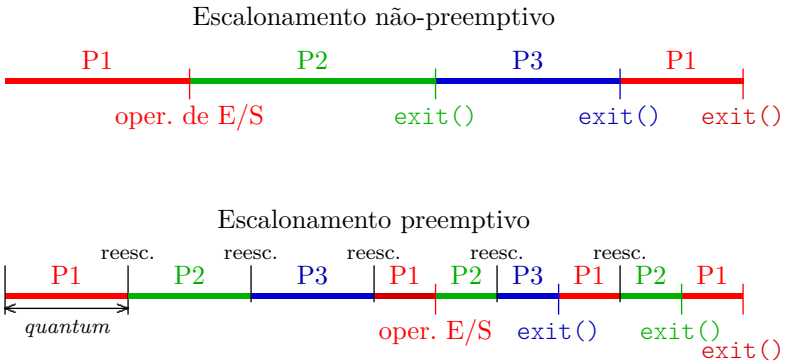


Figura 6.9: Escalonamento não-preemptivo e preemptivo.

Com escalonamento preemptivo, após o decurso do *quantum* (mostrado como *reesc.* no diagrama), P1 é removido do processador e P2 executa até terminar o próximo *quantum*, quando ocorre novo reescalonamento. P3 executa durante seu *quantum* e então perde o

processador. P1 executa até efetuar a operação de E/S quando devolve o processador. P2 executa até o final daquele *quantum* e perde o processador. P3 executa até terminar e P1 retorna ao processador até o final daquele *quantum*. P2 executa até terminar e P3 então executa até terminar. Note que P1 executa durante meio *quantum* e entrega o processador para efetuar uma operação de E/S, e que P2 executa meio *quantum* e entrega o processador porque termina.

O diagrama não mostra, mas cada reescalonamento consome tempo por causa da troca de contextos. Escalonamento preemptivo é menos eficiente do que o não-preemptivo porque cada processo individual toma mais tempo para completar por causa das interrupções. São relativamente raros os processos que executam até esgotar seus *quanta* porque as operações de E/S são frequentes. Processos com simulações complexas e/ou cálculo numérico são os que tendem a fazer uso de todos os seus *quanta*.

6.4.3 Contexto de execução

O *contexto de execução* de um processo é o estado, num dado instante, da computação que o processo executa. O contexto inclui o estado de execução do processo, o conteúdo de todos os registradores do processador, PC, apontador de pilha (SP), registrador de *status* (STAT), todo o conteúdo de suas variáveis em memória, além de arquivos abertos e operações de E/S pendentes.

Quando o processo que está executando muda de estado, o processo que está na cabeça da fila de prontos deve ser carregado no processador. Se o processo que mudou de estado não está terminando, o contexto de execução deve ser preservado no seu descritor na tabela de processos. Todos os registradores, incluindo PC, SP e STAT são copiados para a área de salvamento.; se necessário, o mapa de memória do processo é atualizado. Uma vez que o contexto esteja a salvo, o processo "que sai" fica *suspenso* – numa operação de E/S – ou volta para a fila de prontos, no estado *pronto*, se o sistema emprega escalonamento preemptivo.

O estado da memória, dos arquivos abertos, e o estado das operações de E/S pendentes é preservado pela implementação dos subsistemas de gerenciamento de memória, de arquivos e de E/S. Estas informações não são copiadas para o descritor do processo numa troca de contexto por serem demasiado volumosas.

O controle é devolvido ao SO, que recompõe o contexto do processo que está na cabeça da fila de prontos, porque este voltará a executar. O contexto do processo "que entra" é recuperado da sua área de salvamento, seu estado é alterado para *executando*, e o processador volta a executar as suas instruções. Esta operação é chamada de *troca de contexto*. São as trocas de contexto que efetivam a multiplexação do processador entre os vários processos de um sistema multiprogramado.

Por razões de eficiência, o código do SO que efetua as trocas de contexto deve ser rápido e geralmente é escrito em *assembly* por causa da manipulação direta dos registradores do processador. Na convenção de uso dos registradores do MIPS, os registradores `k0` e `k1` são reservados para uso pelo SO e não devem ser utilizados em programas de usuário. Estes dois registradores possibilitam a manipulação segura dos registradores no salvamento do estado do processador durante o tratamento de eventos tais como trocas de contexto e no tratamento de interrupções.

6.5 Modos de execução e proteção

Um mecanismo de *hardware* é necessário para viabilizar sistemas como os descritos aqui. O processador deve ser capaz de executar num de dois modos: *modo usuário* ou *modo sistema*. Geralmente, bits no registrador de *status* do processador definem o modo de execução.

Quando executa em *modo sistema*, o processador pode executar todas as instruções e pode acessar todas as posições de memória e periféricos. Este é o modo no qual é executado o código das chamadas de sistema que alteram as estruturas de dados importantes do SO, bem como o estado de operação dos periféricos. Um processo que executa em modo sistema pode se utilizar de todos os privilégios disponíveis ao SO – pode, potencialmente, alterar *todos* os bits de estado do computador.

Quando executa em *modo usuário*, o processador pode executar um subconjunto próprio das instruções e os acessos à memória são restritos aos endereços do segmento de dados (`.data`, `.bss`), pilha e segmento de código do processo (`.text`). Qualquer tentativa de executar "instruções privilegiadas", tais como aquelas que alteram o modo de execução, ou de acessar endereços ilegais – periféricos ou endereços fora do mapa de memória do processo – causam uma *exceção*, que possivelmente resultará no término forçado do processo, através das nossas bem conhecidas *segmentation faults*. Em *modo usuário* o processo pode alterar um conjunto relativamente pequeno dos bits de estado no computador, e nenhuma dessas alterações provoca algo pior do que a terminação (saudável) daquele processo.

Existem três maneiras de trocar de modo de execução, de modo usuário para modo sistema: (i) executar uma chamada de sistema que faz parte de uma função de biblioteca tal como `read()` ou `open()`; (ii) ocorrência de uma *interrupção* por um periférico; e (iii) ocorrência de uma *exceção*, tal como uma tentativa de executar uma instrução privilegiada ou acessar um endereço fora do mapa de memória do processo. Geralmente, as duas primeiras são benignas, enquanto que a terceira pode causar a terminação forçada do processo.

Infelizmente, a nomenclatura é confusa. Em textos de Sistemas Operacionais, o termo *syscall* é usado para identificar as funções que dão acesso aos serviços do sistema operacional. Nesse contexto, *syscall* é uma abreviatura para *system call*, e significa "invocação de serviço do sistema operacional". Quando se fala de processadores, geralmente existe uma *instrução* que é chamada de **`syscall`** que tem a funcionalidade de alterar o modo de execução de "modo usuário" para "modo sistema". Em geral, e para aumentar a confusão, uma *system call* deve conter uma instrução **`syscall`**.

Os processadores efetivam uma troca de modo usuário para sistema quando uma instrução privilegiada é executada. No caso do MIPS estas instruções são **`break`**, **`syscall`** ou **`trap`**. O código das funções de biblioteca efetua todas as verificações de proteção e dos parâmetros em modo usuário, e somente após haver a garantia de que o processo tem o direito de receber o serviço solicitado, uma instrução **`syscall`** deve ser executada para mudar o modo de execução de usuário para sistema, aumentando assim o nível de privilégio do processo, que então pode efetuar as operações de que necessita. Em geral, estas operações incluem alterações nas estruturas de dados do SO ou em periféricos, que devem ser efetuadas *exclusivamente* por código confiável, como é caso das bibliotecas do sistema tais como a `libc` e `stdio`. Por "código confiável" entende-se código com "origem confiável".

O código das funções de biblioteca contém instruções **`eret`** (*exception return*) que retornam o modo de execução para usuário, quando os privilégios do modo sistema são revogados.

O Programa 6.2 mostra o pseudocódigo para uma hipotética chamada de sistema `oper()`. O programa invoca `oper()` em modo usuário, e todos os três argumentos são verificados em modo usuário. Caso todos os argumentos sejam razoáveis, a função `syscall_oper()` é

invocada; esta é uma função curta, talvez codificada em *assembly*, e que contém a instrução `syscall`, que provoca a mudança para modo sistema.

Espaço em branco proposital.

Programa 6.2: Pseudocódigo C da chamada de sistema `oper()`.

```
1  int oper(int x, int y, int z) { // função de biblioteca
2     int status = NO_ERR;
3     if ( x != ... )      { ... ; status = ERR_IN_X; }
4     else if (y != ... )  { ... ; status = ERR_IN_Y; }
5     else if (z != ... )  { ... ; status = ERR_IN_Z; }
6     else {                       // argumentos aceitáveis
7        // esta função fica escondida do programador de aplicação
8        // executa a operação solicitada no tratador de exceção
9        status = syscall_oper(x, y, z);
10    }
11    return status;
12 }
```

O pseudocódigo de `syscall_oper()` é mostrado no Programa 6.3. Esta função é quase vazia e contém a instrução **`syscall`**, para provocar a exceção que muda o modo de proteção para sistema. As estruturas de dados do SO são alteradas no código do tratador das chamadas de sistema, e este tratador termina com uma instrução **`eret`**, que retorna o processador para modo usuário. A função `syscall_oper()` é a porta de entrada para o modo sistema, através da instrução **`syscall`**. O tratador da exceção provocada por **`syscall`** executa em modo sistema e efetua a tarefa indicada em `oper()`. Quando o controle é retornado para `syscall_oper()` com **`eret`**, o processador volta a executar em modo usuário.

Programa 6.3: Pseudocódigo *assembly* da chamada de sistema `oper()`.

```
1  # argumentos em a0, a1, a2, status retorna em v0
2  syscall_oper:
3     move a3, COD_SYSCALL_OPER # o serviço desejado é oper()
4     syscall # muda para modo sistema e altera estruturas do SO
5             # o código do tratador da exceção SYSCALL(oper)
6             #   efetua o serviço solicitado em oper(),
7             #   o tratador termina com um eret, e então
8             #   volta a executar em modo usuário
9     nop     # ponto de retorno da exceção SYSCALL(oper)
10    move v0, STATUS_OPER_OK # o serviço foi executado
11    jr ra   # retorna para o local da invocação em oper()
```

O modo de execução – sistema ou usuário – é um conceito ortogonal ao estado de execução dos processos. Quando o processador é re-escalonado, um processo pode devolver o processador em modo sistema – o que é a maneira usual, numa chamada de sistema – ou em modo usuário – em sistemas *preemptivos*, nos quais o decurso do *quantum* causa a retomada do processador pelo SO. O modo de execução é um mecanismo de "baixo nível" necessário para implementar os mecanismos de proteção de (mais) "alto nível", tais como os privilégios do usuário *root*.

Os mecanismos de proteção somente têm eficácia se o sistema for construído e mantido levando-se em conta todos os mecanismos de segurança. Por exemplo, o código das funções de biblioteca somente pode ser alterado pelo usuário *root*; os diretórios em que estas bibliotecas residem devem ser de propriedade de *root* e somente aquele pode alterar seus conteúdos. Evidentemente, sua senha deve ser inquebrável. Do contrário, seria possível a um calouro reformatar os discos dos nossos servidores de arquivos. Existe um SO para pedestres, tão popular quanto inseguro, no qual estes mecanismos foram introduzidos recentemente, enquanto sistemas da família Unix os empregam há mais de meio século.

6.6 Dispositivos de entrada e saída

Além da tabela de processos, o SO mantém uma *tabela de dispositivos*. Esta tabela contém um elemento para cada um dos dispositivos instalados, tais como terminais (`tty`), discos, impressoras, DVDs, *pendrives*, etc.

Os elementos da tabela de dispositivos contêm campos para o estado do dispositivo (pronto, esperando, erro), seu tipo (disco), seu número (disco-1, disco-2), e uma fila de requisições pendentes de/para o dispositivo.

Quando um processo solicita uma leitura do disco, por exemplo, o processo fica suspenso porque esta operação é demorada. O SO insere um registro na fila de requisições pendentes com as informações da requisição, tais como o PID do processo que solicita a leitura, qual o bloco solicitado, qual a operação (leitura/escrita).

Quando a requisição é satisfeita pelo dispositivo, este causa uma interrupção para informar ao SO. O SO consulta a lista de requisições e então insere o processo que foi atendido na fila de prontos e altera seu estado de *suspenso* para *pronto* para executar.

Esta breve descrição ignora vários tópicos que serão retomados em Sistemas Operacionais.

6.6.1 Processos e operações E/S

A Figura 6.10 mostra um diagrama de tempo com a execução de três processos, P1, P2 e P3. As quatro linhas horizontais representam a execução dos processos em modo usuário e em modo sistema, e as operações efetuadas pelos dispositivos de E/S, concorrentemente com a execução do processador.

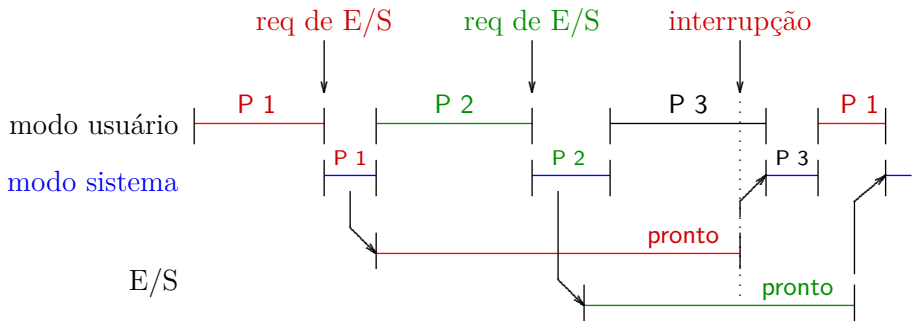


Figura 6.10: Sistema com três processos concorrentes.

Após executar por um certo tempo, P1 efetua uma requisição de E/S, tal como a leitura de alguns caracteres de um arquivo. O código de biblioteca da função `read()` faz as verificações para garantir que P1 tem as permissões para ler o arquivo e então efetua uma chamada de sistema (`syscall`) para invocar os serviços do SO associados à leitura do disco. Efetivada a requisição de leitura, a função `read()` invoca o trocador de contexto porque a leitura do disco é uma operação demorada. P1 fica suspenso, esperando até que a operação de E/S complete.

P2 estava na cabeça da fila de prontos e seu contexto é carregado no processador. Após algum tempo, P2 envia alguns caracteres de controle para configurar a impressora. Esta também é uma operação demorada e que depende de funções privilegiadas, e P2 se suspende enquanto espera pela resposta da impressora.

P3 é o próximo processo pronto e seu contexto é carregado no processador, e este executa durante um certo tempo. Antes que P3 termine, o controlador do disco interrompe o processador avisando que a leitura encomendada anteriormente por P1 completou.

P1 está na cabeça da fila de prontos. P3 devolve o processador, porque tentou ler um caractere do teclado. Na troca de contexto, P1 é recarregado no processador e sua execução continua no ponto do código da função `read()` no qual o processador suspendeu a execução de P1 e em seguida o processo volta a executar em modo usuário. Do ponto de vista de P1, nada aconteceu além de a requisição de leitura ter sido atendida.

Todos os detalhes da programação do controlador do disco, bem como a complexidade da temporização dos acessos ao dispositivo, são escondidos de quem escreveu o programa que executa como P1. O controlador de disco não detém nenhuma informação sobre o processo que solicitou a leitura. A informação que relaciona a requisição de leitura com o processo que a solicitou é mantida no descritor de processos de P1, na sua lista de eventos pendentes. Com a interrupção, o *driver* do disco informa que a leitura do bloco solicitado completou e então o SO consulta a fila de requisições do disco e lá encontra um apontador para P1.

6.7 Exercícios

Ex. 47 Descreva as máquinas virtuais providas por Bash, pela linguagem C, e pelo *assembly* do MIPS; compare os níveis de abstração das três máquinas.

Ex. 48 Escreva uma função recursiva para `bash()` que imprime (i) seu próprio PID, (ii) o PID do seu progenitor, e (iii) o caminho completo do executável – que é o caminho completo do próprio *script*, (iv) cria uma cópia, e (iv) entra num laço infinito ao final das chamadas recursivas. Execute sua função para que ela crie *somente cinco cópias* e observe os números dos processos com `ps` e `top`.

Você pode não gostar das consequências se o número de cópias for elevado.

Ex. 49 Com base nos resultados do Exercício 48, preencha a tabela de processos descrita na Seção 6.4 com os dados da execução de função recursiva e suas cópias.

Ex. 50 Considere, no sistema de três processos da Figura 6.9, que P1 necessita de 2000μs para completar, que P2 necessita de 1800μs, e que P3 necessita de 1600μs, que um *quantum* dura 500μs, e que cada troca de contexto custa 5μs. Quais os tempos totais de execução em cada uma das formas de escalonamento? Qual é o tempo médio de execução por processo? O tempo médio é o tempo total dividido pelo número de processos.

Ex. 51 Existem sistemas em que o escalonamento dos processos é baseado em prioridade. Em todas as trocas de contexto, o processo de maior prioridade é escolhido para executar. Fazendo uso dos dados do Ex. 50, e considerando que as prioridades são os números dos processos ($P3 > P2 > P1$), compute o tempo total de execução dos três processos, e o tempo de execução de cada processo – o tempo de P1 será o mais longo.

Ex. 52 Repita o Ex. 51, invertendo as prioridades: $P1 > P2 > P3$.

Ex. 53 Com base nos resultados dos Ex. 50, 51 e 52, compare o tempo total de execução dos três processos para cada forma de escalonamento.

Ex. 54 Com base nos resultados dos Ex. 50, 51 e 52, calcule o tempo médio de execução de cada processo para cada forma de escalonamento.

Capítulo 7

Entrada e Saída: Contador e Interrupções

God's teeth! Am I to suffer this constant stream of interruptions?
Will, Shakespeare in love

O computador que usaremos nas discussões sobre Entrada e Saída (E/S) é baseado num modelo do MIPS, escrito em VHDL e chamado de *cMIPS*, ou *classical MIPS*. Este modelo é uma "implementação literal" do processador descrito no livro texto de Arquitetura [**?**]. O código fonte do modelo do computador está disponível no GitHub em `https://github.com/rhexsel/cmips.git`.

A Figura 7.1 mostra um diagrama de blocos do modelo do nosso computador. O processador é ligado a uma memória ROM, que contém o(s) programa(s) por executar e é ligada ao estágio de busca do processador. Ao estágio de memória estão conectados a memória RAM e os periféricos. Como este é um modelo para simulações, os periféricos não são aqueles usualmente encontrados em computadores "de verdade".

Há um periférico que permite escrever na saída padrão do simulador (*stdout*), dois periféricos para efetuar acessos a arquivos em disco – um para leitura e outro para a escrita de inteiros em arquivos, os dois mostrados no bloco *file* – e um contador que gera uma interrupção após um número programável de ciclos do relógio (*count*). Há também uma interface serial, que não é mostrada na figura, mas é o objeto do Capítulo 8, e é o periférico empregado no trabalho desta disciplina.

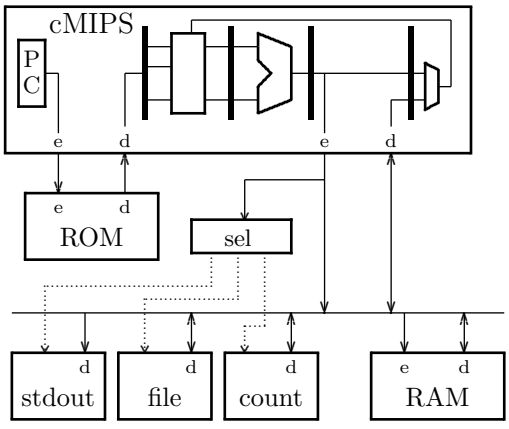


Figura 7.1: Modelo do computador com seus periféricos.

Cada periférico se comporta como se fosse uma 'memória' com umas poucas palavras, e cada uma dessas 'memórias' está alocada a uma faixa de endereços distinta das faixas com RAM e ROM. O seletor de endereços (*sel*), ligado ao barramento de endereços, seleciona

um dos periféricos em função do endereço emitido pelo processador. As faixas de endereços alocadas aos periféricos, RAM e ROM são definidas num arquivo de configuração do modelo VHDL do cMIPS. Na configuração atual, a memória ROM está alocada entre os endereços `0x0000.0000` e `0x0000.4000`, a memória RAM nos endereços de `0x0004.0000` a `0x0006.0000`, e os periféricos acima do endereço `0x3c00.0000`.

Nas próximas seções veremos dois modos de acesso aos periféricos pelo SO, usando o contador como exemplo. Este é um periférico assaz primitivo, o que simplifica a apresentação.

7.1 Modelo do periférico

A Seção 7.5 de [**?**] descreve circuitos de contadores e temporizadores. Estes circuitos são usados como base para gerar a noção de tempo-real, ou de horário do "relógio da parede" mantidos pelos computadores. Em "computadores de verdade", empregam-se circuitos integrados dedicados à manutenção do horário e estes são alimentados por uma bateria para que continuem a operar mesmo quando o computador está desligado. No nosso caso, usaremos um circuito simples para exemplificar os modos de acesso aos periféricos.

Recorde que periféricos se comportam como 'memórias' especializadas. Do ponto de vista do programador, o contador é uma memória de uma única palavra; quando esta 'palavra' é lida, o valor atual da contagem pode ser obtido; quando esta 'palavra' é atualizada, o contador é (re-)programado com o (novo) número de pulsos de relógio por contar.

Nosso periférico consiste de um contador de 30 bits que pode ser inicializado pelo processador com o valor limite da contagem. Uma vez que o processador grava um limite no contador, este é inicializado em zero e incrementa a contagem a cada ciclo do relógio. A contagem paralisa ao atingir o valor limite.

A Figura 7.2 mostra a interface de programação do contador. Numa escrita, que dispara uma nova contagem, o valor limite da contagem é contido nos bits 29 a 0, o bit 31 determina se uma interrupção será gerada (=1) ou não (=0), e o bit 30 habilita (=1) ou paralisa (=0) a contagem. Quando o periférico é lido, o bit 31 mostra o estado de habilitação da interrupção, o bit 30 mostra o estado de habilitação da contagem, e os bits 29 a 0 mostram o valor do contador no instante da leitura.

bit:	31	30	$29 \cdots 0$
escrita:	interrompe	habilita	limite
leitura:	interrompe	habilita	contagem

Figura 7.2: Interface de programação do contador.

7.1.1 Entrada e saída por programa

O modo de acesso a periféricos *por programa* (*programmed I/O*) é também conhecido por *polling*, ou "consultas frequentes". O periférico é programado para operar da maneira desejada e o processador executa um laço no qual efetua consultas frequentes ao estado do periférico. Vejamos um exemplo com o contador. Note que o que é apresentado no Programa 7.1 é pseudo-C e o código não seria compilado sem erros – a ideia é apresentar um modo de acesso e não um programa completo e acabado para acessar o periférico.

O laço externo (linhas 7 a 16) efetua algum processamento sobre os `N` itens de um armazenador (*buffer*). O laço interno (linhas 8 a 15) faz o programa esperar até que, pelo menos,

1.000 ciclos do relógio do processador tenham decorrido antes que o próximo item possa ser processado.

Programa 7.1: Exemplo de E/S por programa (*polling*).

```
1  volatile int *cont;        // aponta para o contador
2
3  cont = (int *)0x3c000080; // endereço do contador
4
5  *cont = 0x400003e8;        // programa contagem de 1.000 ciclos
6                             //   e habilita a contagem
7  for (i=0; i < N; i++) {    // processa armazenador com N itens
8     do {
9          // fica no laço a esperar, durante 1000 ciclos
10       } while ((*cont & 0x3fffffff) > 0);  // lê contador
11
12       fun(N, a[]);     // algum processamento sobre o armazenador
13
14      *cont = 0x400003e8; // inicia novo intervalo de 1000 ciclos
15    }
16 }
```

O valor da contagem é em 30 bits e por isso, na linha 10, o valor apontado por `*cont` é ajustado pela máscara de 30 bits `0x3fff.ffff`. O apontador é declarado como **`volatile`** para evitar que o compilador remova as instruções do laço interno se o código for otimizado. A máscara `0x3f` no byte mais significativo remove os dois bits de controle.

Este modo de acesso, *entrada e saída por programa*, é muito simples e portanto eficiente, mas possui dois defeitos sérios. O primeiro é que o processador fica preso no laço interno a esperar que a contagem chegue a zero. Enquanto espera, o processador não efetua nenhuma computação útil.

O segundo defeito, que não é evidente neste exemplo, é a baixa tolerância a falhas. Suponha que ocorra uma falha no circuito do contador, de tal forma que a contagem nunca chegue ao limite. O processador ficará eternamente preso no laço interno. Este problema é mais evidente em periféricos que dependem de uma ligação ao mundo externo, tal como uma interface USB via cabo, pois o cabo pode ser rompido ou removido acidentalmente.

Uma possível solução para o segundo defeito é esperar por um número limitado de voltas no laço interno. Se o evento esperado não ocorrer dentro de um intervalo razoável, é provável que algo de grave tenha ocorrido e isso deve ser sinalizado de alguma forma. O laço interno pode ser codificado como o mostrado no Programa 7.2. Se o laço fizer mais do que 200 voltas, então o intervalo de 1.000 ciclos deve ter decorrido, embora a contagem não tenha chegado ao limite. Isso pode indicar algum problema com o dispositivo de contagem. Evidentemente, o número de voltas do laço deve ser ajustado para tomar um tempo similar ao intervalo de espera.

Programa 7.2: Exemplo de E/S por programa com espera limitada.

```
1  for (i=0; i < N; i++) {   // processa armazenador com N itens
2     voltas=0;
3     do {
4         // espera por 1000 ciclos ou por 200 voltas no laço
5         voltas = voltas + 1;
6     } while ((*cont & 0x3fffffff) > 0 && (voltas < 200));
7     if (voltas == 200)
8        PANIC();
9     ...
10 }
```

A solução para o primeiro defeito, que é o processador ficar preso no laço, depende do sistema de interrupções, descrito na próxima seção.

Exemplo 7.1 Suponha que precisamos testar, automaticamente, se a sequência de contagem de contador é estritamente monotônica: a contagem do ciclo $i + 1$ é sempre maior do que aquela do ciclo i.

A Figura 7.3 mostra duas sequências de contagem, uma com duração CNT, e outra com duração CNT+1. A contagem é monotônica se, para amostragens c_i nos ciclos t_i, $c_{i+1} > c_i$, para todas as amostras, em todos os intervalos de duração CNT+i.

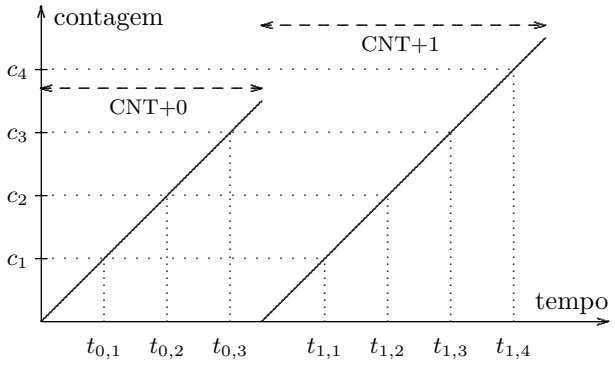


Figura 7.3: Teste para contagem monotônica.

Um teste que observe medidas para uma única sequência de duração CNT não é suficiente porque o contador pode contar corretamente num intervalo curto, tal como $[0, 2^N)$, mas falha para intervalos que provoquem alterações em mais do que N bits, como no intervalo $[0, 2^{N+1})$ ou $[0, 2^{N+2})$.

O Programa 7.3 testa um contador para quatro intervalos de contagem. O intervalo CNT é alargado a cada volta do laço externo. O laço interno acumula os resultados das comparações anteriores; se uma única falhar, então a contagem não é monotônica. ◁

Espaço em branco proposital.

Programa 7.3: Programa de teste para contagem monotônica.

```
1  #define N 4                  // test four intervals
2  #define CNT 0x40000040       // enable, count 64 cycles
3    ...
4    newValue = CNT;
5    increased = TRUE;
6
7    for (i=0; i < N; i++) {
8      newValue = CNT + (i<<3);  // CNT increases with i*8
9      startCounter(newValue, 0);
10     old = 0;
11     do {
12       if ( (new = readCounter()) > old) {
13         increased = increased & TRUE;
14         old = new;
15       } else {
16         increased = FALSE;
17       }
18     } while ( readCounter() < newValue );
19   }
20   report_result(increased);
```

7.1.2 Entrada e saída por interrupções

Quando o telefone toca, largamos o que estamos a fazer, respondemos de forma relativamente educada ao operador de *telemarketing*, e voltamos aos nossos afazeres. Se o telefone possui identificador de chamadas, podemos ignorar o chamado, caso o número seja desconhecido, ou podemos largar o que fazemos, caso a ligação seja de alguém importante. O mecanismo de interrupções de um processador não é muito diferente de um telefone que toca e do humano que o atende.

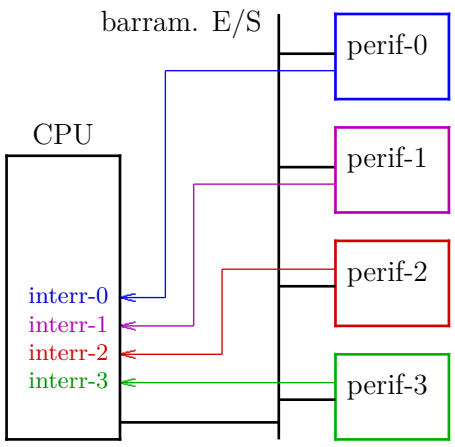


Figura 7.4: Modelo do sistema de interrupções.

O mecanismo de interrupções permite aos periféricos solicitar a atenção imediata do processador, que então trata do evento sinalizado pela interrupção. A Figura 7.4 mostra um diagrama de blocos com quatro periféricos e um processador. Cada periférico pode solicitar a atenção do processador para tratar um dos seus eventos através de seu sinal de interrupção. Cada linha de interrupção é ligada a um sinal na interface do processador, e estes sinais são amostrados a cada ciclo do relógio. Se um ou mais sinais de interrupção estiverem ativos, então o processador salta para um endereço pré-determinado e passa a

executar código para tratar o evento sinalizado. A Seção 7.1 de [**?**] descreve os circuitos necessários a um sistema de interrupções.

Ao atender a solicitação, o processador salta para o endereço do *tratador de interrupção* adequado ao evento que está sendo sinalizado. O tratador é uma função sem parâmetros, que não retorna nenhum valor, e que não perturba o estado de execução do processo que foi interrompido. A Figura 7.5 mostra um diagrama de tempo com um processo que sofre uma interrupção. Do ponto de vista lógico, o dispositivo que interrompe o processador é um 'processo' que executa concorrentemente com o processo que é interrompido.

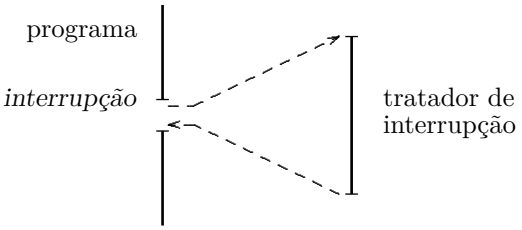


Figura 7.5: Execução concorrente de um processo e um tratador de interrupção.

Este último ponto é importante: o processo interrompido *não pode ser perturbado pela execução de um tratador* e isso significa que nenhum dos registradores usados pelo código do processo pode ser alterado por causa da execução do tratador. O código do tratador deve salvar os registradores que modifica numa área de salvamento em memória, para então tratar do evento, e antes de retornar deve recompor os registradores que foram salvados. Assim, o processo interrompido não é afetado pela ocorrência da interrupção, a menos do tempo dispendido a executar o código do tratador.

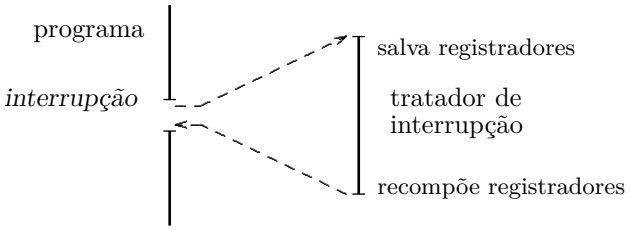


Figura 7.6: Salvamento e recomposição dos registradores pelo tratador.

Ao detectar a interrupção, o processador salva o endereço da instrução que seria executada no registrador EPC (*Exception PC*), e ao PC é atribuído o endereço da primeira instrução do tratador de interrupções. O EPC mantém o endereço da instrução que teria sido executada caso a interrupção não fosse aceita. A última instrução do tratador deve ser uma instrução **`eret`** (*exception return*), que copia o conteúdo do EPC para o PC, retomando assim a execução do processo que foi interrompido.

Como as interrupções são assíncronas aos processos, e estas ocorrem em instantes arbitrários, é necessário um mecanismo distinto e separado daquele empregado com funções, para relembrar o endereço de retorno das interrupções. A Seção 7.3 traz mais detalhes sobre o tratamento de interrupções no MIPS.

No intervalo em que os registradores são salvados, e durante a sua recomposição, as interrupções devem ficar desabilitadas para garantir que as cópias de e para a memória sejam efetuadas completamente: todos os registradores devem salvados ou recuperados. A ocorrência de uma interrupção no meio do salvamento pode provocar erros se um registrador for modificado pela segunda interrupção antes de ser salvado pela primeira.

O código de um tratador típico segue o padrão mostrado no Programa 7.4. O registrador CAUSE indica a causa da interrupção e este registrador deve ser lido no início do tratamento. Uma vez que o conteúdo de CAUSE está a salvo numa variável local ao tratador, os registradores que são alterados pelo código do tratador devem ser salvados na área de salvamento do tratador. Com o estado do processo interrompido a salvo em memória, o código que trata o evento pode ser executado.

Com base no conteúdo do registrador CAUSE determina-se qual é o evento sinalizado e assim o seu tratamento – no caso do contador, o tratamento consiste em reprogramar a contagem no periférico e talvez registrar a ocorrência da interrupção para uso no relógio de tempo real. Isso feito, as interrupções são temporariamente desabilitadas e os valores originais dos registradores são recompostos a partir das cópias salvadas em memória. Com as interrupções habilitadas, o tratador então volta a executar as instruções do processo que fora interrompido.

Geralmente, os processadores permitem a atribuição de prioridades às várias interrupções. No caso do MIPS, em seu modo de operação mais simples, são cinco interrupções por *hardware* e duas por *software*. Dentre as interrupções por *hardware*, a de maior prioridade é a de nível 7 e a de menor prioridade a de nível 3. Se forem detectadas duas interrupções ao mesmo tempo, a de maior prioridade deve ser atendida antes.

Programa 7.4: Esqueleto de um tratador de interrupção.

```
1  void tratador(void) {
2  volatile regCause Cause; // record que descreve o reg Cause
3
4  // neste ponto as interrupções estão desabilitadas (hardware)
5
6  Cause = le_reg_cause();
7
8  salva_registradores_do_processo_interrompido();
9
10 habilita_interrupções();
11
12 // descobre causa da interrupção e trata evento
13 switch (Cause.IP) {
14    case COUNTER:
15       // trata do evento sinalizado pelo contador interno
16    case SERIAL:
17       // trata do evento sinalizado pela interface serial
18    default:
19       // trata dos outros eventos
20 }
21
22 desabilita_interrupções();
23
24 recompõe_registradores();
25
26 habilita_interrupções();
27
28 return_from_interrupt();
29 }
```

O projetista do computador é quem decide qual evento tem maior prioridade. Normalmente, uma interrupção de teclado tem menor prioridade do que uma interrupção da interface de rede porque o humano ao teclado não perceberá a demora de uns poucos microssegundos dispendidos no atendimento de uma interrupção pela interface de rede.

Adicionalmente, o custo para recuperar uma mensagem que é perdida porque não foi tratada a tempo é significativo e portanto alta prioridade é atribuída às interrupções pela interface de rede.

O que fazer se uma interrupção for detectada durante a execução do tratador de alguma outra interrupção? A segunda interrupção pode ser tratada desde que (i) sua prioridade seja mais alta, e (ii) sua execução não perturbe o processo que foi interrompido, e que neste caso é um tratador. A Figura 7.7 mostra um diagrama de tempos com duas interrupções aninhadas: a execução do *tratador 1* é interrompida pela execução do *tratador 2*.

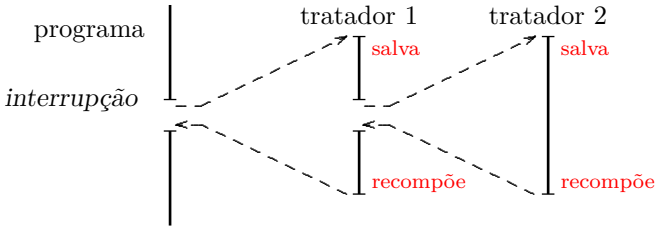


Figura 7.7: Interrupções aninhadas.

Voltemos à habilitação e desabilitação das interrupções na Seção 7.2. O processador inicia o tratamento de uma interrupção com todas as interrupções desabilitadas, e isso garante que o registrador CAUSE não seja alterado pela ocorrência de outra interrupção, bem como que os registradores possam ser preservados em memória. Uma vez que os registradores do processo interrompido estejam a salvo, as interrupções podem ser novamente habilitadas – talvez de forma seletiva se o sistema operacional admite interrupções aninhadas – porque uma nova interrupção não corromperia o estado.

As interrupções são desabilitadas ao final do tratador para que os conteúdos dos registradores sejam recuperados sem nenhuma perturbação. Ao retornar, as interrupções são automaticamente habilitadas.

O código dos tratadores de interrupção deve ser rápido e eficiente para que os eventos sinalizados ao processador não sejam perdidos porque seu tratamento não ocorreu a tempo. Por esta razão, as regiões de código em que as interrupções estão desabilitadas devem ser tão curtas quanto possível. Ao executar com as interrupções desabilitadas, o paralelismo entre o processador e o os periféricos é eliminado, o que serializa os eventos de E/S.

7.2 Concorrência e execução atômica

Do ponto de vista do sistema operacional, os periféricos se comportam como "processos em *hardware*" que executam em paralelo com o processador. Quando o processador programa o contador com uma nova contagem, o periférico passa a executar sua tarefa em paralelo ao(s) processo(s) de *software* que executa(m) no processador. Quando a contagem chegar ao limite, o processador é informado do evento através de uma interrupção.

Interrupções são uma maneira eficiente de tratar eventos externos ao processador mas possuem um efeito colateral causado pela *concorrência*: ao invés de um único processo executando no computador, os processos que podem escrever na memória concorrentemente são tantos quantos os tratadores de interrupção que possam executar simultaneamente, além do processo que é interrompido. O *controle de concorrência* é um tópico amplo e é estudado na disciplina de Sistemas Operacionais. Agora nos interessa somente uma versão simples que envolve habilitar ou impedir interrupções.

Qual o problema com concorrência? Vejamos um exemplo para nos auxiliar na resposta. Considere uma aplicação que lê comandos do teclado e os executa. Esta aplicação usa uma interrupção gerada pelo teclado para sinalizar a disponibilidade de um novo caractere. O tratador da interrupção lê o caractere do dispositivo *teclado*, o insere numa fila circular, e incrementa o contador de caracteres na variável `cont`. Se `cont > 0`, o programa remove um caractere da fila e decrementa aquela variável.

Esta aplicação consiste de dois processos concorrentes, o programa `prog()` e o tratador da interrupção `trat()`, que se comunicam através da fila e de `cont`. O contador de caracteres é a variável que *sincroniza* os dois processos: `prog()` não remove de uma fila vazia (`cont == 0`) e `trat()` não insere numa fila cheia (`cont == MAX`). A Figura 7.8 indica as relações entre os dois processos e as variáveis que compartilham.

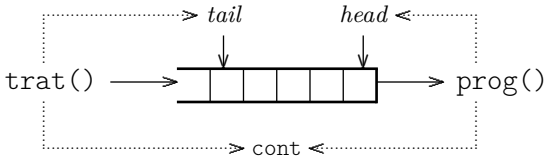


Figura 7.8: Fila circular compartilhada por dois processos.

Os processos `prog()` e `trat()` atualizam os apontadores da fila circular independentemente um do outro. A variável `cont` pode ser atualizada pelos dois processos concorrentemente, dependendo da temporização das interrupções. Considere o trecho de código no Programa 7.5, no qual `prog()` decrementa o contador de caracteres, e `trat()` o incrementa. Os dois processos executam virtualmente em paralelo porque o tratador é executado de forma assíncrona com o processo – os caracteres são recebidos pelo dispositivo em instantes arbitrários do ponto de vista do programa que trata destes caracteres.

Programa 7.5: Exemplo de acesso concorrente à variável.

```
#   prog()                          #   trat()
# processo decrementa               # tratador incrementa
# cont = cont - 1;                  # cont = cont + 1;
1: la   r6, cont                    6: la   k1, cont
2: lw   r5, 0(r6)                   7: lw   k0, 0(k1)
3: addi r5, r5, -1                  8: addi k0, k0, 1
4: sw   r5, 0(r6)                   9: sw   k0, 0(k1)
5: nop
```

Os registradores `r6` e `k1` apontam para o endereço de `cont` – estes dois registradores apontam para o mesmo endereço em memória. Suponha que `cont = 4` e `prog()` executa o `lw` da linha 2 (`r5` ← 4). Neste ponto, entre as linhas 2 e 3, `prog()` é interrompido. `trat()` é invocado e lê `cont` (`k0` ← 4), o incrementa na linha 8 (`k0` ← (4+1)), e este valor é armazenado na memória (`cont` ← 5). Após o retorno da interrupção, `prog()` decrementa `r5` (`r5` ← (4-1)) e armazena este valor em `cont` (`cont` ← 3).

Cada processo executou devidamente suas instruções mas o resultado é errado porque, ao menos, um caractere foi perdido. O problema está no *acesso concorrente à variável `cont`*, e este tipo de erro é extremamente difícil de se detectar e corrigir porque é praticamente impossível que a sequência de execução que provocou o erro seja repetida, o que não significa que o problema possa ser ignorado.

O comando em C "`cont = cont + 1;`" *não* é executado atomicamente pelo processador porque são necessárias três instruções para efetuar esta computação: **`lw`**→ **`add`**→ **`sw`**, e é esta sequência que deve executar atomicamente porque a variável `cont` é compartilhada. Tal se consegue desabilitando as interrupções em `prog()`, antes da linha 2 e habilitando-as

após a linha 4. Isso impede que `trat()` altere `cont` enquanto `prog()` está "no meio" de uma atualização.

O que se deseja é garantir que a alteração da variável `cont` ocorra em *exclusão mútua*; somente um dos dois, o processo ou o tratador da interrupção, pode acessar a variável para alterar seu valor. Para que isso ocorra, o incremento e o decremento devem ocorrer de forma *atômica*: ou a operação iniciou e completou sem a interferência de outro processo, ou ela não foi efetuada.

A forma mais simples de impedir que os dois processos concorrentes alterem o contador é desabilitar as interrupções enquanto o `prog()` decrementa o contador. Assim, o tratador de interrupção não poderá incrementar o contador concorrentemente a `prog()`. O código no Programa 7.6 garante o acesso atômico ao contador: entre as instruções **`di`** (linha 2) e **`ei`** (linha 6), o tratador de interrupções não executará e o decremento é atômico. Quando o tratador executar – somente antes da linha 2, ou somente depois da linha 6 – o processo permanece inativo e *não está decrementando o contador*, e portanto o incremento pelo processo é atômico.

Programa 7.6: Exemplo de acesso atômico à variável compartilhada.

```
# processo decrementa                    # tratador incrementa
# cont = cont - 1;                       # cont = cont + 1;
1: la   r6, cont                         7: la   k1, cont
2: di      # disable interrupts          8: lw   k0, 0(k1)
3: lw   r5, 0(r6)                        9: addi k0, k0, 1
4: addi r5, r5, -1                      10: sw   k0, 0(k1)
5: sw   r5, 0(r6)
6: ei     # enable interrupts
```

O grau de concorrência é diminuído quando as interrupções são desabilitadas, o que pode causar a perda de algum evento pelo processador. Por isso, os trechos de código que necessitam de execução atômica devem ser curtos para não atrasar o atendimento às interrupções.

7.3 Interrupções no MIPS

Nos documentos que descrevem a arquitetura MIPS32 [**?**], emprega-se o termo *exceção* para descrever eventos síncronos e assíncronos[1]. Um evento é dito *síncrono* se ele ocorre sempre na mesma instrução quando o programa é executado com os mesmos dados, como é o caso de uma divisão por zero. Um evento é dito *assíncrono* se não se pode prever sua ocorrência durante a execução de um programa, como é o caso da recepção de uma mensagem pela interface de rede. A Seção 7.4 explicita as diferenças entre estas duas classes de evento.

Os projetistas do MIPS optaram por um projeto simples em que a maioria das alterações no estado de execução deve ser efetivada por *software*, e somente um mínimo de estado é alterado pelo *hardware* na ocorrência de exceções ou interrupções. Tal filosofia de projeto simplifica o *hardware* ao mesmo tempo em que provê aos desenvolvedores do SO uma grande flexibilidade no uso do processador. No outro extremo das possibilidades, os projetistas da arquitetura x86 definiram que a maior parte das alterações de estado é efetivada pelo *hardware*, o que limita as opções disponíveis aos desenvolvedores.

A arquitetura do sistema de interrupções no MIPS32r2 consiste de 30 a 60 registradores do *Co-Processador 0* – ou do CP0. Destes nos interessam três, que são descritos abaixo. Uma implementação completa do CP0 contém mais de 50 registradores que permitem configurar

[1] Este é mais um daqueles casos em que ficamos reféns da terminologia deste ou daquele fabricante.

uma ampla gama de modos de operação do processador. Neste texto abordaremos um pequeno subconjunto do CP0. Todos os registradores do CP0 são descritos no Capítulo 8 de [**?**].

Exception PC (EPC) registrador que mantém o endereço da instrução causadora do evento excepcional. No caso de interrupções, contém o endereço da instrução que deve ser executada para retomar a execução do processo interrompido; no caso de exceções, contém o endereço da instrução na qual o evento foi detectado;

Status Register (STATUS) registrador que define o modo de execução do processador (usuário ou *kernel*), se e quais interrupções estão habilitadas;

Cause Register (CAUSE) registrador que exibe a(s) causa(s) do(s) evento(s) sinalizado(s) por uma interrupção e/ou exceção.

De uma forma muito simplificada, quando o processador detecta um evento excepcional, a execução salta para um endereço fixo, no qual está o "tratador de exceções universal". Este tratador lê o registrador CAUSE para determinar a causa da exceção e então executa a função específica a cada evento.

O tratamento da interrupção ocorre em "nível de exceção" (*exception level*), e todas as interrupções são ignoradas. Para retornar da interrupção, o processador executa uma instrução **`eret`**, quando volta a executar no "nível normal" e com as interrupções habilitadas.

Se o evento excepcional for uma interrupção, o processador salta para o endereço do "tratador de interrupções". Se há mais de um pedido de interrupção pendente, é o programador do tratador de interrupções quem define as prioridades de atendimento, em função do conteúdo de CAUSE.

A instrução **`mtc0`** (*move to CP0*) copia o conteúdo de um registrador para um registrador do CP0; a instrução **`mfc0`** (*move from CP0*) copia o conteúdo de um registrador do CP0 para um registrador de uso geral.

