Princípios Básicos de

Arquitetura *e* Organização *de* Computadores

Segunda Edição

Linda Null *e* Julia Lobur

bookman

N969p Null, Linda.
Princípios básicos de arquitetura e organização de computadores / Linda Null, Julia Lobur ; tradução: Maria Lúcia Blanck Lisbôa ; revisão técnica: Carlos Arthur Lang Lisbôa. – 2. ed. – Porto Alegre : Bookman, 2010.
822 p. : il. color. ; 25 cm.

ISBN 978-85-7780-737-6

1. Ciência da computação. 2. Arquitetura dos computadores. 3. Organização de computadores. I. Lobur, Julia. II. Título.

CDU 004.2

Catalogação na publicação: Ana Paula M. Magnus – CRB-10/Prov-009/10

um byte de endereço de numeração par). Isto desperdiça espaço. Arquiteturas *little endian*, tal como a Intel, permitem leituras e escritas em endereços ímpares, o que torna a programação destas máquinas muito mais fácil. Se um programador escreve uma instrução para ler um valor de tamanho errado de palavra, em uma máquina *big endian* ele é sempre lido como um valor incorreto; em uma máquina *little endian*, isto pode algumas vezes resultar em uma leitura correta do dado. (Note que a Intel finalmente adicionou uma instrução para reverter a ordem dos bytes dentro de registradores.)

Redes de computadores são *big endian*, o que significa que quando computadores *little endian* vão enviar inteiros através da rede (endereços de dispositivos de rede, por exemplo), eles necessitam convertê-los para a ordem de bytes da rede. Do mesmo modo, quando recebem valores inteiros da rede, eles precisam convertê-los de volta para a sua representação interna.

Embora você possa não estar familiarizado com o debate *little versus big endian*, ele é importante para muitas aplicações de software atuais. Qualquer programa que escreva dados ou leia dados de um arquivo deve estar ciente da ordenação de bytes da máquina em particular. Por exemplo, o formato gráfico Windows BMP foi desenvolvido em uma máquina *little endian*, de modo que para ver BMPs em uma máquina *big endian*, a aplicação usada deve primeiro reverter a ordem dos bytes. Projetistas de software de softwares populares estão bem conscientes destas questões de ordenamento de bytes. Por exemplo, Adobe Photoshop usa *big endian*, GIF é *little endian*, JPEG é *big endian*, MacPaint é *big endian*, PC Paintbrush é *little endian*, RTF da Microsoft é *little endian* e arquivos de rastreio da Sun são *big endian*. Algumas aplicações suportam ambos formatos: Microsoft WAV e arquivos AVI, arquivos TIF e XWD (X windows Dump) suportam ambos, geralmente codificando um identificador no arquivo.

5.2.3 Armazenamento interno na UCP: pilhas *versus* registradores

Uma vez determinado o ordenamento de bytes na memória, o projetista de hardware deve tomar algumas decisões sobre como a UCP deve armazenar dados. Este é o meio mais básico de diferenciar ISAs. Existem três alternativas:

1. Uma arquitetura de pilha
2. Uma arquitetura de acumulador
3. Uma arquitetura de registrador de propósito geral (GPR – *general-purpose register*)

Arquiteturas de pilha usam uma pilha para executar instruções, e os operandos são (implicitamente) encontrados no topo da pilha. Mesmo que as máquinas baseadas em pilha tenham uma boa densidade de código e um modelo simples de avaliação de expressões, uma pilha não pode ser acessada randomicamente, o que dificulta a geração de código eficiente. Além disso, a pilha se torna um gargalo durante a execução. **Arquiteturas de acumulador**, como MARIE, com um operando implicitamente no acumulador, minimizam a complexidade interna da máquina e permitem instruções bem curtas. Mas, visto que o acumulador é somente uma memória temporária, o tráfego na memória é muito alto. **Arquiteturas de registrador de propósito geral**, que usam conjuntos de registradores de propósito geral, são os modelos amplamente aceitos para

as arquiteturas de máquinas atuais. Estes conjuntos de registradores são mais rápidos do que a memória, fáceis de lidar para compiladores e podem ser usados de forma muito eficiente e eficaz. Além disso, os preços de hardware diminuíram significativamente, tornando possível adicionar um grande número de registradores a um custo mínimo. Se o acesso à memória é rápido, um projeto baseado em pilha pode ser uma boa ideia; se a memória é lenta, muitas vezes é melhor usar registradores. Estas são razões pelas quais a maioria dos computadores da última década têm sido baseados em registradores. Entretanto, visto que todos os operandos devem ser indicados, usar registradores resulta em instruções mais longas, causando tempos de carga e de decodificação maiores. (Um objetivo muito importante para projetistas de ISAs são instruções curtas.) Projetistas que escolhem uma ISA devem decidir qual irá funcionar melhor em um ambiente em particular e examinar cuidadosamente os prós e os contras.