O Programa 7.7 mostra uma possível codificação para os trechos inicial e final de um tratador de exceções – uma versão mais realista é apresentada na Seção 7.4. As linhas 2 a 4 salvam o conteúdo de CAUSE em memória para uso posterior. Isso feito, o tratador decide qual evento será tratado, de-multiplexando os pedidos de interrupção. Para retornar da interrupção, as instruções devem ser habilitadas novamente. O **`eret`** causa o retorno para o endereço armazenado em EPC e o processador sai do "nível de exceção".

Programa 7.7: Tratador universal de exceções – primeira tentativa.

```
1  univ_excp_entry:              # at exception level
2          mfc0 k0, c0_cause     # save contents of CAUSE
3          la   k1, CAUSE_save_addr
4          sw   k0, 0(k1)
5
6  excp_demux:                   # event demultiplexer
7          ...                   # handle event
8
9  excp_return:
10         eret                  # return from exception
```

O Programa 7.7 emprega somente os registradores `k0` e `k1`, que são reservados para uso pelo SO (*kernel*). Os registradores que serão alterados no tratamento da exceção devem ser preservados no início do código que efetua a seleção e o tratamento do evento. Exemplos mais detalhados são mostrados adiante.

No que segue, adotaremos a seguinte convenção para simplificar a exposição: os sinais

definidos nos registradores de controle e de *status* serão mostrados como um sufixo ao nome do registrador: REG.sinal. Por exemplo, o bit *IV* do registrador CAUSE é indicado como CAUSE.IV.

7.3.1 Registrador *STATUS*

Os bits do registrador que nos interessam são definidos na Figura 7.9, e a definição completa deste registrador está na pág. 79 de [**?**]. Para efeitos da discussão sobre interrupções, dos 32 bits de STATUS, somente os relevantes são mostrados na Figura 7.9.

19	15	14	13	12	11	10	9	8	4	3	2	1	0
NMI	irq7	irq6	irq5	irq4	irq3	irq2	irq1	irq0	KU	0	ERL	EXL	intEn

Figura 7.9: Bits do registrador STATUS (incompleto).

O bit STATUS.NMI (*Non Maskable Interrupt*) é a interrupção de mais alta prioridade e que não pode ser mascarada. As máscaras das interrupções por *hardware* são STATUS.irq7 até STATUS.irq2, enquanto que as duas interrupções por *software* são mascaradas pelos bits irq1 e irq0 – o uso da máscara é explicado na Seção 7.3.2.

Se o bit 4 (STATUS.KU) é 1, então o processador está em modo usuário (U), se 0 o processador está em modo *kernel* (K). Se o bit 2 (STATUS.ERL) é 1, então o nível de proteção é *ERror Level* porque algum erro catastrófico no sistema foi reportado pelo *hardware*. Se o bit 1 (STATUS.EXL) é 1, então o nível de proteção é *EXception Level* e o processador está em modo privilegiado porque há uma exceção pendente. Se o bit 0 (STATUS.intEn) é 1, então as interrupções estão habilitadas.

Todos estes bits podem ser alterados por *software*; os bits KU, ERL e EXL são ligados e desligados pelo *hardware*, mas podem ser modificados por *software*.

As interrupções de 7 a 0 podem ser *mascaradas* e assim desabilitadas. Se o bit correspondente a uma das interrupções é 0, então aquela interrupção está 'mascarada' e pode ser ignorada pelo *software*. O bit STATUS.intEn desabilita todas as interrupções, exceto NMI.

7.3.2 Registrador *CAUSE*

A Figura 7.10 define os bits do registrador CAUSE que nos interessam; a definição completa deste registrador está na pág. 92 de [**?**]. Dos 32 bits de CAUSE, somente os relevantes neste contexto são mostrados na Figura 7.10.

31	23	15	14	13	12	11	10	9	8	6 a 2	1 a 0
inBds	IV	irq7	irq6	irq5	irq4	irq3	irq2	irq1	irq0	código	00

Figura 7.10: Bits do registrador CAUSE (incompleto).

O bit 31 (inBds) indica que a exceção ocorreu numa instrução que está num *branch delay-slot* e portanto o EPC foi ajustado por *hardware* para que o desvio seja executado novamente.

O bit IV, se 1, indica que o *Interrupt Vector* está alocado num endereço 0x0200 acima da base dos tratadores de exceção, e que os tratadores de interrupção estão portanto separados dos demais tratadores de exceção. Se o bit IV é 0, então o código do tratador universal deve separar as interrupções das exceções. Este bit é ligado por *software*, e a separação das interrupções das exceções melhora a eficiência do tratamento das interrupções, porque simplifica o código que decide qual é o evento a ser tratado.

Os bits 15 a 8 indicam que o sinal da interrupção correspondente está ativo e portanto aquela interrupção está pendente. Estes bits são ligados diretamente aos sinais de pedido de interrupção dos dispositivos periféricos.

Os bits de 6 a 2 contém o código da exceção. Alguns dos códigos são mostrados na Tabela 7.1. As exceções de "erro no barramento" são causadas por erros de paridade na memória ou pela demora excessiva na resposta da memória/periférico a uma requisição enviada através do barramento. A exceção de "instrução reservada" ocorre na tentativa de executar um *opcode* inválido. As exceções *syscall*, *breakpoint* e *trap* ocorrem quando as instruções `syscall`, `break` e `trap` são executadas. A exceção *overflow* ocorre quando é detectado o *overflow* na execução de uma instrução de aritmética "com sinal" (*signed*), tal como um `add` ao invés de um `addu`.

binário	dec	causa da exceção
00000	0	interrupção
00001	1	Mod, modificação de página (modificação na TLB)
00010	2	TLBL, exceção da TLB (*load* ou busca)
00011	3	TLBS, exceção da TLB (*store*)
00100	4	erro de endereçamento (*load* ou busca)
00101	5	erro de endereçamento (*store*)
00110	6	erro no barramento de instruções
00111	7	erro no barramento de dados
01000	8	*syscall*
01001	9	*breakpoint*
01010	10	instrução reservada
01100	12	*overflow*
01101	13	*trap*
11111	31	nenhuma exceção pendente

Tabela 7.1: Códigos de exceção, bits 6 a 2 do registrador CAUSE.

Exemplo 7.2 Vejamos um exemplo de uso das máscaras de interrupção. Suponha que a programadora definiu que as interrupções que podem ser atendidas, numa determinada circunstância, sejam todas *exceto* irq7 e irq5. Para tanto, o registrador STATUS deve ser alterado para desligar os bits correspondentes àquelas interrupções. O Programa 7.8 mostra os trechos de código com o mascaramento das interrupções, e o trecho do tratador com o teste para verificar se as requisições pendentes estão habilitadas.

A constante definida na linha 1 contém os bits correspondentes às interrupções 7 e 5 em zero e todos os demais em 1 (`5 = 0101`) e portanto os bits irq7 e irq5 estão em 0. O registrador STATUS é lido, e aos bits s_{15} e s_{13} são atribuídos 0s pelo `and`. O novo valor de STATUS é copiado para CP0.

No tratador da interrupção, o registrador CAUSE é lido e todos os bits, que não sejam os pedidos de interrupção, tornam-se 0 após o `andi`. O registrador `STATUS` é lido com as interrupções de nível 7 e 5 mascaradas – e o `and` garante que somente requisições irq6 e irq4-irq0 serão observadas, uma vez que irq7 e irq5 foram mascarados. ◁

Espaço em branco proposital.

Programa 7.8: Mascaramento de interrupções.

```
1    .set maskOff_7_5, 0xffff5fff # irq7=bit15=0, irq5=bit13=0
2    ...
3    li   t0, maskOff_7_5 # carrega máscara de interrupções
4    mfc0 t1, c0_status   # lê registrador STATUS
5    and  t2, t1, t0      # t2 <- STATUS AND máscara
6    mtc0 t2, c0_status   # atualiza registrador STATUS
7    ...
8  tratador:
9    ...
10   mfc0 k0, c0_cause    # lê conteúdo de CAUSE
11   andi k0, k0, 0xff00  # mantém bits de interrupção
12   mfc0 k1, c0_status   # lê conteúdo de STATUS
13   and  k1, k1, k0      # máscara apaga indesejadas
14   ...                  # escolhe interrupção a tratar
```

No "modo de compatibilidade" (*compatibility mode*) de tratamento de interrupções, a atribuição da prioridade de cada interrupção é definida pelas ligações físicas dos sinais de pedido de interrupção dos dispositivos às entradas dos sinais de interrupção no processador. O *software* deve decidir quais são as prioridades relativas dos pedidos pendentes.

No "modo vetorizado" o *hardware* decodifica pedidos simultâneos, priorizando as requisições, e então direciona o tratamento de cada nível para o endereço do tratador apropriado. Neste texto veremos somente o modo de compatibilidade.

7.3.3 Tratamento de uma interrupção

O tratamento de uma interrupção ocorre nas três fases mostradas na Figura 7.11. Inicialmente, quando o pedido de interrupção é detectado, o estado do processador se altera, e o fluxo de execução é desviado para o código que cria um ambiente de execução mínimo e então *despacha* o tratamento para a função associada ao dispositivo que está sendo atendido.

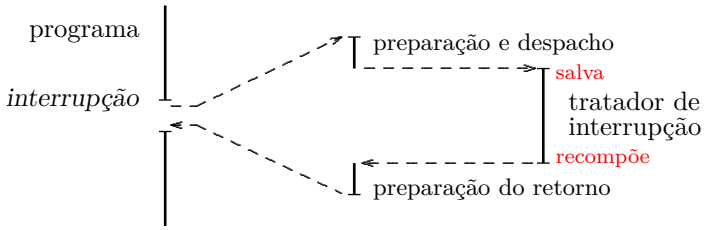


Figura 7.11: Fases da execução de um processo e um tratador de interrupção.

A rotina específica ao evento salva os registradores de que faz uso e então trata do evento. O código que trata dos eventos é tipicamente escrito numa linguagem de alto nível. Findo o tratamento, os registradores são recompostos, e o fluxo de execução retorna para o código do despachante, e a execução retorna para o programa interrompido. Esta sequência de eventos é listada abaixo em mais detalhe.

Espaço em branco proposital.

Bootstrapping cria um contexto de execução usando somente os registradores `k0` e `k1`;

despacho examina CAUSE.excCode para identificar a causa do evento;

prepara ambiente de execução salva os registradores;

tratamento trata o evento associado ao dispositivo que interrompeu;

preparação do retorno recompõe os registradores; e

retorno a instrução **`eret`** atribui o endereço de retorno ao PC (PC←EPC) e as interrupções são novamente habilitadas.

Vejamos o que ocorre com o processador durante o atendimento de uma interrupção. Suponha que o periférico associado ao sinal da interrupção número 3 (`irq3`) sinaliza um evento. A Figura 7.12 mostra as modificações que ocorrem no estado do processador durante o tratamento desta interrupção. A interrupção é detectada na busca da instrução no endereço `1014`, e a `irq3` está habilitada (STATUS.irq3=1).

A Figura 7.12 mostra as alterações nos registradores – indicadas por '*hardware*' – bem como as tarefas que devem estar codificadas no tratador de interrupções – indicadas por '*software*'. O registrador STATUS muda o processador para nível de exceção (STATUS.EXL=1) e portanto desabilita todas as interrupções. CAUSE indica que o evento excepcional é uma interrupção de nível 3, e EPC é carregado com o endereço da instrução que seria executada se não fosse a ocorrência da interrupção.

```
1010: sw   r1, 0(r2)
1014: sub  r4, r5, r6
          interrupção de nível 3
              STATUS.EXL ← 1, at exception_level hardware
              CAUSE  ← irq3=1, excCode ← interr, hardware
              EPC    ← 1014, hardware
              PC     ← endereço do tratador, hardware
                  despachante salta para o tratador de irq3, software
                  trata evento da irq3, software
                  retorna para despachante, software
              eret # STATUS.EXL ← 0, at normal_level, hardware
                   # PC ← EPC, hardware
1014: sub  r4, r5, r6
1018: xor  r7, r8, r9
101c: lw   r10, 0(r11)
```

Figura 7.12: Alterações no processador ao tratar uma interrupção.

O código do tratador da interrupção 3 trata do evento sinalizado pelo periférico e retorna para o tratador de interrupções. A instrução **`eret`** recoloca o processador em "nível normal", habilita as interrupções, e retorna a execução para o endereço `1014` – aquela instrução é buscada novamente e então executada

As interrupções somente são aceitas se STATUS.EXL = 0 e STATUS.ERL = 0 e STATUS.IE = 1. Ao aceitar uma interrupção, o CP0 faz STATUS.EXL = 1, o que automaticamente desabilita outras interrupções. Além disso, o *software* decide pelo tratamento em função da máscara que está vigente.

O Programa 7.9, adaptado da pág. 25 de [**?**], contém um trecho de código que despacha a execução para o endereço do tratador associado a cada nível de prioridade. O código foi

escrito para o caso em que CAUSE.IV=1 e portanto o endereço do código despachante de interrupções é separado do código despachante das exceções. O código pressupõe que a maior prioridade seja irq7=HW5 e que a menor seja irq1=SW0. Os bits que indicam quais as requisições estão pendentes são coletivamente chamados de *bits IP* (CAUSE.IP, *Interrupt Pending*), e os bits de máscara, definidos em STATUS são chamados de *bits IM* (STATUS.IM, *Interrupt Mask*).

A instrução **clz** (*count leading zeros*) conta o número de zeros na parte mais significativa do registrador k0, e este contém somente as requisições de interrupção aceitáveis, por causa da máscara (CAUSE.IP **and** STATUS.IM). Se a interrupção IP7 está ativa, então são 16 os bits em zero; se somente a IP0 está ativa, então são 23 os bits em zero. O **xori** transforma 16..23 em 7..0 e o **sll** ajusta os deslocamentos no endereço de cada vetor:

`10000 xor 10111 = 00111` $= 16_{10} \rightarrow 7_{10}$, e `10111 xor 10111 = 00000` $= 23_{10} \rightarrow 0$.

As rotinas de tratamento são implicitamente associadas ao periférico ligado a cada sinal (fio) de interrupção. Os registradores do periférico devem ser consultados para determinar o evento causador da interrupção, como indicado no Programa 7.4, na página 134.

As interrupções de *software* são usadas para implementar dois níveis de tratamento de eventos. No "baixo nível", próximo ao *hardware*, o tratador atende à interrupção associada ao pedido de interrupção pelo periférico (IRQ); se alguma tarefa de menor urgência é necessária para completar o tratamento, o tratador de baixo nível solicita uma interrupção "por *software*", que será atendida *depois* que o tratador da interrupção por *hardware* retornar. O tratamento desta última interrupção é de mais "alto nível" porque menos próximo ao *hardware*.

O salto na linha 21 é para o endereço de outro salto, este para o endereço do tratador do evento associado ao sinal de interrupção. A *jump table* nas linhas 28 a 43 é uma tabela com instruções que saltam para os endereços dos tratadores dos eventos.

Todos os tratadores devem terminar com uma instrução **eret** (*exception return*), para retornar o controle ao processo interrompido.

Espaço em branco proposital.

Programa 7.9: Tratador de interrupções vetorizadas em *software*.

```
1  # Software priorities are IP7..IP0 (HW5..HW0, SW1..SW0)
2
3  # define constants
4  .set M_CauseIM,0x0000ff00 # keep bits 15..8 -> IM = IP
5  .set IntCtlVS,3           # jump to handlers are 8 bytes apart
6  ...
7  IV_simple_interrupt:
8  mfc0 k0, c0_Cause         # Read Cause register for IP bits
9  mfc0 k1, c0_Status        #   and Status register for IM bits
10 andi k0, k0, M_CauseIM    # Keep only IP bits from Cause
11 and  k0, k0, k1           #   and mask with IM bits
12 beq  k0, zero, Dismiss    # No bits set - spurious interrupt
13 nop
14
15 clz  k0, k0               # Find 1st bit set IP7..IP0; k0 = 16..23
16 xori k0, k0, 0x17         # 16..23 => 7..0
17 sll  k0, k0, IntCtlVS     # Times 8 to find address of handler
18 lui  k1, %hi(VectorBase) # Get base of interrupt vector
19 ori  k1, k1, %lo(VectorBase)
20 addu k0, k0, k1           # Compute target from base and offset
21 jr   k0                   # Jump to specific exception routine
22 nop
23
24 Dismiss:                  # No pending request, maybe was noise
25 eret                      #  do nothing and return
26 ...
27              # jump table
28 VectorBase: j handlerIRQ1 # handle IP0=SW0
29             nop
30 vbPlus1x8:  j handlerIRQ2 # handle IP1=SW1
31             nop
32 vbPlus2x8:  j zzzInt      # handle IP2=HW0
33             nop
34 vbPlus3x8:  j yyyInt      # handle IP3=HW1
35             nop
36 vbPlus4x8:  j xxxInt      # handle IP4=HW2
37             nop
38 vbPlus5x8:  j counterInt  # handle IP5=HW3 counter
39             nop
40 vbPlus6x8:  j uartInt     # handle IP6=HW4 UART
41             nop
42 vbPlus7x8:  j timerInt    # handle IP7=HW5 COUNT=COMPARE
43             nop
44 ...
```

Interrupções atômicas Uma vez que a causa seja conhecida, há duas formas de tratar o evento. A forma mais simples é completar o tratamento do evento com as interrupções desabilitadas – a requisição de interrupção pelo periférico deve ser removida, possivelmente pela leitura do seu registrador de *status*. Esta forma é chamada de 'atômica' porque o tratamento da interrupção não é, ele próprio, interrompido. Veja a Figura 7.5.

O Programa 7.10 mostra o que poderia ser o tratamento de uma interrupção pelo contador da Seção 7.1, considerando que o sinal de interrupção é ligado à irq5=HW3, e portanto sua posição na tabela de saltos é `vbPlus5x8`.

Suponha que o relógio do processador é de 50 MHz (20ns) e que o contador gera uma interrupção a cada 50.000 ciclos, o que significa uma interrupção a cada milésimo de segundo. Se o tratador implementa um relógio de tempo real, então a cada mil interrupções o contador de segundos deve ser incrementado. Esta função do tratador está indicada em pseudocódigo nos comentários.

Programa 7.10: Tratador de interrupções simples, contador da Seção 7.1.

```
1  # Assumption: IP5 = HW3
2
3  # define constants
4  .set HW_cntr_addr, 0x3c000080 # Counter address
5  .set HW_cntr_value,0xc000c350 # Count 50.000 clk cycles + interr
6  ...
7
8  counterInt:                   # Handles IP2=HW0 = counter
9  lui  k0, %hi(HW_cntr_addr)
10 ori  k0, k0, %lo(HW_cntr_addr)
11 sw   zero, 0(k0)              # Reset cntr, remove interrupt request
12 lui  k1, %hi(HW_cntr_value)
13 ori  k1, k1, %lo(HW_cntr_value)
14 sw   k1, 0(k0)                # Reload counter so it starts again
15
16 # salva_registradores();
17 # microseconds += 1;
18 # if (microseconds == 1000) {
19 #    seconds += 1;
20 #    atualiza_mostrador_de_segundos;
21 # }
22 # recompõe_registradores();
23
24 eret                          # Return from the interrupt
```

A requisição de interrupção é removida pela escrita de zero no contador – uma escrita de zero reinicializa a contagem – e o novo valor é então programado no periférico. Este deve contar 50.000 pulsos do relógio e então interromper novamente.

Ao retornar, as interrupções são habilitadas novamente quando a instrução **eret** faz STATUS.EXL=0. O **eret** recarrega o PC com o valor contido em EPC, que é o endereço da instrução que teria sido executada, não fosse a interrupção.

Interrupções aninhadas A segunda forma de tratar interrupções é mais complexa mas permite que o tratamento de uma interrupção seja interrompido por outra interrupção – reveja a Figura 7.7. Para tanto, o tratador deve salvar uma parte do estado do próprio tratador e então habilitar as outras interrupções – o arquiteto do sistema define quais as interrupções que devem permanecer desabilitadas durante o tratamento de um evento, em função dos níveis relativos de prioridade. Tipicamente, somente o bit de STATUS.IM correspondente à interrupção atendida é desligado, ou então algum subconjunto de interrupções de prioridade mais baixa é desabilitado.

O Programa 7.11, adaptado da pág. 26 de [**?**], contém os trechos iniciais e finais do código do despachante de interrupções para tratar de exceções e/ou interrupções aninhadas. O atendimento a exceções aninhadas implica em salvar EPC e STATUS e os registradores de uso geral (*General Purpose Registers*, ou GPRs) que são alterados pelo tratador. Os bits de máscara apropriados de STATUS devem ser desligados para evitar um "laço de interrupções", no qual o processador fica preso atendendo à mesma interrupção. O processador

deve ser colocado em "modo normal" e as interrupções podem ser novamente habilitadas.

7.3.4 Onde salvar os registradores

Num sistema embarcado "pequeno e simples", no qual não são usados processos no modelo Unix, e portanto não ocorrem trocas de contexto mas somente interrupções, os GPRs e os registradores do CP0 podem ser salvados diretamente na pilha. Neste caso, do ponto de vista do uso da pilha, interrupções aninhadas se comportam como funções aninhadas.

Em sistemas mais complexos, nos quais são usados processos no modelo Unix, cuidados especiais devem ser tomados ao se usar a pilha para o salvamento dos registradores. Outra solução, que não seja fazer uso da pilha, aloca a cada tratador de interrupção, ou de exceção, uma área em memória para o salvamento dos registradores.

Se o tratador da irqN pode ser invocado e portanto interromper a execução de outra instância do tratador da mesma irqN, então deve-se evitar que a segunda instância sobrescreva os valores salvados pela primeira. A solução mais simples para evitar a sobrescrita é impedir que outra irqN seja atendida durante o tratamento de uma interrupção de nível N.

O documento que descreve a *Application Binary Interface* (ABI) do MIPS [**?**] define, na Seção 4 pág. 76, uma seção do formato ELF específica para os processadores MIPS denominada `.reginfo`. A primeira palavra desta seção contém uma máscara para os registradores do processador que são utilizados no código contido no arquivo ELF. O parágrafo abaixo é uma reprodução daquele documento.

> `ri_gprmask` *This member contains a bit-mask of general registers used by the program. Each set bit indicates a general integer register used by the program. Each clear bit indicates a general integer register not used by the program. For instance, bit 31 set indicates register $31 is used by the program; bit 27 clear indicates register $27 is not used by the program.*

Espaço em branco proposital.

Programa 7.11: Tratador de interrupções aninhadas.

```
1  # The sample code below cannot cover all nuances of this
2  #   processing and is intended only to demonstrate the concepts.
3
4  # define constants
5  .set NESTED_save_area,0x80008000 # Area for saving state
6  .set N_EPC_save,8        # Where to save EPC
7  .set N_STATUS_save,12    # Where to save Status
8  .set IMbitsToClear,0x0000f800 # Disable irq2..irq0
9
10 IV_nested_interrupt:
11
12 lui  k1, %hi(NESTED_save_area)     # Save GPRs here.
13 ori  k1, k1, %lo(NESTED_save_area) # This code saves only t0,t1
14 sw t0, N_t0_save(k1)
15 sw t1, N_t1_save(k1)
16 ...
17
18 move t1, k1             # t0,t1 were saved to memory
19 mfc0 t0, c0_epc         # Get restart address
20 sw   t0, N_EPC_save(t1) # Save it to memory
21 mfc0 t0, c0_Status      # Get Status value
22 sw   t0, N_STATUS_save(t1) # Save it to memory
23 lui  t1, %hi( ~IMbitsToClear )
24 ori  t1, t1, %lo( ~IMbitsToClear )
25 and  t0, t0, t1          # Clear bits in copy of Status
26 ins  t0, zero, N_Status_clr #  Clear KU,ERL,EXL bits in STATUS
27 mtc0 t0, c0_status       # Modify mask, switch to kernel mode,
28                          # re-enable interrupts (EXL <- 0)
29
30 # Registers k0,k1 might/will be overwritten by nested interrupt
31
32 # Process interrupt here, including clearing device interrupt
33
34 # To complete interrupt processing, the saved values must be
35 #   restored and the original interrupted code restarted.
36
37 di             # Disable interrupts to protect values in k0,k1
38
39 # Restore GPRs
40 lui  k1, %hi(NESTED_save_area)
41 ori  k1, k1, %lo(NESTED_save_area)
42 lw t0, N_t0_save(k1)
43 lw t1, N_t1_save(k1)
44 ...
45 lw   k0, N_STATUS_save(k1) # Get saved STATUS
46 lw   k1, N_EPC_save(k1)    # and EPC
47 mtc0 k0, c0_status         # Restore the original values of
48 mtc0 k1, c0_epc            #  STATUS and EPC
49 eret                      # Return from interrupt
```

Espaço em branco proposital.

7.4 Exceções

Uma *exceção* é similar a uma interrupção exceto que a causa é interna ao processador. Exceções são causadas por: (i) operações aritméticas tais como *overflow* ou divisão por zero; (ii) tentativa de referenciar um endereço inválido, tal como um endereço desalinhado; (iii) tentativa de referenciar um endereço cuja página física não está carregada – uma falta de página, que é definida no Capítulo 9; (iv) tentativa de executar uma instrução inválida – *opcode* ilegal ou violação de privilégio; e (v) outras situações que são discutidas nas disciplinas de Arquitetura de Computadores e Sistemas Operacionais.

O tratamento de uma exceção é similar ao de uma interrupção. A detecção ocorre durante a execução da "instrução causadora" ou da "vítima da exceção". Dependendo do evento, o processo é abortado – violação de privilégio ou endereço inválido – ou a execução é interrompida apenas temporariamente – falta de página, tratamento de *overflow*.

O valor armazenado no EPC deve ser aquele da instrução causadora, para que ela possa ser reexecutada se a exceção for benigna, como é o caso de uma falta de página. Se a exceção for maligna, o EPC aponta para a instrução que causou a terminação do processo e esta informação pode ser usada por um depurador, para localizar a causa do problema.

A Figura 7.13 mostra as alterações nos registradores do processador ao detectar uma exceção, que no exemplo é *overflow* na execução do **sub** no endereço `1014`. Os registradores STATUS, CAUSE e EPC passam a refletir o novo estado. O endereço da instrução causadora da exceção é armazenado em EPC.

Se a condição excepcional pode ser revertida – algo como transformar o *overflow* em 'infinito' – então o programa não é sumariamente terminado, e a instrução causadora é executada novamente, ao final da execução do tratador com o **eret**, que retoma a execução desde o endereço `1014`.

```
1010: sw   r1,0(r2)
1014: sub  r4, r5, r6
          overflow
              STATUS.EXL ← 1, at exception_level hardware
              CAUSE.excCode ← overflow, hardware
              EPC      ← 1014, hardware
              PC       ← endereço do tratador, hardware
                  salta para o tratador de exceções, software
                  trata evento excepcional, software
                  retorna para tratador, software
              eret # STATUS.EXL ← 0, at normal_level, hardware
                  # PC ← EPC, hardware
1014: sub  r4, r5, r6
1018: xor  r7, r8, r9
101c: lw   r10, 0(r11)
```

Figura 7.13: Alterações no processador ao tratar uma exceção.

A correção do *overflow*, neste caso, implica em decodificar a instrução e examinar seus operandos. Um dos operandos dentre `r5` ou `r6` seria alterado pelo tratador, garantindo que `(r5 - r6) < (0x8000.0000 - 1)`.

O Programa 7.12 indica como implementar o tratador de exceções. O código da exceção, mantido em CAUSE.código, é usado para indexar uma tabela com os endereços de todos os tratadores de exceções, cuja base é o endereço `handlers_tbl`. A programadora do SO é

responsável por inicializar esta tabela com os endereços das rotinas de tratamento de todos os eventos excepcionais.

As instruções nas linhas 7 a 13 computam o índice na tabela de saltos. O código da exceção CAUSE.código é um número de 5 bits deslocado de duas posições para a esquerda – alinhado portanto a um endereço de palavra, é deslocado para a esquerda, ficando alinhado ao par `j` ; `nop`, e então adicionado ao endereço da base da tabela de tratadores.

O código da exceção deslocado é o índice na tabela, o que simplifica enormemente o despacho do tratamento das exceções. A "tabela de tratadores" é uma tabela de saltos com instruções de salto para os endereços dos tratadores, organizada de acordo com a Tabela 7.1. A linha 13 salta para uma instrução `j`, que por sua vez salta para o endereço do tratador apropriado.

As exceções da TLB, ou *Translation Lookaside Buffer*, são discutidas brevemente no Capítulo 9. As exceções de erro de endereço e de erro no barramento são sinalizadas pelos controladores de memória e do barramento. A exceção *coprocessor unusable* indica que o programa tenta utilizar um coprocessador que não está instalado no sistema, tal como uma unidade de ponto flutuante (CP1 ou CP3). As exceções restantes são usadas para mudar o modo de execução do processador, e são usadas em funções de biblioteca (`syscall` e `trap`), ou durante a depuração de programas com *breakpoints* (`break`).

Espaço em branco proposital.

Programa 7.12: Tratador de exceções.

```
1  # The sample code below cannot cover all nuances of this
2  #   processing and is intended only to demonstrate the concepts.
3
4  general_excep_handler:
5  mfc0 k0, c0_cause       # Read Cause register for excpCode
6  andi k0, k0, 0x007c     #  and keep only excpCode (bits 6..2)
7  sll  k0, k0, 1          #  and adjust for 8 bytes/element
8  lui  k1, %hi(handlers_tbl) # Get addr of handler table
9  ori  k1, k1, %lo(handlers_tbl)
10 add  k1, k1, k0         #   and add it to the exception code
11 jr   k1                 # Jump to the appropriate handler
12 nop
13
14 general_excep_exit:
15 ...                     # Similar to interrupt return
16 lw   k0, N_STATUS_save(k1) # Get saved STATUS (incl EXL set)
17 lw   k1, N_EPC_save(k1)    # and EPC
18 mtc0 k0, c0_status         # Restore the original values of
19 mtc0 k1, c0_epc            #   STATUS and EPC
20 eret                       # Return from exception
21
22 .org handlers_tbl          # set table address
23
24              # jump table; see Cause.ExcCode
25 handlers_tbl: j interrupt_handlr # interrupt handlers
26              nop
27              j excp_TLBmod      # TLB modified
28              nop
29              j excp_TLBL        # TLB exception on a load
30              nop
31              j excp_TLBS        # TLB exception on a store
32              nop
33              j excp_AddrErrL    # address error on a load
34              nop
35              j excp_AddrErrS    # address error on a store
36              nop
37              j excp_IBusErr     # instruc fetch bus error
38              nop
39              j excp_DBusErr     # load/store bus error
40              nop
41              j excp_Syscall     # syscall instruction
42              nop
43              j excp_Breakpoint  # breakpoint instruction
44              nop
45              j excp_reservInstr # invalid opcode
46              nop
47              j excp_CopUnusable # coprocessor unusable
48              nop
49              j excp_Overflow    # overflow
50              nop
51              j excp_Trap        # trap instruction
52              nop
53              ...
```

Capítulo 8

Entrada e Saída – Interface Serial

R2-D2, you know better than to trust a strange computer.
C-3PO, Star Wars Ep. V: The Empire Strikes Back

Este capítulo descreve brevemente uma interface de *hardware* para a comunicação serial e apresenta os detalhes de implementação do *driver* para esta interface. A comunicação através de uma interface serial é descrita na Seção 8.1. A Seção 8.2 descreve um *driver* para a interface serial e um protocolo primitivo de comunicação que faz uso do *driver*.

O modelo de computador que usaremos para programar a interface serial é mostrado na Figura 8.1, que é similar àquele do Capítulo 7. O periférico indicado como *uart* é uma interface serial do tipo *Universal Asynchronous Receiver Transmitter*, geralmente denominada pela sua abreviatura, 'UART'.

Para que possamos estudar o que ocorre durante a comunicação serial, faremos uso de um segundo 'computador', também equipado com uma UART e que se comunica com o cMIPS. Este segundo 'computador' é chamado de *unidade remota* e seu código VHDL imita o comportamento de uma UART, similar àquela ligada ao cMIPS.

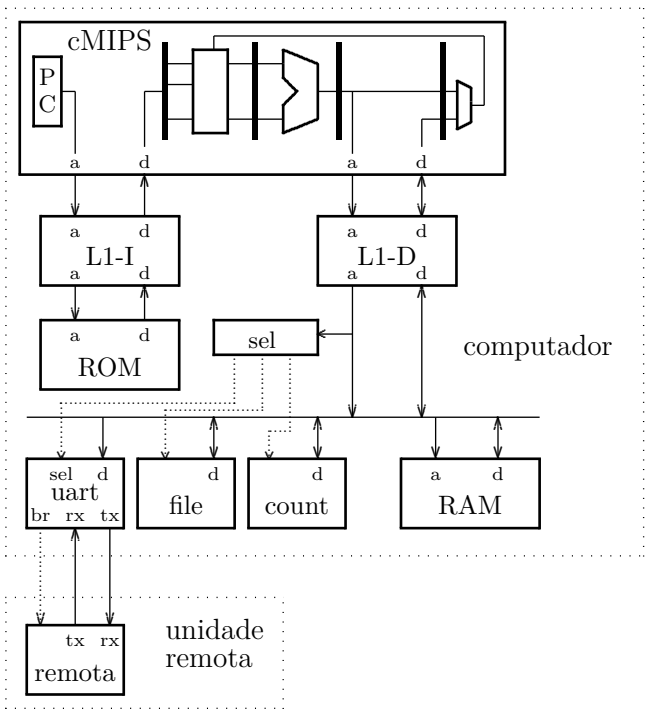


Figura 8.1: Modelo do computador com UART e unidade remota.

8.1 Interface Serial

Sempre que duas entidades se comunicam, sejam elas biológicas ou não, algum tipo de protocolo é empregado. Como vimos no Capítulo 2, o protocolo empregado por humanos na comunicação por escrito consiste de um alfabeto (conjunto de símbolos), uma gramática (conjunto de regras) e uma semântica (conjunto de significados). Se nos ocupamos com algo um tanto mais primitivo do que linguagem natural, podemos empregar versões pequenas dos três conjuntos. O menor conjunto de símbolos pode ser $\{1, 0\}$, a gramática pode conter uma dúzia de regras, e o conjunto de significados pode conter $2^8 = 256$ elementos distintos. Nesta seção estudaremos uma interface com estas características, e que é chamada de *Interface EIA 232-D*, ou *RS 232*.

Suponha que se deseje a troca de informações entre duas máquinas de baixo custo, e a interligação deve custar uma pequena fração do custo de cada uma das máquinas. Neste caso, ao invés de uma extensão do barramento de memória, que poderia ser um cabo com muitas vias – 8, 16 ou 32 vias de dados, mais os sinais de controle – estamos interessados num cabo com o menor número possível de vias, se possível, uma via só ou um só fio. No mínimo são necessários dois fios, um que transporta os sinais elétricos propriamente ditos – alguma representação elétrica para $\{1, 0\}$ – e outro que serve de referência de potencial elétrico para os sinais no primeiro fio. Se a comunicação deve ocorrer nas duas direções simultaneamente então são necessários três fios, um de referência de tensão, mais um fio para transmitir em cada direção.

A Figura 8.2 mostra os componentes de uma interface serial típica. Nas bordas estão os dois computadores que se comunicam através da interface serial. No centro estão os circuitos que convertem os sinais elétricos correspondentes aos níveis lógicos 0 e 1 para os níveis de tensão e corrente empregados para a comunicação de longa distância, e esta é a *interface elétrica* entre os dois computadores. Na figura, o meio físico de comunicação é representado por dois pares de amplificadores, conectados por fios. Nada impede que o meio de comunicação física seja um par de antenas e que o meio físico seja a atmosfera, ao invés de um par de fios de cobre.

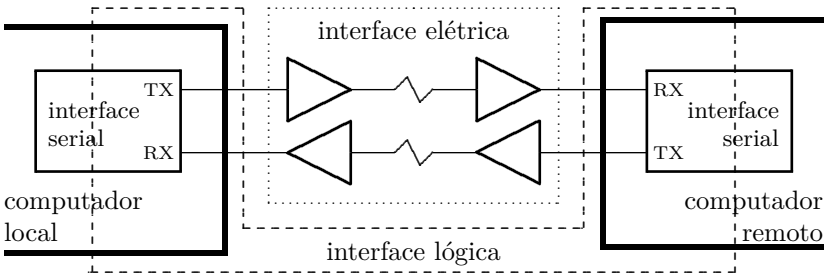


Figura 8.2: Interfaces de um sistema de comunicação serial.

É sempre uma boa prática de projeto separarmos o que pode ser separado. A interface elétrica pode ser implementada com um par de fios de cobre, fibra óptica, antenas ou sinais de fumaça. Se a *interface lógica* entre o circuito de comunicação e o computador for independente do meio físico, então a implementação dos protocolos de comunicação em *software* pode ser genérica e independente do meio físico empregado para interconectar os computadores. No nosso caso, a interface lógica é definida por um *protocolo*, ou um conjunto de definições e regras, que apenas considera bits, e ignora níveis de tensão ou corrente elétrica.

O diagrama mostra um circuito para cada direção. O circuito que transmite os bits é identificado, no computador local, como TX, que é a abreviatura para *transmissão*, e o circuito que recebe os bits é identificado como RX, para *recepção*. O mesmo circuito é

chamado de TX numa das pontas, e de RX na outra. O diagrama não mostra o fio de referência de tensão.

O circuito que implementa a Interface EIA 232 é usualmente chamado de *Universal Asynchronous Receiver Transmitter*, abreviado para *UART*. O 'universal' decorre de o circuito ser programável: o programador pode definir a velocidade de operação, o número de bits transmitidos em cada evento de comunicação, e o uso de detecção de erros, dentre outros parâmetros. A 'assincronia' é uma característica da comunicação: no lado do receptor, não se pode prever o instante exato em que um evento de comunicação ocorrerá; além disso, os circuitos de relógio do transmissor e do receptor operam de forma independente e possivelmente com frequências ligeiramente distintas. O *'receiver transmitter'* indica que o circuito da interface é capaz de efetuar operações de recepção e de transmissão simultâneas.

8.1.1 Comunicação Serial

Inicialmente, consideremos a comunicação unidirecional. A comunicação entre as duas máquinas se dá quando uma certa área de memória do remetente – que é a *mensagem* por transmitir – é copiada para a memória do destinatário. No caso da comunicação serial, são necessários uma interface de comunicação serial em cada máquina e o cabo que interconecta as duas interfaces. Um *enlace* de comunicação é composto (i) pelas duas interfaces –uma no transmissor e outra no receptor, (ii) pelo meio físico –cabo de interligação, par de antenas, e (iii) pelo *software* de comunicação que é executado nos computadores nas duas pontas do enlace.

Se a interligação pode ser feita com dois fios, como é efetuada a cópia dos dados da memória do primeiro computador para a do segundo? Dados são armazenados em formato paralelo na memória, como bytes ou palavras de 32 ou 64 bits de largura. Como é que se transmite um octeto através de um único fio? A solução é *transmitir um bit de cada vez.*

O circuito de transmissão é um registrador de deslocamento que é carregado em paralelo pelo processador e cujo conteúdo é deslocado bit a bit para a saída serial. O circuito de recepção é um registrador de deslocamento, que amostra os bits pela entrada serial e é acessado em paralelo pelo processador.

Alguns parâmetros devem ser fixados de antemão para que o esquema delineado acima funcione corretamente. Primeiro, a duração de cada bit deve ser a mesma no sistema que envia e no que recebe. Isso é necessário para que as máquinas de estado nas duas pontas do enlace trabalhem sincronizadas e que durante um evento de comunicação, as duas pontas estejam de acordo sobre qual bit de qual octeto está sendo transferido de um computador para o outro.

Segundo, o número de bits transferidos a cada evento deve ser pré-definido para que o receptor consiga recuperar toda a informação contida nos bits transmitidos. Do contrário, não há como garantir que a informação emitida pelo transmissor seja reconhecida pelo receptor.

A duração de cada bit é determinada pela *velocidade de transmissão.* As velocidades definidas pelo padrão EIA 232 são múltiplos de 75 bits por segundo (75, 150, 300, 600, 1200, 2400, 4800, 9600, 19200). A unidade da velocidade de transmissão é *bits por segundo*, abreviada *bps.* Assim, para transmissões a 9600 bps, cada bit dura 1/9600 segundos, e este tempo é chamado de *tempo de bit* ou *intervalo de bit.* A unidade de transferência em interfaces seriais padrão EIA 232 é um caractere, tipicamente representado em 8 bits, embora 5 e 7 bits também sejam usados.

A Figura 8.3 mostra os registradores de deslocamento na interface serial do computador

transmissor e na interface do receptor. No transmissor, um registrador faz a conversão do formato paralelo (octeto) para o serial (um bit por vez), enquanto que no receptor é usado um registrador que faz a conversão do formato serial (um bit) para o paralelo (um octeto).

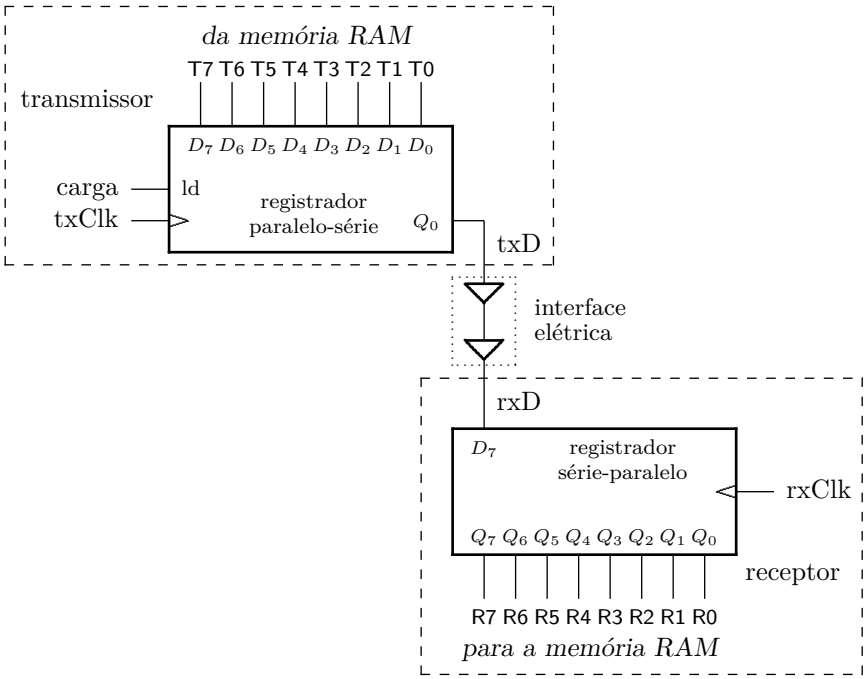


Figura 8.3: Registradores de transmissão e de recepção.

Quando o transmissor deseja enviar um caractere, o octeto por enviar é apresentado pelo processador nas linhas T7-T0, o sinal *carga* é ativado, e no primeiro pulso do *relógio de transmissão* (*txClk*), o valor do bit T0 é apresentado na saída Q_0 do registrador de transmissão, e portanto no sinal *txD* que é o fio que transporta os bits do transmissor até o receptor. O relógio no transmissor gera mais 7 pulsos no sinal *txClk* para transferir os bits T1 a T7 para a saída Q_0, e através da interface elétrica, para o receptor.

A taxa de transmissão, definida pelo relógio de transmissão (*txClk*), é determinada através de algum outro mecanismo de comunicação, e o acordo quanto à velocidade deve ser efetivado por carta, por uma conversa ao telefone, ou por um mensageiro humano. O mecanismo de definição de velocidade é dito "fora de banda" (*out of band*) porque a decisão ocorre "por fora" do sistema de comunicação.

No receptor, o sinal *rxClk* é usado para amostrar o valor recebido no sinal *rxD*. A cada borda do relógio de recepção *rxClk*, um novo bit é amostrado na entrada serial (D_7) do registrador série-paralelo, e os bits amostrados anteriormente são deslocados para a direita. Após 8 ciclos de *rxClk*, os bits R7-R0 contém uma cópia do valor que foi carregado no registrador de transmissão, e este octeto é então gravado na memória do computador.

Por convenção, o primeiro bit transmitido é o bit D_0 e ao final da sequência de transmissão, o valor do bit D_0 emitido pelo transmissor está na posição mais à direita no registrador do receptor, também na posição correspondente ao bit D_0. As frequências dos relógios de transmissão e de recepção (*rxClk* e *txClk*) não podem diferir em mais de 10% porque então o receptor pode amostrar o último bit fora do intervalo correspondente àquele bit. Em equipamentos de baixo custo, os relógios operam com frequências que diferem de ± 1 a $\pm 2\%$ da frequência nominal. O Exercício 55 explora esta questão.

A Figura 8.4 contém um diagrama de tempo com a transmissão de um caractere no circuito descrito nos parágrafos anteriores. Na figura, o octeto `0x6a` (`0110.1010`) é transmitido,

sendo o bit menos significativo (B_0) enviado primeiro. A cada borda ascendente de *txClk*, um novo bit é deslocado para a direita e exibido em *txD*. O sinal *rxClk* amostra o sinal *rxD* na entrada D_7 do registrador de recepção, meio ciclo mais tarde.

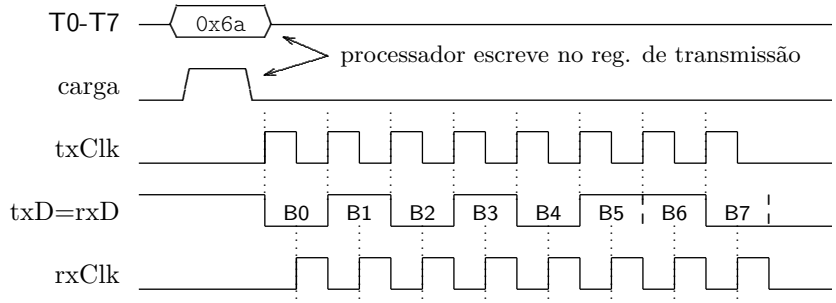


Figura 8.4: Diagrama de tempo da transmissão de um octeto.

A melhor posição para a amostragem pelo receptor é no meio do intervalo de bit porque isso garante que o sinal já estabilizou no nível de tensão que corresponde ao bit transmitido. O enlace de comunicação se comporta como uma antena que capta sinais elétricos externos à comunicação e os adiciona ao nível elétrico que corresponde aos bits sendo transmitidos. O ruído pode ser tal que um bit em 1 seja percebido como um 0, ou um 0 seja percebido como 1. Além disso, o enlace se comporta como uma corda grossa e pesada: no transmissor, as fronteiras de um bit são 'quadradas', enquanto que no receptor, as bordas dos bits são distorcidas pelas características físicas do meio de comunicação. Como um teste, amarre uma corda num ponto fixo, e na outra ponta da corda efetue movimentos verticais, representando 1s e 0s alternados. Qual o tipo de movimento que se observa próximo da ponta fixa da corda? Faça um nó no meio da corda; o que acontece com o formato da onda perto do final da corda?

Se a amostragem pelo receptor fosse no mesmo instante em que o transmissor emite um novo bit, seria difícil discriminar o valor amostrado por causa da transição de um nível para outro, que não é instantânea. Se a mudança de nível no sinal não for perfeita como mostrado na Figura 8.4 – e na prática nunca o é – o receptor poderia amostrar um valor errado para todos os bits, exceto possivelmente no bit B6, que tem o mesmo valor que B5.

Há um problema sério com este projeto: como o receptor descobre que um novo octeto está sendo recebido? Considere a transmissão de uma longa sequência de octetos com todos os bits em 0 ou em 1. Como o receptor garante que está amostrando um octeto? Lembre que não há um sinal que transporta nenhuma informação de sincronismo entre transmissor e receptor.