A arquitetura de propósito geral pode ser repartida em três classificações, dependendo de onde os operandos estão localizados. Arquiteturas **memória-memória** podem ter dois ou três operandos na memória, permitindo a uma instrução realizar uma operação sem requerer que qualquer operando esteja em um registrador. Arquiteturas **registrador-memória** requerem um mix, onde pelo menos um operando esteja em um registrador e um esteja na memória. Arquiteturas **carrega-armazena** requerem que os dados sejam movidos para os registradores antes que quaisquer operações sobre estes dados sejam realizadas. Intel e Motorola são exemplos de arquiteturas registrador-memória; a arquitetura VAX da Digital Equipment permite operações memória-memória; e SPARC, MIPS, Alpha e PowerPC são máquinas carrega-armazena.

Dado que a maioria das arquiteturas atuais são baseadas em GPR, vamos examinar agora duas das principais características de conjuntos de instruções que dividem as arquiteturas de registradores de propósito geral. Estas duas características são o número de operandos e como os operandos são endereçados. Na Seção 5.2.4 vamos examinar o tamanho da instrução e o número de operandos que uma instrução pode ter. (Dois ou três operandos são os mais comuns para arquiteturas GPR, e vamos comparar estas com arquiteturas de zero e um operando.) Depois, vamos investigar tipos de instrução. Finalmente, na Seção 5.4, investigaremos os diversos modos de endereçamento disponíveis.

5.2.4 Número de operandos e tamanho da instrução

O método tradicional para descrever uma arquitetura de computador é especificar o número máximo de operandos ou endereços contidos em cada instrução. Isto tem um impacto direto no próprio tamanho da instrução. MARIE usa uma instrução de tamanho fixo, com um *opcode* de 4 bits e um operando de 12 bits. Instruções em arquiteturas atuais podem ser formatadas de duas maneiras:

- **Tamanho fixo** – Desperdiça espaço, mas é rápida e resulta numa melhor performance quando é usado pipeline em nível de instrução, como veremos na Seção 5.5.
- **Tamanho variável** – Mais complexa de decodificar, mas economiza espaço.

Geralmente o compromisso com a vida real envolve usar dois a três tamanhos de instrução, o que fornece padrões de bits que são facilmente distinguíveis e simples de decodificar. O tamanho da instrução também deve ser comparado com o tamanho da palavra da máquina. Se o tamanho da instrução é exatamente igual ao tamanho da palavra, as ins-

truções se alinham perfeitamente quando armazenadas na memória principal. Instruções sempre necessitam ser alinhadas em palavras por razões de endereçamento. Portanto, instruções que são metade, um quarto, o dobro ou o triplo do tamanho real da palavra podem desperdiçar espaço. Instruções de tamanho variável claramente não são do mesmo tamanho e precisam ser alinhadas por palavra, resultando também em perda de espaço.

Os formatos de instruções mais comuns incluem zero, um, dois ou três operandos. Vimos no Capítulo 4 que algumas instruções de MARIE não possuem operandos, enquanto outras possuem um operando. Operações aritméticas e lógicas geralmente têm dois operandos, mas podem ser executadas com um operando (como vimos em MARIE) se o acumulador for implícito. Podemos estender esta ideia para três operandos se considerarmos o destino final como um terceiro operando. Também podemos usar uma pilha que nos permita ter instruções com zero operandos. Os seguintes formatos de instruções são alguns dos mais comuns:

- ***OPCODE* somente** (zero endereços)
- ***OPCODE* + 1 Endereço** (geralmente um endereço de memória)
- ***OPCODE* + 2 Endereços** (geralmente registradores ou um registrador e um endereço de memória)
- ***OPCODE* + 3 Endereços** (geralmente registradores ou combinações de registradores e memória)

Todas as arquiteturas possuem um limite no número máximo de operandos permitidos por instrução. Por exemplo, em MARIE o máximo era um, embora algumas instruções não possuam operandos (`Halt` e `Skipcond`). Mencionamos que instruções de zero, um, dois ou três operandos são as mais comuns. Instruções de zero, um, dois ou três operandos são razoavelmente fáceis de entender; uma ISA completa construída com instruções com zero operandos pode, em princípio, ser um tanto confusa.