O início e o final de um octeto devem ser demarcados claramente para evitar ambiguidade na recepção de dois octetos transmitidos em seguida. Isso é obtido acrescentando-se uma *moldura* ao octeto, como se ele fosse uma fotografia. Em repouso, a linha permanece sempre em nível lógico 1. O início de um octeto é demarcado por um bit em 0, chamado de *bit de início* (*start bit*). O final de um octeto é demarcado por um bit em 1, chamado de *bit de final* (*stop bit*), o que coloca a linha em repouso. A máquina de estados do circuito de transmissão se encarrega de acrescentar os bits de início e de final a cada octeto transmitido. A Figura 8.5 mostra o diagrama de tempo da transmissão do octeto `0x6a` enquadrado pelos bits de início e de final.

Após um intervalo em repouso, o circuito de recepção espera pela borda descendente em *rxD*, o que demarca o início de um novo octeto. Após a borda do bit de início, o receptor espera meio intervalo de bit, e passa a amostrar os bits recebidos. O receptor descarta os bits de início e de final e entrega somente o octeto recebido ao processador do lado do receptor. O diagrama de tempo mostra uma certa distorção nos sinais em *rxD*, para

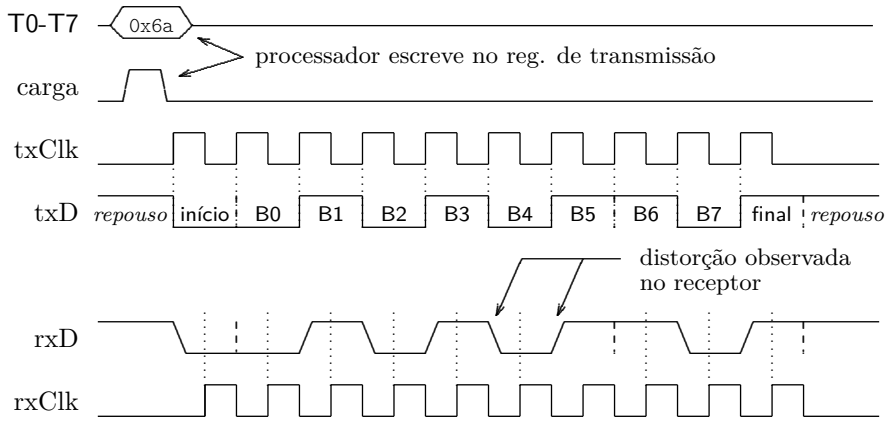


Figura 8.5: Enquadramento de um caractere para transmissão serial.

enfatizar a necessidade de amostragem no meio do intervalo de bit.

A máquina de estados do circuito de transmissão é mostrada na Figura 8.6. A ME fica a esperar que o processador carregue um novo octeto no registrador de deslocamento ($\overline{carga}$). A ME então insere um bit de início, conta oito ciclos para dar tempo à transmissão dos oito bits de dados, insere um bit de final, e volta a esperar pela carga do próximo octeto por transmitir.

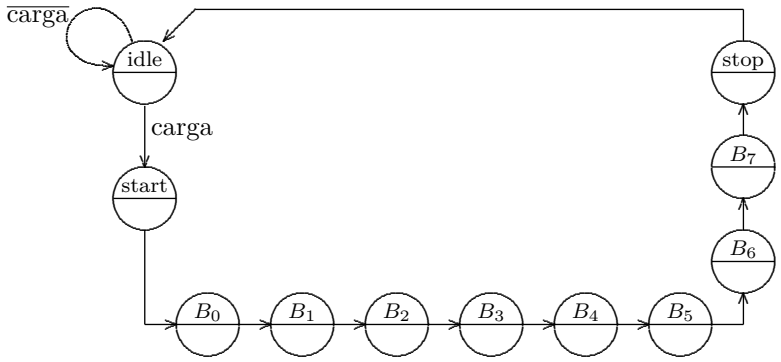


Figura 8.6: Máquina de estados da transmissão.

O controle da recepção é um pouco mais complexo por causa da detecção do bit de início. A ME deve detectar a borda descentende na linha de dados e então esperar durante meio intervalo de bit (*st1*), quando então a linha de dados pode ser amostrada pelo registrador de recepção. Se o sinal amostrado permanece em zero, então ocorreu um bit de início e o receptor deve iniciar a amostragem dos dados depois de decorrido um tempo de bit (*st2*). Se o sinal não é zero, então ocorreu um "falso início" e a ME retorna ao estado em que espera pelo bit de início.

Confirmado o início, a ME conta os 8 intervalos de bit para amostrar o octeto e espera para garantir que um novo bit de início não seja amostrado antes do término do bit de final. O diagrama de estados é mostrado na Figura 8.7, em versão simplificada porque o estado *st1* dura meio intervalo de bit, enquanto que os demais duram um intervalo completo.

Até aqui, nosso protocolo contém as seguintes definições:

(1) são transferidos oito bits a cada evento de comunicação;
(2) o bit B_0 é o primeiro bit transmitido;
(3) cada octeto é precedido de um bit de início e seguido de um bit de final;
(4) a taxa de transmissão (*txClk*=*rxClk*) é definida de alguma forma *externa* ao sistema de comunicação;

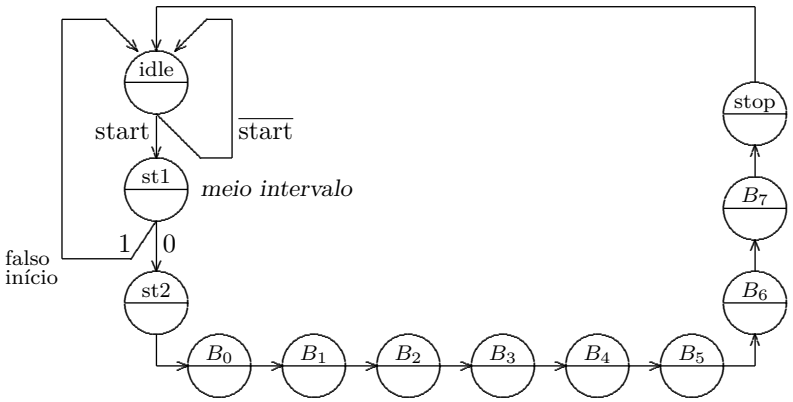


Figura 8.7: Máquina de estados da recepção.

(5) um octeto pode ser enviado a qualquer tempo.

O protocolo descrito acima é conhecido como *serial assíncrono* e partes dele são definidas pelo padrão EIA 232-D, que era o protocolo usado na interface serial de computadores pessoais, antes do advento das interfaces USB.

Este protocolo é dito *assíncrono* porque cada caractere é transmitido independentemente dos demais, o que permite acomodar ligeiras diferenças de frequência entre os relógios de transmissor e receptor, causadas pelo uso de geradores de relógio de baixo custo. Protocolos *síncronos* necessitam que alguma informação de sincronismo seja transferida juntamente com os caracteres, o que possibilita a transmissão de longas sequências de caracteres sem bits de início ou de final a cada caractere. Protocolos síncronos são mais eficientes que os assíncronos mas são um tanto mais complexos.

Este protocolo assíncrono não é muito eficiente: para transmitir oito bits com informação, são necessários dois bits adicionais para enquadramento. A eficiência de transmissão é de $8/10 = 0,8$, ou 80%. Esta eficiência é aceitável por causa da simplicidade dos circuitos de dados e de controle, e da implementação com geradores de sincronismo de baixo custo.

Exercícios

Ex. 55 Considerando que um caractere é transmitido em 10 bits (bits de início e de final, mais um octeto), estime qual é a diferença máxima aceitável para as frequências dos relógios de transmissão e de recepção, visando minimizar os erros de amostragem no circuito de recepção – *rxClk* amostra os últimos bits cedo demais, ou tarde demais.

Ex. 56 Na descrição do circuito de recepção é dito que o receptor espera meio intervalo de bit após a borda do bit de início. Isso implica em que o relógio da máquina de estados do circuito receptor deve operar com uma velocidade que é, no mínimo, o dobro da velocidade do relógio de recepção. Supondo que o relógio da máquina de estados opera numa velocidade oito vezes maior que a velocidade de recepção, projete uma máquina de estados para o circuito receptor que detecta a borda do bit de início e então passa a amostrar os bits recebidos no meio do intervalo de bit, como mostra o diagrama de tempos na Figura 8.5.

Ex. 57 Suponha que interferência eletromagnética cause um pulso de ruído elétrico de curta duração, quando nada está sendo transmitido e os sinais estão no nível de repouso. O receptor pode interpretar o pulso de ruído como se fosse um bit de início. O que pode ser feito para detectar a ocorrência de *falsos bits de início*? *Pista 1*: se o bit de início é

verdadeiro, então o bit de final *deve* ser 1, do contrário ocorreu um erro de "início falso"; *pista 2*: veja sua solução para o Ex. 56.

Ex. 58 O circuito descrito nesta seção permite a comunicação numa única direção. O que é necessário para permitir a comunicação nas duas direções ao mesmo tempo, no que é chamado de *transmissão dúplex*, com 3 fios: transmissão, recepção, e referência de tensão? Repita, considerando comunicação nas duas direções alternadamente, no que é conhecido como *transmissão semidúplex*, com somente 2 fios: um para transmissão/recepção, e outro para referência de tensão.

Ex. 59 A paridade de uma sequência de bits é a contagem de bits em 1 na sequência. Se a paridade é *ímpar* em uma sequência com $n-1$ bits, então o bit de paridade, que é o n-ésimo bit, deve ser tal que o número de bits em 1 na sequência seja ímpar. Projete uma máquina de estados que computa a paridade ímpar de uma sequência de bits. Esta máquina de estados possui três entradas e uma saída: a entrada *din* recebe a sequência de bits cuja paridade deve ser computada. A entrada *rel* é o sinal que cadencia a operação do circuito. A saída *parid* contém a paridade ímpar da sequência apresentada em *din* até o ciclo anterior de *rel*. A entrada *reset* coloca a saída em zero. (a) Desenhe um diagrama de tempos mostrando a operação do circuito. (b) Desenhe um diagrama de blocos com o circuito e explique seu funcionamento.

Ex. 60 Usando o circuito do Ex. 59, acrescente ao projeto da interface serial um detector de erros de paridade. O circuito de transmissão computa a paridade do caractere por enviar e transmite o bit de paridade computado após o oitavo bit. O circuito receptor re-computa a paridade do caractere recebido e compara o valor que computou com o bit de paridade recebido; se os dois forem iguais, não houve erro de transmissão; se forem distintos, o erro é sinalizado nos bits apropriados do registrador de *status*.

8.1.2 Ligação ao Processador

Até o momento, a discussão se concentrou no funcionamento dos circuitos de transmissão e recepção da interface de comunicação serial. Os dois registradores de deslocamento (paralelo-série e série-paralelo) e o circuito de controle são normalmente encapsulados em um único bloco funcional, usualmente chamado de *Universal Asynchronous Receiver-Transmitter* (UART). O 'universal' é devido à versatilidade do dispositivo, que pode operar em um de vários modos diferentes, com as opções selecionadas pelo programador. Além dos registradores de deslocamento, uma UART possui um registrador de controle que mantém o modo de operação selecionado pelo programador, e um registrador de *status* que indica a ocorrência de erros, tais como paridade.

A Figura 8.8 mostra um diagrama de blocos da UART. No lado esquerdo do diagrama é mostrada a interface entre UART e processador. As oito linhas de dados ($D_{7..0}$) permitem a carga e a leitura dos registradores de deslocamento da UART pelo processador. O sinal *csIser* (*chip select*) habilita o acesso ao periférico pelo processador. Para o programa obter acesso aos registradores da UART basta executar instruções de leitura ou escrita nos endereços associados à porta serial. O circuito de seleção dos periféricos ativa o sinal *csIser* sempre que o endereço da UART for emitido pelo processador. O sinal de escrita (*wr*), emitido pelo processador, determina a direção da referência e não é mostrado no diagrama.

Quando *csIser* está ativo, as linhas de endereço ($E_{1,0}$) e o sinal *wr* escolhem um dos cinco registradores de acordo com a Tabela 8.1. Em geral, quando o computador é inicializado, o modo de operação da UART (velocidade) é programado no registrador de controle (ctrl) e, se interrupções serão usadas, o registrador de interrupção (interr) também é programado. Os registradores de controle e de interrupção podem ser lidos e escritos; o registrador de

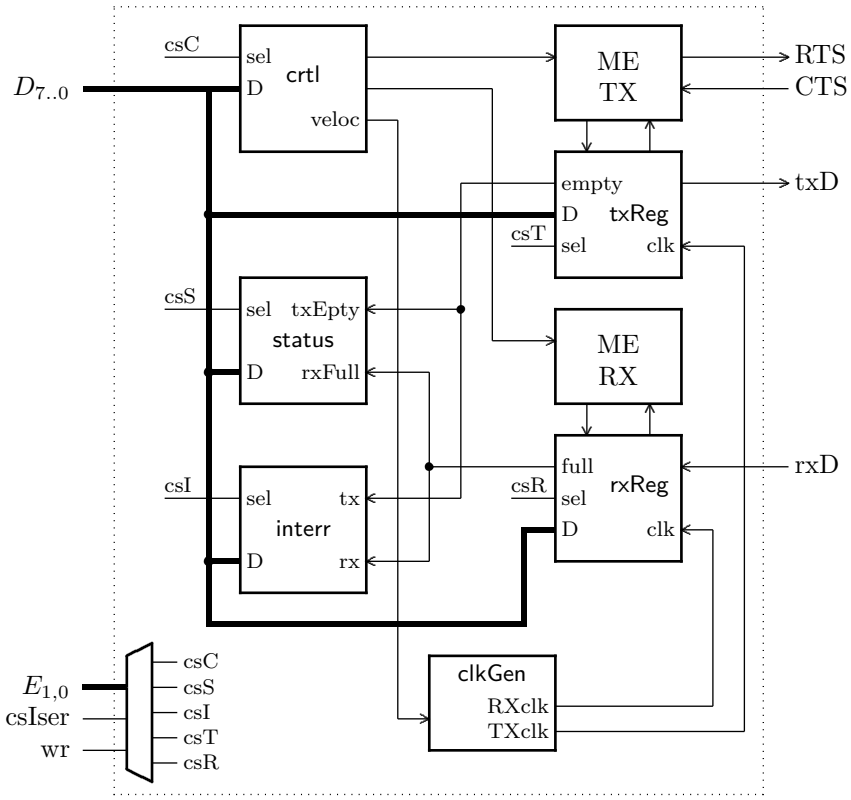


Figura 8.8: Sinais da interface com processador.

status pode ser lido mas não alterado; o registrador de transmissão pode ser escrito e o registrador de recepção pode ser lido.

Durante a transmissão de uma mensagem, os registradores de *status* e de transmissão (txreg) são acessados frequentemente; o mesmo vale para a recepção de uma mensagem, embora neste caso o registrador de recepção (rxreg) é acessado com frequência, juntamente com o registrador de *status*. A Seção 8.2 mostra exemplos de código para a transmissão e a recepção de mensagens.

Tabela 8.1: Endereçamento dos registradores da interface serial.

csIser	E_1E_0	wr	registrador
0	X	X	*desconectados*
1	0 0	X	controle (ctrl)
1	0 1	0	*status*
1	1 0	X	interrupção (interr)
1	1 1	0	transmissão (txreg)
1	1 1	1	recepção (rxreg)

Quando o processador escreve um octeto no registrador de transmissão, com

$$csIser \wedge E_0 \wedge E_1 \wedge \overline{wr},$$

os oito bits são emitidos um por vez na saída *txD* da UART. Após a emissão do bit de final, o bit *txEmpty* do registrador de *status* fica ativo, para indicar que um novo octeto pode ser gravado pelo processador no registrador txreg.

Quando um octeto é recebido através da linha *rxD*, o bit *rxFull* do registrador de *ststus* indica que o octeto está disponível e que o registrador rxreg deve ser lido pelo processador

com

$$csIser \wedge E_0 \wedge E_1 \wedge wr\,.$$

O registrador de *status* contém um bit (*txEmpty*) que informa se há espaço no registrador de transmissão, e um bit (*rxFull*), que informa se há um novo caractere no registrador de recepção. O registrador de *status* também aponta eventuais erros que sejam detectados pelo circuito de recepção. Os bits do registrador *status* são definidos na Seção 8.2.

A Figura 8.8 contém um diagrama da versão (quase) completa da UART. Além dos registradores de transmissão e recepção, A UART contém registradores de *status*, de controle, e de interrupção. A Figura 8.8 mostra ainda o circuito de geração dos sinais de relógio para a recepção e a transmissão. A velocidade de operação da UART pode ser programada gravando-se os valores adequados nos bits $c_{2..0}$ do registrador de controle. O circuito de relógio é um um contador programável.

8.2 Modelo da UART

A Figura 8.9 mostra um diagrama de blocos da nossa UART. A interface com o processador é mostrada no topo da figura e consiste de cinco registradores: interrupções (interr), controle (ctrl), *status*, transmissão (txreg) e recepção (rxreg). Os nomes destes registradores estão grafados sem serifa. O circuito de transmissão está representado em azul, e o de recepção em vermelho. A largura dos caminhos de dados está indicada junto ao traço diagonal que 'corta' os fios.

O circuito de transmissão consiste do registrador de transmissão *txreg*, do registrador de deslocamento *transmit*, de um contador que gera o sinal de relógio de transmissão (*bit_rt_tx*), e de duas máquinas de estado, uma que controla a transmissão serial (*TX SM*) e outra que faz a interface com o barramento de dados (*TXCPU SM*). Esta última implementa o mecanismo de *double buffering* na transmissão: os registradores *txreg* e *transmit* se comportam como uma fila com duas posições. Se o registrador *transmit* está vazio, um octeto (8 bits) escrito pelo processador em *txreg* é imediatamente transferido para *transmit*, assim 'esvaziando' *txreg*, que fica disponível para uma nova escrita pelo processador.

O circuito de recepção consiste do registrador *rxreg* e do registrador de deslocamento *receive*, um gerador de relógio de recepção (*bit_rt_rx*) e duas máquinas de estado, uma para controlar o circuito de recepção (*RX SM*) e outra que faz a interface com o barramento de dados (*RXCPU SM*). Esta última implementa o mecanismo de *double buffering* na recepção, como uma fila de duas posições: assim que um octeto é recebido pela linha *rxdat*, ele é transferido para rxreg, em preparação para a chegada do próximo octeto.

Para transmitir um octeto, o processador deve gravá-lo no registrador txreg, e então as máquinas de estado da transmissão o transmitem pela linha de saída *txdat*. Quando um octeto é recebido pela linha de entrada *rxdat*, ele é transferido para o registrador rxreg, de onde pode ser lido pelo processador.

A interface da UART com o processador se dá através do barramento de dados, que tem 32 bits de largura, mas somente o byte menos significativo é usado na interface da UART. Para garantir que o acesso pelo processador seja alinhado com o byte menos significativo do barramento – onde estão alocados os registradores da UART – as estruturas de dados que modelam os registradores contém três bytes de enchimento, mais o byte com o conteúdo dos registradores, como mostra a Figura 8.10.

Espaço em branco proposital.

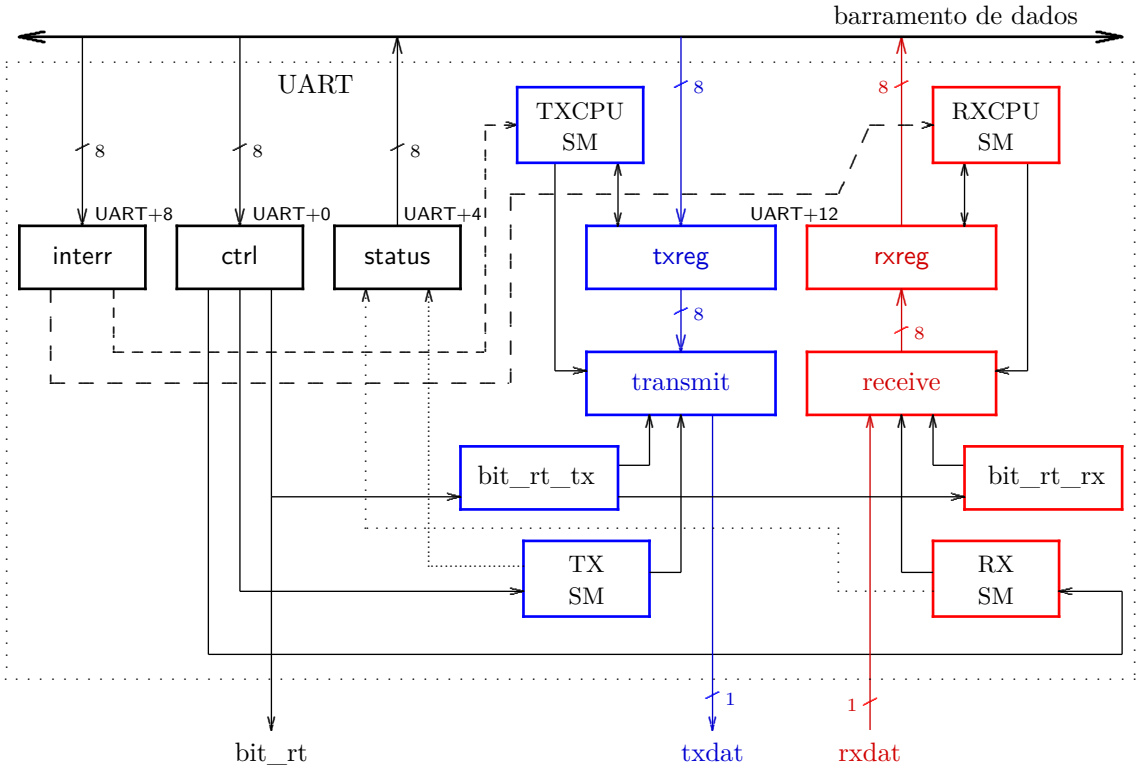


Figura 8.9: Diagrama de blocos da UART.

bits:	31..24	23..16	15..8	7 ..	0
uso:	ignorados			registrador	

Figura 8.10: Alinhamento dos registradores da UART no barramento.

Registrador de controle A Figura 8.11 mostra o modelo de programação do registrador de controle. Os bits 2 a 0 (c_2, c_1, c_0) programam a velocidade de transmissão e recepção da UART[1], ou sua taxa de sinalização, ou sua *Baud rate*, ou ainda a sua *bit rate*. Quando $c_2, c_1, c_0 = 000$, a frequência de transmissão e recepção é 1/4 da frequência do relógio do processador. A cada incremento no número representado por estes bits, a taxa de sinalização é reduzida pela metade.

bit:	7	6..3	2..0
	rts	ign_4	*bit_rate*

Figura 8.11: Registrador de controle (ctrl).

Os bits c_3 a c_6 são ignorados pelo controlador da UART. O bit c_7 ativa o sinal *request to send*, que controla a interface entre computador e modem, indicando ao modem que o computador está prestes a transmitir. O conteúdo do registrador de controle pode ser lido pelo processador.

O Programa 8.1 contém a estrutura de dados que corresponde ao registrador de controle. A estrutura `Tcontrol` é uma representação exata do registrador de controle.

[1]Para simplificar os testes, a velocidade de transmissão é excepcionalmente elevada para uso com interfaces seriais – do contrário, a transmissão de um octeto demoraria 10^3 a 10^4 ciclos do processador.

Programa 8.1: Estrutura de dados para acessar ctrl.

```
1 typedef struct control { // control register fields, ls byte
2   unsigned int ign : 24, // ignore uppermost 3 bytes (bits 8..31)
3     rts     : 1,         // Request to Send output (bit 7)
4     ign4    : 4,         // bits 6,5,4,3 ignored
5     speed   : 3;         // 1/4,1/8,1/16..data rates (bits 2,1,0)
6 } Tcontrol;
```

Registrador de *status* A Figura 8.12 mostra o modelo de programação do registrador de *status* da UART. O bit $s_0 = 1$ indica a ocorrência de um erro de *overrun* – o octeto recebido anteriormente foi sobrescrito pelo último octeto recebido. O bit $s_1 = 1$ indica um erro de enquadramento – o último octeto recebido não estava 'enquadrado' por bits de *start* e de *stop* – o que pode indicar a ocorrência de um "falso *start*" provocado por ruído eletromagnético, ou que as duas pontas do enlace trabalham com velocidades diferentes.

bit:	7	6	5	4	3	2	1	0
	cts	txEmpty	rxFull	intTX	intRX	ign	*framing*	*overrun*

Figura 8.12: Registrador de *status* (status).

O bit s_2 é sempre zero. O bit $s_3 = 1$ indica que há uma interrupção de recepção pendente, e o bit $s_4 = 1$ indica que há uma interrupção de transmissão pendente. Estes bits são ativados se os bits i_3, i_4 de interr forem programados em 1 na inicialização da UART, ou se os bits i_5, i_6 foram ativados através de um *store* em interr. O bit s_7 indica que o modem ativou o sinal *clear to send* e portanto está pronto para transmitir os dados emitidos pelo computador. O bit $s_5 = 1$ indica que o registrador de recepção rxreg está cheio, e o bit $s_6 = 1$ indica que o registrador de transmissão txreg está vazio. Quando o registrador status é lido pelo processador, os bits s_0 e s_1 mudam para 0.

O Programa 8.2 mostra a estrutura que corresponde ao registrador de `status`.

Programa 8.2: Estrutura de dados para acessar status.

```
1  typedef struct status {  // status register fields, ls byte
2    unsigned int ign : 24, // ignore 3 MS bytes (bits 8..31)
3      cts     : 1,         // Clear To Send input=1 (bit 7)
4      txEmpty : 1,         // TX register is empty (bit 6)
5      rxFull  : 1,         // RX register is full (bit 5)
6      int_TX  : 1,         // inter pending on TXempty (bit 4)
7      int_RX  : 1,         // inter pending on RXfull (bit 3)
8      ign1    : 1,         // ignored (bit 2)
9      framing : 1,         // framing error (bit 1)
10     overrun : 1;         // overrun error (bit 0)
11 } Tstatus;
```

Registrador de interrupção Há ainda um registrador para controlar os bits de pedido de interrupção. Este registrador tem dois bits que permitem ativar, e dois que permitem desativar os pedidos de interrupção de recepção e de transmissão, separadamente. Quando ocorre uma escrita em interr, e o bit $i_6 = 1$, ocorre uma interrupção de transmissão; com bit $i_5 = 1$, ocorre uma interrupção de recepção; se o bit $i_4 = 1$, então o sinal de interrupção de transmissão é desativado na UART; se o bit $i_3 = 1$, então o sinal de interrupção de recepção é desativado na UART.

O bit $i_1 = 1$ programa a geração de uma interrupção quando o registrador de transmissão txreg está vazio e um novo octeto pode ser transmitido. O bit $i_0 = 1$ programa uma interrupção quando há um octeto novo no registrador de recepção rxreg e aquele deve ser lido pelo processador.

Ao ser lido, o registrador de interrupção retorna somente os valores dos bits `intTX` e `intRX`: $0000.00i_1i_0$. Sempre que alguma das interrupções for ativada, ou desativada, o registrador deve ser lido, e então atualizado com um (ou mais) dentre `setTX`, `setRX`, `clrTX`, ou `clrTX` ativados, *mantendo* os valores originais de i_1i_0.

bit:	7	6	5	4	3	2	1	0
	igna	setTX	setRX	clrTX	clrRX	ignb	intTX	intRX

Figura 8.13: Registrador de interrupções (interr).

O Programa 8.3 mostra a estrutura que corresponde ao registrador de interrupções (interr).

Programa 8.3: Estrutura de dados para acessar *interr*.

```
 1 typedef struct interr {  // status register fields, ls byte
 2   unsigned int ign : 24, // ignore 3 MS bytes (bits 8..31)
 3     igna   : 1,          // ignored (bit 7)
 4     setTX  : 1,          // set IRQ on TXempty (bit 6)
 5     setRX  : 1,          // set IRQ on RXfull (bit 5)
 6     clrTX  : 1,          // clear IRQ on TXempty (bit 4)
 7     clrRX  : 1,          // clear IRQ on RXfull (bit 3)
 8     ignb   : 1;          // ignored (bit 2)
 9     intTX  : 1,          // do interrupt on TXempty (bit 1)
10     intRX  : 1,          // do interrupt on RXfull (bit 0)
11 } Tinterr;
```

Registrador de dados Octetos gravados no registrador txreg são transmitidos pela interface serial. A leitura do registrador rxreg retorna o último octeto recebido pela interface serial.

8.2.1 Entrada e saída por programa

O Programa 8.4 contém as estruturas de dados que correspondem aos registradores da UART. As estruturas `Tcontrol`, `Tstatus` e `Tinterr` estão nas linhas 3 a 32. A estrutura `Tserial` reflete o leiaute dos registradores, como se eles fossem uma 'memória' de 4 palavras.

O registrador de controle está no endereço base do periférico (`UART_BASE_ADDR`), e é um registrador que pode ser atualizado e lido. O registrador de *status* está no endereço seguinte (`UART_BASE_ADDR+4`) e escritas neste endereço são ignoradas. O registrador de controle das interrupções está no endereço (`UART_BASE_ADDR+8`) e sua leitura retorna zero. O registrador de dados está no endereço `UART_BASE_ADDR+12`; quando lido, o byte menos significativo é uma cópia de rxreg; nas escritas, o octeto no byte menos significativo é gravado no registrador txreg.

A modelagem do periférico com estruturas de dados como estas permite que quase todo o código para acessá-lo seja escrito em linguagem de alto nível, que é C neste caso. Apenas partes do tratador de interrupções devem ser escritas em *assembly*.

Programa 8.4: Estruturas de dados para acessar a UART.

```
1  #include "cMIPS.h"
2
3  typedef struct control { // control register fields, only ls byte
4    unsigned int ign : 24, // ignore uppermost 3 bytes
5      rts     : 1,         // Request to Send output (bit 7)
6      ign4    : 4,         // bits 6,5,4,3 ignored
7      speed   : 3;         // 4,8,16.. tx,rx-clockRate (bits 2,1,0)
8  } Tcontrol;
9
10 typedef struct status {  // status register fields, only ls byte
11   unsigned int ign : 24, // ignore uppermost 3 bytes
12     cts     : 1,         // Clear To Send input=1 (bit 7)
13     txEmpty : 1,         // TX register is empty (bit 6)
14     rxFull  : 1,         // new octet on RX register (bit 5)
15     intTX   : 1,         // interr pending on TX empty (bit 4)
16     intRX   : 1,         // interr pending on RX full (bit 3)
17     ign1    : 1,         // ignored (bit 2)
18     framing : 1,         // framing error (bit 1)
19     overrun : 1;         // overrun error (bit 0)
20 } Tstatus;
21
22 typedef struct intreg  { // interrupt clear bits, only ls byte
23   unsigned int ign : 24, // ignore uppermost 3 bytes
24     igna    : 1,         // bit 7 ignored
25     setTX   : 1,         // set   IRQ on TX buffer empty (bit 6)
26     setRX   : 1,         // set   IRQ on RX buffer full  (bit 5)
27     clrTX   : 1,         // clear IRQ on TX buffer empty (bit 4)
28     clrRX   : 1,         // clear IRQ on RX buffer full  (bit 3)
29     ignb    : 1,         // bit 2 ignored
30     intTX   : 1,         // do interrupt on TX empty (bit 1)
31     intRX   : 1;         // do interrupt on RX full  (bit 0)
32
33 } Tintreg;
34
35 typedef union interr {
36   Tintreg      s;        // structure bits
37   unsigned int i;        // integer
38 } Tinterr;
39
40 typedef struct serial {
41   volatile       Tcontrol   ctl; // read-write, (int *)UART
42   const volatile Tstatus   stat; // read-only,  (int *)(UART+1)
43   volatile       Tinterr interr; // read-write, (int *)(UART+2)
44   volatile       int       data; // read-write, (int *)(UART+3)
45 } Tserial;
```

O registrador interr é modelado como uma **union** para permitir a atualização de um bit por vez, através da estrutura `Tintreg`, ou da programação de vários bits simultaneamente.

O compilador é idiossincrático ao gerar código para acessar dispositivos de *hardware*. Ao traduzir o código C, para o que seria a modificação de um único bit, o compilador pode gerar sequências de leitura-modificação-escrita que produzem um resultado distinto que que o programador pretendia. Por isso, é desejável que se possa acessar o registrador como a organização da estrutura `Tserial`, ou como um inteiro no qual se pode ativar simultaneamente tantos bits quanto seja necessário.

Os componentes da estrutura `Tserial` são declarados com os qualificadores **`volatile`** e **`const`**. O registrador `status` é um registrador só de leitura, e o qualificador **`const`** provocará um erro de compilação em qualquer tentativa de escrita neste registrador.

O qualificador **`volatile`** indica ao compilador que este endereço é 'volátil' e que portanto seu conteúdo pode ser alterado por código que não está na vizinhança da referência. A variável declarada como **`volatile`**, se corresponder ao registrador de periférico, pode ser alterada pela máquina de estados que controla o periférico. Se o endereço volátil é o de uma palavra de memória, então seu conteúdo pode ser alterado por um tratador de interrupção. Em qualquer dos casos, o programador informa ao compilador que nenhuma forma de otimização pode ser aplicada nas referências à variável que é volátil.

O Programa 8.5 contém o código que recebe uma cadeia de octetos pela UART, e então transmite uma outra cadeia. Este programa consiste de duas fases: primeiro, o processador fica a esperar pela recepção de uma cadeia de octetos através da entrada serial. O código da recepção emprega *espera ocupada*: o processador fica num laço lendo o registrador `status` à espera de cada novo octeto. A recepção termina quando um caractere `EOT` (*End Of Transmission* `0x04`) é recebido.

Espaço em branco proposital.

Programa 8.5: Acesso à UART por programa.

```
1  #define SPEED 1                    // choose a data rate
2  #define COUNTING ((SPEED+1)*100) // wait for end of transmission
3
4  char r[16] = "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"; // where to put received text
5  char s[] = "123\n456\n";         // text which will be sent
6
7  int main(void) {
8    volatile int i, val;
9    volatile int state;      // GCC must not optimize away these
10   volatile Tserial *uart;
11   volatile Tstatus status;
12   Tcontrol ctrl;
13   volatile int *counter;
14
15   counter = (void *)IO_COUNT_BOT_ADDR; // counter address
16   uart    = (void *)IO_UART_BOT_ADDR;  // UART address (base)
17
18   ctrl.rts   = 1;        // make RTS=1 to activate RemoteUnit
19   ctrl.ign4  = 0;
20   ctrl.speed = SPEED;
21   uart->ctl = ctrl;      // write config to control register
22
23   // -- receive a string via UART serial interface ------------
24   i = 0;
25   do {
26     while ( (int)uart->stat.rxFull == 0 )
27       {};                      // busy wait
28     r[i] = (char)uart->data; // get new character
29     to_stdout( r[i] );       //   and print it on stdout
30   } while (r[i++] != EOT);   // end of string?
31
32   // -- send a string through the UART serial interface -------
33   i = 0;
34   do {
35     while ( (int)uart->stat.txEmpty == 0 )
36       {};                      // busy wait
37     uart->data = (int)s[i];  // send out next character
38   } while (s[i++] != EOT);   // EOT was sent in previous line
39
40   // wait until last char is sent out of the shift-register.
41   // Otherwise, the simulation will end too soon producing
42   //    wrong result(s).
43   startCounter(COUNTING, 0);
44   while ( (val=(readCounter() & 0x3fffffff)) < COUNTING )
45     {};
46
47   return val;
48 }
```

A transmissão também emprega *espera ocupada*: o processador fica num laço, lendo o registrador status, até que o registrador txreg fique vazio, quando então o próximo octeto pode ser enviado. As variáveis `r[]` (*receive*) `s[]` (*send*) são os vetores para conter os octetos recebidos e por transmitir.

As variáveis que correspondem aos registradores da UART são declaradas como **`volatile`** para impedir que o GCC elimine o código quando este for otimizado – do ponto de vista

do compilador, os laços de espera ocupada são código inútil e não efetuam nenhuma computação, sendo portanto candidatos à eliminação.

Acertadamente, o compilador considera uma estupidez rematada a leitura repetida de um mesmo endereço, pois as variáveis de um programa *não variam* sozinhas. Registradores de periféricos são variáveis muito peculiares porque elas podem ser atualizadas independentemente das ações do processador. O mesmo se pode dizer de variáveis compartilhadas entre processos, ou entre um processo e o tratador de uma interrupção.

Dois apontadores são inicializados com os endereços do contador e da UART. A razão para usar o contador será explicitada adiante.

Os campos apropriados do registrador de controle são inicializados e estes valores são gravados em ctrl.

O primeiro laço recebe uma cadeia de octetos através da UART. O laço interno mantém o processador lendo o registrador status, até que o registrador de recepção contenha um octeto. Este é lido do registrador de recepção rxreg e então exibido na saída padrão do simulador, com a função `to_stdout()`.

O segundo laço é similar com a direção da comunicação revertida. Quando o registrador de transmissão está vazio, um novo octeto é gravado em txreg. A unidade remota exibe os octetos na saída padrão do simulador. O *casting* para `int` alinha o registrador, conforme a Figura 8.10.

Quando o `EOT` é enviado, o laço termina. Isso não significa que a simulação do programa possa terminar. É necessário esperar que o `EOT` seja transferido para o registrador de deslocamento, e de lá para a unidade remota. Esta é a função do laço de espera pelo contador – enquanto a contagem não expira, a simulação continua para que o último octeto seja transmitido e exibido na saída padrão.

Com a taxa de transmissão a $1/N$ da velocidade do processador, são necessários $10N$ ciclos do processador para transmitir um octeto. O fator de 10 inclui os *start* e *stop* bits mais os 8 bits do octeto. Por isso, para $N = 1$, o contador é programado para contar 200 ciclos, e o último laço faz o processador esperar até que o `EOT` seja transmitido, para que só então a simulação do programa termine, com alguma folga.

Aqui enfrentamos um problema inerente ao nosso ambiente de simulação: a taxa de transmissão é uma fração F da velocidade do processador; como cada octeto tem 10 bits, o processador executa $\approx 10F$ instruções a cada octeto transmitido. Quando o último octeto, `EOT`, é gravado em txreg, a transmissão deste demorará o tempo de $10F$ instruções, mas a simulação do programa terminaria antes disso. Para evitar erros de simulação – terminar cedo demais – o programa de testes faz uso do contador para 'segurar' a simulação por um tempo longo o bastante para que o `EOT` seja transmitido. Isso feito, o programa pode terminar sem erros.

Espaço em branco proposital.

8.2.2 Entrada e saída por interrupções

O Programa 8.6 mostra o trecho do tratador que é executado após o despacho da interrupção, e que salta para a rotina `UARTinterrHandler`, escrita em C, e que trata dos eventos causados pela UART.

O driver para a UART, descrito na Seção 8.2.4, emprega uma estrutura de dados com duas filas circulares – uma para a recepção, e uma para a transmissão – e esta estrutura é definida no Programa 8.7, na pág. 172. O espaço em memória para a estrutura `UARTdriver` é alocado nas linhas 5 a 13 do Programa 8.6. O programa em C deve declarar esta estrutura como **`extern`** porque o espaço em memória é alocado no tratador da interrupção.

O prólogo deste tratador é similar ao do contador: nas linhas 33-35 o registrador de *status* do periférico é lido. O conteúdo do registrador `status` é armazenado em memória na linha 36. A área em que o *status* é preservado é um vetor que deve ser cuidadosamente definido pelo programador, porque nesta área também devem ser armazenados os registradores que são modificados durante a execução do tratador. Os registradores utilizados pelo tratador são armazenados nas linhas 26-31 – dependendo da complexidade do código, mais registradores devem ser preservados.

Se o tratador da UART pode ser interrompido por outros tratadores, então as interrupções devem ser habilitadas antes do salto para a função que trata do evento sinalizado, como indicado na linha 41.

Um tratador para interrupções da UART é mais complexo que o tratador do contador, e portanto uma possibilidade é que ele seja escrito em C ao invés de *assembly*. As linhas 43-44 saltam para a função que trata a interrupção, escrita em C, cujo argumento é o *status* da UART. Escrever esta função fica como um exercício para a interessada leitora.

As interrupções devem ser novamente desabilitadas antes da recomposição dos registradores preservados e do retorno da interrupção, como indicado na linha 46. Os registradores são recompostos nas linhas 48-53.

Isso feito, o processador volta a executar o programa interrompido, com a instrução **`eret`**.

Espaço em branco proposital.

Programa 8.6: Tratador de interrupções da UART.

```
1          # interrupt handler for UART attached to IP6=HW4
2          .bss
3          .equ Q_SZ, (1<<4) # 16, MUST be a power of two
4          .global Ud        # make Ud's address visible to main()
5  Ud:
6  rx_hd:  .space 4          # reception queue head index
7  rx_tl:  .space 4          # rx queue tail index
8  rx_q:   .space Q_SZ       # rx queue
9  tx_hd:  .space 4          # transmission queue head indexes
10 tx_tl:  .space 4          # tx queue tail index
11 tx_q:   .space Q_SZ       # tx queue
12 nrx:    .space 4          # num. of characters in RX_queue
13 ntx:    .space 4          # num. of spaces left in TX_queue
14
15 _uart_buff: .space 16*4 # up to 16 registers to be saved here
16         # _uart_buff[0]=UARTstatus, [1]=UARTcontrol,
17         #           [2]=v0, [3]=v1,
18         #           [4]=ra, [5]=a0, [6]=a1, [7]=a2, [8]=a3,
19
20         .equ USTAT,  4      # UART Status register
21         .equ UINTER, 8      # UART Interr register
22         .equ U_clr_rx_irq, 0x08 # clr RX irq, bit 3
23
24         .text
25 UARTinterr:
26         lui   k0, %hi(_uart_buff)  # get buffer's address
27         ori   k0, k0, %lo(_uart_buff)
28         sw    ra, 4*4(k0)  # create a small context
29         sw    a0, 5*4(k0)  #   for C code to execute in
30         sw    a1, 6*4(k0)
31         sw    v0, 2*4(k0)
32
33         lui   a0, %hi(HW_uart_addr) # get device's address
34         ori   a0, a0, %lo(HW_uart_addr)
35         lw    k1, USTAT(a0) # Read status
36         sw    k1, 0*4(k0)   #   and save UART status to memory
37
38         lw    a1, UINTER(a0)      # remove interrupt request
39         nop
40         ori   a1, a1, U_clr_rx_irq
41         sw    a1, UINTER(a0)
42
43         # re-enable interrupts here?  Enable all but UART's?
44
45         jal UARTinterrHandler     # jump to C function
46         move a0, k1               #   UARTinterrHandler(status)
47
48         # disable interrupts here?
49
50         lui   k0, %hi(_uart_buff) # restore registers
51         ori   k0, k0, %lo(_uart_buff)
52         lw    ra, 4*4(k0)
53         lw    a0, 5*4(k0)
54         lw    a1, 6*4(k0)
55         lw    v0, 9*4(k0)
56
57 return: eret                     # Return from interrupt
```

8.2.3 Concorrência e assincronia no acesso à UART

Antes de discutirmos o *driver* para a UART, é necessário recordar a discussão sobre acesso atômico a variáveis na Seção 7.2. A Figura 8.14 é uma versão expandida da Figura 7.8 (pág. 136). No lado esquerdo do diagrama está o *processo* que executa toda a vez em que ocorrer uma interrupção de recepção, enquanto que à direita é representado o *processo* que executa o programa `main()`. O uso da palavra *processo* é proposital para enfatizar que a execução destes dois trechos de código, `trat()` e `prog()`, é concorrente e assíncrona. `prog()` executa 'constantemente' no processador, enquanto que `trat()` é executado somente quanto um octeto é recebido através da UART, e *a priori*, não há como saber quando o próximo octeto será recebido – por isso os dois processos são chamados de *assíncronos*.

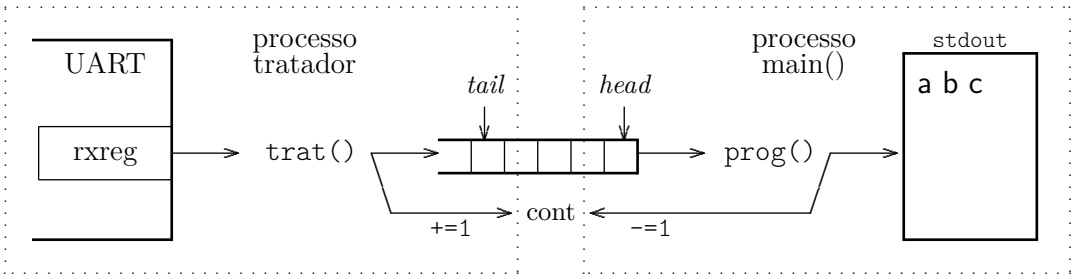


Figura 8.14: Fluxo de dados no *driver* da UART.

Por causa da natureza concorrente e assíncrona, a estrutura para o *driver* que melhor reflete a concorrência dos dois processos é um modelo em duas camadas, uma próxima da UART e outra próxima da aplicação. Estes dois processos se comunicam através da fila de octetos e de um contador (`cont`) que indica o número de octetos na fila.

Quando o tratador da interrupção executa, `prog()` não está a executar porque foi interrompido. `trat()` incrementa `cont` sem que haja interferência pelo outro processo. Quando `prog()` acessa o contador para decrementá-lo, durante o decremento – leitura, atualização, escrita: `lw`;`addi`;`sw` – pode ocorrer uma interrupção que incrementará o contador, possivelmente deixando-o com um valor inconsistente. Por causa do risco de interferência, `prog()` deve desabilitar as interrupções enquanto altera o valor de `cont`, assim garantindo a execução do decremento em exclusão mútua.

O diagrama de tempo da Figura 8.15 mostra a execução concorrente de `prog()` e do tratador da interrupção. A UART gera uma interrupção para sinalizar quando um novo octeto é recebido. Quando a interrupção é atendida, o processo `trat()` copia o octeto recebido de rxreg para a fila de octetos, decrementa o apontador da cauda da fila, incrementa `cont`, e retorna ao executar a instrução `eret`. O intervalo entre a sinalização de octeto recebido pela UART e o início do tratamento é variável, porque as interrupções podem estar desabilitadas. Para que todos os octetos recebidos sejam capturados pelo tratador, é importante que os intervalos com as interrupções desabilitadas sejam curtos.

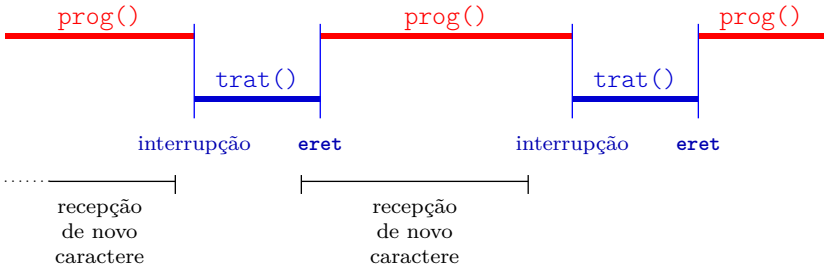


Figura 8.15: Execução assíncrona do tratador e do programa.

8.2.4 Um *driver* para a UART

A Figura 8.16 mostra um diagrama do *driver* simplificado para a UART. Este *driver* é dividido em duas camadas: (i) o tratador de interrupções (*handler*) que interage diretamente com a UART; e (ii) um conjunto de funções que permitem ao programador enviar e receber através da interface serial. A este conjunto de funções é que chamamos de *driver*.

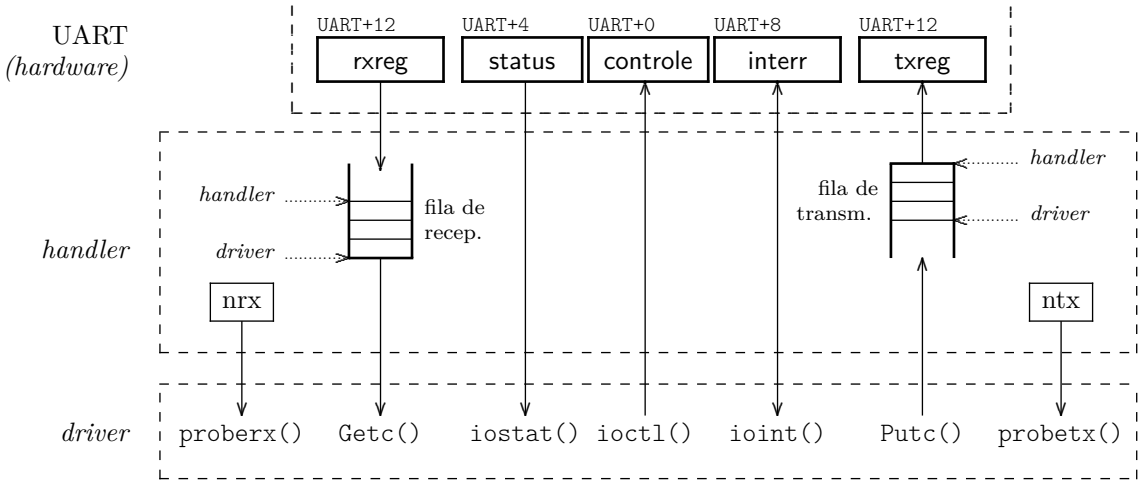


Figura 8.16: Fluxo de dados no *driver* da UART.

O tratador de interrupções gerencia duas filas de octetos, uma associada à recepção e outra à transmissão, ambas com capacidade para 16 octetos. O tratador mantém dois contadores: `nrx` indica o número de octetos disponíveis na fila de recepção; e `ntx` indica o número de espaços na fila de transmissão.

As funções `proberx()` e `probetx()` retornam os valores de `nrx` e `ntx`, respectivamente. A função `iostat()` retorna o conteúdo do registrador de *status* da UART, e a função `ioctl()` permite escrever no seu registrador de controle e assim alterar as configurações da UART. A função `Getc()` retorna o octeto que está na cabeça da fila de recepção, e decrementa `nrx`. A função `Putc()` insere um octeto na fila de transmissão, e decrementa `ntx`.

O programa que usa o *driver* deve garantir, testando `nrx`, que a fila de recepção não esteja vazia, e ainda deve garantir, testando `ntx`, que a fila de transmissão não esteja cheia.

O Programa 8.7 mostra a versão em C da estrutura de dados e as variáveis de controle do *driver*, e que corresponde exatamente àquelas definidas em *assembly* no Programa 8.6.

Programa 8.7: Estrutura de dados do *driver* da UART.

```
#define Q_SZ (1<<4)      // 16, MUST be a power of two
typedef struct UARTdriver {
   int      rx_hd;       // reception queue head index
   int      rx_tl;       // reception queue tail index
   char     rx_q[Q_SZ]; // reception queue
   int      tx_hd;       // transmission queue head index
   int      tx_tl;       // transmission queue tail index
   char     tx_q[Q_SZ]; // transmission queue
   int      nrx;         // number of characters in rx_queue
   int      ntx;         // number of spaces in tx_queue
} UARTdriver;
```

Os componentes da estrutura `UARTdriver` podem ser acessados diretamente no código do *driver* uma vez que ela seja declarada no programa em C. No *handler*, os endereços para acessar os componentes da estrutura devem ser computados explicitamente. O trecho de código no Programa 8.8 insere um octeto na fila de recepção.

O contador de octetos recebidos, `nrx`, é incrementado. O índice da cauda da fila é incrementado módulo `Q_SZ` – o **`andi`** com `(Q_SZ-1)` garante o uso correto da fila circular. O registrador de dados da UART é lido, apontado por `k1`.

O valor do índice é somado ao endereço da base de `Ud` e o **`sb`** usa este valor, mais o deslocamento de 8 bytes até o início da fila de recepção para depositar o octeto recebido na posição apontada por `rx_tl`. O endereço em que este octeto deve ser escrito é `Ud+RX_Q+rx_tl`.

Programa 8.8: Inserção de um octeto na fila de recepção.

```
1          .global Ud                  # see UARTinterr
2
3          .equ Q_SZ, (1<<4)           # 16, MUST be a power of two
4          .equ RXHD, 0                # define constants for
5          .equ RXTL, 4                #   addressing members of
6          .equ RX_Q, 8                #   structure Ud
7          .equ TXHD,24
8          .equ TXTL,28
9          .equ TXQ, 32
10         .equ NRX, 48
11         .equ NTX, 52
12
13         #
14         # arrived here because of JAL from UARTinterr
15         #
16 UARTinterrHandler:
17         lui   k0, %hi(Ud)           # UART's data structure
18         ori   k0, k0, %lo(Ud)
19         lw    a0, NRX(k0)           # increment nrx
20         nop
21         addiu a0, a0, 1
22         sw    a0, NRX(k0)           #   and save it
23
24         lui   k1, %hi(HW_uart_addr) # get device's address
25         ori   k1, k1, %lo(HW_uart_addr)
26         lw    a1, UDATA(k1)         # read data from UARTrxreg
27
28         lw    a0, RXTL(k0)          # read rx_tail
29         nop
30         addu  a2, a0, k0            # Ud + M(rx_tl)
31         sb    a1, RX_Q(a2)          # M[Ud+tail+RX_Q] <- data
32
33         addiu a0, a0, 1             # increment rx_tail
34         andi  a0, a0, (Q_SZ - 1)    #   modulo Q_SZ
35         sw    a0, RXTL(k0)          # save new tail
36
37         # if there is a TX interrupt pending
38         # then go handle it (not shown here)
39         # else all work is done, return
40
41         jr    ra                    # return to UARTinterr
```

A função `Getc()` decrementa `nrx` e remove o octeto na cabeça da fila, conforme mostra o Programa 8.9. A função deve manipular a estrutura `Ud` com as interrupções desabilitadas, conforme discutido na Seção 8.2.3.

Programa 8.9: Função `Getc()`.

```
char Getc(void) {
   char c;
   extern   UARTdriver Ud;
   volatile UARTdriver *ptr;

   ptr = &Ud;

   disable_interr();    // exclui handler enquanto altera Ud

   ptr->nrx -= 1;
   c = ptr->rx_q[ptr->rx_hd];
   ptr->rx_hd = (ptr->rx_hd+1) & (Q_SZ-1);  // (hd++) mod Q_SZ

   enable_interr();

   return c;
}
```

O Programa 8.10 indica a forma de uso de `Getc()`, tal como seria num sistema embarcado. O programa deve verificar se um novo octeto foi recebido, para só então executar `Getc()`. Num sistema Unix, quando um processo executa `Getc()` e a fila de recepção está vazia, este processo fica suspenso até que um octeto seja recebido. Um sistema embarcado geralmente não executa um sistema operacional 'pesado' como o Unix e neste caso a espera ocupada pode ser aceitável, dependendo das restrições de temporização do projeto.

Programa 8.10: Trecho de código que usa `Getc()`.

```
   extern UARTdriver Ud;
   ...
   while (proberx() == 0)   // espera por um novo octeto
     { };
   vetor[i++] = Getc();
   ...
```

8.3 Exercícios

Ex. 61 A função `Putc()` é mais complexa do que `Getc()`. O que acontece se a fila de transmissão estava inicialmente vazia, e o *status* da UART indica que o registrador de transmissão está vazio?

Ex. 62 Ainda quanto à transmissão, o que o *handler* deve fazer, se a fila de transmissão estiver vazia ao atender uma interrupção de transmissão?

Ex. 63 Num sistema com comunicação *dúplex*, em que transmissão e recepção podem ser concomitantes, quais são as combinações possíveis de interrupções simultâneas? Como isso afeta a organização do *handler*?

Ex. 64 Uma vez detectado um erro de enquadramento, ou *overrun*, como o *handler* informa essa condição ao *driver*?

Capítulo 9

Memória Virtual e Paginação

Um *espaço de endereçamento* é o conjunto de endereços que pode ser referenciado por um programa. De uma forma simplificada, pode-se dizer que na linguagem C o espaço de endereçamento é a faixa de endereços que pode ser alcançada por um apontador, que num processador de 32 bits seria de 2^{32} bytes. Este espaço de endereçamento é dito *linear* porque é uma sequência contínua de posições de memória.

A alternativa a um espaço de endereçamento linear é um espaço *segmentado*, tipicamente com os segmentos de texto (código), dados e pilha. Cada segmento é apontado por um *registrador base*, que aponta seu endereço mais baixo, e é limitado pelo *registrador limite*, que aponta seu endereço mais alto. O PC aponta para um endereço no segmento de texto. Para referenciar uma variável, seu endereço é computado pela adição de um deslocamento ao conteúdo do registrador base de dados. O apontador de pilha pode tomar valores entre a base e o limite do segmento de pilha.

Desde a década de 1940, do ponto de vista do espaço de endereçamento visível ao programador, os computadores foram construídos segundo três modelos, que evoluíram na medida em que a tecnologia de fabricação assim o permitiu. Vejamos os três modelos.

O primeiro modelo esteve em uso até o início da década de 1960. As máquinas eram equipadas com pouca memória – 8 a 16 Kbytes – e seus programas eram carregados manualmente por um operador humano[1]. Os dispositivos de entrada eram terminais eletromecânicos, os de saída eram impressoras eletromecânicas, e os dispositivos de armazenagem eram tambores com 10^4 a 10^6 bytes de capacidade. O espaço de endereçamento era contínuo e se estendia do endereço zero ao topo da memória instalada. Os programas eram sempre carregados num mesmo endereço inicial, e a memória podia conter *somente um programa em execução*. Quando o programa efetuava operações de E/S, sempre demoradas, o processador ficava num laço a esperar que as operações completassem, para então voltar a efetuar trabalho útil.

O segundo modelo vigorou até 1972. As máquinas desta época eram equipadas com algo como 64 Kbytes de memória e empregavam registradores base-limite para suportar *multiprogramação*. Estes registradores permitem que os programas possam ser carregados em qualquer endereço. Tais máquinas executavam sistemas operacionais primitivos e seus periféricos se assemelhavam aos que estamos habituados, com terminais eletromecânicos (TTY) ou de vídeo (VT100). As máquinas podiam ser equipadas com mais de uma unidade de disco, o espaço de endereçamento era segmentado, e os programas podiam ser maiores do que a capacidade da memória física instalada. Se um segmento era grande demais para o espaço disponível, a programadora era obrigada a empregar *overlays*: o programa deveria

[1] Um dos mecanismos de carga consistia da montagem na unidade de leitura de um rolo de fita de papel perfurada com a codificação do programa – um humano, de fato, carregava um programa na máquina.

ser dividido em módulos, sendo que cada módulo continha todas as funções e estruturas de dados com dependências entre aquelas. O nome *overlay* descreve uma técnica pela qual o módulo M_j é carregado em memória sobre o módulo M_k, desde que o módulo M_j não dependa do código ou dados contidos no módulo M_k que será sobrescrito. Toda a complexidade do gerenciamento da alocação de módulos em memória ficava para a programadora, o que sem demora se torna entretenimento excessivo.

O terceiro modelo é aquele empregado até hoje pelos sistemas operacionais de uso geral. O IBM 370, lançado em 1972, foi a primeira máquina comercial a implementar *paginação sob demanda* [**?**]. Este mecanismo esconde do programador o gerenciamento da memória: o espaço de endereçamento visível ao programador é linear, ao invés de segmentado, e pode ser maior do que a capacidade da memória instalada. A ideia de paginação sob demanda foi publicada em 1962 [**?**] e a tecnologia só permitiu sua implementação comercial uma década mais tarde[2].

Em que pese a antiguidade, as técnicas de gerenciamento de memória dos primeiro e segundo modelos são amplamente empregadas em sistemas embarcados. Estes sistemas são construídos com a menor capacidade de memória possível mas que seja suficiente para executar as aplicações do sistema. Isso força a programadora destes sistemas a gerenciar explicitamente a alocação da memória necessária à execução dos programas, tanto dos segmentos de texto quanto dos segmentos de dados e pilha. Considerando o número de dispositivos vendidos anualmente, a quantidade de sistemas embarcados supera aquela dos sistemas de uso geral por algumas ordens de magnitude. Esta conta exclui *smartphones* e *tablets*, que em sua maioria empregam sistemas operacionais de uso geral.

Limitações do Endereçamento Físico

Em máquinas construídas sob os primeiro e segundo modelos, todos os programas compartilham um *espaço de endereçamento físico*, tendo portanto acesso a todos os recursos do computador, e não há modo de evitar que um programa acesse qualquer recurso da máquina. A Figura 9.1 mostra um diagrama com a ligação direta entre o processador e a memória, que é acessada diretamente com *endereços físicos*. Nestas máquinas, uma instrução é acessada diretamente com o conteúdo do PC, e são parcos os mecanismos para limitar danos causados por programas em erro, ou cheios de malícia.

Figura 9.1: Endereçamento físico sem proteção.

A *virtualização* da memória é obtida pelo uso de um conjunto de mecanismos em *hardware* e algoritmos que oferece ao programador um espaço de endereçamento unidimensional (*flat*), com capacidade 'infinita' que é limitada somente pela implementação dos barramentos de endereços no processador.

[2]Estes eram os anos iniciais da vigência da Lei de Moore, *viz.* "*a quantidade de transistores disponíveis em um circuito integrado dobra a cada 18 meses*" [**?**].

9.1 Memória virtual

O nome atribuído pelo programador a uma função é traduzido pelo compilador para o *endereço* da primeira instrução da função. A programadora pensa em nomes simbólicos, tais como `fun()`, e estes são associados pelo compilador aos seus *endereços*. O mesmo se passa com as variáveis: o que o programador denomina `vetor` é associado pelo compilador ao conjunto de *endereços* em memória no qual os elementos do vetor são armazenados.

Considere um programa escrito em C e compilado para um processador de 32 bits. Em tese, um apontador pode percorrer 4 Gbytes de 'memória' mesmo que a memória DRAM instalada na máquina tenha capacidade bem menor do que 4 Gbytes. Os endereços referenciados pela programadora são chamados de *virtuais* porque, na máquina virtual provida pelo compilador de C, sempre se pode alcançar 2^{32} bytes de memória, independentemente da máquina real em que o programa é executado.

O resultado do processo de compilação é um arquivo executável, possivelmente chamado de `a.out`. Para criar um novo processo para executar o `a.out`, o processo pai invoca a chamada de sistema `fork()`, que cria um novo *espaço de endereçamento* para o processo filho, que é uma cópia exata do processo pai. Quando o processo filho invoca `execve()`, o executável armazenado em `a.out` é *carregado* sobre o espaço de endereçamento do filho; as instruções e dados – da cópia do processo pai – são sobrescritas pelo conteúdo de `a.out`. Isso feito, o *carregador* inicializa a pilha do processo filho com `argc` e `argv[]`, inicializa algumas estruturas de dados do SO, e carrega o PC com o endereço da primeira instrução do programa. Este último passo dispara a execução do programa armazenado em `a.out`.

Isso tudo posto, definamos então o que vem a ser *memória virtual*. O material nesta seção é uma adaptação de [**?**], e emprega a mesma notação e nomenclatura. Esta *survey* foi publicada por Denning pouco antes da difusão de sistemas com memória virtual como conhecemos hoje.

O modelo de máquina que usamos para definir memória virtual é mostrado na Figura 9.2. O programa que executa no processador só tem acesso direto à *memória principal*, que é implementada como um vetor linear de *posições*, e cada posição pode armazenar o conteúdo de uma unidade de informação – no nosso caso, um byte ou uma palavra de 32 bits. O tempo médio de cada referência à memória é de Δ ciclos do processador, e Δ é da ordem de 2 a 10 ciclos. No tempo médio estão considerados todos os efeitos de memórias cache, *refresh* da DRAM, e interrupções sofridas pelo programa.

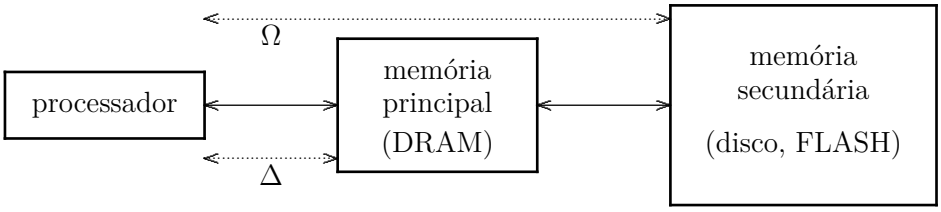


Figura 9.2: Modelo do computador (*hardware*).

A *memória secundária* só é referenciada indiretamente pelo programa – através de serviços solicitados ao SO – e a organização da memória secundária é escondida do programador. O tempo médio de acesso à memória secundária é de Ω ciclos, e este valor inclui o tempo necessário para movimentar o braço do disco, espera em filas do SO, e transferências de dados entre dispositivo e memória principal. Ω é da ordem de 10^5 a 10^6 ciclos para discos magnéticos e 10^3 ciclos para 'discos' com memória FLASH. A relação Δ/Ω é da ordem de 10^2 a 10^3 para dispositivos com memória FLASH, e 10^5 para discos magnéticos.

Usamos a notação de intervalos para definir as faixas de valores dos espaços de endereçamento. O intervalo $[0, x)$ corresponde ao conjunto de inteiros $\{0, 1, \ldots, x-1\}$.

O conjunto de nomes simbólicos usados pela programadora é chamado de *espaço de nomes*, com seu correspondente *espaço de endereçamento virtual*. O espaço de endereçamento virtual tem $N = [0, n)$ endereços.

O local na memória principal no qual uma função ou variável é alocada pelo carregador é a sua *posição na memória física*, ou o seu *endereço físico*. O conjunto de posições alocadas a um programa é o seu *espaço de endereçamento físico*. O espaço de endereçamento físico tem $M = [0, m)$ posições. Em geral, $n \neq m$.

O mapeamento de endereços virtuais para posições em memória *deve* ser uma função na qual cada endereço virtual mapeado em *uma única* posição em memória, ou então mapeado para uma posição inválida. O domínio da função de mapeamento é todo o espaço de nomes do programa, e sua imagem é um conjunto de posições, que geralmente é menor que o domínio, $n \geq m$. Suponha que a memória física tem m posições, e que o programa nomeia n endereços. Então, $N = [0, n)$ é o *espaço de endereçamento virtual*, o domínio do mapeamento, e $M = [0, m)$ é o conjunto de posições de memória, a sua imagem.

Num determinado instante, o mapeamento de endereços virtuais em posições de memória física é dado pela função $f : N \to M \cup \{\varphi\}$, definida na Equação 9.1.

$$f(a) = \begin{cases} a' & \text{se } a \text{ está alocado na posição } a' \in M \\ \varphi & \text{se } a \text{ não está alocado a uma posição em } M \end{cases} \tag{9.1}$$

A função f é implementada na *Unidade de Gerenciamento de Memória* (UGM) da Figura 9.3, que mostra um mapeamento do endereço a na posição a'. O endereço virtual (EV) é usado para indexar a tabela f, e se o mapeamento é válido, então o endereço físico (EF) a' é usado para acessar a posição correspondente na memória principal. Se $f(a) = \varphi$, então o mapeamento é inválido e SO é chamado a intervir, como veremos adiante.

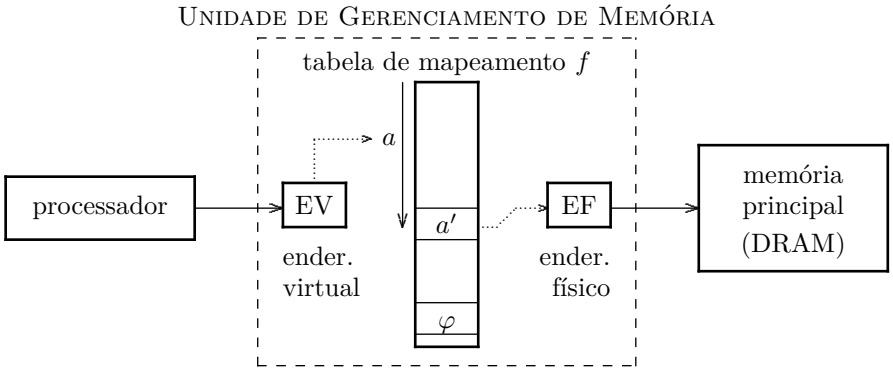


Figura 9.3: Unidade de Gerenciamento de Memória.

A programadora escreve os programas considerando que os nomes $a, b, c \in N$ são referenciados em posições consecutivas de N, embora o mapeamento f permita que estes itens sejam armazenados em posições arbitrárias de M, porque f é um *mapeamento totalmente associativo* de N em M. O domínio de f é 'ordenado' mas os elementos correspondentes na sua imagem podem estar numa 'ordem' arbitrária.

Quando o processo p inicia, o carregador preenche o seu mapeamento f_p com um número de pares (a, a') suficiente para que a execução se inicie e progrida. f_p deve ser atualizada quando o programa referencia um endereço x, tal que $f_p(x) = \varphi$. Esta referência inválida pode ser benigna, e neste caso o par (x, x'), com $x' \in M$, é acrescentado a f_p, ou então a

referência é maligna, e neste caso o processo é terminado com uma *falta de segmentação* (*segmentation fault*), que é a exceção que causa a terminação forçada do processo.

A Equação 9.1 é definida para "um determinado instante" porque os mapeamentos de endereços virtuais em físicos podem mudar ao longo da vida de um processo. Por exemplo, no início da execução do processo K, o endereço virtual a é mapeado na posição a'; por razões discutidas no que segue, este mapeamento pode ser alterado para (a, x') por causa do comportamento dinâmico do conjunto de todos os processos em execução.

Quais as possíveis causas para alterações em f? (i) Na inicialização do processo, algum subconjunto adequado de f deve ser preenchido pelo carregador; (ii) o processo termina e toda a memória física que usava deve ser retornada ao SO, para uso pelos outros processos; (iii) o processo referencia um endereço válido pela primeira vez; e (iv) por erro de programação, ou por malícia, o processo referencia um endereço considerado inválido.

As consequências das alterações em f são determinadas pela regra de substituição, pela regra de alocação, ou pela regra de busca.

A *regra de alocação* é usada para determinar qual posição da memória principal será alocada para conter um item nomeado pelo programa. O sistema operacional deve manter uma lista com todas as posições de memória disponíveis para uso, e esta lista é consultada e atualizada quando uma nova posição deve ser alocada, ou liberada.

A *regra de busca* determina o momento em que um item é acrescentado ao domínio de f. São duas as possibilidades para a regra de busca. Na *busca sob demanda*, um novo item é mapeado somente quando referenciado pelo programa. Na *busca antecipada*, um item pode ser mapeado antes que seja referenciado pelo programa. Esta última pode causar a 'poluição' da memória pela carga de itens que não seriam nunca referenciados.

A *regra de substituição* é invocada quando $f(x) = \varphi$. Se M não está esgotado, o mapeamento (x, x'), com $x' \in M$ e $f^{-1}(x') = \{\}$, é acrescentado a f. Se M está esgotado porque toda a memória principal está alocada, então um item y' de M é escolhido pela regra de substituição, o mapeamento (y, y') é removido de f, e o novo par (x, y') é acrescentado. A função f deve ser atualizada para que $f(y) = \varphi$.

Ocorre uma falta de segmentação quando $f(x) = \varphi$. Se a referência é benigna e considerada legal, então o item que não está em memória principal deve ser buscado da memória secundária e copiado para alguma posição na memória principal. Esta operação de cópia da memória secundária para a memória principal demora Ω ciclos.

Vejamos três possibilidades para a regra de substituição.

Exemplo 9.1 Considere que esta sequência de endereços virtuais ocorre durante a execução de um programa: `3 4 2 6 4 7 1 3 2 6 3 5 1 2 3`. A memória primária tem capacidade para quatro itens. Quando um item deve ser substituído, o item que está na memória há mais tempo deve ser substituído pelo novo. Esta *regra de substituição* implementa uma *fila*, porque o primeiro que entra é o primeiro que sai, e é conhecida como *FIFO – First In First Out*.

Numa fila, novos itens são inseridos no fim da fila, e a *vítima* da reposição é o elemento que está no início da fila.

A Tabela 9.1 mostra o conteúdo da fila após cada referência e no topo da tabela é mostrado o item por inserir. Os quatro primeiros são inseridos diretamente, e as colunas mostram o estado da fila após a inserção. Itens em negrito são aqueles que estavam na fila e foram referenciados novamente, e portanto não causam faltas de segmentação.

Para *essa* sequência, e com memória de capacidade 4, somente três itens são reaproveitados. ◁

Tabela 9.1: Substituição do mais velho – FIFO.

		4	7	1	3	2	6	3	5	1	2	3
fim	6	6	7	1	3	2	6	6	5	1	1	3
	2	2	6	7	1	3	2	2	6	5	5	1
	4	**4**	2	6	7	1	3	**3**	2	6	6	5
início	3	3	4	2	6	7	1	1	3	2	**2**	6

Exemplo 9.2 Considere a mesma sequência de itens e a memória de capacidade quatro do Exemplo 9.1. Agora, a escolha do item por substituir recai naquele que não é usado há mais tempo. Esta política implementa uma *pilha*, e é conhecida como *LRU – Least Recently Used*.

Para escolher o item LRU para capacidade quatro, emprega-se uma pilha de 4 posições. Numa inserção, se o elemento não está na pilha, ele é inserido no topo, e o elemento da base é descartado. Se o elemento já está na pilha, ele é movido para o topo. No topo da pilha fica o item *MRU – Most Recently Used*.

A Tabela 9.2 mostra o conteúdo da pilha LRU após cada referência. No topo da tabela é mostrado o item por inserir. As colunas mostram o estado da pilha após a inserção. Os itens em negrito foram referenciados novamente e portanto não causam falta de segmentação.

Para *essa* sequência, e com memória de capacidade 4, somente três itens são reaproveitados. ◁

Tabela 9.2: Substituição do usado menos recentemente – LRU.

		4	7	1	3	2	6	3	5	1	2	3
topo	6	**4**	7	1	3	2	6	**3**	5	1	2	**3**
	2	6	4	7	1	3	2	6	3	5	1	2
	4	2	6	4	7	1	3	2	6	3	5	1
base	3	3	2	6	4	7	1	1	2	6	3	5

A política de substituição LRU é muito eficaz quando as referências exibem a propriedade de *localidade temporal*, assim definida: *se um item foi referenciado pelo programa recentemente, existe uma (alta) probabilidade de que ele seja referenciado novamente num futuro próximo.* As referências ao índice de um vetor que é percorrido num laço exibem localidade temporal. Para um conjunto de C items, a profundidade máxima da pilha LRU é C, quando todos os elementos estão contidos na pilha. Uma simulação com uma pilha de profundidade C contém as simulações de todas as pilhas de profundidade menor que C.

Exemplo 9.3 Considere a mesma sequência de itens e a memória de capacidade quatro do Exemplo 9.1. Agora, a escolha do item por substituir é feita por um oráculo que vitimiza o item que causaria a menor perturbação no conteúdo da memória.

O oráculo examina o futuro de toda a sequência de referências e escolhe a vítima de forma a minimizar as substituições – se um item que está na memória não será mais referenciado, este é escolhido para a substituição.

A Tabela 9.3 mostra o conteúdo da memória após cada referência. As colunas mostram o estado da memória após a inserção. Os itens em negrito foram referenciados novamente e portanto não causam falta de segmentação.

Para *essa* sequência, e memória, das 15 referências, 8 itens são reaproveitados.

O oráculo é mais eficiente no reaproveitamento dos itens, e portanto na redução das substituições. Evidentemente, um oráculo não pode ser implementado, mas seus resultados servem como uma quota superior – nenhuma outra política produziria resultados *melhores* do que o oráculo. Esta política é chamada de OPT, porque ela é ótima. ◁

Tabela 9.3: Substituição com oráculo – OPT.

	4	7	1	3	2	6	3	5	1	2	3
3	3	3	3	**3**	3	3	**3**	3	3	3	**3**
4	**4**	7	1	1	1	1	1	1	**1**	1	1
2	2	2	2	2	**2**	2	2	2	2	**2**	2
6	6	6	6	6	6	**6**	6	5	5	5	5

Até o momento ignoramos o fato de que, para mapear um espaço de endereçamento com N endereços, é necessária uma tabela f com N posições. Ademais, frequentemente temos $N \geq M$. A solução para este problema é fazer com que f mapeie blocos de endereços, ao invés de um único endereço. Sistemas modernos empregam blocos com tamanho fixo chamados de *páginas*. Uma página é um conjunto contíguo de z endereços e o conjunto de páginas $P = [0, p)$ contém p páginas que cobrem todos os $n = p \times z$ endereços virtuais.

A memória física é logicamente dividida em $q = M/z$ *quadros*, cada quadro com z posições, de forma que o espaço de endereçamento físico contenha quadros que iniciam nos endereços $\{0, z, 2z, \cdots, (q-1)z\}$, e $Q = [0, q)$ é o conjunto de quadros. A memória principal é alocada em unidades de quadros.

Um endereço virtual a é interpretado como $a = \langle \mathrm{p}, \mathrm{w} \rangle$, sendo p o número de uma página, e w o número de uma palavra na página p, segundo as relações

$$a = p \cdot z + w, \quad 0 \leq w < z, \quad p = a/z, \quad w = a\%z$$

sendo / e % a divisão inteira e o resto (mod), respectivamente. Se z for uma potência de dois, os circuitos para efetuar estas operações são triviais: para escolher o número da página, basta ignorar os z bits menos significativos, ou efetua-se a divisão com um deslocamento para a direita de z posições; o resto é dado pelos z bits menos significativos, ou pela conjunção do endereço com $z - 1$.

Com paginação e uma escolha adequada para z, o domínio da tabela de mapeamento f é reduzido substancialmente. Valores típicos para z são 4Kbytes e 8Kbytes. Para um processador de 32 bits e páginas de 4Kbytes, $|f| = 2^{32}/2^{12} = 2^{20}$ mapeamentos. No que segue, usamos páginas de 4Kbytes.

9.2 Paginação

O mapeamento de endereços virtuais para posições em memória física foi inventado para facilitar o gerenciamento eficiente de memórias físicas de pequena capacidade. Esta deixou de ser a motivação principal porque o custo da memória física vem diminuindo com o tempo. Atualmente, a principal utilidade da memória virtual é a *separação dos espaços de endereçamento* dos vários processos que executam concorrentemente, o que tem o efeito colateral importante que é a *proteção* na execução concorrente dos processos.

Máquinas modernas empregam um mecanismo que efetua o gerenciamento da memória física que provê aos programas de usuário em *espaço de endereçamento virtual*, contínuo, linear e que suporta funcionalidades tais como proteção e o compartilhamento da memória física entre muitos processos. A Figura 9.4 mostra um diagrama com a ligação indireta

entre o processador e a memória – os endereços virtuais emitidos pelo processador são traduzidos para endereços físicos. A *unidade de gerenciamento de memória* (UGM, ou *memory management unit*, MMU), é implementada parte em *hardware* e parte em *software*, e é gerenciada pelo sistema operacional. A UGM mapeia endereços virtuais em endereços físicos, que são usados nos acessos à memória física e aos periféricos.

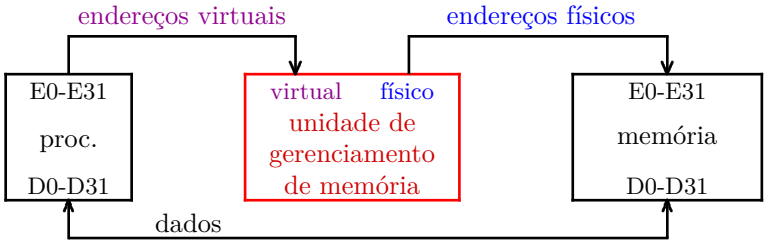


Figura 9.4: Tradução de endereços virtuais para físicos.

A *máquina virtual* provida pela linguagem C para o processador MIPS de 32 bits tem um espaço de endereçamento linear (*flat*) de 4 Gbytes porque os ponteiros em C tem 32 bits e podem portanto referenciar qualquer endereço entre 0 e $2^{32}-1$. A memória física instalada na máquina pode ser de qualquer capacidade razoável – de 64 Mbytes a 64 Gbytes, com endereços físicos de 26 a 36 bits – e a unidade de gerenciamento de memória mapeia o espaço de endereços virtuais de 4 Gbytes em posições na memória física, qualquer que seja a sua capacidade.

Sistemas de memória virtual com paginação permitem a alocação simultânea de memória a muitos processos, cuja demanda agregada por espaço de memória pode ser *muito* maior do que a memória física instalada. Isso é possível porque, num dado intervalo de tempo, um processo referencia um subconjunto pequeno de todas as suas páginas. Este comportamento é chamado de *localidade de referência*. Por causa da localidade, somente as páginas que são referenciadas ativamente pelos processos são mantidas em memória física, enquanto que as páginas que não são referenciadas podem ser mantidas em memória secundária, na "área de troca" ou *swapping*. A Figura 9.5 mostra quatro processos distintos carregados em memória, cada um deles capaz de acessar 4 Gbytes, compartilhando uma memória física com 256 Mbytes de capacidade.

Este é um conceito importante: o compartilhamento da memória física é viável porque, em intervalos mais ou menos longos, os processos referenciam somente um subconjunto pequeno de seus espaços de endereçamento. Para cada processo, o subconjunto ativo com algo como 10^2 a 10^3 páginas é determinado pela sua localidade de referência.

Exemplo 9.4 Para explicitar a ideia de localidade, considere dois tipos de programas:

(i) programas para aplicações científicas geralmente obedecem à "regra da localidade 90/10": os programas passam 90% do tempo executando 10% das linhas de código fonte. Tipicamente, os 10% do código são os laços mais internos que efetuam a computação desejada [**?**]; e

(ii) "código de produção" é aquele de produtos comerciais robustos e bem protegidos contra usuários não muito sagazes, e que empregam de 25 a 50% das linhas de código fonte para detecção e tratamento de erros nos dados de entrada. Se o usuário for marginalmente esperto, raramente os testes de qualidade dos dados resultarão negativos e portanto o volumoso código de recuperação de erros raramente é executado. ◁

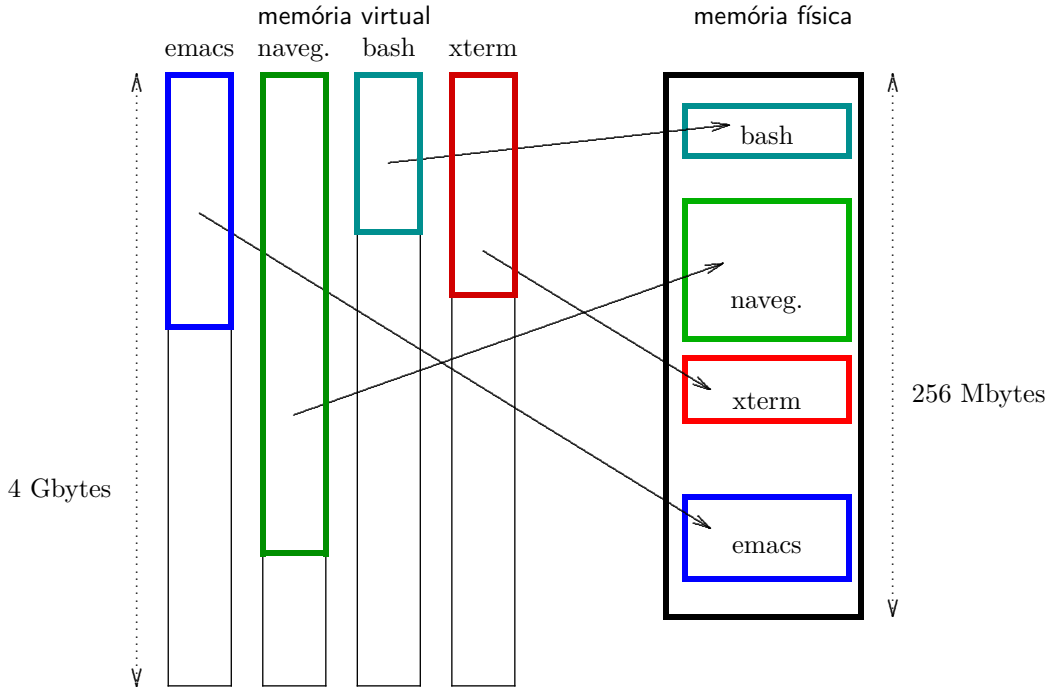


Figura 9.5: Processos de ≤ 4 Gbytes alocados em memória física de 256 Mbytes.

9.2.1 Tradução de endereços

Em sistemas com paginação, cada processo pode usar todo o espaço de endereçamento suportado pela linguagem. No caso de C em sistemas Unix, todos os processos tem seus segmentos lógicos de código, dados e pilha alocados a partir de um mesmo conjunto de endereços. Na ABI do Unix para processadores MIPS de 32 bits, o segmento de código inicia em `0x0040.0000`, o segmento de dados em `0x1000.0000`, e o de pilha em `0x7fff.fffc`.

Considere duas cópias de um mesmo programa, tal como `bash`, executando num computador. Embora os endereços virtuais dos processos `bash` coincidam – os domínios das funções de mapeamento f_{b1}, f_{b2} podem ser idênticos, cada processo tem seu espaço de endereçamento físico alocado num conjunto de páginas físicas que é disjunto daqueles dos demais processos – as imagens das f_{b1}, f_{b2} são disjuntas – e é isso o que garante a proteção.

Ao espaço de endereçamento virtual de cada processo – domínio de f – corresponde um conjunto de páginas físicas – imagem de f – que compõem o espaço de endereçamento físico do processo. O mapeamento entre endereços virtuais e físicos efetuado pela função f é implementado com uma *tabela de páginas* (TP). A TP é gerenciada exclusivamente pelo sistema operacional.

A Figura 9.6 mostra um diagrama com o mapeamento de endereços virtuais em físicos. Como as páginas são de 4Kbytes (2^{12} bytes), o mapeamento dentro de uma página é a função identidade (id) porque nenhum outro tipo de mapeamento faz sentido – a função identidade é implementada com 12 fios. Assim, o que a TP contém é um mapeamento entre os *números de páginas virtuais* e os *números de páginas físicas*, porque num endereço físico ou virtual somente os bits acima do bit 11 são potencialmente distintos, pois a UGM ignora os bits de 0 a 11.

O mapeamento é totalmente associativo: uma página virtual pode ser mapeada em qualquer página física, e duas páginas virtuais contíguas a e b podem ser mapeadas em páginas físicas a' e b' que não são contíguas.

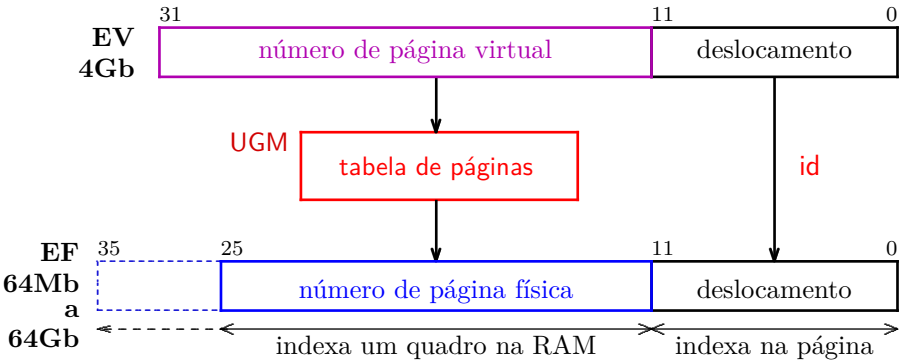


Figura 9.6: Mapeamento de endereço virtual em endereço físico.

Quando um processo é criado, o SO aloca um certo número de páginas físicas e preenche a tabela de páginas com os números destas páginas. Na medida em que o processo executa, e sua localidade se altera, se o espaço mapeado crescer, novas páginas físicas são alocadas para acomodar os acréscimos. O mesmo ocorre com a alocação de espaço na *heap* via `malloc()`; as novas páginas devem ser mapeadas através da TP.

Na sua forma mais básica a TP implementa uma função injetora cujo domínio é todo o espaço de endereçamento virtual, e cuja imagem é o conjunto de páginas físicas que estão alocadas. Esta função raramente é sobrejetora porque, neste caso, *todo* o espaço de endereçamento estaria alocado em páginas físicas, e são relativamente raros os programas que ocupam (quase) todo o espaço de endereçamento virtual[3].

A Figura 9.7 mostra a implementação do mapeamento de endereços com um pouco mais de detalhe, com páginas de $z = 4\,\text{Kbytes}$ e $M = 256\,\text{Mbytes}$ de memória física. O processador contém um registrador que aponta para a base da tabela de páginas do processo que está executando. Este registrador é chamado de *Base da Tabela de Páginas* (BaseTP).

O processador emite um endereço virtual (EV $a = \langle \text{p}, \text{w} \rangle$) de 32 bits; os 20 bits mais significativos ($p = a/z$) são somados ao conteúdo do R e o resultado aponta a posição da TP que contém o número da página física q que corresponde ao endereço virtual p. Este número é concatenado com os 12 bits menos significativos do EV ($w = a\%z$), formando o endereço físico (EF $a' = \langle q, w \rangle$) da palavra referenciada pelo processador. Se o mapeamento não for válido ($f(a) = \varphi$), então a referência provoca uma *exceção de falta de página* (*page fault*).

9.2.2 Paginação sob demanda

A memória física é um recurso escasso e é compartilhada entre todos os processos ativos no sistema. A localidade de referência permite que processos executem com uma fração relativamente pequena de seus espaços de endereçamento carregados em memória física, e esta fração é o conjunto das páginas que são acessadas ativamente num determinado intervalo de tempo.

Páginas que não são referenciadas há tempo, ou que ainda não foram referenciadas, podem ser mantidas em disco, assim liberando páginas físicas para uso pelos demais processos. O SO mantém uma lista de páginas físicas livres, e quando um processo é criado, um certo número de páginas é alocado ao novo processo. Quando o processo morre, suas páginas são devolvidas à lista de páginas livres.

Uma região do disco rígido é reservada para memória secundária, que acomoda as páginas

[3] O autor faz uso do compilador/sintetizador Quartus, da Altera, na síntese de código VHDL para FPGAs que ocupa cerca de 900 Mbytes durante sua execução.

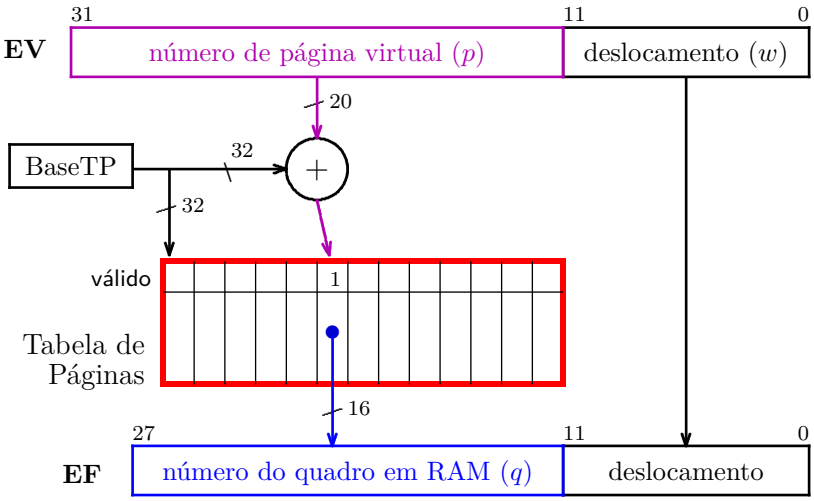


Figura 9.7: Mapeamento através da Tabela de Páginas.

dos processos que não estão carregadas em memória. Esta região é chamada de *área de troca*, ou de *swapping*, porque espaço em disco "é trocado" por espaço em RAM, e vice-versa. Geralmente, a área de *swapping* é uma partição do disco especialmente formatada para acomodar as páginas inativas dos processos. O espaço restante em disco é destinado ao sistema de arquivos, que é construído numa camada do SO que fica acima do sistema de memória virtual.

Quando um processo referencia uma página virtual que não está mapeada em memória, mas está disponível na área de *swapping*, a sequência de eventos para recuperar esta página é a seguinte: (i) a referência à página não-mapeada causa uma exceção de falta de página; (ii) a execução do processo é suspensa; (iii) o SO se encarrega de obter uma página física livre para mapear no espaço do processo causador; (iv) o SO copia a página virtual que está armazenada na área de *swapping* do disco para a página física recém alocada; (v) o novo mapeamento é inserido na TP do processo suspenso; e (vi) quando a página estiver carregada e mapeada, o processo volta a executar assim que o processador estiver disponível. A Figura 9.8 mostra um processo com páginas em memória e páginas na área de *swapping*.

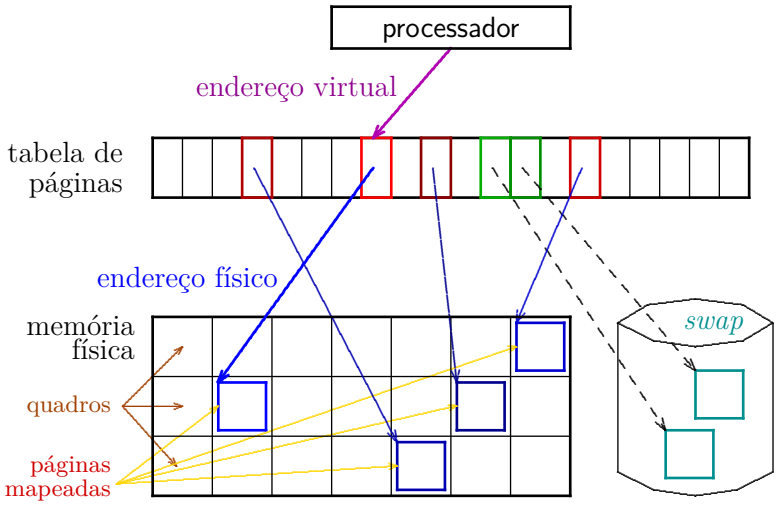


Figura 9.8: Páginas alocadas em memória física e na área de *swapping*.

Este mecanismo se chama *paginação sob demanda* e foi originalmente definido por Kilburn *et.alli* em 1962: "*...a system has been devised to make the core–drum combination*

to appear to the programmer as a single level store, the requisite transfers taking place automatically". Numa tradução atualizada: num sistema com paginação sob demanda, a combinação memória–disco é percebida pelo programador como uma memória 'plana' (sem hierarquia, *flat*), e as transferências entre disco e memória ocorrem automaticamente e de forma transparente ao programador, que codifica considerando um modelo de memória que é plano, ao invés de uma hierarquia com parte do espaço de endereçamento em RAM (acesso em Δ ciclos) e parte em disco (acesso em Ω ciclos) [**?**].