Instruções de máquina que não possuem operandos devem usar uma pilha (uma estrutura de dados último a entrar, primeiro a sair, introduzida no Capítulo 4 e descrita em detalhes no Apêndice A, na qual todas as inserções e retiradas são feitas no topo) para realizar aquelas operações que logicamente requerem um ou dois operandos (tal como um `Add`). Em vez de usar registradores de propósito geral, uma arquitetura baseada em pilha armazena os operandos no topo da pilha, tornando o elemento do topo acessível à UCP. (Note que a mais importante estrutura de dados nas arquiteturas de máquinas é a pilha. Esta estrutura não fornece apenas um meio eficiente de armazenar valores de dados intermediários durante cálculos complexos, mas também um método eficiente para passagem de parâmetros durante a chamada de procedimentos, bem como serve como um meio de salvar a estrutura de bloco local e definir o escopo de variáveis e sub-rotinas.)

Em arquiteturas baseadas em pilhas, a maioria das instruções consiste somente em *opcodes*; entretanto, existem instruções especiais (aquelas que adicionam elementos a e removem elementos de uma pilha) que têm apenas um operando. As arquiteturas de pilha necessitam uma instrução para inserir e uma instrução para retirar, sendo a cada uma destas permitido um operando. `Push X` coloca na pilha o valor do dado encontrado na posição de memória *X*; `Pop X` remove o elemento do topo da pilha e o armazena na posição *X*. Somente certas instruções têm permissão para acessar a memória; todas as outras devem usar a pilha para qualquer operando requerido durante a execução.

Para operações que requerem dois operandos, os dois elementos do topo da pilha são usados. Por exemplo, se executamos uma instrução `Add`, a UCP soma os dois elementos do topo da pilha, retirando ambos e depois colocando a soma no topo da pilha. Para operações não comutativas tal como a subtração, o elemento do topo da pilha é subtraído do elemento próximo ao topo, sendo ambos retirados, e o resultado é colocado no topo da pilha.

Esta organização de pilha é muito eficaz para avaliação de expressões aritméticas longas escritas em **notação polonesa reversa (RPN – *reverse Polish notation*)**. Esta representação coloca o operador após os operandos, sendo conhecida como **notação pós-fixada** (em comparação com a **notação infixada**, que coloca o operador entre os operandos e a **notação pré-fixada**, que coloca o operador antes dos operandos). Por exemplo:

$$X + Y \text{ está na notação infixada}$$
$$+ \, X \, Y \text{ está na notação pré-fixada}$$
$$X \, Y \, + \text{ está na notação pós-fixada}$$

Todas as expressões aritméticas podem ser escritas usando-se qualquer destas representações. Entretanto, a representação pós-fixada combinada com uma pilha de registradores é o meio mais eficiente de avaliar expressões aritméticas. De fato, algumas calculadoras eletrônicas (tais como as da Hewlett-Packard) exigem que o usuário entre com expressões na notação pós-fixada. Com uma pequena prática nestas calculadoras, é possível avaliar rapidamente expressões longas contendo muitos parênteses aninhados sem nunca parar para pensar como os termos são agrupados.

Considere a seguinte expressão:

$$(X + Y) \times (W - Z) + 2$$

Escrita em RPN, isto se torna:

$$XY + WZ - \times 2 +$$

Note que a necessidade de parênteses para preservar a precedência é eliminada quando se usa RPN.

Para ilustrar o conceito de zero, um, dois e três operandos, vamos escrever um programa simples para avaliar uma expressão aritmética, usando cada um destes formatos.