O tempo necessário para efetuar uma leitura num disco rígido (Ω) é da ordem de 10^5 a 10^6 ciclos do processador. Portanto, quando ocorre uma falta de página, o processo pode ficar esperando pela leitura do disco durante um milhão de ciclos. Este tempo poderia ser melhor empregado se o processo que sofreu a falta for suspenso, e o processador passar a executar instruções de outro(s) processo(s). A cada falta de página ocorre uma troca de contexto, e o processo que sofreu a falta é reativado depois que a leitura do disco completar. Trocas de contexto ajudam a esconder a latência dos acessos a disco. Revise o diagrama da Figura 6.10, na página 125, com os acessos ao disco efetuados por três processos.

A paginação sob demanda reduz o tempo total dispendido nos acessos a disco, ao minimizar o número de páginas que são carregadas em memória – as páginas são carregadas para a memória física somente quando o processo as referencia. Páginas inativas permanecem em memória secundária, na área de *swapping*. O princípio da localidade nos acode – se uma página física não é referenciada durante um intervalo razoável, ela é uma candidata a ser descartada e retornada para a lista de páginas livres – se a página não foi modificada – ou ser gravada em disco, caso tenha sido modificada.

9.2.3 Proteção

Uma primeira forma de proteção entre os processos é obtida com a própria tabela de páginas. Toda vez que uma referência ao endereço x retorna um mapeamento inválido, $f(x) = \varphi$, o sistema operacional verifica se a referência é para: (i) uma página que está na área de troca, quando então esta página é recarregada; (ii) uma página adicionada por `malloc()`, que é então alocada na memória principal; ou (iii) é uma referência para um endereço que não pertence ao espaço de endereçamento do processo, e neste caso o processo é abortado. A referência para um endereço que não está mapeado pode decorrer de um erro de programação, tal como de-referenciar um *pointer* nulo, ou pode ser maliciosa, numa tentativa de acessar a área de dados de outro processo. Este processo deve morrer.

Uma segunda forma de proteção depende de bits de *status* mantidos nos elementos da TP. Estes contêm, além de um número de página física, alguns bits de proteção que determinam quais referências podem ocorrer na página mapeada. Tipicamente as páginas são "só de leitura" (*read-only*, RO), com "código executável" (EX), e/ou "com dados variáveis" (*writable*, WR). Se o bit correspondente estiver desligado, uma referência do tipo proibido causa uma exceção de *violação de proteção* – tal como uma tentativa de escrita numa página com código, por exemplo.

Além do tipo de referência, acesso com o nível errado de privilégio de execução provoca uma exceção de *violação de privilégio*. Por exemplo, páginas reservadas ao SO não podem ser acessadas em modo usuário. A Figura 9.9 mostra a TP com as funções de mapeamento de endereços e de validação de referências.

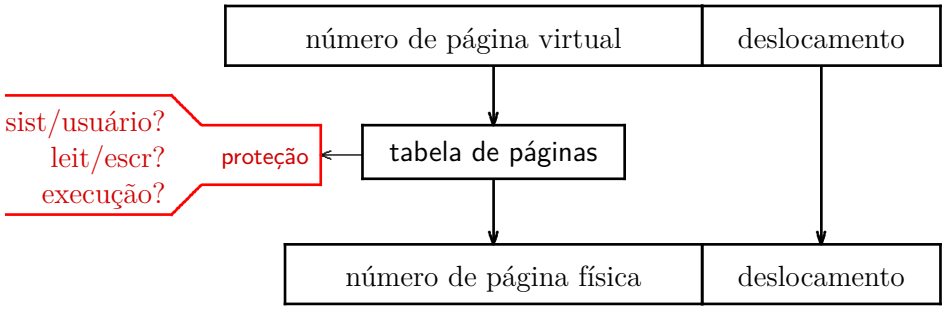


Figura 9.9: Proteção através da Tabela de Páginas.

9.2.4 Implementação da tabela de páginas

Se uma página tem 4 Kbytes, quantas são as páginas num espaço de endereçamento virtual? São necessários 12 bits para indexar todos os bytes de uma página, e em máquinas com endereços de 32 bits, sobram 20 bits para indexar todas as páginas virtuais. A TP tem 2^{20} elementos, e com 4 bytes em cada elemento, perfaz 4 Mbytes, ou 1024 páginas. Por razões de eficiência, a tabela de páginas deve ser mantida em memória física.

Cada processo necessita de sua própria TP, e por isso uma fração enorme da memória física poderia ser alocada para manter as TPs de todos os processos. Evidentemente, manter uma tabela linear com 1024 páginas para cada processo é inviável. Uma *tabela de páginas hierárquica* é uma implementação assaz eficiente, portanto popular, que reduz dramaticamente a demanda por espaço para acomodar as TPs. A Figura 9.10 mostra uma TP hierárquica que é similar à implementação empregada nos processadores da família x86.

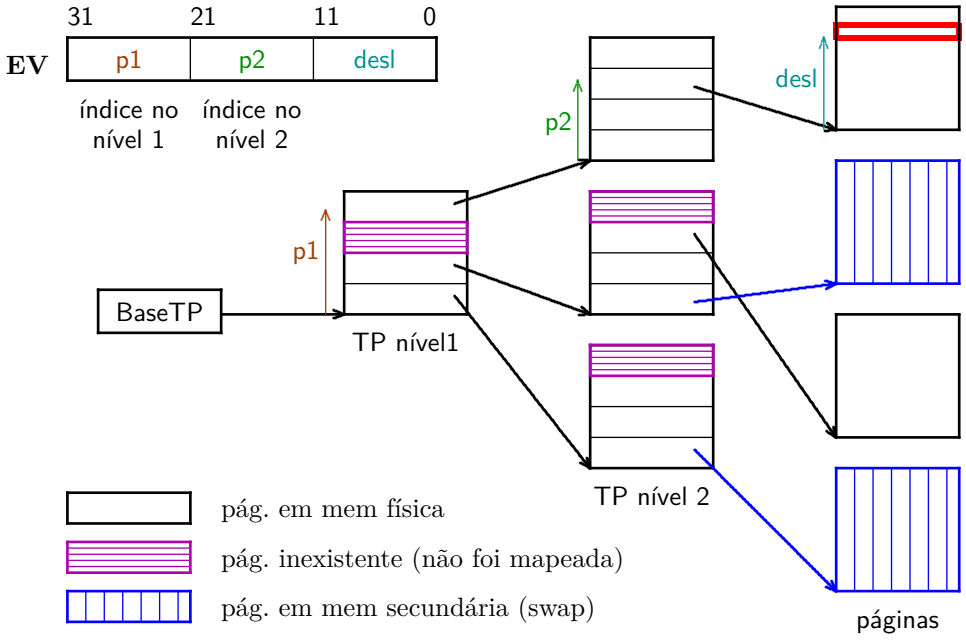


Figura 9.10: Tabela de Páginas Hierárquica.

Para acessar a TP hierárquica, um endereço virtual é dividido em três campos: o primeiro campo, com os 10 bits mais significativos do endereço, indexa a *TP de primeiro nível* (`p1`); o segundo campo, com os próximos 10 bits indexa uma das 1024 *TPs de segundo nível* (`p2`); e o terceiro campo, com os 12 bits restantes, indexa um byte dentro da página.

A TP de primeiro nível 'cabe' exatamente numa página, se cada elemento for um inteiro de 4 bytes, e esta TP aponta para $2^{10} = 1024$ TPs de segundo nível. Se cada elemento

da TP de segundo nível é um inteiro, cada uma das 1024 TPs deste nível mapeia 1024 páginas físicas. Este projeto é extremamente eficiente porque todos os componentes da TP, do primeiro e do segundo níveis, são eles próprios páginas.

Esta implementação é mais econômica do que uma tabela linear porque, num certo intervalo de tempo, apenas um subconjunto pequeno do espaço de endereçamento é referenciado pelo processo – a localidade indica que apenas uma fração de todas as páginas é referenciada numa certa janela de tempo. As regiões do espaço de endereçamento – e da TP – que não estão sendo referenciadas não necessitam ser mapeadas e portanto não ocupam espaço em memória. Partes do segundo nível da TP podem ser mantidas na área de troca – a própria TP pode ser paginada. Como mencionado acima, isso é possível porque as TPs de segundo nível são organizadas em páginas. A página com o primeiro nível é sempre mantida na memória principal.

Espaço em branco proposital.

Exemplo 9.5 Vejamos como deve ser interpretado um endereço virtual, num acesso à TP hierárquica. Lembre que um endereço de 32 bits é mostrado em 8 quartetos de bits, em formato hexadecimal. Assim, os 10 bits dos índices no primeiro e no segundo nível devem ser 'ajustados' para uma visualização correta. O diagrama na Figura 9.12 mostra como o endereço virtual `0x03c0.3200` é interpretado para acessar a TP hierárquica.

Para o primeiro campo de 10 bits, separamos os oito bits mais significativos (`0x03`) e mais dois bits do próximo quarteto (`0xc = 1100`) que são `11`. Os bits `0000.0011.11`, reagrupados em quartetos alinhados à direita ficam `00.0000.1111 = 0x00f`.

Com o mesmo procedimento, o segundo campo de 10 bits fica `00.0000.0011 = 0x003`.

O deslocamento já está em 3 quartetos: `0010.0000.0000 = 0x200`.

A Figura 9.11 mostra a separação do endereço nos três campos: p1, p2 e desl. O endereço físico correspondente é a concatenação do número da página física `0x14c38` com o deslocamento `0x200`. que é `0x14c3.8200`. ◁

0	3	c	0	3	2	0	0
0000	0011	1100	0000	0011	0010	0000	0000
0000	0011	11	00 0000	0011	2	0	0
00 0000	1111		00 0000	0011	2	0	0
00f			003			200	
p1			p2			desl	

Figura 9.11: Separação do EV nos campos para acessar a TP Hierárquica.

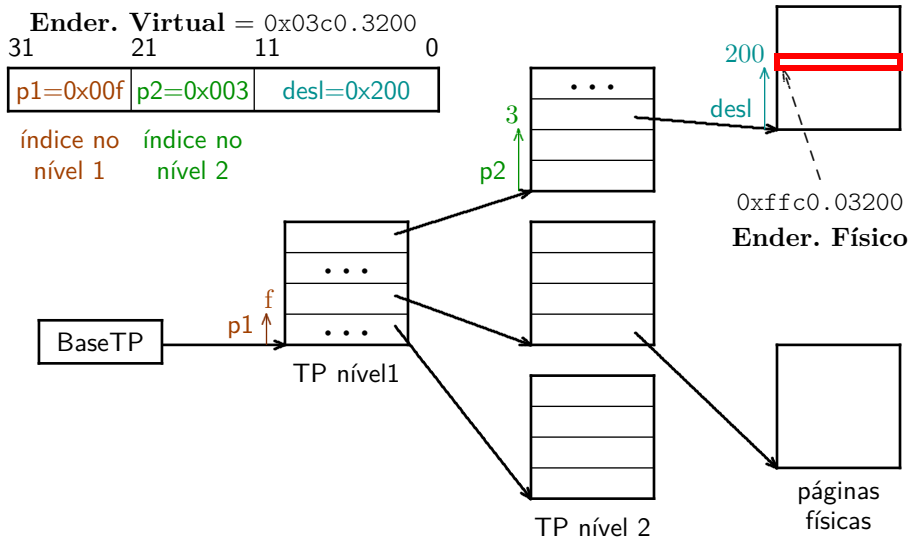


Figura 9.12: Exemplo de mapeamento em Tabela de Páginas Hierárquica.

9.3 Arquivos mapeados em memória

Uma técnica empregada para acelerar a carga de programas e para economizar memória física é o *mapeamento de arquivos em memória*. Em sistemas com memória virtual baseada em paginação, a área de *swapping* é usada para armazenar partes do espaço de endereçamento de um processo. O sistema de arquivos pode ser usado de maneira similar. Um processo solicita ao SO que um determinado arquivo seja mapeado em memória. As páginas virtuais que seriam alocadas para conter o arquivo não são inicializadas com uma

cópia do arquivo; ao invés disso, a tabela de páginas é ajustada para apontar os blocos em disco que correspondem ao arquivo, como mostra a Figura 9.13.

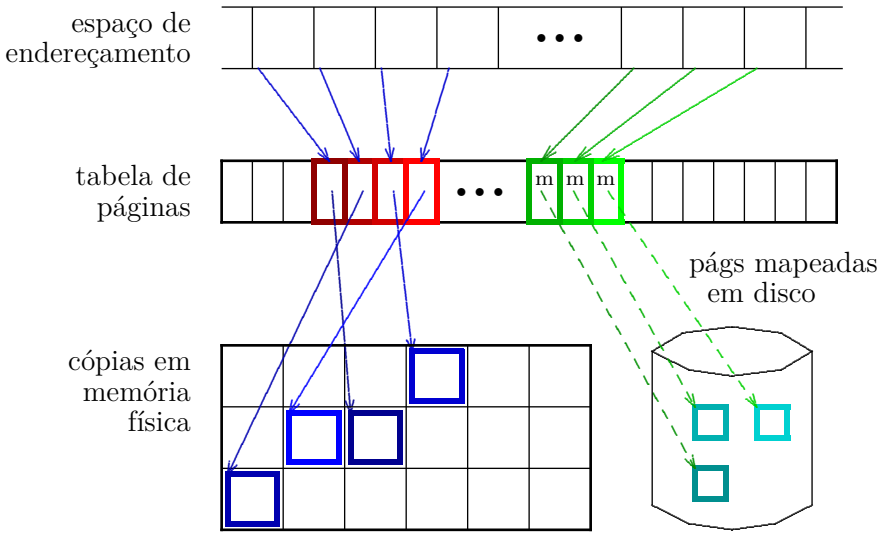


Figura 9.13: Arquivo mapeado em memória.

Se as páginas mapeadas em disco são acessadas somente para leitura, como é o caso do compartilhamento de código entre processos, não é necessária nenhuma providência adicional, além de proteger as páginas contra escrita, ligando o bit RO na tabela de páginas. O programa é interrompido se ocorrer uma escrita em páginas de código.

Se alguma das páginas é atualizada, então quando o arquivo é 'des-mapeado', as atualizações devem ser propagadas até o disco. Na tabela de páginas estas páginas devem ser marcadas como RW. Note que somente um processo pode atualizar o arquivo sob pena de que este seja corrompido por atualizações concorrentes.

Para permitir o compartilhamento para escrita, pode-se usar a técnica chamada de *copy-on-write* (COW). Com esta técnica as páginas físicas são marcadas como COW e são mapeadas no espaço de vários processos. Quando um processo atualiza sua cópia, a página que será modificada é copiada para uma página física de acesso exclusivo e esta é então atualizada. Dessa forma todos os processos compartilham o arquivo mas cada um deles possui uma versão privativa com as suas alterações.

9.4 Exercícios

Ex. 65 Considere que esta sequência de números de páginas virtuais ocorre durante a execução de um programa: `0 1 2 3 4 6 4 7 1 3 0 6 3 5 1 2 3 4 6`. Para substituir as páginas, empregue a política "substitua a página acessada menos recentemente" (LRU). Ocorre uma *falta de página* (*page fault*) quando uma página virtual que não está em memória física é referenciada; ocorre um *acerto* (*hit*) quando a página referenciada está em memória física. Calcule a taxa de acertos – número de acertos dividido pelo número total de referências – para uma memória com 4, e com 8 páginas físicas [**?**].

Ex. 66 Repita usando a política de substituição "substitua a mais velha" (FIFO).

Ex. 67 Repita usando um oráculo infalível (OPT).

Ex. 68 Para a sequência de referências do Exercício 65, qual das três políticas (LRU, FIFO, OPT) produz a menor taxa de faltas? Este resultado pode ser generalizado para qualquer sequência de referências?

Ex. 69 Escreva, em pseudocódigo, uma função com o protótipo abaixo que percorre uma tabela de páginas de dois níveis e retorna **1** se a página está em memória, ou **0** numa falta. O endereço físico é atribuído à `*enderfis` num acerto. Endereços virtuais tem 32 bits, endereços físicos tem 34 bits e páginas tem 4Kbytes [**?**].

```
int buscatp(void *basetp, void* endervirt, void** enderfis);
```

Ex. 70 Considere o programa de multiplicação de matrizes abaixo. Suponha que as matrizes contém 1024x1024 elementos, cada elemento um `double` (8 bytes). O programa é executado num único processador com páginas de 4Kbytes. (a) Descreva o comportamento do sistema de memória virtual durante a execução deste programa; (b) sugira uma ou mais maneiras de melhorar o desempenho da multiplicação de matrizes, envolvendo paginação. Explicite quaisquer suposições usadas na sua resposta.

```
for (i = 0; i < 1024; i++)
   for (j = 0; j < 1024; j++) {
      for (sum = 0, k = 0; k < 1024; k++)
         sum += a[i][k] * b[k][j];
      c[i][j] = sum;
   }
```

Ex. 71 Considere um sistema de memória virtual com as seguintes características: (i) endereço virtual de 32 bits; (ii) páginas com 4Kbytes; e (iii) endereço físico com 38 bits. A tabela deve conter bits de válido, read-only, executável, modificado e usado. (a) Qual é o tamanho de uma tabela de páginas linear? (b) Mostre como implementar a tabela de páginas hierárquica em dois níveis. (c) Suponha que na sua implementação do item (b), 3/4 dos elementos da tabela de primeiro nível sejam nulos. Quais os tamanhos máximo e mínimo do espaço de endereçamento mapeado pela tabela de páginas?

Capítulo 10

Infraestrutura para Ligadores

As próximas seções versam sobre material aparentemente disperso e desconectado, e em alguma medida isso é correto. Estes tópicos são necessários para a compreensão da construção e utilização de bibliotecas ligadas dinamicamente, que é o assunto dos dois últimos capítulos. Portanto é necessário algo de paciência.

Comecemos com as tabelas de *strings* e tabelas com espalhamento (*hashing*), e então uma visão geral do processo desde a compilação até a execução.

10.1 Tabela de cadeias de caracteres

A estrutura de dados empregada para armazenar cadeias de caracteres (*strings*) é composta de duas partes: (i) um vetor com apontadores; e (ii) um vetor com as cadeias propriamente ditas. Cada elemento do vetor de apontadores aponta para o endereço do primeiro caractere de uma cadeia, e este endereço é usado para acessar a cadeia propriamente dita.

Exemplo 10.1 O vetor com seis elementos, mostrado na Figura 10.1, contém apontadores para cinco cadeias, armazenadas a partir do endereço 0x100. Cada elemento de `V` aponta para o endereço do primeiro caractere da *string* que lhe corresponde. O último elemento do vetor aponta para uma cadeia vazia, que sinaliza o final da área de armazenagem. Quais são as virtudes desta construção? ◁

```
vetor[100,106,10a,110,112,117]

100:  abcde\0
106:  xyz\0
10a:  pqrst\0
110:  x\0
112:  mnop\0
117:  \0
```

Figura 10.1: Exemplo de vetor de cadeias.

10.2 Tabelas com espalhamento

Uma vez que as cadeias de caracteres podem ser armazenadas numa tabela de cadeias, nos falta uma estrutura para armazenar os nomes das funções e variáveis de um programa. Por

razões que veremos em breve, tal estrutura deve permitir buscas eficientes.

Como os nomes de funções em C iniciam com caracteres alfabéticos, além de '_' e '.', poderíamos indexar a tabela pela primeira letra do nome do símbolo. Esta tabela deve permitir a armazenagem de mais de um símbolo que inicie com uma certa letra – cada elemento da tabela é a cabeça de uma lista encadeada, e cada elemento da lista aponta para um elemento da tabela de cadeias. As listas encadeadas são chamadas de *listas de colisão*. Uma tabela com espalhamento é mostrada na Figura 10.2. Para simplificar o diagrama, a tabela contém somente minúsculas.

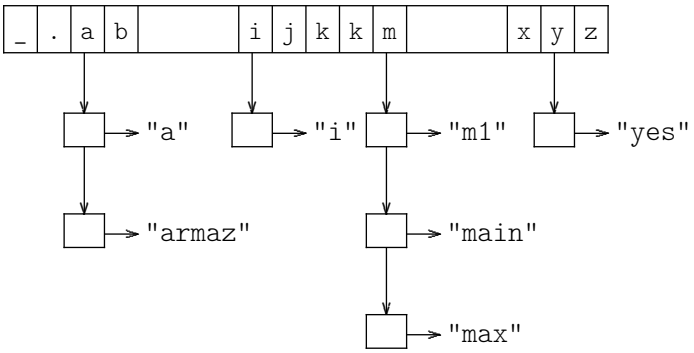


Figura 10.2: Tabela indexada pelo início do nome.

Para buscar um símbolo da tabela, basta indexar a tabela com a primeira letra do nome, e então percorrer a lista encadeada, comparando as cadeias armazenadas com a cadeia procurada. Aqui enfrenta-se um problema de ordem prática: alguns nomes são mais populares do que outros, e as letras mais populares podem conter listas longas, enquanto que outras podem conter listas vazias. Listas longas implicam em buscas demoradas, portanto ineficientes.

Uma melhor solução consiste em usar uma função que 'espalhe' mais uniformemente a alocação de símbolos aos elementos da tabela. Esta *função de espalhamento* (*hashing*) pode ser tão simples como a soma dos caracteres da cadeia, módulo o tamanho da tabela.

Uma palavra sobre a notação: um vetor é denotado por uma maiúscula (V) e cada um dos seus elementos por uma minúscula com o índice de sua posição no vetor (v_i). Se V tem n elementos, então $V = v_0, v_1, \ldots v_{n-1}$, ou ainda $V = v_i,\ 0 \leq i < n$.

Considere uma função de espalhamento h tal que, para cada símbolo S, o valor $h(S)$ é dado pela soma (com, ou sem vai-um) dos valores binários dos caracteres. Usando-se soma com vai-um, $h(S) = \sum s_i$, ou usando-se soma sem vai-um (ou-exclusivo), $h'(S) = \bigoplus s_i$. Estas são funções demasiadamente simples mas servem como exemplo.

Além dos apontadores para o próximo elemento da lista encadeada, e para a cadeia de caracteres, o elemento da tabela que representa o símbolo S armazena também o valor de $h(S)$. Um elemento da tabela de símbolos é mostrado no Programa 10.1. Tipicamente, o tamanho destas tabelas é uma potência de dois, para simplificar o cálculo do módulo.

Programa 10.1: Elemento da Tabela de Símbolos.

```
typedef struct sym {
   struct sym *next; // aponta próximo na lista de colisões
   int fullhash;     // valor de h(S) para acelerar buscas
   char *symname;    // aponta cadeia com nome do símbolo
} symTab;

struct symTab *symhash[256];
```

Exemplo 10.2 A posição do símbolo "main" numa tabela com 256 elementos é dada por $I = h(\texttt{main})$. O índice de "main" numa tabela com 256 elementos é 165 porque

$$h(\texttt{'m'+'a'+'i'+'n'}) = 421, \;\; 421 \,\%\, 256 = 165.$$

É (muito) provável que a função $h(\,)$ produza o mesmo valor I para mais de um símbolo (main, mina, mani, iman), porque sua imagem é o tamanho da tabela, que é finito. Assim, cada elemento na imagem de $h(\,)$ deve ser cabeça de uma lista de colisão.

Na busca por um símbolo S, o valor de $K = h(S)$ é computado e a tabela indexada com aquele valor. Cada elemento da lista de colisões é comparado com K, e se o valor armazenado é igual a K, então o símbolo (talvez) foi encontrado – veja o próximo parágrafo. Se K não é encontrado, então S não está na lista e deve ser inserido nela. ◁

Nomes usados em programas podem ser longos, o que torna a pesquisa lenta; se o valor da função $h(S) = I$ é armazenado junto ao símbolo (em sym.fullhash), então durante uma busca compara-se o valor da função $h(S)$ computada para o novo símbolo com o valor armazenado em sym.fullhash. Se os dois inteiros são iguais, só então compara-se as duas cadeias. Por que esta segunda comparação é necessária?

10.3 O Processo Completo: do Código Fonte à Carga

Quando se empregam boas práticas de programação, um programa complexo é dividido em vários *módulos* e cada módulo define uma estrutura de dados e um conjunto de operações sobre aquela estrutura. Do ponto de vista da organização, um módulo consiste de um ou mais arquivos com código C, mais um arquivo com definições de tipos de dados, de funções e de constantes.

Ao invocar o compilador, o que é executado é o chamado "*driver* de compilação" – um programa que coordena a execução de vários passos de tradução e otimização para produzir um executável a partir de um ou mais arquivos com código fonte. As operações necessárias podem ser agregadas em três etapas. A primeira etapa é a *compilação* do código fonte, no qual os comandos na linguagem de alto nível são traduzidos para *assembly*. A segunda etapa é a *montagem* dos programas em *assembly* que são traduzidos para *arquivos objeto*, que contém o código traduzido para binário. A terceira etapa é a *ligação* de vários arquivos objeto para produzir um *executável*.

Após a ligação, o executável deve ser carregado em memória para então iniciar sua execução. O *carregador* é a parte do sistema operacional que instala em memória a imagem executável do programa que foi compilado, e se for o caso, efetua a sua transformação num processo.

Símbolos e endereços Um *símbolo* corresponte ao nome de uma função ou de uma variável. O *valor* de um símbolo é o endereço da função ou variável. Neste contexto, o valor do símbolo "β" corresponde ao endereço no qual a variável β foi alocada, e não ao conteúdo daquela variável. No caso de funções, β corresponde ao endereço da primeira instrução da função.

Se o endereço correspondente a um símbolo é desconhecido, porque o processo de ligação ainda não resolveu todos os endereços, o símbolo é dito *indefinido*. Estritamente falando, um programa com símbolos indefinidos não pode executar porque referências aos endereços indefinidos causariam falhas de segmentação e/ou erros de endereçamento. Sistemas que usam bibliotecas para ligação dinâmica são um tanto mais liberais quanto a símbolos indefinidos, como nos revela o Capítulo 15.

O Figura 10.3 mostra o processo de compilação (cpp e gcc), montagem (as), ligação (ld), e carga de um programa (carr). Os próximos parágrafos descrevem cada um destes passos.

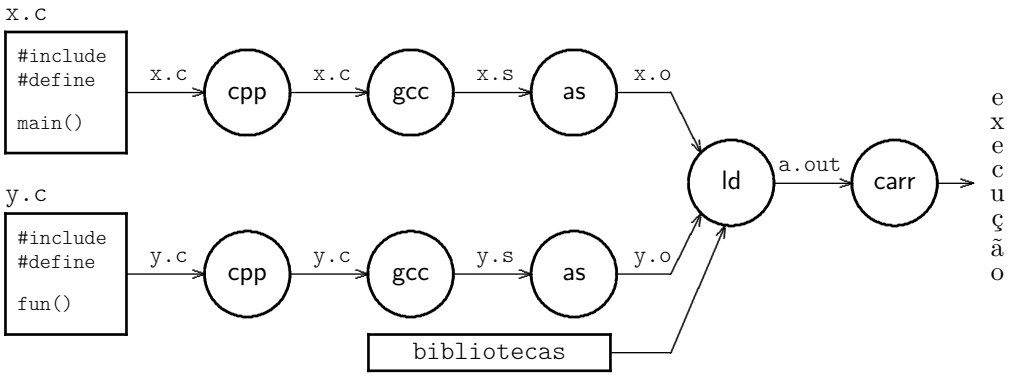


Figura 10.3: Etapas do processo de compilação.

Compilação O preprocessador cpp faz a expansão de macros e a inclusão de arquivos, e o compilador gcc traduz o código fonte para uma versão em *assembly* que é logicamente equivalente ao programa em C. O arquivo `x.s` produzido pelo compilador contém as declarações dos símbolos que devem ser exportados para outros módulos (`.global`), para que estes fiquem visíveis aos outros módulos, mais uma lista de símbolos que devem ser importados de outros módulos (`.extern`), porque estes símbolos não são definidos no arquivo compilado `x.c`.

Montagem O montador traduz a versão *assembly* – que é uma representação simbólica do programa – para código binário e gera um arquivo objeto `x.o`, que não é executável. O arquivo objeto gerado pelo montador contém uma tabela de símbolos que é construída a partir das declarações de funções e variáveis globais (`.global`). Além da tradução de *assembly* para binário, o montador efetua uma série de operações que facilitam a vida do programador:

- expandir as pseudoinstruções e as macros;
- efetuar conversões de base; e
- computar operações aritméticas com endereços e/ou com símbolos.

O arquivo objeto (`x.o`) produzido pelo montador é dividido em várias *seções*:

- um cabeçalho que descreve o tamanho e posição das demais seções;
- seção de texto (`.text`) com o código do programa;
- seção de dados estáticos inicializados (`.data`);
- seção de dados estáticos não-inicializados (`.comm`, `.bss`);
- informação de relocação que identifica instruções e variáveis que dependem de endereços absolutos;
- tabela de símbolos com referências indefinidas a endereços externos (`.extern`); e
- quando relevante, informações de depuração para permitir associação de instruções com comandos em C, e descrição do leiaute das estruturas de dados.

Exemplo 10.3 Compile um programa que não usa nenhuma biblioteca com o comando

```
mips-gcc -O2 -c prog.c -o prog.o
```

e examine as seções do arquivo objeto gerado com o comando

```
mips-objdump -D -EL -h -x -t -show-raw-insn prog.o
```

◁

Ligação O ligador ld concatena os arquivos objeto `x.o` e `y.o`, ajusta as referências externas nos arquivos objeto, e produz o executável `a.out`. As funções do ligador são:

- alocação de memória – as seções de texto e dados de todos os arquivos objeto são concatenadas e alocados na memória, resultando numa primeira versão do mapa de memória do processo;
- relocação – uma vez determinados os endereços do texto e dos dados, o ligador ajusta os endereços nas referências internas – que apontam para endereços internos ao módulo, e nas referências externas – que apontam para símbolos globais que estão definidos em outros módulos;
- geração do arquivo executável – o arquivo que contém todas as seções e/ou segmentos definidos pelo formato padrão do sistema (ELF ou a.out) é gravado. Estes formatos são apresentados no Capítulo 11.

Os arquivos objeto são divididos em *seções* para código (`.text`) e dados (`.data`, `.comm`, `.bss`). O ligador concatena as seções de código de todos os objeto e produz o *segmento* de código que é chamado `TEXT`. Da mesma forma, todos as *seções* de dados são concatenadas no *segmento* de dados, que é chamado de `DATA`.

Carga O carregador carr lê o arquivo executável e instala a imagem executável em memória. O carregador inicializa o espaço de endereçamento, efetua a ligação com bibliotecas (se for o caso) e dispara a execução do processo. O carregador é invocado implicitamente pela execução das funções de sistema `fork()` e `execve()`. O carregador tem as seguintes funções:

- ler o cabeçalho do executável e determinar o tamanho dos segmentos de texto e dados;
- criar o espaço de endereçamento com tamanho adequado;
- copiar instruções e dados inicializados para memória;
- criar o ambiente de execução do processo e inserir os argumentos do programa na pilha (Seção 10.4);
- inicializar registradores do processador e ajustar o apontador de pilha; e
- saltar para a rotina de inicialização (`start.s`), que prepara o ambiente em que `main()` executará. Quando `main()` retorna, é executada uma rotina de finalização, que contém uma chamada de sistema `exit()` para encerrar a execução do processo.

10.3.1 Exemplo de compilação e ligação

O que se segue é um exemplo que ilustra, de forma muito simplificada, as operações necessárias a cada passo do processo de compilação-ligação.

Relembrando: programas complexos e grandes são divididos em *módulos* e cada módulo define uma estrutura de dados e um conjunto de operações sobre aquela estrutura. Programadores usam as funções definidas no módulo para consultar e/ou alterar os conteúdos da estrutura de dados. A documentação do módulo descreve sua *Application Programming Interface* (API), que define a funcionalidade das funções bem como seus parâmetros e resultados. Geralmente, a cada módulo correspondem dois arquivos, um arquivo de cabeçalho (*headers*) e um ou mais arquivos com o código fonte das funções do módulo.

O programa do nosso exemplo está dividido em três módulos, cada módulo contido num arquivo com código fonte C. Neste exemplo, os valores são transferidos entre as funções através dos argumentos e dos valores de retorno, bem como através das variáveis globais declaradas nos módulos. Esta não é uma boa maneira de organizar e escrever programas

e é usada aqui apenas para indicar como os símbolos são tratados durante a compilação. Uma variável global é usualmente exportada – seu endereço é disponibilizado ao ligador e portanto aos demais módulos – mas só é importada se for declarada como **extern**.

O Módulo 1 (Prog. 10.2) exporta os símbolos `k,l,m` e importa os símbolos `r` (declarado explicitamente como **extern**), `f()`, `g()` e `printf()`, declarados como funções importadas. O Módulo 2 (Prog. 10.3) exporta os símbolos `p,q,r` e `f()` e importa os símbolos `y` (declarado como **extern**), e as funções importadas `g()` e `printf()`. O Módulo 3 (Prog. 10.4) exporta os símbolos `x,y,z` e a função `g()` e importa os símbolos `k` (declarado como **extern**), e `printf()`.

Programa 10.2: Módulo 1, no arquivo `mod1.c`.

```
int k[64], l, m; // variáveis globais

int f(int, int); // definida em outro módulo
int g(int, int); // definida em outro módulo
void printf(char *, ...);
extern int r;    // definida em outro módulo

int main(int argc, char **argv) {
   k[0] = 2;
   l = f(k[1], r);
   m = g(l, 5);
   printf("k=%d␣l=%d␣m=%d\n", k[0], l, m);
   return(0);
}
```

Programa 10.3: Módulo 2, no arquivo `mod2.c`.

```
int p, q, r;     // variáveis globais

int g(int, int); // definida em outro módulo
void printf(char *, ...);
extern int y;    // definida em outro módulo

int f(int i, int j) {
   printf("arquivo␣2:␣i=%d␣j=%d\n", i, j);
   return( g(i, j/y) );
}
```

Programa 10.4: Módulo 3, no arquivo `mod3.c`.

```
int x[128], y, z; // variáveis globais

void printf(char *, ...);
extern int k[];  // definida em outro módulo

int g(int i, int j) {
   printf("arquivo␣3:␣i=%d\n", i);
   return( i*k + k[j] );
}
```

Quando um módulo é compilado, além do código traduzido para *assembly*, o arquivo resultante contém uma tabela de símbolos cujo conteúdo é mostrado na Figura 10.4, para os Módulos 1, 2 e 3.

Cada elemento da tabela possui três atributos: o primeiro define se o símbolo é um endereço de uma variável na pilha (*auto*), ou em memória (*data*), ou de uma função (*text*); o segundo atributo informa se o endereço do símbolo é definido no seu módulo ('–' ou *global*), ou se o endereço não é definido neste módulo (*indef*); o terceiro atributo é o *valor* do símbolo, que é um endereço. Um símbolo com atributo *global* é o nome para um endereço estático e que é exportado pelo módulo; um símbolo com o atributo *auto* é uma variável local ao módulo, e que é alocada na pilha da função em que é declarado.

Módulo 1
argc, auto, –, sp+40
argv, auto, –, sp+44
k, data, global, ?
l, data, global, ?
m, data, global, ?
main, text, global, ?
f, text, indef, ?
r, data, indef, ?
g, text, indef, ?
printf, text, indef, ?

Módulo 2
i, auto, –, sp+8
j, auto, –, sp+12
p, data, global, ?
q, data, global, ?
r, data, global, ?
y, data, indef, ?
f, text, global, ?
g, text, indef, ?
printf, text, indef, ?

Módulo 3
i, auto, –, sp+8
j, auto, –, sp+12
x, data, global, ?
y, data, global, ?
z, data, global, ?
k, data, indef, ?
g, text, global, ?
printf, text, indef, ?

Figura 10.4: Tabelas de símbolos dos três módulos antes da ligação.

A Figura 10.5 mostra trechos do código gerado para os três módulos. As seções de código (`.text`) iniciam no endereço `0x000` e as seções de dados (`.data`) em `0x800`. A Seção 10.4 descreve o registro de ativação de `main()`, que é distinto daquele de uma função comum.

```
# módulo 1
     .text
     .global main
     .extern r,f,g
     .extern printf
main:
000: addi sp,sp,-64
     ...
010: la s2, k
014: lw a0, 4(s2)
018: la t1, r
01c: lw a1, 0(t1)
020: jal f
024: nop
028: sw v0, 256(s2)
     ...
1f8: jr, ra
1fc: addi sp,sp,64

     .data
     .global k,l,m
     .org 0x800
k:   .space 0x100
l:   .space 4
m:   .space 4
```

```
# módulo 2
     .text
     .global f
     .extern y,g
     .extern printf
f:
000: addi sp,sp,-16
     ...
020: la s3, p
024: lw a0, 0(s3)
028: la t2, y
02c: lw a1, 0(t2)
030: jal g
034: nop
038: sw v0, 8(s3)
     ...
0f8: jr, ra
0fc: addi sp,sp,16

     .data
     .global p,q,r
     .org 0x800
p:   .space 4
q:   .space 4
r:   .space 4
```

```
# módulo 3
     .text
     .global g
     .extern k
     .extern printf
g:
000: addi sp,sp,-32
     ...
040: la s4, x
044: lw a0, 512(s4)
048: la s5, k
04c: lw a1, 0(s5)
     ...
2f8: jr, ra
2fc: addi sp,sp,32

     .data
     .global x,y,z
     .org 0x800
x:   .space 0x200
y:   .space 4
z:   .space 4
```

Figura 10.5: Código gerado para os três módulos.

O ligador aloca memória para os três módulos ao concatenar as três seções com código (`.text`), e as três seções com dados (`.data`). Uma vez concatenadas as seções `.text` e `.data`, ficam determinados os endereços de todas as funções e de todas as variáveis, e então o ligador percorre os segmentos e resolve todos os símbolos que estavam indefinidos nos arquivos objeto dos três módulos. Além dos símbolos definidos nos três arquivos objeto, o ligador deve acrescentar ao programa a biblioteca `stdlib.a` que contém a função `printf()`. Isso feito, o ligador gera o arquivo executável. A Figura 10.6 mostra arquivo objeto resultante, *sem* o código da biblioteca. O endereço inicial do segmento `TEXT` é `0x1000` e o endereço inicial do segmento `DATA` é `0x8000`.

```
# TEXT segment
main: # módulo 1
1000: addi sp,sp,-64
      ...
1010: la s2, k
1014: lw a0, 4(s2)
1018: la t1, r
101c: lw a1, 0(t1)
1020: jal f
1024: nop
1028: sw v0, 256(s2)
      ...
11f8: jr, ra
11fc: addi sp,sp,64

f:    # módulo 2
1200: addi sp,sp,-16
      ...
1220: la s3, p
1224: lw a0, 0(s3)
1228: la t2, y
122c: lw a1, 0(t2)
1230: jal g
1234: nop
1238: sw v0, 8(s3)
      ...
12f8: jr, ra
12fc: addi sp,sp,16

g:    # módulo 3
1300: addi sp,sp,-32
      ...
1340: la s4, x
1344: lw a0, 512(s4)
1348: la s5, k
134c: lw a1, 0(s5)
      ...
15f8: jr, ra
15fc: addi sp,sp,32
```

```
# DATA segment
8000: .space 0x100 // k[64]
8100: .space 4     // l
8104: .space 4     // m

8108: .space 4     // p
810c: .space 4     // q
8110: .space 4     // r

8114: .space 0x200 // x[128]
8314: .space 4     // y
8318: .space 4     // z
```

Figura 10.6: Leiaute do arquivo objeto com os três módulos.

A Figura 10.7 mostra a tabela de símbolos com os valores dos símbolos após a alocação e a resolução dos símbolos pelo ligador. No final da ligação, o ligador percorre os segmentos e edita os endereços que estavam indefinidos, corrigindo todas as referências à variáveis e à funções. O arquivo `a.out` resultante da ligação pode ser carregado em memória para execução porque não contém nenhum símbolo/endereço indefinido.

Módulo 1
argc, auto, –, sp+40
argv, auto, –, sp+44
k, data, –, 8000
l, data, –, 8100
m, data, –, 8104
main, text, –, 1000
f, text, –, 1200
r, data, –, 8110
g, text, –, 1300
printf, text, –, 5000

Módulo 2
i, auto, –, sp+8
j, auto, –, sp+12
p, data, –, 8108
q, data, –, 810c
r, data, –, 8110
y, data, –, 8314
f, text, –, 1200
g, text, –, 1300
printf, text, –, 5000

Módulo 3
i, auto, –, sp+8
j, auto, –, sp+12
x, data, –, 8114
y, data, –, 8314
z, data, –, 8318
k, data, –, 8000
g, text, –, 1300
printf, text, –, 5000

'–' são símb. resolvidos

Figura 10.7: Tabelas de símbolos dos três módulos após a ligação.

10.4 Interface sistema–`main()`

Esta seção descreve o mecanismo usado pelo carregador para passar os argumentos de linha de comando para a função `main()`. Além dos argumentos em `argv[]`, `main()` executa num ambiente que é herdado do processo pai de `main()`. O *ambiente de execução* é uma lista de pares $\langle$nome, valor$\rangle$, *viz.* $\langle$ $\langle$`HOME`, `"/home/roberto"`$\rangle$, $\langle$`SHELL`, `"/bin/bash"`$\rangle$, ... $\rangle$.

A função `main()` aceita dois argumentos, `argc` e `argv`. `argc` é um inteiro que indica o número de componentes em `argv[]`, e `argv[]` é um vetor de cadeias, e cada um de seus elementos é uma cadeia que contém um dos argumentos de linha de comando, incluindo `argv[0]=nomeDoPrograma`.

Exemplo 10.4 Considere um programa que copia uma cadeia de caracteres sobre uma outra. Este programa seria invocado com a linha de comando:

```
cpcad pqrstuv ABCDEFG
```

O carregador preenche a pilha inicial de `cpcad()` com os valores de `argc` e `argv` no registro de ativação de `main()`. A Seção 10.4.1 mostra o registro de ativação completo.

Conforme definido na ABI do MIPS [**?**], ao iniciar a execução, `argc=3` é atribuído ao registrador `a0`, e ao registrador `a1` é atribuído `*argv`, com o endereço na lista de cadeias. Os três argumentos são `argv[0]="cpcad"`, `argv[1]="pqrstuv"`, `argv[2]="ABCDEFG"`, e `argv[3]=0`, como mostra a Figura 10.8. ◁

	pilha	*conteúdo*
	...	
	0x0000.0000	`argv[3]=0`, palavra com zero
	G\0--	dois bytes de enchimento ou *padding*
	CDEF	
	v\0AB	`argv[2] = "ABCDEFG"`
	rstu	
	d\0pq	`argv[1] = "pqrstuv"`
	cpca	`argv[0] = "cpcad"`, nome do executável
	...	
sp →	3	3 argumentos: `"cpcad"`, `"pqrstuv"`, `"ABCDEFG"`

Figura 10.8: Parâmetros de `cpcad()` em seu registro de ativação.

10.4.1 Registro de ativação de `main()`

Um *processo* é um programa em execução e consiste de uma cópia em memória do código do "programa" que está sendo executado, de áreas de memória para dados e para a pilha, além de estruturas de dados gerenciadas pelo sistema operacional.

Quando um processo é criado, com a chamada de sistema `execve()`, os argumentos para esta função são o nome do programa – o caminho completo até o executável, a lista de argumentos, e uma lista de cadeias com o ambiente de execução. O ambiente de execução e os argumentos são colocados na pilha do processo antes que este inicie a execução.

A pilha inicial de um processo é organizada como mostra a Figura 10.9 [**?**].

endereço alto	não especificado
Bloco de Informação	cadeias com argumentos
(com tamanho variável)	cadeias do ambiente
	informação auxiliar
	não especificado
Vetor auxiliar	elemento nulo
(elementos são doubleword)	vetor auxiliar
	palavra de zeros
Vetor de (apontadores para) ambiente	um endereço para cada elemento
	palavra de zeros
Vetor de (apontadores para) argumentos	um endereço para cada elemento
sp →	número de argumentos ‡
	indefinido

‡ alinhado como *double-word*

Figura 10.9: Registro de ativação de `main()`.

Os componentes do *Bloco de Informação* não ficam numa ordem específica. O sistema pode deixar uma quantidade indeterminada de espaço entre o elemento nulo do vetor auxiliar e o início do Bloco de Informação.

O *Vetor de Argumentos* e o *Vetor de Ambiente* são usados para transmitir informação do processo pai para o processo filho. O *Vetor Auxiliar* é usado para transmitir informação do SO para o processo. Este vetor é composto por elementos definidos no Programa 10.5.

Programa 10.5: Vetor auxiliar.

```
typedef struct {
   int a_type;
   union {
      long a_val;
      void *a_ptr;
      void (*a_fcn)();
   } a_un;
} auxv_t;
```

Alguns dos valores de `a_type` são:

`AT_NULL` último elemento do Vetor Auxiliar;
`AT_IGNORE` elemento não significa nada; `a_un` é indefinido;
`AT_PAGESZ` `a_val` contém o tamanho da página de memória virtual em bytes;
`AT_ENTRY` `a_ptr` contém endereço inicial do programa;

`AT_UID` `a_val` contém UID real do usuário;
`AT_GID` `a_val` contém GID real do usuário; e
`AT_EXECFD` `a_val` contém um descritor de arquivo, herdado do processo pai.

Exemplo 10.5 A pilha inicial de um programa que é invocado com "`cp src dst`" é mostrada na Figura 10.10. A pilha é alocada em `0x7fc0.0000`, o programa recebe duas variáveis de ambiente `HOME=/home/dir` e `PATH=/home/dir/bin:/usr/bin`, e seu vetor auxiliar é inicializado com o descritor de arquivo número 13 (`a_type = AT_EXECFD = 2`). Note que `argv[argc]=0`. ◁

	b0 b1 b2 b3	*quatro bytes numa palavra*
0x7fc0.0000	...	*endereço alto*
	n \0 *pad pad*	dois bytes de enchimento
	r / b i	
	: / u s	
0x7fbf.fff0	/ b i n	
	/ d i r	
	h o m e	
	T H = /	
0x7fbf.ffe0	r \0 P A	PATH=/home/dir/bin:/usr/bin\0
	e / d i	
	/ h o m	
	O M E =	
0x7fbf.ffd0	s t \0 H	HOME=/home/dir\0
	r c \0 d	
	c p \0 s	cp\0 src\0 dst\0
	0	espaço vazio
0x7fbf.ffc0	0	vet_aux[1] = {0,0} (elmto. nulo)
	13	
	2	vet_aux[0] = {2,13} (file descr.)
	0	separador
0x7fbf.ffb0	0x7fbf.ffe2	"PATH=..."
	0x7fbf.ffd3	vetor de ambiente: "HOME=..."
	0	separador, argv[argc]==0
	0x7fbf.ffcf	"dst"
0x7fbf.ffa0	0x7fbf.ffcb	"src"
	0x7fbf.ffc8	vetor de argumentos: "cp"
sp→	3	número de argumentos

Figura 10.10: Pilha inicial do processo `"cp src dst"`.

Exceto para aqueles registradores especificados abaixo, os registradores de inteiros e de ponto flutuante contém valores indeterminados quando um processo inicia sua execução – o SO *não inicializa registradores em zero*. Ao iniciar, os registradores listados abaixo contém os seguintes valores.

`v0 (r2)` um valor diferente de zero especifica um ponteiro para uma função que o processo deve registrar com a chamada `atexit()`. Se `v0` contém zero, não é necessário fazer nada ao final do programa;
`a0 (r4)` `argc`;
`a1 (r5)` `*argv`;
`sp (r29)` contém o endereço da base da pilha, alinhado como *double-word* (8 bytes); e
`ra (r31)` inicializado com zero para que depuradores possam identificar a base da pilha.

O endereço da pilha de um processo é definido arbitrariamente a cada execução, e portanto processos devem usar o endereço em `sp`. O conteúdo da memória abaixo do endereço apontado por `sp` é indeterminado.

10.4.2 Alocação de memória no MIPS

Quando um programa é compilado num sistema Unix–MIPS, as seções de texto dos arquivos objeto são agrupadas no segmento `TEXT`, as seções com dados inicializados são agrupadas no segmento `DATA`. Além de **`.data`**, a seção chamada de BSS (*Block Started by Symbol*, **`.bss`**) acomoda variáveis não inicializadas. Esta seção não é armazenada no arquivo objeto, apenas o seu tamanho é registrado. Quando o programa é carregado, o carregador inicializa com zeros toda a região da memória que corresponde às seções **`.bss`** dos módulos.

Normalmente, o início do segmento de texto é alocado no início de uma página, e o tamanho deste segmento deve ser arredondado para um número inteiro de páginas. Da mesma forma, o segmento de dados também é alocado numa fronteira de página, seguido imediatamente pelo BSS. Em média, meia página é desperdiçada no segmento de texto, e mais meia página no dados/BSS.

Em processadores como o MIPS, nos quais o campo de deslocamento nas instruções **`lw`** e **`sw`** tem 16 bits, o compilador cria duas áreas de dados estáticos, uma para *dados pequenos* e outra para o restante dos dados. A definição de *pequeno* é escolhida pelo programador do SO ou da(s) biblioteca(s), e tipicamente é de 4 ou 8 bytes.

Por convenção, o registrador `r28` é usado como *global pointer* (`gp`) e aponta para o centro da *área global*, com 64 Kbytes, que é usada para conter as "variáveis globais pequenas", além de estruturas de dados usadas na ligação dinâmica de programas. Para acessar uma variável alocada na área global, o compilador usa somente uma instrução **`lw`** ao invés de duas (**`lui`**`;`**`lw`**) ou três (**`lui`**`;`**`ori`**`;`**`lw`**), tornando o código mais rápido.

Os dados estáticos pequenos são alocados no entorno do endereço `0x1000.8000`. Durante a carga do programa, o registrador `gp` é inicializado com este endereço, e variáveis estáticas podem ser acessadas diretamente com relação ao `gp`, com deslocamentos de ±32Kbytes.

Depois que o registrador `gp` é inicializado, o programa pode referenciar variáveis estáticas com instruções como **`lw`** `s1,200(gp)` ou **`sw`** `s2,-800(gp)`. Nos acessos à variáveis fora do alcance do *global pointer*, o endereço pode ser construído com um par de instruções:

```
lui t0, %hi(ender)     # 16 bits mais significativos de ender
lw  s3, %lo(ender)(t0) # 16 bits menos significativos
```

As variáveis pequenas não inicializadas são armazenadas na seção chamada *small BSS* (**`.sbss`**), que é alocada na região alcançada pelo deslocamento de ±32Kbytes com relação ao *global pointer*.

O Programa 10.6 mostra duas sequências de instruções, uma com o acesso a uma "variável pequena" e armazenada na área global e outra com acesso a uma variável que é armazenada fora dela.

Programa 10.6: Acesso na área global e fora dela.

```
# a = a + 40;          // a é uma variável pequena
lw   r5, desl_a(gp)   # desl_a é deslocamento na Global Area
addi r5, r5, 40       #   com relação ao apontador gp
sw   r5, desl_a(gp)

# b = b + 40;          // b não é uma variável pequena
lui  r1,0x6060        # r1 = %hi(6060.4040) = 0x6060
lw   r5,0x4040(r1)    # desl = %lo(6060.4040) = 0x4040
addi r5, r5, 40
sw   r5, 0x4040(r1)
```

10.4.3 Mapa de memória

Tipicamente, o espaço de endereçamento de sistemas com processadores MIPS de 32 bits é dividido em cinco segmentos [**?**], como mostra a Figura 10.11.

O registrador `gp` é inicializado com `0x1000.8000`, o que permite acesso rápido à faixa entre `0x1000.0000` e `0x1001.0000`, que é a área global. Dados dinâmicos são alocados com a função de biblioteca `malloc()`, que usa a chamada de sistemas `sbrk()` para aumentar o espaço de endereçamento do processo com a adição de páginas de memória virtual.

área	de	até
reservado	0x0000.0000	0x003f.ffff
texto	0x0040.0000	0x0fff.ffff
dados estáticos	0x1000.0000	0xiiii.iiii
dados dinâmicos	0xjjjj.jjjj	0xkkkk.kkkk
pilha	0xnnnn.nnnn	0x7fff.fffc

`iiii`, `jjjj`, `kkkk`, `nnnn` são números quaisquer

Figura 10.11: Mapa do espaço de endereçamento do MIPS.

10.4.4 Variáveis globais não-inicializadas

Programas frequentemente contém variáveis inicializadas com zero. Para evitar que estas variáveis ocupem espaço no arquivo que contém o executável, elas são alocadas na seção chamada de *Block Started by Symbol* (**`.bss`** ou **`.comm`**). Na inicialização, uma função preenche toda a BSS com zeros. A diretiva do montador para alocar variáveis não-inicializadas na BSS chama-se **`.comm`** que é a abreviação para *common*, um nome herdado da linguagem Fortran.

Como um exemplo da alocação de variáveis nas seções **`.data`** e na área global, o Programa 10.7 multiplica duas matrizes e atribui o resultado a uma terceira, sendo que `A[]` e `B[]` são parcialmente inicializadas e a matriz `C[]` não é inicializada. O código compilado com `mips-gcc -S -O2 aloca.c -o a02.s` produz a saída mostrada no Programa 10.8. O corpo da função `main()` não é mostrado para poupar espaço.

Programa 10.7: Programa C que multiplica duas matrizes.

```
#define SZ 8

int A[SZ][SZ] = {-1,-1,-1,-1,-1,-1,-1,-1}; // inicializa 1a linha
int B[SZ][SZ] = {2,2,2,2,2,2,2,2};         // inicializa 1a linha
int C[SZ][SZ];                             // inicializa com zero
int i,j,k;                          // variáveis globais pequenas

int main (int argc, char** argv) {
    int num=0;
    for (i = 0; i < SZ; i++)
      for (j = 0; j < SZ; j++)
        for (k = 0; k < SZ; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    num=i+j+k;
    return(num);
}
```

Entre as linhas 41 e 49 é alocada e inicializada a matriz `A[]` (oito palavras mais 224 bytes). Entre as linhas 26 e 35 é alocada a matriz `B[]` (oito palavras mais 224 bytes). Na linha 19 é alocada a matriz `C[]`, com 256 bytes e alocada na BSS (`.comm`). Nas linhas 15 a 17 são alocadas as variáveis globais `i`, `j`, `k`, com tamanho de 4 bytes.

Todas as variáveis são alocadas em endereço múltiplo de 4, seja com os `.align` nas linhas 23 e 28 – o alinhamento é a potência de dois do argumento do `.align` e $2^2 = 4$ – seja com o terceiro argumento dos `.comm`, que é o alinhamento.

Espaço em branco proposital.

Programa 10.8: Início da versão *assembly* da multiplicação de matrizes.

```
1           .file   1 "aloca.c"
2           .section .text.startup,"ax",@progbits
3           .align  2
4           .globl  main
5           .ent    main
6           .type   main, @function
7   main:
8           lui     $10,%hi(A)
9           lui     $12,%hi(C)
10          ...
11          jr      $31
12          li      $2,24  # 0x18
13          .end    main
14
15          .comm   k,4,4
16          .comm   j,4,4
17          .comm   i,4,4
18
19          .comm   C,256,4
20
21          .globl  B
22          .data
23          .align  2
24          .type   B, @object
25          .size   B, 256
26  B:
27          .word   2
28          .word   2
29          .word   2
30          .word   2
31          .word   2
32          .word   2
33          .word   2
34          .word   2
35          .space  224
36
37          .globl  A
38          .align  2
39          .type   A, @object
40          .size   A, 256
41  A:
42          .word   -1
43          .word   -1
44          .word   -1
45          .word   -1
46          .word   -1
47          .word   -1
48          .word   -1
49          .word   -1
50          .space  224
51
52          .ident  "GCC: (GNU) 5.1.0"
```

Espaço em branco proposital.

10.5 Tabelas de símbolos para ligação

Quando o ligador lê seus arquivos de entrada, a tabela de símbolos é preenchida. A ligação consiste da *resolução dos símbolos*, quando lhes são atribuídos os endereços para execução. Isso feito, o ligador emite o arquivo de saída, que é o executável.

Ao final da primeira passagem sobre os arquivos de entrada, a tabela de símbolos contém simbolos ditos *resolvidos* porque existe um endereço definido associado ao símbolo, e a tabela contém símbolos por resolver porque não há (ainda) um endereço associado ao símbolo. Um *endereço definido* corresponde a um endereço em memória, atribuído pelo montador, ou que será atribuído durante a segunda fase do processo de ligação.

Uma vez que a tabela de símbolos esteja populada com todos os símbolos dos arquivos de entrada, o ligador percorre novamente a tabela, e para cada símbolo indefinido que encontre, seu valor é buscado na tabela e então o endereço é atribuído ao símbolo. Esse processo é chamado de *resolução dos símbolos*.

Se, ao final desta fase houver algum símbolo ainda indefinido, o ligador informa do erro e termina sem emitir o arquivo executável.

Um ligador deve tratar outros tipos de símbolos, além de variáveis e funções. Os tipos de símbolos incluem os listados a seguir.

- *Símbolos globais definidos* e referenciados dentro de um módulo/arquivo;
- símbolos globais referenciados mas não-definidos, ou *símbolos externos*;
- *nomes de segmentos*, que são considerados símbolos globais, e são sempre definidos como o início de um segmento – a relocação se dá com relação a estes símbolos;
- símbolos usados por depuradores – estes não são usados na ligação mas devem ser repassados para o arquivo objeto ou executável; e
- números de linha de código fonte, também repassados para depuradores.

Um ligador emprega mais de uma tabela de símbolos para efetuar a ligação de vários arquivos objeto. São necessárias (i) uma *tabela de módulos* que mantém informação sobre os arquivos de entrada e módulos de bibliotecas; (ii) uma *tabela de símbolos* propriamente dita que é usada para a resolução de símbolos globais definidos e referenciados nos arquivos de entrada; e (iii) uma *tabela de depuração* com os símbolos de depuração em cada módulo. Esta última pode ser a concatenação das tabelas de depuração dos módulos individuais. Estas três tabelas empregam uma única, ou três distintas *tabelas de cadeias*, ou *tabelas de strings*, que são descritas na Seção 10.1.

O ligador mantém uma tabela de símbolos como um vetor de elementos, tal como aquele mostrado na Figura 10.13, e usa uma função de *hashing* para localizar um elemento da tabela. Geralmente, esta tabela emprega um vetor de apontadores que são indexados com a função de *hashing*, mais listas de colisão. Tais estruturas são definidas na Seção 10.2.

10.5.1 Tabelas de módulos

O ligador mantém uma tabela com informações sobre todos os arquivos de entrada e módulos extraídos de bibliotecas. Esta *tabela de módulos* ajuda na localização dos símbolos nos vários segmentos agrupados a partir de fontes distintas. Para cada arquivo objeto, o cabeçalho do arquivo é copiado para esta tabela, ou cria-se um apontador para o cabeçalho. Esta tabela contém apontadores para a cópia em memória da tabela de símbolos referente ao arquivo/módulo, e para as tabelas de *strings* e de relocação. As posições dos segmentos de texto, dados e BSS deste módulo/arquivo no arquivo de saída são registradas também.

Durante uma primeira passada sobre os arquivos por ligar – o *primeiro passo da ligação*, o ligador lê para memória a tabela de símbolos de cada arquivo, bem como a tabela de *strings*. Para acelerar o processamento, cada referência a um elemento na tabela de *strings* é transformada num apontador para o endereço em memória onde a *string* está armazenada.

No *segundo passo da ligação*, o ligador percorre as tabelas criadas no primeiro passo e resolve todos os símbolos, ao atribuir um endereço à cada referência a um símbolo.

10.5.2 Tabela de símbolos global

A *tabela de símbolos global* contém um elemento para cada símbolo que é referenciado ou definido em qualquer dos arquivos de entrada. A cada arquivo de entrada que é processado, o ligador adiciona todos os seus símbolos à tabela global, mantendo uma lista de locais nos quais o símbolo é definido ou referenciado. Na medida em que os símbolos são adicionados à tabela global, cada elemento na tabela de símbolos local ao módulo é ligada ao correspondente elemento na tabela global.

Para cada módulo é mantida uma tabela local de símbolos e é criado um vetor de apontadores para os elementos correspondentes na tabela global de símbolos. A lista de apontadores permite associar referências num certo módulo com os símbolos correspondentes na tabela global de símbolos, e é esta associação o que permite resolver os símbolos externos.

O trecho de pseudo-código na Figura 10.12 mostra uma versão resumida dos módulos dos Programas 10.2, 10.3 e 10.4. A função `printf()` é ignorada nesta seção. Dois módulos definem funções, e o outro importa as funções e define variáveis. Estes módulos são usados no que segue para exemplificar a construção da tabela de símbolos global.

```
// módulo 1
extern f();
extern g();
extern int r;
int k,l,m;
```

```
// módulo 2
extern g();
extern int y;
int r;
int f(){..y..};
```

```
// módulo 3
extern int k;
int y;
int g(){..k..};
```

Figura 10.12: Versão resumida dos três módulos.

A função `g()` é referenciada nos módulos 1 e 2 e é definida no módulo 3 (`g:`); a função `f()` é referenciada no módulo 1 e é definida no módulo 2 (`f:`). A variável `k` é definida no módulo 1 e referenciada no módulo 3.

A Figura 10.13 mostra a tabela global de símbolos após a leitura do arquivo do primeiro módulo (*módulo 1*). À esquerda é mostrado o arquivo com sua tabela local e o vetor de apontadores para as referências. Na direita é mostrada a tabela global de símbolos já com os símbolos do *módulo 1*, mais a tabela de símbolos, dividida em texto e dados, para aquele módulo. Apontadores são representados por letras do alfabeto grego.

A Figura 10.14 mostra a tabela completa, depois que os três módulos foram lidos e seus símbolos processados. À esquerda da figura é mostrado o *módulo 1*, com o vetor de apontadores para a tabela global. Na direita é mostrada a tabela global completa e as tabelas de símbolos dos três módulos. Para simplificar o diagrama são mostrados apenas os símbolos do *módulo 1*.

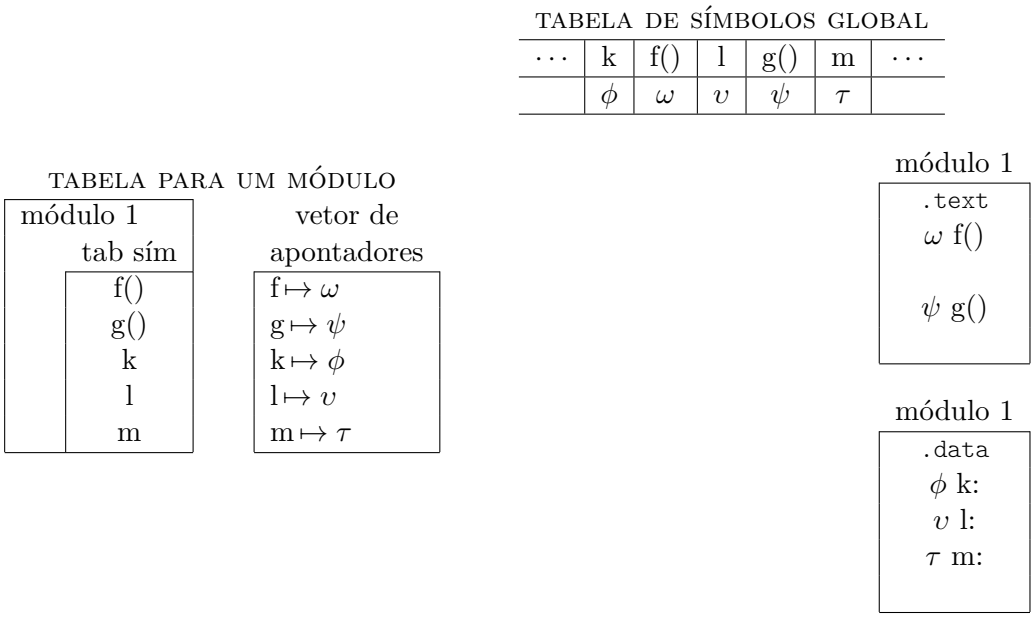


Figura 10.13: Construção da tabela de símbolos — inclusão do primeiro módulo.

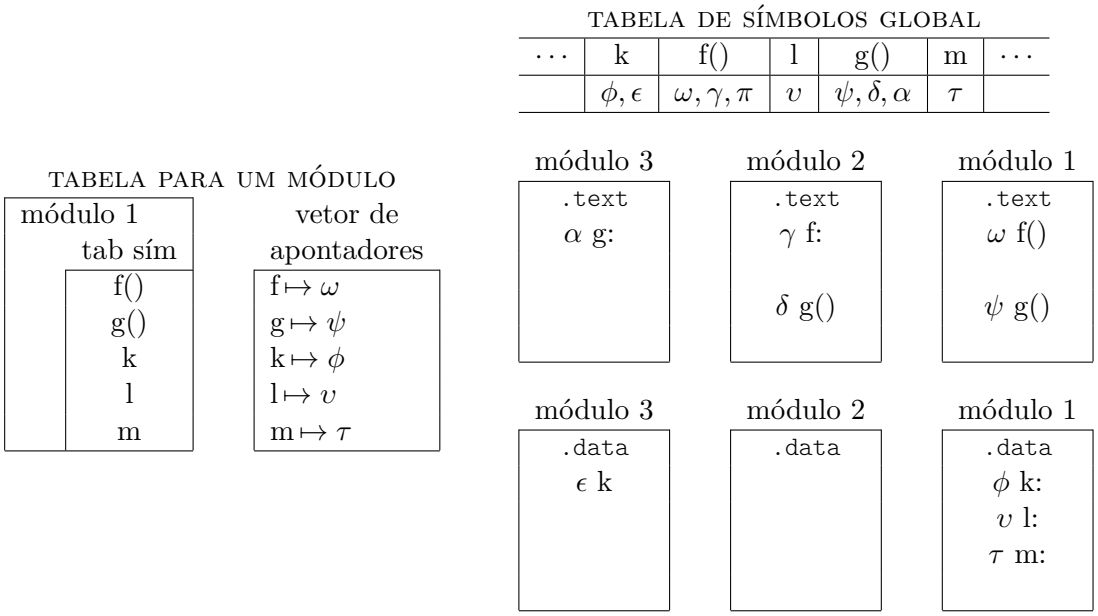


Figura 10.14: Construção da tabela de símbolos — tabela completa.

10.5.3 Resolução de símbolos

Durante o segundo passo, o ligador resolve as referências na medida em que o arquivo de saída é criado. Por exemplo, numa referência absoluta a uma variável do tipo inteiro, o endereço do símbolo é inserido no local da referência: se o símbolo (nome de variável) `matriz` é alocado no endereço `data+2048 = 0x1000.2048`, cada referência ao símbolo `matriz` é substituída pelo endereço `0x1000.2048`.

10.6 Exercícios

Ex. 72 As cadeias mostradas abaixo devem ser armazenadas numa tabela de cadeias. Mostre o vetor de símbolos e a tabela depois que todas as cadeias tenham sido inseridas. A tabela inicia no endereço `1000`.

```
The quick brown fox\n\0
\tjumped over\n\0
\tthe lazy\n\0
\tsleepy dog.\n\0
```

Ex. 73 Insira os símbolos ao lado numa tabela de espalhamento com 16 elementos e com a função $h(\,)$ dada pela soma dos caracteres do símbolo. Armazena o valor da função $I = h(S)$ junto com o símbolo S.

```
a   b   a1   a2   fun   gun   pun   cat   rat
```

Ex. 74 Escreva um conjunto de funções em C que permita criar uma tabela de *strings* como a mostrada na Seção 10.1. Seu código deve ser genérico e permitir a construção de tabelas de cadeias de tamanhos não-triviais.

Ex. 75 Escreva um conjunto de funções em C que permita recuperar qualquer dos componentes da tabela de cadeias do Ex. 74: dado o índice no vetor, retorna a cadeia.

Ex. 76 Escreva um conjunto de funções em C que permita criar uma tabela de símbolos como a mostrada na Seção 10.5. Seu código deve ser genérico e permitir a construção de tabelas de símbolos de tamanhos não-triviais.

Capítulo 11

Formatos de arquivos objeto

Arquivos objeto são criados por montadores e ligadores e contém o código binário e dados resultantes da compilação de um arquivo fonte, além de informações necessárias para a ligação e a carga. Um arquivo objeto contém ao menos cinco tipos de informação:

cabeçalho informação sobre o arquivo tais como nome do arquivo fonte que lhe deu origem, tamanho do código e data de criação;

código objeto instruções e dados gerados pelo montador ou compilador;

informação de relocação lista de lugares no código que devem ser ajustados quando o ligador alocar os segmentos e ajustar os endereços no código objeto;

símbolos lista com símbolos que são definidos neste módulo (arquivo fonte) e símbolos que devem ser importados de outros módulos, ou definidos pelo ligador; e

informação para depuração informações sobre o objeto que não são usadas usadas pelo ligador mas que são úteis para o depurador, tais como leiaute de estruturas de dados, mapeamento entre número de linhas de código fonte e instruções, e definições de símbolos locais.

O conteúdo de um arquivo objeto depende de seu uso. Se o arquivo será tratado por um ligador, então este deve conter informação para relocação e tabela de símbolos; se o arquivo será carregado em memória para executar, então necessita de pouco mais que o código objeto; se o arquivo será manipulado pelo ligador dinâmico/carregador, então deve ser carregado em memória como uma biblioteca, para ser posteriormente ligado a um programa. Combinações destes três formatos podem ser úteis.

11.1 Arquivos `.COM`

O *formato .COM* é usado no *Disk Operating System* (DOS) e não contém nenhuma informação além do código executável. Quando o DOS dispara a execução de um arquivo `.COM`, este é carregado a partir do endereço 0x100 de um segmento livre. O carregador copia os parâmetros de execução e argumentos da linha de comando para os endereços `0x00` a `0xFF` – que é o *Program Segment Prefix* (PSP), faz todos os registradores de segmento apontarem para o PSP, faz o apontador de pilha apontar para o final do segmento, e salta para o endereço `0x100`.

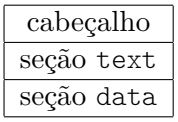
Este formato depende fortemente da arquitetura de segmentação do x86. Se o programa cabe num segmento de 64Kbytes, o ligador não tem nada a fazer. Para programas que não

cabem num único segmento, a responsabilidade sobre as referências aos outros segmentos é empurrada para o programador.

11.2 Arquivos `a.out`

Computadores que possuem relocação em *hardware* normalmente criam um processo novo com espaço de endereçamento inicialmente vazio. O executável é criado para ser carregado a partir de um endereço fixo e neste caso, não é necessário relocar o programa ao carregá-lo.

Em sua forma mais simples, um arquivo no formato a.out consiste de um cabeçalho seguido pelo código executável (`text`), e os dados inicializados (`data`), possivelmente seguido de outras seções.



O cabeçalho de um arquivo `a.out` tem os componentes mostrados no Programa 11.1 – veja a definição completa em `/usr/include/a.out.h`.

Programa 11.1: Cabeçalho do formato a.out.

```
struct exec {             // All lengths given in bytes
   unsigned long a_info;  // Use macros *MAGIC for access
   unsigned int a_text;   // Length of text -- RO
   unsigned int a_data;   // Length of data -- RW
   unsigned int a_bss;    // Length uninitialized data area -- RW
   unsigned int a_syms;   // Length symbol table data in file
   unsigned int a_entry;  // Start address
   unsigned int a_trsize; // Length of relocation info for text
   unsigned int a_drsize; // Length of relocation info for data
};
```

O campo `a_info` contém o número mágico (*magic number*) que identifica o formato do arquivo, que é um dentre os listados no Programa 11.2. O campo `a_bss` contém o tamanho da área para dados não-inicializados, que serão inicializados com zero pelo carregador. Os campos `a_syms`, `a_trsize`, e `a_drsize` contém os tamanhos das tabelas de símbolos que seguem o segmento de dados no arquivo. Em arquivos executáveis estes campos são inicializados com zero.

Programa 11.2: Números mágicos do formato a.out.

```
#define OMAGIC 0407  // Object file or impure executable
#define NMAGIC 0410  // Pure executable
#define ZMAGIC 0413  // Demand-paged executable
#define QMAGIC 0314  // Demand-paged executable w header in .text
                     // The first page is unmapped to help trap
                     //    NULL pointer references
#define CMAGIC 0421  // Core file
```

A carga de um arquivo `a.out` no formato `NMAGIC`, que era empregado em sistemas sem memória virtual paginada – nas décadas de 1970-80 – com segmentos de texto e dados é simples e consiste dos seguintes passos:

1. criar um novo espaço de endereçamento para o processo;
2. ler o cabeçalho para descobrir os tamanhos dos segmentos;
3. alocar um segmento de texto com permissões EX+RO, e copiar o segmento de texto do arquivo para a memória;

4. alocar um segmento de dados com permissões RW e tamanho suficiente para comportar segmento de dados e BSS juntos, copiar dados inicializados (`.data`) e preencher com zeros o BSS (`.bss`);
5. alocar um segmento de pilha com permissões RW, copiar os argumentos da linha de comando e parâmetros de execução para a pilha; e
6. inicializar os registradores do processador e saltar para o endereço inicial.

Em sistemas com paginação, ao invés de copiar todo o arquivo para memória, o arquivo `a.out` é mapeado no espaço de endereçamento do processo, e somente é necessário alocar aquelas páginas nas quais o processo escreve. Como o arquivo é mapeado em memória, somente aquelas partes que serão efetivamente usadas necessitam ser copiadas do disco, acelerando o processo de carga.

Arquivos de formato `QMAGIC` são usados em sistemas com paginação, e o início dos segmentos é alocado numa fronteira de página. O cabeçalho e o segmento de texto são alocados na página virtual de número 1 – a primeira instrução do programa é alocada no nono inteiro daquela página. A página virtual de número 0 não é alocada nem mapeada e isso ajuda na 'captura' de referências através de apontadores nulos. Note que o endereço inicial da carga não é muito importante, desde que o carregador e o SO estejam de acordo quanto a qual seja este endereço.

O formato `a.out` não é adequado para programas em C++ e não facilita a utilização de bibliotecas dinâmicas, por isso o formato `a.out` foi substituído pelo formato *ELF* em meados da década de 1980.

11.3 Arquivos ELF

Os formatos `.COM` (Sec. 11.1) e `a.out` (Sec. 11.2) resolvem parte do problema que é 'transportar' informações entre os estágios finais do processo de compilação, embora estes formatos sejam pouco flexíveis. Por causa desta deficiência, o formato *ELF* (*Executable and Linking Format*) foi desenvolvido. O formato é extremamente flexível e configurável; o outro lado da moeda é que as estruturas de dados necessárias são mais complexas e sofisticadas que os formatos simples.

Vejamos três exemplos da flexibilidade: (i) os nomes de seções, tais como `.text` e `.data`, não são fixos – as cadeias de caracteres com os nomes das seções são armazenados num vetor de cadeias, o que significa que os nomes podem ter qualquer tamanho; (ii) a tabela que descreve as seções presentes no arquivo é uma lista, terminada por um elemento nulo – esta tabela se assemelha a uma cadeia de caracteres terminada por `'\0'`, e pode conter tantas seções quantas sejam necessárias e não tem tamanho fixo; e (iii) a localização do conteúdo das várias seções no arquivo é especificada por um deslocamento a partir do início do arquivo, mais o seu tamanho – as seções podem ser armazenados no arquivo numa ordem arbitrária. Vamos pois aos detalhes.

O formato *ELF* é geralmente empregado em uma de três maneiras: (i) para conter código relocável que será processado pelo ligador; (ii) para conter código executável que será carregado e executado; e (iii) para conter um objeto compartilhado, que é uma biblioteca compartilhada para ligação dinâmica.

Arquivos ELF relocáveis são criados por compiladores e montadores e devem ser processados pelo ligador antes de executar.

Arquivos ELF executáveis estão com todas as relocações concluídas e todos os símbolos resolvidos, exceto símbolos de bibliotecas, que devem ser resolvidos em tempo de execução.

Objetos compartilhados ELF são bibliotecas compartilhadas que contém informação para a resolução de símbolos pelo ligador, além de código executável. Por "bibliotecas compartilhadas" entenda-se bibliotecas que são compartilhadas dinamicamente pelos processos; as funções da biblioteca são incorporadas ao processo em tempo de execução, e não em tempo de ligação – mais detalhes no Capítulo 15.

Arquivos ELF podem ser interpretados de duas formas. Compiladores, montadores e ligadores tratam um arquivo ELF como um conjunto de seções – que é descrito por uma *tabela de cabeçalhos de seções* (*Section Header Table*), enquanto que o carregador trata um arquivo ELF como um conjunto de segmentos – descrito por uma *tabela de cabeçalhos de programa* (*Program Header Table*). Um segmento normalmente contém várias seções. Objetos compartilhados possuem as duas tabelas. As seções são processadas pelo ligador, enquanto que segmentos são mapeados em memória pelo carregador. O diagrama na Figura 11.1 mostra a visão de ligação à esquerda, e a visão de execução/carga à direita.

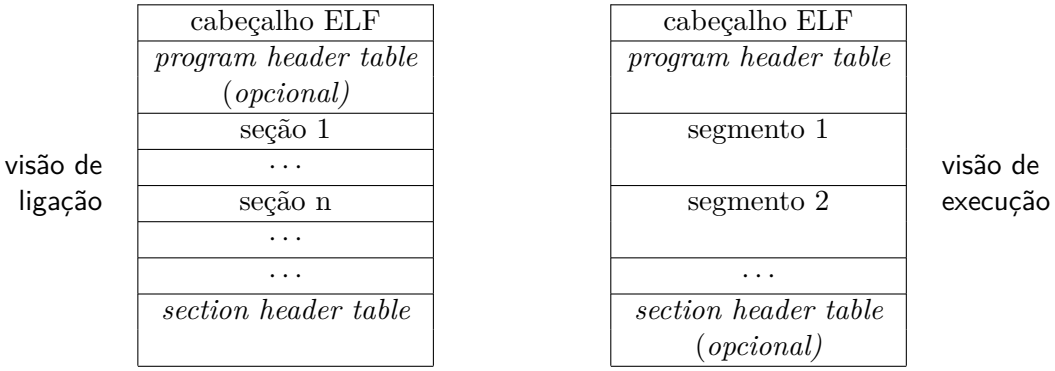


Figura 11.1: Dupla personalidade do formato ELF.

11.3.1 Cabeçalho ELF

A estrutura de um cabeçalho ELF é mostrada no Programa 11.3. Os primeiros 4 bytes contêm o número mágico que identifica o arquivo ELF; os próximos 2 bytes descrevem o formato do restante do arquivo: (i) se a arquitetura do processador é de 32 ou 64 bits; (ii) se o arquivo está codificado como *little-* ou *big-endian*. O campo `e_type` indica o tipo de arquivo, e o campo `e_machine` indica para qual arquitetura do processador o código foi gerado. Para mais detalhes diga `man elf`.

Espaço em branco proposital.

Programa 11.3: Cabeçalho do formato ELF.

```
typedef struct {
   unsigned char magic[4]={'0x7f','E','L','F'}; // numero magico
   unsigned char class;       // tamanho: 1= 32 bits, 2= 64 bits
   unsigned char byteorder;   // 1= little-endian, 2= big-endian
   unsigned char hversion;    // versão do cabeçalho, sempre 1
   unsigned char pad[9];      // completa 16 bytes
   uint16_t      e_type;      // 1= relocável, 2= executável,
                              // 3= objeto compartilhado, 4= core
   uint16_t      e_machine;   // 2= sparc, 3= x86, 8= mips, ...
   uint32_t      e_version;   // versão do arquivo, sempre 1
   Elf32_Addr    e_entry;     // primeira instrução, se executável
   Elf32_Off     e_phoff;     // posição do program header, ou 0
   Elf32_Off     e_shoff;     // posição do section header, ou 0
   uint32_t      e_flags;     // específico ao processador, ou 0
   uint16_t      e_ehsize;    // tamanho deste cabeçalho
   uint16_t      e_phentsize; // tamanho de elmto do progr hdr
   uint16_t      e_phnum;     // núm de elmtos no prgr hdr, ou 0
   uint16_t      e_shentsize; // tamanho do elmto no sectn hdr
   uint16_t      e_shnum;     // núm de elmtos no sectn hdr, ou 0
   uint16_t      e_shstrndx;  // núm da seção com strings de seção
} ElfN_Ehdr;
```

O campo `e_phoff` indica a posição no arquivo em que inicia a *program header table*; o campo `e_phentsize` define o tamanho de um elemento daquela tabela, e o campo `e_phnum` define o seu número de elementos – o número de segmentos de um arquivo para carga ou compartilhamento não é limitado *a priori* pelo formato.

O campo `e_shoff` indica a posição no arquivo em que inicia a *section header table*; o campo `e_shentsize` define o tamanho de um elemento daquela tabela, e o campo `e_shnum` define o seu número de elementos – o número de seções de um arquivo para ligação ou compartilhamento não é limitado pelo formato ELF. O campo `e_shstrndx` indica o número da seção que contém as *strings* com os nomes das seções.

11.3.2 Arquivos para relocação — tabela de cabeçalhos de seções

Arquivos relocáveis contém várias seções, conforme mostrado na Figura 11.2, e descrito pela tabela de seções do Programa 11.4. Cada seção contém um único tipo de informação (código, símbolos), e cada símbolo definido no código fonte é definido com relação a uma seção. A seção zero é uma seção nula e seu descritor é preenchido com zeros.

Arquivos ELF podem conter uma seção que é uma tabela de cadeias, ou *strings*, cada uma delas é o nome textual de uma seção do arquivo. O campo `sh_name` é o índice na tabela de cadeias para o elemento que é o nome daquela seção. Isso permite que as seções tenham nomes arbitrariamente longos.

Espaço em branco proposital.

cabeçalho ELF	*não é considerado uma seção*
tabela de cabeçalhos de segmentos	*não é considerada uma seção*
.text	
.data	
.rodata	
.bss	
.sym	
.rel.text	
.rel.data	
.rel.rodata	
.strtab	
tabela de cabeçalhos de seções	*não é considerada uma seção*

Figura 11.2: Arquivo ELF para relocação.

Programa 11.4: Tabela de Cabeçalhos de Seções.

```
typedef struct {
   uint32_t   sh_name;      // nome, índice na tabela de strings
   uint32_t   sh_type;      // tipo da seção
   uint32_t   sh_flags;     // bits para WR, EX, se aloca memória
   Elf32_Addr sh_addr;      // endereço em memória ou 0
   Elf32_Off  sh_offset;    // posição no arquivo do inicio da seção
   uint32_t   sh_size;      // tamanho, em bytes
   uint32_t   sh_link;      // número de seção relacionada, ou 0
   uint32_t   sh_info;      // mais informação ref a esta seção
   uint32_t   sh_addralign; // tipo de alinhamento (2^n)
   uint32_t   sh_entsize;   // se seção eh vetor, tamanho dos elmtos
} Elf32_Shdr;
```

Os tipos de seção, campo `sh_type`, incluem:

`SHT_PROGBITS` seção contém informação definida pelo programa (código, dados, informação para o depurador);

`SHT_NOBITS` similar a `PROGBITS` nas não ocupa espaço no arquivo, área reservada para BSS que é alocada e inicializada somente em tempo de carga;

`SHT_SYMTAB` seção contém tabela de símbolos;

`SHT_STRTAB` seção contém tabela de cadeias;

`SHT_RELA` seção contém informação para relocação (similar ao a.out);

`SHT_DYNAMIC` seção contém informação para ligação dinâmica;

`SHT_HASH` seção contém tabela *hash* para ligação dinâmica.

Um arquivo relocável contém várias seções; algumas são usadas pelo ligador, enquanto que outras são descartadas ou incluídas na saída sem nenhum processamento. O atributo `ALLOC` significa que o conteúdo da seção ocupa espaço – o espaço deve ser alocado em memória, mas não no arquivo de saída. Os atributos `EX` e `WR` indicam que o conteúdo da seção é respectivamente executável ou alterável. O atributo `RD` indica que a seção pode ser lida. Algumas seções usuais são:

`.text` contém código, de tipo `SHT_PROGBITS` e atributos `EX+ALLOC`;

`.data` contém dados inicializados, de tipo `SHT_PROGBITS` e atributos `WR+ALLOC`;

`.rodata` contém dados inicializados só de leitura, de tipo `SHT_PROGBITS` e atributos `ALLOC`;

`.bss` contém dados não-inicializados e que não ocupam espaço no arquivo objeto mas que devem ser alocados, de tipo `SHT_NOBITS`, atributos `WR+ALLOC`;

`.rel.text .rel.data .rel.rodata` informação de relocação para as seções correspondentes, tipo `SHT_RELA`;

`.symtab .strtab` contém tabela de símbolos e tabela de cadeias, tipos `SHT_SYMTAB` e `SHT_STRTAB`;

`.dynsym .dynstr` contém tabela de símbolos e tabela de cadeias para ligação dinâmica, tipos `SHT_DYNSYM` e `SHT_STRTAB`, com atributo `ALLOC` porque devem ser carregadas em memória na execução.

`.got .plt` *global offset table* e *procedure linkage table*, também usadas na ligação dinâmica, de tipo `SHT_PROGBITS`.

Tabela de símbolos A tabela de símbolos é um vetor de elementos do tipo `Elf32_Sym`, que é mostrado no Programa 11.5. O nome textual do símbolo é mantido na tabela de cadeias. Cada símbolo é definido com relação a uma seção (nome de função associado à `.text`, nome de variável associado à `.data`, etc) e o campo `st_shndx` faz a associação entre o símbolo e o elemento correspondente da tabela de seções. Para símbolos na `.bss`, o campo `st_size` determina o tamanho da área a ser reservada.

Programa 11.5: Tabela de Símbolos.

```
typedef struct {
   uint32_t      st_name;  // índice do símbolo na tabela de cadeias
   Elf32_Addr    st_value; // valor do símbolo
   uint32_t      st_size;  // tamanho do símbolo, ou 0
   unsigned char st_info;  // tipo e atributo de ligação
   unsigned char st_other; // contém 0 (uso não definido)
   uint16_t      st_shndx; // contém o índice da seção associada a
                           //    esta tabela
} Elf32_Sym;
```

Os tipos dos símbolos (`st_info`), e seus atributos, incluem:

`STT_NOTYPE` indefinido;

`STT_OBJECT` símbolo associado a um dado;

`STT_FUNC` símbolo associado a uma função ou outro elemento de código;

`STT_SECTION` símbolo associado a uma seção – normalmente usado para relocação, deve ser o símbolo associado ao primeiro endereço da seção;

`STB_LOCAL` atributo de símbolo que é local e invisível fora do arquivo em que é definido;

`STB_GLOBAL` atributo de símbolo que é visível a todos os arquivos objeto sendo ligados;

`STB_WEAK` atributo de baixa precedência – a versão 'forte' do símbolo sobrescreve esta definição (usado em C++);

11.3.3 Arquivos para execução — tabela de cabeçalhos de segmentos

O formato executável do ELF é similar ao formato para relocação, mas seu conteúdo é organizado de forma a simplificar a carga em memória para execução. A *tabela de cabeçalhos de segmentos* (*Program Header Table*) segue o cabeçalho do arquivo, e define o mapeamento dos segmentos. A tabela de segmentos é um vetor de descritores de segmentos, com tipo `Elf32_Phdr`, mostrado no Programa 11.6.

Programa 11.6: Tabela de Cabeçalhos de Segmentos.

```
typedef struct {
   uint32_t   p_type;    // tipo do segmento: código, dados
   Elf32_Off  p_offset;  // início do segmento no arquivo
   Elf32_Addr p_vaddr;   // endereço em memória virtual do segmento
   Elf32_Addr p_paddr;   // 0, ou endereço físico do segmento
   uint32_t   p_filesz;  // tamanho no arquivo da imagem do segmento
   uint32_t   p_memsz;   // tamanho da imagem do segmento em memória
   uint32_t   p_flags;   // EX e/ou WR e/ou RD
   uint32_t   p_align;   // alinhamento = 0 ou tamanho da página
} Elf32_Phdr;
```

Normalmente são poucos os segmentos, com código, dados (`.data` e `.bss` combinados), e informação para ligação dinâmica, se for o caso, o que simplifica e acelera a carga em memória. Os segmentos no arquivo, e na tabela de segmentos, são ordenados pelo endereço virtual de carga (`p_vaddr`). Os tipos de segmento (`p_type`) incluem os listados abaixo.

`PT_LOAD` define um segmento a ser carregado no endereço contido em `p_filesz` ou `p_memsz`; se (`p_memsz > p_filesz`), a área além de `p_filesz` deve ser preenchida com zeros.

`PT_DYNAMIC` elemento especifica informação para ligação dinâmica.

`PT_INTERP` elemento especifica o caminho completo do programa que deve ser usado para interpretar o arquivo, de forma similar ao `#!/bin/bash` em *scripts*.

`PT_PHDR` se presente na tabela, deve preceder todos os segmentos a carregar, e descreve a localização e o tamanho da tabela de segmentos, usado em arquivos para ligação dinâmica.

De forma similar ao formato `QMAGIC` do a.out, o cabeçalho ELF e a tabela de segmentos são copiados para a memória, junto ao segmento de código. Um segmento pode iniciar em qualquer posição no arquivo; sua imagem em memória deverá ser carregada "na mesma posição" em uma página – por "mesma posição" entenda-se a posição no arquivo *módulo* o tamanho da página.

A Figura 11.3 mostra um exemplo de arquivo executável. A coluna POS. ARQ mostra a posição da tabela ou segmento no arquivo – que é um deslocamento com relação ao primeiro byte – e a coluna ENDEREÇO mostra o endereço de carga. As páginas são de 4Kbytes (`0x1000`) e o endereço inicial de carga é `0x40.0000`.

SEGMENTO	POS. ARQ.	ENDEREÇO	ATRIBUTOS
cabeçalho ELF	0	0x40.0000	
tab de segmentos	0x40	0x40.0040	PT_PHDR
código, tam: 0x4500	0x100	0x40.0100	PT_LOAD, RD, EX
dados, tam arq: 0x2200	0x4600	0x40.5600	PT_LOAD, RD, WR
mem: 0x3500		0x40.7bff	
outras seções	0x6800		não são carregadas

Figura 11.3: Exemplo de arquivo ELF para execução.

O cabeçalho ELF e a tabela de segmentos são carregados junto com o segmento de texto, na mesma página no início do código. O segmento de dados é carregado logo acima do segmento de texto. Esta página do arquivo é mapeada com os atributos *read-only* na última página do código e *copy-on-write* na primeira página dos dados. O segmento de texto termina em `0x40.45ff` mas o segmento de dados é carregado na página seguinte, em `0x40.5600`. Uma parte da última página do segmento de texto, somente `0x600` bytes, são

de fato utilizados, enquanto que na primeira página do segmento de dados os primeiros 0x600 bytes são ignorados. A seção .bss é alocada junto à seção de dados, com tamanho de 0x1300 bytes. A última página de dados é mapeada diratamente do arquivo, mas assim que ela for inicializada com zeros, (.bss), a página é copiada para memória física por causa do *copy-on-write*.

11.3.4 Arquivos para ligação dinâmica

Arquivos para ligação dinâmica (bibliotecas) contêm tanto as seções de arquivos para relocação quanto os segmentos de arquivos para execução. O arquivo contém uma tabela de segmentos – que descreve os segmentos de código e dados da biblioteca, a informação para relocação na tabela de símbolos, mais a tabela de seções no final do arquivo. A Figura 11.4 mostra um exemplo de alocação destes últimos.

SEGMENTO	POS. ARQ.	ENDEREÇO	ATRIBUTOS
cabeçalho ELF	0	0x40.0000	
tab de segmentos	0x40	0x40.0040	PT_PHDR
código, tam: 0x4500	0x100	0x40.0100	PT_LOAD, RD, EX
dados, tam arq: 0x2200	0x4600	0x40.5600	PT_LOAD, RD, WR
mem: 0x3500		0x40.7bff	
tab símbolos (0x200)	0x6800	0x40.7c00	PT_DYNAMIC
tab de seções (0x100)	0x6a00	0x40.7e00	PT_DYNAMIC
	0x6aff	0x40.7eff	

Figura 11.4: Exemplo de arquivo ELF para ligação dinâmica

11.4 Exercícios

Ex. 77 Escolha um programa em C de tamanho que não seja trivial, que chamaremos de x.c, execute os seguintes comandos, e compare os resultados:

```
gcc -o x.o -c x.c ; file x.o
gcc x.c ; file a.out
file /usr/lib/libXXX.so.N.M
```

Na terceira linha, troque XXX por algum nome de biblioteca do seu sistema, e N.M pelo número da versão instalada.

Ex. 78 Leia os resultados de:

```
man elf ld nm objdump readelf
```

Ex. 79 Execute os comandos abaixo e observe os resultados:

```
gcc -o x.o -c x.c ; nm x.o
gcc x.c ; nm a.out
```

A saída da segunda linha contém nomes de símbolos contidos em biblioteca(s) do Linux ligada(s) ao executável.

Capítulo 12

Código Independente de Posição e Relocação

Para ligar arquivos objeto, o ligador faz a alocação dos arquivos no mapa de memória e então ajusta os endereços de todas as referências para as suas novas posições. Este capítulo discute a relocação de endereços durante a ligação, e a geração de código que executa corretamente em qualquer faixa de endereços em que seja carregado.

12.1 A ligação com código independente de posição

O código gerado para ser mantido numa biblioteca para ligação dinâmica deve ser compilado de tal forma que referências a dados ou a funções sejam independentes da posição em que a biblioteca venha a ser carregada – as razões para tal são apresentadas no Capítulo 15. Isso é necessário porque a biblioteca pode ser alocada em uma região diferente do espaço de endereçamento em cada um dos programas a que possa ser ligada. A Figura 12.1 mostra dois programas que usam as mesmas bibliotecas dinâmicas, mas estas são mapeadas em endereços distintos nos dois programas.

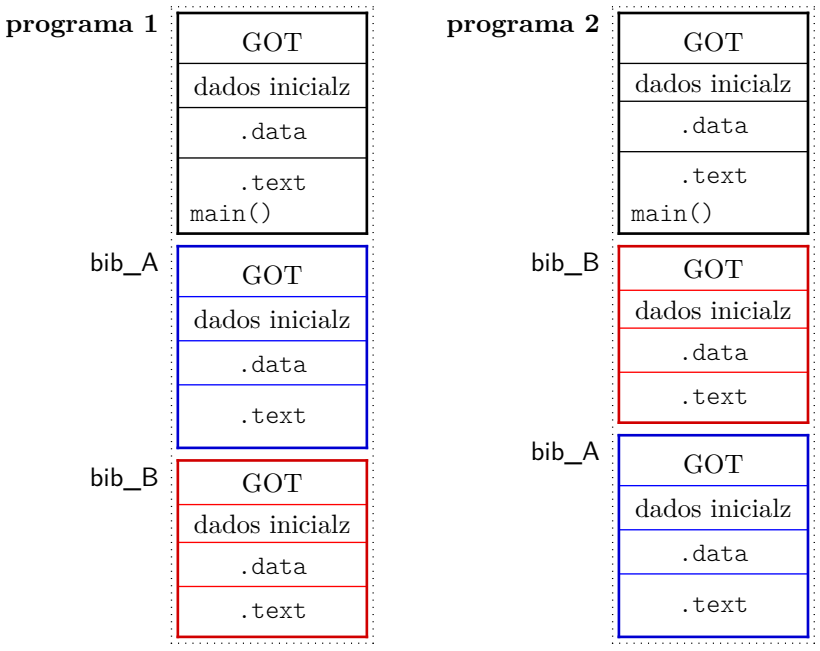


Figura 12.1: Bibliotecas dinâmicas alocadas em áreas distintas em dois programas.

Quando o compilador gera *código independente de posição* (*Position Independent Code*, PIC), tal código contém somente referências indiretas à variáveis ou à funções. Cada biblioteca possui uma *Global Offset Table* (GOT) e esta tabela contém um vetor de endereços, com um endereço para cada *símbolo externo* que é definido na biblioteca e que pode ser referenciado em outros objetos.

O preenchimento da GOT se dá em duas fases: na primeira, o arquivo ELF é criado com uma GOT 'vazia'; na segunda fase, durante a ligação dinâmica em tempo de carga, a tabela é completada com os endereços que estão indefinidos.

Quando o compilador gera o arquivo ELF, aquele cria a GOT e a preenche com zeros, e o ligador associa a cada símbolo externo indefinido um elemento da GOT. Na ligação dinâmica, o ligador resolve as referências externas depois que as bibliotecas são alocadas e mapeadas em memória, e então preenche a GOT com os endereços correspondentes aos símbolos – dados ou funções – que até então eram referências indefinidas. Isso feito, as referências aos símbolos externos ocorrem indiretamente através da GOT.

A Figura 12.1 mostra uma GOT em cada um dos três objetos que compõem programa_1 e programa_2. Há uma GOT em cada biblioteca além de uma GOT no arquivo objeto com o programa principal. Esta última mantém a tabela de apontadores para as referências indiretas a todos os símbolos que são definidos nas duas bibliotecas. O ligador dinâmico preenche a GOT do programa da mesma forma que o faz com as bibliotecas. A GOT da biblioteca é necessária para o acesso aos símbolos externos que são referenciados pela biblioteca. Por exemplo, se bib_A referencia uma função `f()` que é definida na bib_B, então a GOT de bib_A contém um elemento que apontará para o endereço absoluto de `f()` *depois* que a ligação dinâmica for concluída, conforme discutido no Capítulo 15.

12.2 Código independente de posição

Código independente de posição (IdP) permite carregar um arquivo objeto executável em endereços distintos, possivelmente num endereço diferente em cada processo criado com aquele objeto.

Código IdP é gerado de forma a que não necessite ser alterado se o endereço de carga mudar. Quando se usa código IdP, somente as páginas com dados de bibliotecas dinâmicas são privativas; as páginas com código podem ser compartilhadas porque o endereço de carga não interfere com a execução do(s) programa(s).

Gerar *código IdP* é relativamente simples: todos saltos e desvios devem ser relativos ao PC ou a um registrador que é inicializado durante a execução.

O acesso aos *dados IdP* não pode se dar com endereçamento direto porque então as referências não seriam independentes de posição. Para referências a dados é necessário um nível de indireção para que o conteúdo das variáveis seja acessado através de um apontador, que no caso do MIPS é mantido no registrador `gp` (*global pointer*). Para acessar dados de forma IdP é necessário criar uma tabela de endereços numa *página de dados* e inicializar registrador `gp` com o endereço desta tabela; os dados são referenciados com endereços relativos ao início da tabela.

A *Global Offset Table* deve ficar no início do segmento de dados. O código para o acesso à variáveis globais contém um deslocamento com relação ao início da tabela e na posição da tabela apontada pelo deslocamento encontra-se o endereço da variável. O registrador que aponta para o início da GOT é o registrador `gp` (`r28`).

Note que esta é uma questão de tradução de código C para *assembly*, e não necessariamente uma técnica particular de programação em C – o compilador deve ser invocado com as opções apropriadas para gerar código IdP. Com o `mips-gcc`, objetos devem ser compilados com as opções `-fPIC -mshared`. Leia com atenção o manual do GCC porque estas opções tem comportamentos que diferem para cada arquitetura.

12.2.1 Código com endereçamento absoluto

Veremos adiante exemplos de código que é independente de posição, e portanto pode ser carregado em qualquer posição do espaço de endereçamento. Antes disso, vejamos como se escreve código com *endereços absolutos*, que só pode ser carregado a partir de um endereço inicial que é fixado no próprio código.

Estes exemplos foram copiados, e ligeiramente adaptados, de [**?**], pg. 3.38 e seguintes. Nos exemplos mostrados, "`dst >> 16`" equivale a "`%hi(dst)`" e "`dst & 0xffff`" equivale a "`%lo(dst)`".

Load e store absolutos *Endereçamento absoluto* é a forma mais comum de uso destas instruções. O valor que é o resultado de uma instrução **`lw`** não pode ser usado imediatamente após aquela instrução – veja a Seção 4.3.2. É necessário que a instrução que usa o valor do **`lw`** esteja separada por uma instrução do **`lw`**, por causa do *load delay slot*, e nos exemplos o *slot* é preenchido com uma instrução útil ou com um **`nop`**.

```
C                   assembly
-------------------------------------------------------------------------
extern int src;     .globl src,dst,ptr;
extern int dst;
extern int *ptr;

ptr = &dst;         lui   t6, dst >>16              # endereço de dst
                    ori   t6, t6, dst & 0xffff
                    lui   t7, ptr >>16
                    sw    t6, ptr & 0xffff (t7)     # ptr <- &dst

*ptr = src;         lui   t6, src >>16
                    lw    t6, src & 0xffff (t6)     # t6 <- mem[&src]
                    lui   t7, ptr >>16              # load delay-slot
                    lw    t7, ptr & 0xffff (t7)     # *ptr <- mem[&src]
                    nop                             # load delay slot
                    sw    t6, 0 (t7)
-------------------------------------------------------------------------
```

Chamada de função absoluta direta A *chamada absoluta direta* é a forma comum de chamada de funções e pode ser usada para funções que são invocadas dentro de um mesmo módulo e que não são referenciadas externamente ao módulo. O **`nop`** após o **`jal`** é necessário para preencher o *branch delay-slot* – veja a Seção 4.3.2.

```
C                     assembly
-------------------------------------------------------------------------
extern void func();   jal func
func();               nop           # branch delay-slot
-------------------------------------------------------------------------
```

Chamada de função absoluta indireta Numa *chamada de função indireta* o endereço da função é atribuído a um apontador "`ptr=func`" e então o apontador é de-referenciado.

```
C                     assembly
-------------------------------------------------------------------------
extern void (*ptr)();
extern void func();

ptr = func;           lui   t6, func >>16          # t6 <- &(func())
                      ori   t6, t6, func & 0xffff
                      lui   t7, ptr >>16
                      sw    t6, ptr & 0xffff (t7)  # mem[ptr] <- &(func())
  ...                 ...
(*ptr)();             lui   t6, ptr >>16
                      lw    t6, ptr & 0xffff (t6)  # t6 <- mem[ptr] = func
                      nop                          # load delay-slot
                      jalr  ra, t6                 # salta para mem[t6]
                      nop                          # branch delay-slot
-------------------------------------------------------------------------
```

Switch absoluto O trecho de código abaixo implementa um *switch* com endereços absolutos. O código emprega uma tabela de *labels* para efetuar o salto para o endereço do início de cada cláusula. Note que não há uma cláusula para `j=1`, e que portanto o elemento da tabela que corresponderia a este caso é preenchido com o *label* do caso *default*. Para simplificar o código, o registrador `t7` contém o resultado da avaliação da condição (`t7` $\leftarrow$ `j`), as cláusulas começam de zero, e os rótulos das cláusulas são `.Lcase`N, `.Ldef` para a cláusula N e *default*, respectivamente.

```
C                     assembly
-------------------------------------------------------------------------
switch (j) {                sltiu at, t7, 4         # t7 = valor de seleção
case 0:                     beq   at, $zero, .Ldef  # j >= 4 -> default:
  ...                       sll   t7, t7, 2         # índice * 4
case 2:                     lui   t6, .Ltab >>16    # t6 <- %hi(.Ltab)
  ...                       ori   t6, t6, .Ltab & 0xffff
case 3:                     addiu t6, t6, t7        # t6 <- .Ltab[j]
  ...                       lw    t7, 0(t6)         # t7 <- cláusula
default:                    nop
  ...                       jr    t7                # salta para cláusula
}                           nop
                            ...
                     .Ltab: .word .Lcase0           # end. cláusula 0
                            .word .Ldef             # não há cláusula 1
                            .word .Lcase2           # end. cláusula 2
                            .word .Lcase3           # end. cláusula 3
                            .word .Ldef             # end. cláusula default
-------------------------------------------------------------------------
```

12.2.2 Código com endereçamento relativo

Os exemplos que seguem mostram como deve ser gerado o código que pode ser carregado em qualquer posição no espaço de endereçamento. Para efetuar as referências indiretas é usada a *Global Offset Table*, que é mantida num endereço pré-definido pelo ligador, e o registrador `gp` (*global pointer*) é carregado com o endereço da GOT. Cada elemento desta

tabela corresponde a um símbolo externo – variável ou função. Nos exemplos, a posição do elemento na tabela é mostrada como `VAR_got_off` para variáveis ou `FUN_got_off` para funções. Note que os `X_got_off` não são gerados automaticamente pelo montador – estes não são macros tais como `%hi()` e `%lo()`. Os deslocamentos com relação ao início da tabela são computados pelo ligador ao criar o arquivo objeto. Estes exemplos foram copiados de [**?**], pg. 3.38 e seguintes.

Load e Store independentes de posição Para implementar acessos à dados *independentes de posição*, a GOT contém apontadores para `src`, `dst` e `ptr`. As referências indiretas usam estes apontadores para acessar os conteúdos das variáveis. Note que `GOT[ptr_got_off]` contém o *endereço do apontador* `ptr`.

```
C                       assembly
-------------------------------------------------------------------------
extern int src;         .globl src, dst, ptr
extern int dst;
extern int *ptr;        la     gp, GOT               # gp aponta para a GOT
ptr = &dst;             lw     t7, dst_got_off(gp)   # posição na GOT de dst
                        lw     t6, ptr_got_off(gp)   # posição na GOT de ptr
                        nop                          # load delay-slot
                        sw     t7, 0(t6)             # ptr <- &dst
*ptr = src;             lw     t7, src_got_off(gp)   # posição na GOT de src
                        nop
                        lw     t7, 0(t7)             # t7 <- mem[src]
                        lw     t6, ptr_got_off(gp)   # posição na GOT de ptr
                        nop
                        lw     t6, 0(t6)             # t6 <- *ptr (= &dst)
                        nop
                        sw     t7, 0(t6)             # *ptr <- mem[src]
-------------------------------------------------------------------------
```

Desvios O endereçamento independente de posição é trivial nos desvios condicionais porque o modo de endereçamento do MIPS nos desvios é naturalmente relativo ao PC. O código pode ser carregado em qualquer endereço que as distâncias relativas entre a instrução de desvio e a instrução de destino não se alteram.

```
C                       assembly
-------------------------------------------------------------------------
label:                  .L32:
      ...                     ...
      goto label;             b   .L32              # beq, bne, ...
                              nop                   # branch delay slot
-------------------------------------------------------------------------
```

Chamada de função independente de posição A GOT contém o endereço absoluto de todas as funções externas que são independentes de posição. Por uma convenção empregada na ligação dinâmica, o endereço da função é armazenado no registrador `t9` *antes* da chamada da função – a razão para isso é discutida no Capítulo 15.

Por causa da ligação dinâmica, e conteúdo de `gp` deve ser recarregado a partir da pilha após um retorno de uma função externa. Isso é necessário porque a biblioteca em que a função é definida possui a sua própria GOT, que é separada da GOT do programa que invocou a função de biblioteca.

O código da função atualiza o valor do gp, fazendo com que ele aponte para a GOT da biblioteca. Após o retorno da função, o gp deve ser recarregado com o endereço original da GOT porque a função chamada pode ter alterado o valor do gp. Neste exemplo o gp fica armazenado na sexta posição do registro de ativação (`24(sp) = (6*4)+sp`).

```
C                       assembly
-------------------------------------------------------------------------
extern void (*ptr)();   .globl ptr, func
extern void func();
                        # invocação direta de função externa
func();                 lw     t9, func_got_off(gp) # t9 <- ender da função
                        nop
                        jalr   ra, t9               # destino em t9
                        nop                         # branch delay slot
                        lw     gp, 24(sp)           # re-carrega gp
                        nop                         # load delay slot

                        # invocação indireta de função externa
ptr = func;             lw     t7, func_got_off(gp) # posição na GOT de func
                        lw     t6, ptr_got_off(gp)  # posição na GOT de ptr
                        nop
                        sw     t7,0(t6)             # ptr <- func

(*ptr)();               lw     t7, ptr_got_off(gp)
                        sw     gp, 24(sp)           # salva gp na pilha
                        lw     t9, 0(t7)            # destino em t9
                        nop
                        jalr   ra, t9               # chamada indireta
                        nop
                        lw     gp, 24(sp)           # re-carrega gp da pilha
                        nop
-------------------------------------------------------------------------
```

Espaço em branco proposital.

Switch independente de posição Para implementar um *switch independente de posição*, a tabela de saltos para as cláusulas – a tabela com os endereços das cláusulas – é armazenada na seção `.rdata` (dados relocáveis) para que o ligador possa fazer os ajustes nos endereços da tabela depois que estes tenham sido alocados pelo ligador. Com código independente de posição, as instruções de seleção devem calcular o deslocamento na tabela de cláusulas para então obter o endereço da cláusula a partir da GOT.

```
C                      assembly
-------------------------------------------------------------------------
switch (j) {                  sltiu at, t7, 4   # t7 contém valor da seleção
case 0:                       bne   at, $zero, .Lnorm # j <= 4, trat. normal
 ...                          nop
case 2:               .Ldef:  li    t7, 4       # vai para default
 ...                  .Lnorm: sll   t7, t7, 2   # índice * 4
case 3:                       lw    at, .Ltab_got_off(gp) # end tab cláusulas
 ...                          nop
default:                      addu  at, at, t7  # ender. da cláusula .Ltab[]
 ...                          lw    t6, 0(at)   # t6 <- .Ltab[cláusula]
}                             nop
                              addu  t6, t6, gp  # t6 <- GOT[.Ltab[cláusula]]
                              jr    t6          # salta para cláusula
                              nop
                               ...
                              .rdata
                              # tabela de (endereços de) cláusulas
                      .Ltab:  .word .Lcase0_gp_off # pos na GOT de case0
                              .word .Ldef_gp_off   # pos na GOT de default
                              .word .Lcase2_gp_off # pos na GOT de case2
                              .word .Lcase3_gp_off # pos na GOT de case3
                              .word .Ldef_gp_off   # pos na GOT de default
-------------------------------------------------------------------------
```

12.3 Relocação

A relocação de símbolos é a última tarefa do ligador antes de emitir o arquivo de saída. Após ler todos os arquivos de entrada e módulos de bibliotecas, o ligador concatena os vários segmentos de cada tipo e cria a tabela global de símbolos. A próxima tarefa é ajustar no código as referências aos endereços que resultaram da concatenação dos segmentos.

12.3.1 Relocação de símbolos e de segmentos

Na primeira passagem sobre os arquivos objeto, o ligador concatena os segmentos de mesmo tipo e coleta os valores dos símbolos, valores estes que são um deslocamento com relação ao início dos seus segmentos originais. Depois que a posição de cada segmento está definida, o ligador deve ajustar os endereços, para que apontem para as novas posições. Ainda, os símbolos indefinidos na tabela global de símbolos devem ser resolvidos.

Os endereços de dados e endereços absolutos 'dentro' de um segmento são ajustados pela adição do novo deslocamento da base do segmento. Por exemplo, se uma referência aponta para o endereço 100 mas a base do segmento foi relocada para a posição 1000, então esta referência deve ser ajustada para a posição 1100.

As referências intersegmento são ajustadas com os valores da tabela de símbolos globais. Endereços absolutos devem ser ajustados para refletir as novas posições dos destinos, en-

quanto que endereços relativos devem ser ajustados para refletir as posições relativas entre o segmento de onde é feita a referência e o segmento que é referenciado.

Referências a símbolos globais são resolvidas com relação ao deslocamento do segmento a que pertencem. Se uma instrução salta para uma função `f()` que está no endereço 500 de um segmento que foi relocado para a posição 2000, a instrução de salto para a função deve ser ajustada para a posição 2500.

Felizmente, o número de segmentos é relativamente pequeno na relocação: `text` e `data`. Para a relocação, cada endereço deve ser recalculado com relação à base de seu segmento antes de que as instruções possam ser ajustadas. Na resolução de símbolos, o ligador deve substituir o valor original do símbolo pelo novo valor que é obtido da tabela global, somente depois que o símbolo for relocado.

Se a um símbolo especial é atribuído o endereço do início de cada segmento, que é a sua *base*, a relocação dos símbolos se resume à soma de duas parcelas *viz.* a base do segmento é somada ao deslocamento do símbolo no segmento. A relocação pode ser efetuada em duas fases: (i) os valores de todos os símbolos na tabela global são ajustados em função das novas bases dos segmentos; e (ii) a relocação das referências é efetuada no código.

Em sistemas Unix, um arquivo objeto não é relocado depois que foi ligado *mas* o código em bibliotecas deve ser relocável. No caso de bibliotecas compartilhadas dinâmicas, código e dados são relocados toda vez em que a biblioteca é ligada a um executável.

Exemplo 12.1 Considere um programa composto por três módulos: Ω, Σ e Δ. O módulo Ω invoca as funções `f()` e `g()`, que são definidas nos módulos Σ e Δ, respectivamente. A Figura 12.2 mostra os três módulos após a compilação. Nas três seções de texto o endereço inicial é zero, e estes endereços são denotados por ω, σ, e δ. Todos os endereços 'dentro' dos segmentos estão a uma certa distância, ou deslocamento, com relação ao endereço inicial da seção.

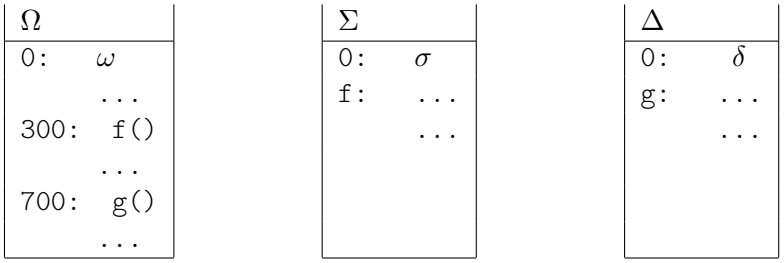


Figura 12.2: Seções de texto dos módulos compilados, antes da alocação.

No que segue, mostraremos com algum detalhe o que ocorre com o segmento de código do módulo Ω durante a alocação de memória e a relocação.

O código compilado de Ω é mostrado abaixo; a tabela de símbolos (TS) de Ω contém dois registros: $\langle$`f()`, indef, 300$\rangle$ e $\langle$`g()`, indef, 700$\rangle$. Estes registros indicam o local no código que deve ser editado quando as referências estiverem definidas.

```
      ...
300:  jal  0    # referência a f() indefinida
      ...
700:  jal  0    # referência a g() indefinida
      ...
```

Alocação: considere que os módulos tem tamanhos de 3.000, 2.000 e 1.000 bytes, e que o endereço inicial de carga seja 4.000. Após a alocação, $\omega = 4.000$, $\sigma = 7.000$, e $\delta = 9.000$. A Figura 12.3 mostra os módulos após a alocação.

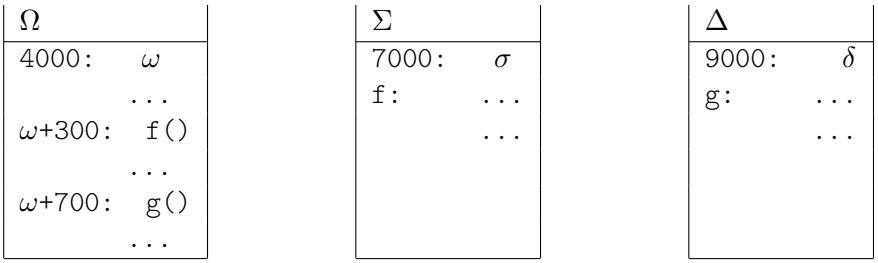


Figura 12.3: Segmentos de texto após a alocação.

Relocação: O segmento de código de Σ é relocado e o endereço de `f()` é atualizado, na TS de Σ para $\sigma = 7.000$. Da mesma forma, o endereço de `g()` é relocado para $\delta = 9.000$.

Edição: após a relocação de Ω, Σ e Δ, a tabela de símbolos de Ω é atualizada para: $\langle$`f()`$, 7.000, 300\rangle$ e $\langle$`g()`$, 9.000, 700\rangle$. Isso feito, o código de Ω pode ser então editado pelo ligador para que as referências indefinidas tornem-se definidas.

```
        ...
4300: jal 7000    # referencia a f() ajustada pelo ligador
      ...
4700: jal 9000    # referencia a g() ajustada pelo ligador
      ...
```

Uma vez relocado, o arquivo objeto está pronto para carga e execução. ◁

12.3.2 Relocação de código gerado para MIPS

Os modos de endereçamento do MIPS32 não são particularmente fáceis de relocar. Contudo, considerando-se a frequência com que as instruções com endereços absolutos são usadas, a tarefa não é assim tão extenuante [**?**].

Intruções de saltos incondicionais, com endereçamento *pseudo-absoluto*, são cerca de 2% de todas as instruções executadas nos programas em C. Desvios condicionais, por outro lado, são de 20 a 30% das instruções executadas, e felizmente, estas instruções usam endereçamento *relativo ao PC*, que prescinde de relocação explícita.

Nas referências à memória, os *loads* perfazem de 20 a 25% de todas as referências, e os *stores* outros 10%. No caso geral, estas referências fazem uso de endereços que necessitam ser relocados. Vejamos todas as possiblidades. A base do segmento sendo relocado fica no endereço `S`.

salto incondicional (`j ender`) o endereço de destino é pseudo-absoluto e necessita ser relocado. O endereço relocado é `S+ender`, se `ender` está na mesma seção em que o `j`, ou o endereço global já relocado que deve ser obtido da tabela de símbolos (TS) global;

salto para função (`jal fun`) mesmo que `j`;

salto indireto a registrador (`jr r`) são dois os casos: (i) o registrador `r` contém o endereço de retorno de uma função – neste caso, quando o código for relocado, o endereço de retorno (`PC+8`) é naturalmente relocado; (ii) o destino é computado a partir de uma tabela de endereços, como num comando `switch` em C, e neste caso, quando o código é relocado, os endereços na tabela também devem ser relocados;

desvio condicional (`beq r1,r2,ender`, `b ender`) o endereço de destino fica a uma certa distância do PC, e quando o código é relocado, o destino é naturalmente relocado;

cálculo de endereços absolutos (`la ender`) o endereço absoluto deve ser relocado de uma forma similar aos saltos incondicionais: o endereço relocado é `S+ender`;

cálculo de endereços absolutos (`lui r,%hi(ender); ori r,r,%lo(ender)`) mesmo que para `la ender`, relocando separadamente as partes mais e menos significativas;

loads e *stores* (`lw r1, desl(r2)`) o modo de endereçamento é *base-deslocamento*, e o endereço efetivo de uma referência à memória é dado pela soma do conteúdo de um registrador, a base, mais um deslocamento. Tanto a base, quanto o deslocamento devem ser relocados, e o ligador deve percorrer o código, "para trás", em busca das instruções que computam o conteúdo do registrador base (`lui`; `ori`) e efetuar a relocação daquelas instruções.

Exemplo 12.2 Vejamos como fica a Tabela de Relocação do arquivo objeto que resulta da ligação dos Programas 10.2, 10.3 e 10.4. O segmento de dados inicia no endereço D = 0x8000, e o segmento de código no endereço T = 0x1000.

A tabela é mostrada na Figura 12.4, e o formato dos seus elementos é

⟨endereço/símbolo, tipo, deslocamento, lista de endereços por editar⟩.

Se a lista está vazia, então o elemento não precisa ser relocado. Veja a Figura 10.7 para o código relocado. ◁

```
main, FUN, T+0, <>              // função
1010, LA, D+0, <1010,1348>      // k, endereço absoluto
1014, LS, 0, <>                 // load sem relocação
1018, LA, D+110, <1018>         // r, endereço absoluto
101c, LS, 0, <>
1020, FUN, T+200, <>
1028, LS, 0, <>                 // store sem relocação
11f8, FUN, 0, <>                // retorno de função
f,    FUN, T+200, <1020,1350>   // função
1220, LA, D+108, <1220>         // p, endereço absoluto
1224, LS, 0, <>
1228, LA, D+314, <1228>         // endereço absoluto
122c, LS, 0, <>
1230, FUN, T+300, <>
1238, LS, 0, <>
12f8, FUN, 0, <>                // retorno de função
g,    FUN, T+300, <1230>        // função
1340, LA, D+114, <1340>         // x, endereço absoluto
1344, LS, 0, <>
1348, LA, D+0, <1340>           // k, endereço absoluto
134c, LS, 0, <>
1350, FUN, T+200, <>
1358, LS, 0, <>
15f8, FUN, 0, <>                // retorno de função
```

Figura 12.4: Tabela de Relocação dos três módulos da Seção 10.3.1.

Espaço em branco proposital.

Exemplo 12.3 Considere as instruções por relocar no Programa 12.1.

A função `f()` fica na seção `.text`, local ao arquivo objeto, e é um endereço absoluto. Na linha 3, `g()` é uma função de biblioteca e este será relocado na ligação dinâmica – mais detalhes nos Capítulos 14 e 15.

Na linha 6, a instrução **`la`** é expandida para

```
lui at, %hi(vetorLoc)
ori, r5, at, %lo(vetorLoc)
```

`vetorLoc` é um endereço de 32 bits, local, na seção `.data`.

Na linha 7, ocorre o mesmo com o endereço `vetorExt`. Este vetor é declarado em outro arquivo fonte/objeto:

```
lui at, %hi(vetorExt)
ori, r6, at, %lo(vetorExt)
```

`vetorExt` é um endereço de 32 bits, externo, seção `.data`.

De novo, na linha 9:

```
lui at, %hi(mygot)
ori, gp, at, %lo(mygot)
```

`mygot` é um endereço de 32 bits, local, seção `.got`, ou `.data`. A utilidade da *Global Offset Table* (`.got`) é demonstrada na Seção 12.2.

Na linha 11, `X_gotOff` é uma constante de 16 bits, e é um deslocamento na GOT, relativa ao endereço apontado pelo *global pointer*, que é mantido no registrador `gp`.

Na linha 13, o desvio incondicional (*branch-always*) **`b`** `.L5` é trocado para **`beq`** `r0,r0,desloc`, sendo `desloc` um endereço relativo ao PC, computado pelo montador a partir do *location counter*: `desloc=(.-.L5)/4`, e é uma constante de 16 bits. O endereço `.L5` é local na seção **`.text`**.

Na linha 15 não é necessário relocar o endereço de retorno porque este é computado em tempo de execução. ◁

Programa 12.1: Exemplos de relocação.

```
1  f:    addi sp,sp,-32       # f ender absoluto, local, em .text
2        ...
3        jal g                # funcao de biblioteca
4        move r2,v0
5  .L5:  add r5,r4,r2         # .L5 ender local, desvio na linha 13
6        la r5, vetorLoc      # dado local
7        la r6, vetorExt      # dado externo
8        ...
9        la gp, mygot         # apontador para Global Offset Table
10       ...
11       sw r5, X_gotOff(gp)  # deslocamento na GOT
12       ...
13       b .L5                # (.-.L5) é relativo ao PC, local
14       ...
15       jr ra                # nao há relocação a fazer
16       ...
```

12.4 Carga de código relocado

Dependendo do nível de sofisticação do sistema, a carga de executáveis se dá numa das duas formas abaixo.

Sistemas sem memória virtual O arquivo objeto é lido e copiado para a faixa pré-definida de endereços em memória física, através de operações simples de E/S tais como `open()`, `read()`, `write()`.

Sistemas com memória virtual O arquivo objeto é mapeado no espaço de endereçamento do processo, com as permissões apropriadas para cada página.

12.4.1 Processo de carga em sistemas com memória virtual

A função `fork()` replica o espaço de endereçamento (EdE) do processo pai e a função `execve()` mapeia o novo executável no EdE do processo filho. O executável é carregado num endereço virtual pré-definido pelo ligador; para tanto são necessários os sete passos listados abaixo.

1. Ler o cabeçalho do objeto/executável para determinar o tamanho do EdE com base nos segmentos do objeto;
2. alocar o EdE, criar e preencher a tabela de páginas, com regiões separadas se o objeto contém segmentos com permissões distintas (RW, RO, EX, COW);
3. mapear os segmentos do objeto para a memória, no EdE criado no passo anterior;
4. preencher com zero o segmento BSS se o sistema de memória virtual não o fizer; o arquivo objeto contém a tabela de símbolos e outras informações após a seção `.data` (ou `.bss`) e estas não podem ser copiadas para a memória porque não fazem parte do processo;
5. criar um segmento para a pilha, se este é o modelo de execução do processo;
6. inserir na pilha as variáveis de ambiente e os argumentos do programa; e
7. inicializar registradores e saltar para a primeira instrução do programa.

12.4.2 Carga com relocação

Alguns sistemas efetuam a relocação durante a carga – tipicamente em sistemas sem paginação ou com paginação deficiente. Já a carga de bibliotecas compartilhadas dinâmicas quase sempre exige relocação, durate a sua 'carga' no espaço de endereçamento do processo a que são ligadas.

A relocação durante a carga é simples: todo o programa é deslocado de um mesmo tanto. Com base na tabela de relocação, todos os endereços relocáveis são igualmente deslocados, quando uma mesma constante é somada em todas as relocações.

12.5 Exercícios

Ex. 80 Para o trecho de código mostrado abaixo, indique em quais instruções pode ser necessário aplicar relocação, e nos casos em que há relocação, de quais tipos pode ser a relocação.

```
1  R:   la t4, X
2       la t5, Y
3       lw s3, constA(t4)
4       sw s3, constB(t5)
5  E:   addi t1, t4, constC
6       jal T
7       nop
8       ori t1, v0, constD
9       beq s3, t1, E
10      j S
11      nop
```

12.6 Relocação com arquivos `a.out`

No formato a.out, cada arquivo objeto contém duas tabelas de relocação, uma para o segmento de texto e outra para o segmento de dados. O segmento BSS (*Block Started by Symbol*) não precisa ser relocado.

Arquivos `a.out` para relocação Sistemas Unix sempre usaram um único formato para arquivos executáveis e arquivos para ligação. Um arquivo *a.out relocável*, com o leiaute simplificado, é mostrado na Figura 12.5.

cabeçalho
seção de texto
seção de dados
vetor de relocação de texto
vetor de relocação de dados
tabela de símbolos
tabela de cadeias

Figura 12.5: Arquivo a.out para relocação

Os elementos de relocação tem duas funções: (i) quando uma seção de código é relocada para um outro endereço base distinto, elementos de relocação marcam os lugares no código que devem ser modificados; (ii) num arquivo que será ligado a outros objetos, os elementos de relocação descrevem os símbolos indefinidos para que o ligador consiga ajustar os valores dos símbolos quando estes forem resolvidos durante a ligação.

A estrutura `relocation_info`, mostrada abaixo, descreve uma única relocação que deve ser efetuada, que pode ser um salto, uma chamada de função, ou uma referência a dados. A seção de relocação de texto é um vetor destas estruturas, e a seção de relocação de dados também é um vetor de estruturas com um elemento para cada variável a ser relocada.

Programa 12.2: Informação de relocação do formato a.out.

```
struct relocation_info {
  int r_address;               // ender. na seção de texto/dados
  unsigned int r_symbolnum:24; // índice na tabela de símbolos
  unsigned int r_pcrel:1;      // endereço relativo ao PC
  unsigned int r_length:2;     // tam. do apontador = 2**r_length
  unsigned int r_extern:1;     // veja abaixo
  unsigned int r_pad:4;        // nulo, para completar um inteiro
};
```

Os campos mais interessantes são:

`r_address` contém o deslocamento, em bytes, com relação ao início do segmento (`.text`, `.data`, `.bss`) de um apontador que deve ser ajustado; o ligador adiciona ao valor armazenado no endereço apontado por este registro o novo valor que computou para a base do segmento relocada;

`r_symbolnum` contém o índice (número ordinal) na tabela de símbolos para este símbolo; depois que o ligador resolve o endereço absoluto para este símbolo, adiciona o endereço absoluto ao apontador que está sendo relocado;

`r_extern` se `r_extern=0` este é um símbolo interno, para ser relocado, e `r_symbolnum` indica qual segmento é endereçado (`N_TEXT`, `N_DATA`, `N_BSS`), e o bit `r_pcrel` indica se a referência é absoluta ou relativa ao PC;
se `r_extern=1` este é um símbolo externo e `r_symbolnum` indexa a tabela de símbolos do arquivo.

As duas últimas seções de um arquivo a.out são a *tabela de símbolos* e a *tabela de cadeias* (*strings*). Símbolos mapeiam nomes em endereços, ou cadeias em valores. Como o ligador ajusta endereços, o nome de um símbolo deve ser usado como se fosse um endereço até que um valor absoluto lhe seja atribuído. Símbolos são representados por um registro de tamanho fixo na tabela de símbolos, e um nome de tamanho variável é mantido na tabela de cadeias. A tabela de símbolos é um vetor de elementos do tipo `nlist`, mostrados no Programa 12.3.

Programa 12.3: Elemento da tabela de símbolos do formato a.out.

```
struct nlist {
   union {
      char *n_name;
      long  n_strx;
   } n_un;
   unsigned char n_type;
   char     n_other;
   short    n_desc;
   unsigned long n_value;
};
```

O primeiro elemento da estrutura `nlist` é um deslocamento na tabela de cadeias até o início do nome do símbolo (cadeia terminada em `'0'`), que pode ser interpretado como

`n_un.n_strx` contém um deslocamento na tabela de cadeias para o primeiro caractere do nome deste símbolo;

`n_un.n_name` depois que a tabela de cadeias em memória é acessada com `n_un.n_strx`, o endereço em memória da cadeia é atribuído a este apontador.

O campo `n_value` contém um endereço se o símbolo é definido em `.text`, `.data` ou `.bss`. O campo `n_desc` é usado por depuradores.

O campo `n_type` define o tipo do símbolo, e os mais interessantes são listados aqui, enquanto que a tabela completa é mostrada no Programa 12.4.

`N_UNDF` símbolo não é definido neste módulo, bit `externo=1`, valor deve ser zero;

`N_ABS` símbolo absoluto não-relocável, seu valor é o valor absoluto do símbolo (usado por depuradores);

`N_TEXT,N_DATA,N_BSS` símbolo definido neste módulo, seu valor é o endereço relocável no módulo correspondente ao símbolo; e

`N_EXT` (externo) símbolo global, visível em outros módulos.

Programa 12.4: Elemento da tabela de símbolos.

```
#define N_UNDF   0      // Undefined symbol
#define N_ABS    2
#define N_TEXT   4
#define N_DATA   6
#define N_BSS    8      // Symbol has no address in binary file
#define N_FN     15     // Symbol names an object file (debugging)
#define N_EXT    1
#define N_TYPE   036
#define N_STAB   0340   // Symbol for debugging
#define N_INDR   0xa
#define N_SETA   0x14   // Absolute set element symbol
#define N_SETT   0x16   // Text set element symbol
#define N_SETD   0x18   // Data set element symbol
#define N_SETB   0x1A   // BSS set element symbol
#define N_SETV   0x1C   // Pointer to set vector in data area
```

Relocação com `a.out` A forma de relocação depende do tipo de símbolo e do segmento. Considere que `TR`, `DR`, `BR` sejam as novas bases (relocadas) dos segmentos de texto, dados e BSS respectivamente.

Para um apontador ou endereço absoluto no mesmo segmento, o ligador adiciona o adendo `A` ao valor que está armazenado na instrução/símbolo, que é relativo à `TR` ou `DR`. Para um apontador ou endereço de um outro segmento, o ligador adiciona a base do segmento destino (`dest_TR`, `dest_DR`, `dest_BR`) ao valor que já está armazenado.

Exemplo 12.4 Considere um arquivo de entrada, cujo texto inicia em 0 e dados em 2.000. Um apontador no segmento de texto aponta para a posição 200 no segmento de dados. No arquivo de entrada este apontador contém o valor 2.200. Se o endereço relocado do segmento de dados é 15.000, então `A`=13.000 (15.000−2.000) e o ligador adiciona 13.000 ao valor original de 2.200, e o endereço relocado é 15.200. ◁

12.7 Relocação com arquivos ELF-MIPS

No formato ELF as tabelas de relocação ficam nas seções `.rel.SEC`, sendo `SEC` o nome da seção à qual a relocação se aplica. Por exemplo, as informações de relocação da seção `.text` são mantidas na seção `.rel.text`. O formato ELF define dois tipos de elementos de relocação, *sem adendo*, ou *com adendo* (parcela de adição) – `Elf32_Rel`, e `Elf32_Rela`, respectivamente. O ligador para o MIPS usa somente elementos sem o adendo.

No descritor da seção de relocação (Prog. 11.4), os campos `sh_info` e `sh_link` apontam para a seção com a tabela de símbolos relevante e para a seção sobre a qual as relocações devem ser aplicadas.

Os elementos das tabelas de relocação são mostrados no Programa 12.5.

Programa 12.5: Elemento da tabela de símbolos.

```
//  /usr/include/elf.h

// Relocation table entry WO addend (in section of type SHT_REL)
typedef struct {
  Elf32_Addr    r_offset;  // Address in section/segment
  Elf32_Word    r_info;    // Relocation type and symbol index
} Elf32_Rel;

// Relocation table entry w addend (in section of type SHT_RELA)
typedef struct {
  Elf32_Addr    r_offset;  // Address in section/segment
  Elf32_Word    r_info;    // Relocation type and symbol index
  Elf32_Sword   r_addend;  // Addend
} Elf32_Rela;
```

`r_offset` aponta a posição na seção ou segmento na qual a relocação deve ser aplicada. Num *arquivo objeto relocável*, aponta o deslocamento com relação ao início da seção da unidade de armazenagem que é afetada pela relocação. Num *arquivo executável*, contém o endereço virtual da unidade afetada pela relocação.

`r_info` contém dois campos: um aponta o índice na tabela de símbolos do símbolo para a relocação; o outro campo indica o tipo de relocação a ser aplicado. Por exemplo, o elemento de relocação para uma chamada de função contém o índice na tabela de símbolos da função a ser invocada. Se o índice é `STN_UNDEF` (símbolo indefinido) a relocação usa zero como o valor do símbolo.

`r_addend` especifica uma constante a ser usada para computar o valor a ser armazenado no campo relocável. *Não é usado com processadores MIPS.*

As macros `ELF32_R_SYM` e `ELF32_R_TYPE` permitem extrair o índice na tabela de símbolos e o tipo da relocação, respectivamente, e são mostradas no Programa 12.6.

Programa 12.6: Informação do campo `r_info`.

```
// How to extract and insert information held in the r_info field.
#define ELF32_R_SYM(val)          ((val) >> 8)
#define ELF32_R_TYPE(val)         ((val) & 0xff)
#define ELF32_R_INFO(sym, type)   (((sym) << 8) + ((type) & 0xff))
```

Os tipos de relocação específicos ao MIPS estão listados no Programa 12.7, e são aqueles que representam os formatos de codificação de endereços nas instruções daquele processador. Considera-se a relocação de um arquivo relocável para produzir um executável ou uma biblioteca compartilhada.

Programa 12.7: Tipos de símbolos para relocação.

```
// .../sde/include/sys/elfmips.h

// MIPSABI relocation (rel type)
#define R_MIPS_NONE     0
#define R_MIPS_16       1
#define R_MIPS_32       2
#define R_MIPS_REL32    3
#define R_MIPS_26       4   // j jal - operando de 26 bits
#define R_MIPS_HI16     5   // la (lui) - operando de 32 bits HI
#define R_MIPS_LO16     6   // la (ori) - operando de 32 bits LO
#define R_MIPS_GPREL16  7   // relativo ao Global Pointer 16b
#define R_MIPS_LITERAL  8
#define R_MIPS_GOT16    9   // relativo à Global Offset Table 16b
#define R_MIPS_PC16     10  // beq, bne - operando relativo ao PC
#define R_MIPS_CALL16   11  // bal - função em ender relativo ao PC
#define R_MIPS_GPREL32  12  // relativo ao Global Pointer 32b
```

Para definir os cálculos para a relocação, o suplemento do MIPS da ABI [**?**] faz uso da seguinte notação:

- `A` adendo para computar o valor do campo relocável, cujo valor é mantido na imagem da memória no arquivo (na própria instrução no caso de código);
- `AHL` outra representação para o adendo: dois campos de 16 bits, `HI` e `LO`;
- `P` posição (deslocamento na seção ou endereço virtual) da unidade de armazenagem sendo relocada, valor em `r_offset`;
- `S` valor do símbolo apontado pelo elemento de relocação; se o símbolo é `STB_LOCAL` com tipo `STT_SECTION`, então `S` representa o `sh_addr` (deslocamento do início da seção) original menos o `sh_addr` final;
- `EA` endereço efetivo do símbolo antes da relocação.

O adendo `AHL` é computado a partir dos adendos de dois elementos de relocação consecutivos. Cada relocação do tipo `R_MIPS_HI16` deve ser seguida por um elemento de tipo `R_MIPS_LO16` na tabela de relocações. Estes elementos são processados como um par e os dois campos de adendo contribuem para o adendo `AHL`. Se `AHI` e `ALO` são os adendos de dois elementos `HI-LO` emparelhados, o adendo `AHL` é dado por `(AHI << 16) + (short)ALO)`. Para um elemento tipo `R_MIPS_LO16` desemparelhado deve ser usado o elemento `R_MIPS_HI16` anterior mais próximo. A Tabela 12.1 define o cômputo de cada relocação. Os nomes dos campos tem os sufixos V (*verify*) ou T (*truncate*) para definir o tipo de relocação: pode ser necessário verificar a ocorrência de *overflow* no cálculo – e possivelmente refazer a relocação – ou truncar o resultado para o número de bits apropriado.

Tabela 12.1: Cálculo dos endereços de relocação.

relocação	campo	símbolo	cálculo	
`R_MIPS_16`	V-half16	ext/loc	`S + signExtend(A)`	base_seção+ender
`R_MIPS_32`	T-word32	ext/loc	`S + A`	base_seção+ender
`R_MIPS_REL32`	T-word32	ext/loc	`A - EA + S`	ligação dinâmica
`R_MIPS_26`	T-targ26	local	`( (A << 2) + (P & 0xf000.0000)+S )>>2`	
`R_MIPS_26`	T-targ26	externo	`(signExtend(A<<2) + S )>>2`	

Na coluna *símbolo*, *local/loc* representa um símbolo referenciado pelo índice na tabela de símbolos no elemento de relocação `STB_LOCAL/STT_SECTION`; nos demais, a relocação é

considerada *externa*. A relocação de tipo `R_MIPS_REL32` é efetuada na ligação dinâmica – detalhes no Capítulo 15.

12.7.1 ELF com código independente de posição

O executável ELF para ser *usado em bibliotecas dinâmicas*, consiste de um certo número de páginas de código seguido de um certo número de páginas de dados. As distâncias entre instruções e variáveis permanecem constantes independentemente do endereço de carga, porque o código para estas bibliotecas é gerado para ser independente de posição.

Se o código, em tempo de execução, 'descobre' seu próprio endereço e o carrega num registrador, os dados estarão a uma distância fixa deste endereço e as referências aos dados podem ocorrer com instruções `lw` e `sw` simples e eficientes.

```
.text ...
      x = ...;  <--+
      ...         |
.data ...         |
      [x]       <--+
      ...
```

O ligador cria a *Global Offset Table* que contém apontadores para todas as variáveis *globais* referenciadas pelo texto. Cada biblioteca tem a sua própria GOT – porque todos os programas compilados para ser IdP contém uma GOT. Como é o ligador quem cria a GOT, só existe um apontador para cada variável, não importa quantas rotinas no executável a referenciem.

Na carga de bibliotecas dinâmicas, os endereços de variáveis globais só são conhecidos em tempo de execução. Para referenciar variáveis globais, o código deve carregar um registrador com o endereço da GOT e então de-referenciar o apontador. Estas referências mais custosas são um preço relativamente pequeno a se pagar pela flexibilidade que se ganha com bibliotecas dinâmicas.

Capítulo 13

Bibliotecas estáticas simples

Bibliotecas são coleções de arquivos objeto que podem ser incluídos pelo ligador, se necessário, na criação de um executável. Tipicamente uma biblioteca contém um diretório para acelerar a busca pelos seus componentes. Nesta seção, o termo *arquivo* é usado para descrever um "arquivo objeto" produzido pelo montador, enquanto que um *módulo* é o conteúdo de um arquivo objeto que faz parte de uma biblioteca, e tipicamente contém uma estrutura de dados e funções para manipular aquela estrutura.

13.1 Bibliotecas em diretórios

Uma biblioteca com formato muito simples é organizada num diretório que contém os arquivos objeto. Cada arquivo objeto contém a definição de um ou mais símbolos, e a cada símbolo definido num objeto corresponde um nome para o arquivo – há um link simbólico para cada símbolo. Durante a ligação, o ligador procura pelos nomes dos símbolos no diretório e os arquivos correspondentes aos nomes dos símbolos são incorporados ao executável.

13.2 Bibliotecas em arquivos

Uma biblioteca organizada como um *archive* é uma versão compacta de um diretório que contém vários arquivos. Um *archive* é um arquivo que contém uma tabela – o *diretório* do *archive* – que descreve seu conteúdo, seguido de seus *arquivos membros*. O formato mais comum em sistemas Unix contém um cabeçalho de texto que descreve a biblioteca e cada membro é precedido de um cabeçalho. Os dois são descritos adiante, e em `/usr/include/ar.h`. A Figura 13.1 mostra o diretório `bib_exemplo` com três arquivos objeto `p.o`, `q.o` e `r.o`, e o *archive* correspondente.

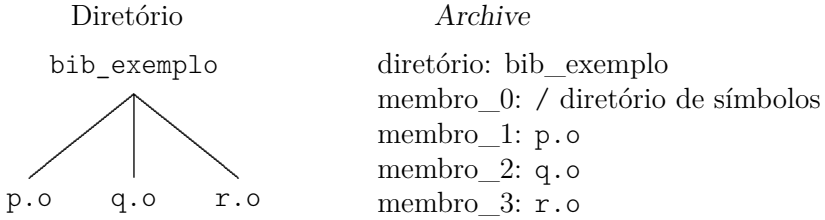


Figura 13.1: *Archive*: diretório e arquivo

O cabeçalho de uma biblioteca é indicado pela cadeia `!<arch>n`, e é descrito no Programa 13.1.

Programa 13.1: Cabeçalho de um *archive*.

```
// String that begins an archive file
#define ARMAG   "!<arch>\n"
#define SARMAG  8                  // Size of that string
// String in ar_fmag at end of each header
#define ARFMAG  "`\n"

// Member header - one per member
struct ar_hdr {
  char ar_name[16];  // Member name, sometimes / terminated
  char ar_date[12];  // File date, decimal secs since 01jan1970
  char ar_uid[6],    // User ID, in ASCII decimal
  char ar_gid[6];    // Group IDs, in ASCII decimal
  char ar_mode[8];   // File mode, in ASCII octal
  char ar_size[10];  // File size, in ASCII decimal
  char ar_fmag[2];   // Always contains ARFMAG = "`\n"
};
```

Os membros da estrutura significam:

`ar_name` nome do membro, terminado com '/' e seguido de espaços (ASCII 0x20) até completar 16 caracteres;

`ar_date` data de criação, número de segundos desde 01jan1970, em ASCII;

`ar_uid,ar_gid` identificador de usuário e de grupo, raramente usado;

`ar_mode` proteção de arquivo (`rwxrwxrwx`);

`ar_size` tamanho do arquivo com o membro, adiciona '\n' se tamanho é ímpar;

`ar_fmag` contém terminador de cabeçalho, '\n .

O primeiro membro de uma biblioteca é o diretório de símbolos. Este membro é chamado de '/'. O formato do diretório de símbolos é definido no Programa 13.2.

Programa 13.2: O primeiro membro de um *archive* é o seu diretório.

```
int nsymbols;      // Number of symbols in directory
int member[];      // Member offsets
char strings[];    // Null terminated strings '\0'
```

`nsymbols` determina o número de símbolos contidos na biblioteca. Para o M-ésimo símbolo, `member[M]` aponta a posição no arquivo onde o membro inicia e `strings[M]` contém o nome associado ao símbolo.

Neste formato o nome dos membros é limitado a 15 caracteres, com '/' no final. Para acomodar nomes longos, o arquivo pode conter um membro adicional com a lista de nomes longos. Este membro é chamado de '//' e os nomes de componentes são separados por '/\n'. No cabeçalho do membro, o nome é representado por '/nnn' sendo 'nnn' a distância em bytes do início do membro '//' até a cadeia com o nome longo. A distância é representada em decimal. Se este membro existe, ele fica logo após o diretório de nomes.

O *archive* representado na Figura 13.1 é organizado como mostra o Programa 13.3.

Programa 13.3: Exemplo de *archive* com três membros.

```
!<arch>\n                                    // magic number
struct ar_hdr {
   char ar_name[16] = "bib_exemplo/␣␣␣␣" // name w 15 chars
   ...
   char ar_fmag[2] ="`\n"
};
int nsymbols=3;          // Member 0 - symbol directory
int member[3]={p,q,r}    // Referenced below by label (p: q: r:)

char strings[3]={"simbP","simbQ","simbR"}
// if it exists, member with long names goes here

p: struct ar_hdr {                          // member 1 - simbP
     char ar_name[16] = "simbP/␣␣␣␣␣␣␣␣␣" // name with 15 chars
     ...
     char ar_fmag[2] ="`\n"
   };
   // p.o  code and data
   ...
q: struct ar_hdr {                          // member 2 - simbQ
     char ar_name[16] = "simbQ/␣␣␣␣␣␣␣␣␣" // name with 15 chars
     ...
     char ar_fmag[2] ="`\n"
   };
   // q.o  code and data
  ...
r: struct ar_hdr {                          // member 3 - simbR
     char ar_name[16] = "simbR/␣␣␣␣␣␣␣␣␣" // name with 15 chars
     ...
     char ar_fmag[2] ="`\n"
   };
   // r.o  code and data
  ..
// EOF (End Of File)
```

13.2.1 Criação de bibliotecas

O programa `ar` (ar*chive*) é usado para criar e manter bibliotecas. `ar` cria o diretório de símbolos e os cabeçalhos e concatena os membros. Existem opções para examinar, remover, adicionar e substituir membros da biblioteca. Para detalhes, diga "`man ar`".

Bibliotecas em *archive*s tem o sufixo `.a`. Por exemplo, o comando abaixo cria a biblioteca `str.a` com funções para o tratamento de cadeias de caracteres definidas nos arquivos objeto `str*.o`:

```
ar cr str.a str*.o
```

É possível que um membro dependa de símbolos definidos em outro membro da biblioteca, e isso pode implicar em que os membros devam ser concatenados numa certa ordem. Os ligadores são capazes de resolver estas dependências com mais de uma passagem sobre os símbolos importados de bibliotecas, embora este processo possa ser lento. Os programas `lorder` e `tsort` ajudam a criar índices de símbolos já ordenados segundo as dependências, acelerando a resolução de símbolos importados.

`tsort` efetua uma ordenação topológica nas dependências, definida como segue. Se as dependências entre símbolos e arquivos são representadas como um grafo direcionado acíclico, a relação R é uma ordem parcial no grafo, e $x\,R\,y$ se e somente se existe um caminho direcionado de x para y. A ordenação topológica é uma ordenação total que é compatível com a ordenação parcial R.

A entrada para `lorder` é um conjunto de arquivos objeto e sua saída é uma lista de dependências que indica quais arquivos dependem de que símbolos em quais outros arquivos. `tsort` ordena esta lista de arquivos de tal forma que todos os símbolos sejam definidos *após* todas as referências a eles. Isso permite que uma única passada sobre os arquivos objeto resolva todas as referências indefinidas – o endereço de cada referência indefinida é registrado na tabela de símbolos; quando o símbolo é definido, aqueles endereços são editados para resolver a referência. A saída de `tsort` é usada para criar a biblioteca, como mostrado abaixo com os arquivos do exemplo anterior:

```
ar cr str.a $(lorder str*.o | tsort) .
```

A título de exemplo[1], a função `main()` invoca uma serie de outras funções, que por sua vez dependem de outras funções. A Figura 13.2 mostra, na esquerda, a lista de dependências extraída de um grafo de chamadas de funções (*call graph*), e a coluna da direita mostra a ordenação produzida por `tsort`. Na figura, "*main* ← *parse_options*" indica que `main()` invoca a função `parse_options()`.

call graph	ordem de chamadas
main ← parse_options	main
main ← tail_file	tail_forever
main ← tail_forever	tail_file
tail_file ← pretty_name	parse_options
tail_file ← write_header	recheck
tail_file ← tail	tail
tail_forever ← recheck	write_header
tail_forever ← pretty_name	pretty_name
tail_forever ← write_header	tail_bytes
tail_forever ← dump_remainder	tail_lines
tail ← tail_lines	start_bytes
tail ← tail_bytes	xlseek
tail_lines ← start_lines	pipe_lines
tail_lines ← dump_remainder	file_lines
tail_lines ← file_lines	dump_remainder
tail_lines ← pipe_lines	start_lines
tail_bytes ← xlseek	
tail_bytes ← start_bytes	

Figura 13.2: Exemplo de ordenação topológica.

13.2.2 Ligação com bibliotecas

A busca em bibliotecas geralmente ocorre no primeiro passo da ligação, depois que os arquivos individuais tenham sido processados e a tabela de símbolos criada. O ligador lê o diretório de símbolos da biblioteca e compara cada símbolo com aqueles na sua tabela global de símbolos. Se um símbolo da biblioteca aparece na tabela global como usado mas

[1] Copiado de `info coreutils 'tsort invocation'`.

não-definido, o membro (módulo) é lido e seus segmentos são incorporados aos segmentos do arquivo de saída, e todos os novos símbolos são inseridos na tabela global. Isso é necessário porque funções de biblioteca frequentemente apontam para símbolos em outra função de biblioteca – `printf()` (da `stdlib.a`) usa `putc()` ou `write()` (da `stdio.a`), por exemplo, embora a coisa toda seja ligeiramente mais complicada do que este exemplo sugere.

O resolução de símbolos de bibliotecas é um processo iterativo. Depois que o ligador examinou os símbolos do diretório de uma biblioteca, se quaisquer símbolos foram incluídos naquela passada, o ligador deve fazer uma nova passada sobre a tabela global para resolver quaisquer símbolos que sejam apontados pelas funções da biblioteca que foram recém-incluídas. Isso deve ser repetido até que não haja nenhum símbolo por resolver dentre aqueles no diretório da biblioteca.

Pode ser necessário fazer uso de uma função específica para alguma aplicação com mesmo nome mas com funcionalidade diferente de uma função de biblioteca. Neste caso, o ligador deve ser instruído a incorporar a função específica ao invés da função de biblioteca. Isso se consegue ao especificar a ordem em que os arquivos devem ser ligados. Por exemplo, se o programador escreveu uma nova versão das funções `malloc()` e `free()` e estas se encontram no arquivo `meuMalloc.o`, os argumentos para o ligador devem ser como o mostrado abaixo. Os arquivos objeto `a.o` e `b.o` dependem das novas versões de `malloc()` e portanto o arquivo com este código é ligado antes (na linha de comando) da biblioteca do sistema. As referências à `malloc()` e `free()` em `a.o` e `b.o` serão resolvidas antes que `stdlib.a` seja examinada e portanto as versões originais não serão usadas:

```
ld a.o b.o meuMalloc.o c.o stdlib.a
```

13.3 Símbolos externos fracos

Um *símbolo externo fraco* é um símbolo que é referenciado mas não é resolvido durante a ligação sem que isso cause um erro. Existem situações nas quais é desejável que símbolos possam ficar indefinidos, sem que isso seja sinalizado como erro. Por exemplo, a função `printf()` emprega a biblioteca de ponto-flutuante para as conversões da representação interna para o formato de impressão. O símbolo `fcvt` é referenciado por `printf()` e o código das funções de conversão seria incluído mesmo que um programa não use ponto-flutuante. Se ao final da ligação de um programa que não usa números em ponto flutuante o símbolo `fcvt` – que é um símbolo fraco – não for resolvido, a ele é atribuído o valor zero sem que isso seja considerado um erro. Em programas que usam ponto-flutuante, o símbolo `fltused` é definido e as rotinas de ponto-flutuante são incluídas, e estas definem o símbolo `fcvt`. Neste caso, o símbolo é resolvido com a inclusão da biblioteca de ponto-flutuante.

Existe também a *definição de símbolos externos fracos*. Uma definição fraca define um símbolo caso não haja uma definição normal; se esta existe, a definição fraca é ignorada.

13.4 Exercícios

Não há exercícios sobre este material.

Capítulo 14

Bibliotecas Compartilhadas Estáticas

The devil is in the details.
Fonte sob disputa.

Bibliotecas contém funções usadas frequentemente, tais como funções matemáticas da biblioteca `libm.a` (`sqrt()`) e de entrada/saída (`read()`, `write()`), por exemplo. Um certo conjunto de funções populares pode ser usado simultaneamente por inúmeros processos e portanto é uma boa ideia manter bibliotecas com estas funções populares carregadas permanentemente em memória.

Num sistema Unix que não faz uso de bibliotecas compartilhadas, em 1000 executáveis seriam 1000 cópias de `printf()` carregadas em memória. Bibliotecas compartilhadas economizam espaço em disco – sem cópias redundantes os executáveis são menores – e economiza espaço em memória – pela mesma razão, as imagens são menores – e estas reduções combinadas melhoram o desempenho da paginação, ao reduzir o tráfego de/para discos.

Uma *Biblioteca Compartilhada Estática* (BCE) é ligada aos executáveis antes da carga, e os endereços do código e dados são resolvidos em tempo de ligação.

O diagrama na Figura 14.1 mostra o executável `meuProg` ligado a uma biblioteca do sistema, que chamaremos de `libio.a`, e a uma biblioteca de aplicação, `myLib.a`. Para que `meuProg` execute corretamente, as duas bibliotecas devem estar carregadas em memória quando o programa for carregado. `fun()` é definida em `myLib.a`, e usa a função `open()` de `libio.a`.

14.1 Espaço de endereçamento

A cada BCE é alocada uma porção fixa do espaço de endereçamento – a biblioteca é alocada e ligada na mesma faixa de endereços em todos os processos, e portanto *cada uma* das bibliotecas deve usar uma faixa de endereços que é disjunta das faixas de todas as outras BCEs.

Em alguma parte do sistema de compilação e ligação deve ser mantida uma tabela com os endereços das bibliotecas, com endereços iniciais próximos ao topo do espaço de usuário – no Linux em `0x6000.0000`, no BSD em `0xA000.0000`. Pode existir uma faixa para bibliotecas de sistema e uma faixa separada para bibliotecas de aplicação ou proprietárias. Na medida do possível, as bibliotecas devem ser compactas para não desperdiçar páginas de memória virtual.

Geralmente, numa biblioteca compartilhada estática os endereços de código e de dados são definidos explicitamente, com os dados iniciando uma ou duas páginas além do final do

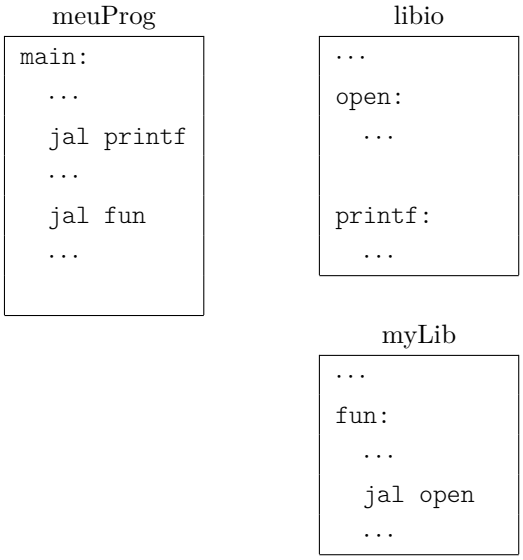


Figura 14.1: Dependências de `meuProg` em duas bibliotecas.

código. Isso é necessário para permitir consertos que alteram o tamanho do código. Se o código for bem projetado, consertos raramente alteram a localização e/ou leiaute dos dados. Cada biblioteca exporta símbolos de código e de dados, e possivelmente importa símbolos de outras bibliotecas das quais ela própria dependa. Para facilitar consertos, as bibliotecas exportam uma tabela de endereços de funções, ao invés dos próprios endereços das funções – a invocação de funções ocorre com referências indiretas através da tabela.

São dois os casos: (i) *código* – os símbolos exportados são os endereços de instruções de salto, ao invés do endereço do código da própria função; (ii) *dados* – é mais difícil usar indireção com dados e portanto os endereços dos dados são exportados como variáveis simples (tal como a variável `errno`) ou tabelas que mudam raramente (como o vetor de estruturas `FILE` na `libc`).

O programador da biblioteca deve agrupar os dados exportados no início da seção de dados, alocando-os antes dos dados anônimos usados internamente pelas funções. Um projeto cuidadoso das estruturas de dados exportadas – o que é obtido com bons modelos – é o que garante que eventuais consertos não causem mudanças nos dados.

14.2 Estrutura de bibliotecas compartilhadas estáticas

Uma BCE é mantida num arquivo com formato executável contendo todo o código e os dados e que está pronto para ser mapeado no espaço do programa ao qual a biblioteca é ligada. O diagrama da Figura 14.2 mostra a organização de uma BCE. A rotina de inicialização é usada para inicializar as variáveis da biblioteca.

14.3 Tempo de ligação *versus* tempo de execução

Para executar um programa ligado estaticamente a uma biblioteca compartilhada, esta deve ser carregada em memória, antes do início da execução do processo, senão ocorre um erro fatal e o processo é abortado.

cabeçalho a.out ou ELF		
rotina de inicialização		← alinhada em fronteira de página
tabela de saltos	`.text`	← alinhada em fronteira de página
código	`.text`	
dados globais	`.data`	← alinhada em fronteira de página
dados privativos	`.data`	
dados não-inicializada	`.bss`	← não existe no arquivo

Figura 14.2: Estrutura de arquivo com uma biblioteca compartilhada estática.

Num programa que não usa bibliotecas, os símbolos são resolvidos para endereços absolutos, e todo o código é agregado ao executável pelo ligador em tempo de ligação, e o executável é portanto autocontido.

Num programa ligado a uma BCE, os símbolos são resolvidos pelo ligador, mas o código da biblioteca só é agregado ao executável em tempo de carga/execução. Isso significa que a ligação se dá em duas fases: na primeira – a ligação propriamente dita – os objetos do programa são ligados a uma "biblioteca falsa" que contém a tabela com as referências absolutas; na segunda fase – na carga – o executável é ligado ao arquivo objeto da biblioteca.

Se a biblioteca for alterada, e uma nova versão instalada no sistema, os programas ligados à versão antiga podem produzir resultados errados. Por isso, uma nova versão não pode alterar a tabela de saltos – nem sua posição e nem seu conteúdo – a ponto de mudar endereços que foram ligados anteriormente a programas. Se as mudanças forem extensas, mais de uma versão da biblioteca deve ser mantida no sistema, ou então isso implica na religação de todos os executáveis que usavam a biblioteca obsoleta, o que pode ser extremamente custoso.

14.4 Criação de bibliotecas compartilhadas

Uma biblioteca compartilhada consiste de dois arquivos, a própria biblioteca e uma *biblioteca falsa* que é usada pelo ligador. A biblioteca falsa não contém código nem dados, mas apenas as definições dos símbolos da biblioteca para permitir a ligação. Estas 'funções' que somente definem os símbolos são chamadas de *stubs* porque não são funções "de verdade" mas apenas auxiliam na primeira fase da ligação.

O ligador liga o programa à biblioteca falsa para resolver os símbolos exportados pela biblioteca. O código da biblioteca só é mapeado no espaço de endereçamento do processo quando este for carregado.

São seis os passos para a criação de uma BCE:

1. determinar em que endereços o código e os dados da biblioteca serão carregados – esta é uma definição de sistema e cada biblioteca é alocada a uma faixa predeterminada de endereços;
2. percorrer os arquivos objeto da biblioteca e coletar todos os símbolos de código exportados. Se existem símbolos que podem/devem ser escondidos, estes permanecem escondidos;
3. construir a tabela de saltos com um elemento para cada símbolo de texto que é exportado;
4. se existe uma rotina de inicialização a ser incluída no início da biblioteca, esta deve ser compilada e agregada à biblioteca;

5. criar a biblioteca: usar o ligador para criar o executável da biblioteca, alocando texto e dados nos endereços absolutos apropriados; e
6. criar a biblioteca falsa: extrair os símbolos exportados pela nova biblioteca, criar uma rotina falsa (com corpo vazio) para cada função – um *stub* por símbolo exportado – compilar e gerar a biblioteca falsa.

Criação da tabela de saltos A maneira mais simples de criar a tabela de saltos é escrever um programa em *assembly* com instruções de salto e então montá-lo. Os *labels* dos saltos devem ser criados sistematicamente para que seja fácil extraí-los da tabela de símbolos, para então criar a biblioteca falsa. Esta tabela deve estar alinhada com o início de uma página. As versões dos nomes das funções internos à biblioteca são prefixados com um *underscore* para diferenciá-los de nomes publicados aos usuários.

O trecho de código abaixo mostra a tabela de saltos de uma biblioteca gerada a partir do objeto `readwrite.c`, que exporta seis símbolos: `read()`, `write()`, `open()` e `close()`, a função de inicialização da biblioteca, e a variável `status`. Os símbolos `_init_rw`, `_read`, `_write` `_open` e `_close` só são visíveis internamente à biblioteca. A tabela de saltos deve ser montada junto com o código *assembly* resultante da compilação de `readwrite.c`.

Programa 14.1: Tabela de saltos de uma BCE.

```
# tabela de saltos (tabSaltos.s) ------------------------------
            .org 0x6000.1000 # alinhado em fronteira de página
JUMP_init:  j _init_rw  # ender absoluto 0x6000.1000
            nop
JUMP_read:  j _read     # ender absoluto 0x6000.1008
            nop
JUMP_write: j _write    # ender absoluto 0x6000.1010
            nop
JUMP_open:  j _open     # ender absoluto 0x6000.1018
            nop
JUMP_close: j _close    # ender absoluto 0x6000.1020
            nop
# -------------------------------------------------------------

# código assembly (readwrite.s) -------------------------------
_init_rw:     ...   # código que inicializa a biblioteca
              ...
_read:        ...   # código de read()
              ...
_write:       ...   # código de write()
              ...
_open:        ...   # código de open()
              ...
_close:       ...   # código de close()
              ...
              .data
              .org 0x6000.7000 # alinhado em fronteira de página
status:       .space 4,0 # ender absoluto 0x6000.7000
              ...
# -------------------------------------------------------------
```

Geração da biblioteca compartilhada Uma vez que a função de inicialização das variáveis estáticas da biblioteca e a tabela de saltos estejam prontos, basta invocar o ligador com os argumentos apropriados para alocar as faixas de endereços e a biblioteca é gerada.

Criação da biblioteca falsa A biblioteca falsa deve conter uma seção que define todos os símbolos globais que são exportados por esta biblioteca e importados de outras bibliotecas. Os símbolos para os dados globais podem ser extraídos da tabela de símbolos da BCE. Os símbolos para as funções exportadas podem ser extraídos da tabela de saltos.

A biblioteca falsa, com os *stubs*, contém somente as definições de símbolos, que são tuplas ⟨símbolo, endereço⟩. O programa que cria a biblioteca falsa extrai todos os símbolos globais, definidos e indefinidos, de código e de dados, e então cria um programa em *assembly* com todos os símbolos de dados em seus endereços absolutos, e os símbolos de código com seus endereços na tabela de saltos. Note que os símbolos devem ser definidos como absolutos porque a relocação já aconteceu na criação da biblioteca. Este programa é montado e a biblioteca falsa está pronta, como mostra o Programa 14.2.

Programa 14.2: Biblioteca falsa da BCE do Programa 14.1.

```
.global init,read,write,open,close,status
.set init_rw, 0x6000.1000
.set read,    0x6000.1008
.set write,   0x6000.1010
.set open,    0x6000.1018
.set close,   0x6000.1020
.set status,  0x6000.7000
```

Repare que os endereços absolutos de código, um para cada função exportada, são os endereços dos saltos para aquelas funções, na tabela de saltos.

Ligação com bibliotecas compartilhadas estáticas O ligador deve coletar a informação sobre bibliotecas contidas nos arquivos objeto de entrada, para criar uma tabela com os nomes das bibliotecas necessárias a um executável, e para cada biblioteca é discriminada a sua faixa de endereços. As bibliotecas falsas fornecem todos os endereços dos símbolos e a ligação pode ser concluída. A tabela de bibliotecas é usada na carga e execução do programa, para que o carregador possa mapear as bibliotecas no espaço de endereçamento do processo.

Note que o que é ligado ao programa é a *biblioteca falsa*, que não contém código nem dados, mas somente os *símbolos resolvidos* da biblioteca verdadeira. Ao ligar com a biblioteca falsa, o ligador resolve todas as referências indefinidas no programa, sem que seja necessário agregar o código da biblioteca verdadeira ao arquivo executável uma vez que a biblioteca é residente em memória.

Execução com bibliotecas compartilhadas estáticas São necessários três passos para iniciar a execução de um programa ligado a uma BCE:

1. carregar o executável em memória;
2. com base na lista de bibliotecas, mapear as bibliotecas no espaço de endereçamento do processo; e
3. executar as funções de inicialização das bibliotecas, e então saltar para `main()`.

Exemplo 14.1 Considere a invocação da função `read()`, mostrada na tabela de saltos do Programa 14.1. No código do programa a invocação se dá com "`jal read`", que aponta para a instrução na segunda posição da tabela se saltos, que é "`j _read`", que por sua vez salta para o endereço na BCE que contém o código da função `read()`. Esta sequência é mostrada na Figura 14.3. O endereço de retorno é preservado porque a instrução `j` não altera conteúdo do registrador `ra`. ◁

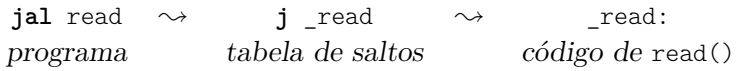


Figura 14.3: Sequência de invocação de função de uma BCE.

Pode-se esperar que uma certa biblioteca esteja permanentemente carregada em memória? *Depende.* Se se trata de uma biblioteca 'popular', tal como a `libc.a`, é provável que ela já esteja mapeada no espaço de algum outro processo – isso não traz problema porque as faixas de endereços das várias bibliotecas são disjuntas – e neste caso o sistema usa o arquivo objeto que já está aberto, e o mapeia novamente no espaço de endereçamento do programa que está sendo carregado. Se a biblioteca não está carregada em memória então o carregador abre o seu arquivo objeto e o mapeia no espaço de endereçamento do processo.

O que ocorre com as variáveis exportadas pelas BCEs? Todos os programas que compartilham uma certa biblioteca também compartilham as variáveis nos mesmos endereços? Sim, nos mesmos *endereços virtuais*. Para as versões das variáveis que são locais a cada processo, deve-se empregar *copy on write*: a página com as variáveis é compartilhada para leitura. Quando o processo P_i tenta atualizar uma variável, é criada uma cópia privativa da página no espaço de endereçamento de P_i, e então a variável pode ser atualizada naquela cópia, de acesso exclusivo a P_i. Esta cópia privativa é disjunta das cópias exclusivas dos processos P_j, P_k, etc.

14.5 Exercícios

Espaço em branco proposital.

Ex. 81 Usando o código abaixo e as duas bibliotecas, execute o algoritmo da ligação com BCEs e preencha as estruturas de dados com os valores indicados no código. O código segue a convenção: (i) a definição de um símbolo é denotada nos diagramas por um *label* como '`simb:`'; (ii) uma referência a uma função é denotada pelos parênteses como '`fun()`'; (iii) uma referência a uma variável é denotada pelo seu nome como '`var`'.

No código C, o endereço nos comentários é o endereço da instrução que invoca a função. Para simplificar sua resposta, variáveis atribuídas são referenciadas em endereço que é 8 bytes maior que o da instrução `jal`, enquanto que variáveis lidas são referenciadas num endereço que é 4 bytes menor do que o da instrução **`jal`**: em `main()`, `alpha` é referenciada em `0x0408` e `x` em `0x03fc`. Para o primeiro **`jal`** de `main()` temos:

```
  lw  a0, 0(t0)  # 0x03fc, t0 aponta x
  jal fun        # 0x0400
  nop
  sw  v0, 0(s0)  # 0x0408, s0 aponta alpha
```

```
extern int foo(int);
extern int bar(int);
extern int cat(int);
extern int rat(int);
int fun(int);
extern int alfa, beta, gama;

int main(void) {
  int x,y,z;
  ...
  alfa = fun(x); // 0x0400
  ...
  beta = foo(z); // 0x1000
  ...
  y = bar(alfa); // 0x2400
  ...
  return(z);     // 0x24fc
}

int fun(int) {
   int x,y,z;
   ...
   gama = cat(x); // 0x3000
   ...
   beta = rat(y); // 0x3800
   ...
   z = foo(bar(alfa)); //0x4000
   ...
   return(y);     // 0x4400
}
```

```
// libfoobar
.org  0x1000 # ender = 0x8.1000
foo: ...
...
.org  0x2000 # ender = 0x8.2000
bar: ...
...
.data
.org  0x3000 # ender = 0x8.3000
alfa: .space 4,0 # int
beta: .space 4,0 # int
.end libfoobar
//-----------------------------

// libcatrat
.text
.org  0x2000 # ender = 0x9.2000
cat: ...
...
.org  0x3000 # ender = 0x9.3000
rat: ...
...
.data
.org  0x4000 # ender = 0x9.4000
gama: .space 4,0 # int
.end libcatrat
//-----------------------------
```

Capítulo 15

Bibliotecas Compartilhadas Dinâmicas

Hell is empty and all the devils are here.
William Shakespeare, The Tempest, I,2.

A ligação dinâmica posterga o processo de ligação até que um processo inicie sua execução. Comparada com a ligação estática, as vantagens da ligação dinâmica incluem:

1. a criação de Bibliotecas Compartilhadas Dinâmicas (BCDs) é mais simples do que a criação de bibliotecas compartilhadas estáticas (BCEs);
2. a atualização das BCDs é mais simples que a de BCEs – os endereços das BCDs não são fixados em tempo de instalação do sistema operacional; e
3. a semântica de BCDs é mais próxima daquela de bibliotecas não-compartilhadas (*archives*).

Os custos em termos de tempo de execução são substancialmente maiores do que na ligação estática: a cada vez em que um processo inicia, o esforço de ligação deve ser repetido para todas as bibliotecas de que o processo dependa – cada símbolo externo referenciado deve ser buscado numa tabela de símbolos e então resolvido.

15.1 Arquivos ELF com BCDs

Um arquivo ELF pode ser encarado como um conjunto de seções interpretadas pelo ligador, ou como um conjunto de segmentos interpretados pelo carregador. Executáveis e BCDs tem a mesma estrutura mas com distintos conjuntos de segmentos e seções. Tipicamente uma biblioteca compartilhada dinâmica tem dois segmentos, um de texto (RO) e um de dados (RW), conforme mostra o diagrama na Figura 15.1. Além dos segmentos de texto e dados, BCDs tem um "segmento dinâmico" que é usado para efetuar a ligação dinâmica. Este segmento dinâmico é um conjunto de seções que inclui `.dynamic`, `.liblist`, `.rel.dyn`, `.dynstr`, `.dynsym`, e uma tabela *hash* para acelerar a resolução das referências.

Uma biblioteca dinâmica pode ser carregada em qualquer endereço, e num dado instante uma mesma biblioteca pode estar carregada em endereços distintos, potencialmente numa faixa de endereços diferente em cada processo a que está ligada. Portanto, o código de BCDs é gerado para ser independente de posição e com uma *Global Offset Table* (GOT) com apontadores para todos os dados estáticos referenciados na biblioteca. A seção `.plt` (*Procedure Linkage Table*) contém apontadores para as funções exportadas pela biblioteca e tem função similar à GOT. BCDs para processadores MIPS não contém a seção `.plt`.

BCDs ELF contém toda a informação necessária para o ligador relocar os segmentos da biblioteca e resolver os símbolos indefinidos. A seção `.dynsym` (tabela de símbolos dinâmica)

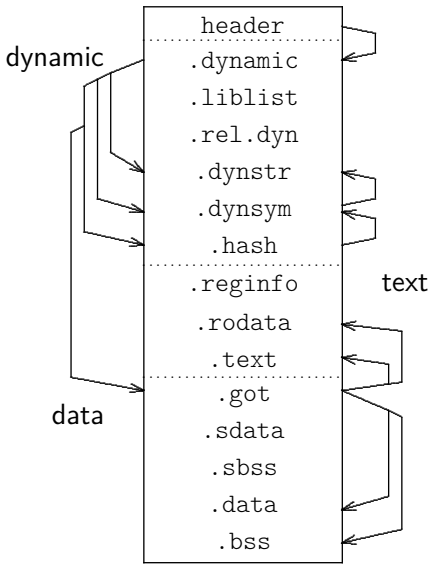


Figura 15.1: Biblioteca compartilhada dinâmica ELF para o MIPS.

contém os símbolos do arquivo que são exportados e importados. As seções `.dynstr` e `.hash` mantém os 'nomes' dos símbolos e uma tabela *hash* para acelerar sua resolução.

A seção `.dynamic` contém uma lista de tuplas ⟨*tipo, valor, apontador*⟩ para que o ligador possa resolver os símbolos indefinidos com base nos elementos desta lista. Alguns dos tipos ocorrem só em executáveis (EXs), outros só em BCDs e outros em ambos, e uma lista parcial é mostrada abaixo.

`NEEDED` nome de uma BCD de quem o objeto depende; ocorre em executáveis e em BCDs;

`SYMTAB, STRTAB, HASH, SYMENT, STRSZ` apontam para a tabela de símbolos e para as tabelas de *strings* e *hash* associadas, o tamanho de um elemento da TS, e o número de *strings* (em EXs e BCDs);

`PLTGOT` aponta para a *Global Offset Table* (GOT, em EXs e BCDs);

`REL, RELSZ, RELENT` apontam para a tabela de relocações, o tamanho e o número de elementos na tabela (em EXs e BCDs);

`INIT, FINI` apontam as rotinas de inicialização e finalização de programas escritos em C++ (em EXs e BCDs);

`INTERP` um executável contém uma seção `.interp` (interpretador) no início do arquivo para especificar o nome do ligador dinâmico, que usualmente é `ld.so`.

Um programa contido num *executável ELF* é similar ao mostrado na Figura 15.1, mas contém seções `.init` e `.fini` no segmento de texto, e uma seção `.interp` que aponta para o ligador dinâmico (`ld.so`) que deve ser invocado antes da execução. Um programa num arquivo ELF executável não contém uma seção `.got` porque ele não é relocado depois da ligação.

15.2 Carga de um programa ligado dinamicamente

Quando carrega um programa para execução, o SO cria seu espaço de endereçamento (EdE), e se existe uma seção `.interp`, o SO carrega o ligador dinâmico (`ld.so`) numa região adequada do EdE, e insere na pilha do novo processo um *vetor auxiliar* com informações necessárias ao ligador. O vetor auxiliar contém:

`AT_PHDR`, `AT_PHENT`, `AT_PHNUM` endereço do cabeçalho ELF do programa, o tamanho e o número de elementos do cabeçalho. Estes descrevem os segmentos do arquivo carregado;

`AT_ENTRY` endereço inicial do programa, para onde o controle é passado depois que o ligador terminar a inicialização; e

`AT_BASE` endereço em que o ligador está carregado.

O próprio ligador dinâmico é mantido como uma biblioteca dinâmica porque ele é carregado toda vez que um novo processo inicia. Além disso, o ligador deve ser capaz de relocar a si próprio antes de iniciar a ligação do executável com biblioteca(s), se esse for o caso.

Ao iniciar, o primeiro elemento da GOT do ligador aponta para a sua seção `.dynamic` e com base no conteúdo desta seção o ligador pode ajustar-se ao endereço em que foi carregado – autorelocação de texto e dados – e então iniciar a ligação do executável.

O ligador cria uma lista de tabelas de símbolos, com apontadores para a tabela de símbolos (TS) do executável, da(s) biblioteca(s) a que é ligado, e uma TS global que será usada pelo ligador. Ao invés de criar uma nova tabela de símbolos, com todos os símbolos de todos os arquivos objeto participantes da ligação, esta lista de tabelas é suficiente porque cada arquivo contém sua própria tabela *hash* para acelerar a resolução de símbolos.

15.2.1 Lista de bibliotecas necessárias

A seção `DT_NEEDED` contém descritores das bibliotecas que devem ser carregadas e incorporadas ao executável. O ligador procura nos lugares habituais (`/lib`, `/usr/lib`, `/usr/local/lib`) ou em lugares especificados pelo programador – a seção `.dynamic` pode conter um elemento `DT_PATH` com caminhos, ou a variável de ambiente `LD_LIBRARY_PATH` contém uma lista de caminhos. Após encontrar um arquivo indicado em `DT_NEEDED`, o ligador lê seu cabeçalho, aloca espaço para seu texto, dados e BSS, liga a TS desta biblioteca à lista de TSs, e se for o caso, incorpora outras bibliotecas das quais ela dependa.

Isso feito, o ligador percorre cada biblioteca efetuando relocação, preenchendo a GOT da biblioteca e ajustando os endereços dos dados para acesso indireto através da GOT.

Se a biblioteca contém uma seção `.init`, esta é preparada para executar antes do que qualquer função do executável; se contém uma seção `.fini`, esta é marcada para executar na finalização do processo. Quando completa estas tarefas, o ligador salta para o ponto de entrada do programa na função `_start()`, para que todos os `.init` executem, e então `main()` inicie.

15.2.2 Ligação de funções

Bibliotecas compartilhadas dinâmicas ELF são geradas com código independente de posição para que possam ser carregadas em qualquer endereço. Isso permite que o mesmo código seja carregado em endereços distintos nos vários processos que usam a BCD simultaneamente. *Código independente de posição* (IdP) é escrito para usar uma *Global Offset Table* (GOT) com apontadores para todas as variáveis estáticas referenciadas pelo programa. O ligador resolve os símbolos e reloca a GOT de cada novo processo em tempo de execução.

A GOT fica no mesmo arquivo em que o código que a referencia, e portanto as distâncias relativas não se alteram, independentemente do endereço de carga. O endereço da GOT é carregado no registrador *global pointer* (`gp`) e todas as referências indiretas à GOT são

baseadas no conteúdo deste registrador. Uma vez que o `gp` é atualizado, todas as variáveis estão automaticamente relocadas.

As BCDs do MIPS não usam a seção `.plt` (*Procedure Linkage Table*). Por isso a resolução dos símbolos de código também ocorre através da GOT. Nos símbolos de código marcados como indefinidos, com tipo `STT_FUNC`, o ligador efetua a relocação do endereço, adicionando ao endereço do *stub* da função a distância entre o endereço virtual no arquivo e o endereço virtual de carga.

Nem todas as funções presentes na BCD são de fato invocadas. Por causa disso, o ligador resolve os símbolos de código em tempo de execução, e somente para funções que são de fato invocadas – isso é chamado de *avaliação preguiçosa* (*lazy evaluation*).

Quando a biblioteca é criada, e contém uma certa função `f()`, uma versão falsa `stub_f()` é gerada e esta é ligada ao executável, enquanto que o endereço de `stub_f()` é colocado na GOT do executável. Na primeira vez em que a função é invocada, indiretamente através da GOT, `stub_f()` é executada. O código no *stub* invoca o ligador dinâmico, que resolve o endereço real do símbolo `f()`, atualiza a GOT com o endereço do símbolo (sobrescreve o endereço do *stub*), e salta para o código de `f()`. Quando esta função termina, do código do *stub* o controle retorna para o ponto em que `f()` foi originalmente invocada no executável. Na próxima invocação, o endereço na GOT já é o endereço real da função e nenhum trabalho adicional é necessário. O Programa 15.1 mostra o *stub* da função `f()` e seu uso é discutido a seguir.

Programa 15.1: *Stub* da função `f()`.

```
1 stub_f:
2     lw   t9, 0(gp)  # ender do ligador na 1a posição da GOT
3     move t7, ra     # ender de retorno será alterado pelo jalr
4     li   t8, .dynsym_indx_f  # índice de f() na seção .dynamic
5     jalr t9         # salta para ld.so, altera ender de retorno
6     nop
```

A primeira posição na GOT da biblioteca *sempre* contém o endereço do ligador dinâmico. Por convenção, o registrador `t8` deve ser carregado com o índice do símbolo `f()` na seção dinâmica (linha 4), e o endereço de retorno de `f()` é salvado em `t7`. Isso é necessário porque o salto para o ligador dinâmico sobrescreve o endereço de retorno em `ra`. O ligador dinâmico resolve o endereço de `f()` e atualiza a posição na GOT que apontava para `stub_f()`. Finda a ligação, o ligador retorna para o endereço salvado em `t7`.

Resumindo: a GOT é gerada com os endereços dos *stubs*. Na primeira vez em que uma função é invocada, o código no *stub* invoca o ligador dinâmico, que sobrescreve o elemento na GOT com o endereço real da função, e então salta para a função. Nas próximas invocações, a GOT aponta para o endereço real da função.

Como exemplo, considere um programa que contém chamadas para três funções de biblioteca `f()`, `g()` e `h()`, e que durante a execução somente as funções `f()` e `h()` são de fato invocadas. O Programa 15.2 contém um fragmento de código que contém apenas os trechos em que as funções são invocadas indiretamente através da GOT.

Programa 15.2: Exemplo de ligação dinâmica com três funções.

```
1  # no programa em C: f();
2  lw t1, gotOff_f(gp) # gotOff_f = índice do ender de f() na GOT
3  nop
4  jalr t1
5  ...
6  # no programa em C: g();
7  lw t5, gotOff_g(gp) # gotOff_g = índice do ender de g() na GOT
8  nop
9  jalr t5
10 ...
11 # no programa em C: h();
12 lw t3, gotOff_h(gp) # gotOff_h = índice do ender de h() na GOT
13 nop
14 jalr t3
15 ...
```

O executável é ligado com os *stubs* de `f()`, `g()` e `h()`, como mostra o Programa 15.3.

Programa 15.3: *Stubs* das funções `f()`, `g()` e `h()`.

```
1  stub_f: lw t9, 0(gp)          # ender inicial do ligador na GOT
2          move t7, ra           # salva ender de retorno
3          li t8, .dynsym_idx_f  # índice de f() na seção .dynamic
4          jalr t9               # salta para ligador dinâmico
5          nop                   # branch delay slot
6
7  stub_g: lw t9, 0(gp)
8          move t7, ra
9          li t8, .dynsym_idx_g  # índice de g() na seção .dynamic
10         jalr t9
11         nop
12
13 stub_h: lw t9, 0(gp)
14         move t7, ra
15         li t8, .dynsym_idx_h  # índice de h() na seção .dynamic
16         jalr t9
17         nop
```

A GOT do executável é inicializada com os endereços absolutos dos *stubs*, como mostra o Programa 15.4.

Programa 15.4: Estado inicial da GOT.

```
1          .got       # GOT no segmento de dados, com permissão RW
2  ld_so:  ld_exec    # endereço do ligador dinâmico
3  ...     ...
4  f:      stub_f     # stub de f()
5  g:      stub_g     # stub de g()
6  h:      stub_h     # stub de h()
7  ...     ...
```

Supondo que a função `h()` seja a primeira função da biblioteca que é invocada, quando o controle retornar ao ponto de invocação, a GOT estará no estado mostrado no Programa 15.5.

Programa 15.5: GOT após invocação de `h()`.

```
1           .got          # GOT no segmento de dados, com permissão RW
2  ...      ...
3  f:       stub_f        # stub de f()
4  g:       stub_g        # stub de g()
5  h:       0x4000.f8c0   # endereço absoluto de h()
6  ...      ...
```

Ao retornar da invocação inicial de `f()`, a GOT estará no estado mostrado no Programa 15.6.

Programa 15.6: GOT após invocação de `h()` e de `g()`.

```
1           .got          # GOT no segmento de dados, com permissão RW
2  ...      ...
3  f:       0x4000.c44C   # endereço absoluto de f()
4  g:       stub_g        # stub de g(), símbolo por resolver
5  h:       0x4000.f8c0   # endereço absoluto de h()
6  ...      ...
```

Quando as funções `f()` e `h()` forem invocadas novamente, serão usados os endereços absolutos destas funções. Enquanto `g()` não for invocada, seu endereço permanece indefinido, o que evita o custo da resolução desnecessária do símbolo.

15.2.3 Desempenho da ligação dinâmica

A ligação dinâmica pode provocar perdas de desempenho por ser lenta. A lentidão advém da combinação de, ao menos, cinco fatores: (i) relocação de bibliotecas em tempo de execução; (ii) resolução de símbolos em tempo de execução, tanto os símbolos dos executáveis quanto das bibliotecas; (iii) instruções adicionais no código IdP na invocação de funções; (iv) referências adicionais no acesso aos dados do código IdP; e (v) uso de registradores como apontadores para a GOT e no prólogo das funções IdP. Isso não é muito relevante para processadores como o MIPS com seus 32 registradores, mas acarreta consequências sérias com o x86 porque somente 6 registradores estão disponíveis ao compilador.

Em sistemas modernos com espaços de endereçamento grandes, é razoável usar uma mistura da ligação dinâmica com a ligação estática. Uma região do espaço de endereçamento é reservada para um conjunto popular de bibliotecas, que são relocadas para endereços nesta faixa. Quando um processo inicia, se é possível ligá-lo à biblioteca no endereço fixo, mantendo-a na faixa de endereços pré-alocada, a biblioteca é carregada e ligada, *sem que seja necessária a sua relocação*. Se, para algum processo não é possível usar aquela faixa de endereços, então a biblioteca é relocada normalmente como na ligação dinâmica e então ligada ao processo.

Por outro lado, com ligação estática pura, se uma determinada biblioteca não pode ser mapeada no espaço de endereçamento de um processo, porque sua faixa de endereços já está ocupada, então o processo deve ser abortado. Bibliotecas estáticas não são escritas para ser relocadas.

Espaço em branco proposital.

Exemplo 15.1 Os Programas 15.7 e 15.8 mostram o estado da GOT na primeira e na segunda invocações da função `h()`. Suponha que esta função seja o oitavo elemento da GOT. Na primeira invocação, a chamada indireta de função (`jalr` na linha 4) resulta na execução do `stub_h`, buscado em `GOT[8]`. O *stub* causa a invocação do ligador dinâmico, que resolve o endereço absoluto de `h()`, e atualiza a GOT.

O Programa 15.8 mostra o estado da GOT imediatamente após a primeira invocação. O ligador dinâmico atualizou `GOT[8]` com o endereço absoluto da função `h()`. Na próxima invocação, o `jalr` da linha 4 efetuará um salto para o endereço da primeira instrução de `h()`. ◁

Programa 15.7: Ligação dinâmica da função `h()` -- `primeira invocação`.

```
1           # no programa em C: h();
2           lw t3, 8*4(gp)  # gotOff_h = 8*4, GOT[8]=h()
3           nop
4           jalr t3
5           ...
6
7   stub_h: lw t9, 0(gp)
8           move t7, ra
9           li t8, .dynsym_idx_h # índice de h() na seção .dynamic
10          jalr t9
11          nop
12          ...
13
14          .got
15  ld_so:  ld_exec   # GOT[0] endereço do ligador dinâmico
16  ...     ...
17  h:      stub_h    # GOT[8] aponta para stub de h()
```

Programa 15.8: Ligação dinâmica da função `h()` -- `segunda invocação`.

```
1           # no programa em C: h();
2           lw t3, 8*4(gp)  # gotOff_h = 8*4, GOT[8]=h()
3           nop
4           jalr t3
5           ...
6
7           .got
8   ld_so:  ld_exec      # GOT[0] endereço do ligador dinâmico
9   ...     ...
10  h:      0x4000.fc80  # GOT[8] endereço absoluto de h()
```

Espaço em branco proposital.

15.3 Exercícios

Ex. 82 Usando o código abaixo e as duas bibliotecas, simule a compilação das bibliotecas e efetue sua ligação dinâmica com o programa. Preencha as estruturas de dados com os valores indicados no código. O código segue a convenção: (i) a definição de um símbolo é denotada nos diagramas por um *label* como 'simb:'; (ii) uma referência a uma função é denotada pelos parênteses como 'fun()'; (iii) uma referência a uma variável é denotada pelo seu nome como 'var'.

No código C, o endereço nos comentários é o endereço da instrução que invoca a função. Para simplificar sua resposta, variáveis atribuídas são referenciadas em endereço que é 4 bytes maior que o da instrução jal, enquanto que variáveis lidas são referenciadas num endereço que é 4 bytes menor do que o da instrução jal: em main(), alpha é referenciada em 0x0404 e x em 0x03fc.

```
extern int foo(int);
extern int bar(int);
extern int cat(int);
extern int rat(int);
int fun(int);
extern int alfa;
extern int beta;
extern int gama;

int main(void) {
  int x,y,z;
  ...
  alfa = fun(x); // 0x0400
  ...
  beta = foo(z); // 0x1000
  ...
  y = bar(alfa); // 0x2400
  ...
  return(z);     // 0x24fc
}

int fun(int) {
   int x,y,z;
   ...
   gama = cat(x); // 0x3000
   ...
   beta = rat(y); // 0x3800
   ...
   z = foo(bar(alfa)); //0x4000
   ...
   return(y);     // 0x4400
}
```

```
// libfoobar
.org  0x1000 # ender = 0x8.1000
.text
foo: ...
...
.org  0x2000 # ender = 0x8.2000
.text
bar: ...
...
.org  0x3000 # ender = 0x8.3000
.data
alfa: .space 4,0 # int
beta: .space 4,0 # int
.end libfoobar
//------------------------------

// libcatrat
.org  0x2000 # ender = 0x9.2000
.text
cat: ...
...
.org  0x3000 # ender = 0x9.3000
.text
rat: ...
...
.org  0x4000 # ender = 0x9.4000
.data
gama: .space 4,0 # int
.end libcatrat
//------------------------------
```

Índice Remissivo