Exemplo 5.1 Suponha que queremos avaliar a seguinte expressão:

$$Z = (X \times Y) + (W \times U)$$

Usualmente, quando três operandos são permitidos, pelo menos um operando deve ser um registrador, e o primeiro operando normalmente é o destino. Usando instruções de três endereços, o código para avaliar a expressão para Z é escrito como segue:

```
Mult  R1, X,  Y
Mult  R2, W,  U
Add   Z,  R2, R1
```

Quando são usadas instruções de dois endereços, normalmente um endereço especifica um registrador (instruções de dois endereços raramente permitem que

ambos os operandos sejam endereços de memória). O outro operando pode ser um registrador ou um endereço de memória. Usando instruções de dois endereços, nosso código se torna:

```
Load   R1, X
Mult   R1, Y
Load   R2, W
Mult   R2, U
Add    R1, R2
Store  Z,  R1
```

Note que é importante saber se o primeiro operando é a origem ou o destino. Nas instruções acima, assumimos que é o destino. (Isto tende a ser um ponto de confusão para aqueles programadores que devem alternar entre a linguagem simbólica da Intel e a linguagem simbólica da Motorola – a simbólica da Intel especifica o primeiro operando como destino, enquanto na simbólica da Motorola o primeiro operando é a origem.)

Usando instruções de um endereço (como em MARIE), devemos considerar um registrador (normalmente o acumulador) implícito como sendo o destino do resultado da instrução. Para avaliar *Z*, nosso código se torna:

```
Load   X
Mult   Y
Store  Temp
Load   W
Mult   U
Add    Temp
Store  Z
```

Note que à medida que reduzimos o número de operandos permitidos por instrução, aumenta o número de instruções requeridas para executar o código desejado. Isto é um exemplo de uma típica troca no projeto da arquitetura – instruções mais curtas, mas programas mais longos.

Como será que este programa fica em uma máquina baseada em pilha com instruções de zero endereços? Arquiteturas baseadas em pilha não usam operandos para instruções tais como `Add`, `Subt`, `Mult` ou `Divide`. Necessitamos de uma pilha e de duas operações sobre esta pilha: `Pop` e `Push`. Operações que se comunicam com a pilha devem ter um campo de endereço para especificar o operando a ser retirado ou inserido na pilha (todas as outras operações não possuem endereço). `Push` coloca o operando no topo da pilha. `Pop` remove o topo da pilha e o coloca no operando. Esta arquitetura resulta no maior programa para avaliar a nossa equação. Assumindo que operações aritméticas que usam dois operandos no topo da pilha retiram os operandos e inserem o resultado da operação, nosso código é como segue:

```
Push  X
Push  Y
Mult
```

```
Push  W
Push  U
Mult
Add
Pop   Z
```

O tamanho de instrução é certamente afetado pelo tamanho do *opcode* e pelo número de operandos permitidos na instrução. Se o tamanho do *opcode* é fixo, decodificar é muito fácil. Entretanto, para fornecer flexibilidade e compatibilidade com versões anteriores, *opcodes* podem ter tamanho variável. *Opcodes* de tamanho variável apresentam os mesmos problemas que instruções de tamanho variável *versus* fixo. Um ajuste usado por muitos projetistas são os *opcodes* expandidos.

5.2.5 *Opcodes* expandidos

***Opcodes* expandidos** representam um ajuste entre a necessidade de um conjunto rico de *opcodes* e o desejo de ter *opcodes* curtos e, assim, instruções curtas. A ideia é fazer alguns *opcodes* curtos, mas ter um meio de prover outros, mais longos, quando necessário. Quando o *opcode* é curto, vários bits são reservados para conter os operandos (o que significa que podemos ter dois ou três operandos por instrução). Quando você não necessita de espaço para operandos (para uma instrução tal como `Halt` ou porque a máquina usa uma pilha), todos os bits podem ser usados para o *opcode*, o que permite muitas instruções distintas. Entre estes limites, existem *opcodes* mais longos com menos operandos, bem como *opcodes* mais curtos com mais operandos.

Considere uma máquina com instruções de 16 bits e 16 registradores. Visto que agora temos um conjunto de registradores em vez de um único acumulador (como em MARIE), precisamos usar 4 bits para especificar um determinado registrador. Poderíamos codificar 16 instruções, cada uma com três registradores de operandos (o que implica que qualquer dado a ser operado precisa primeiro ser carregado em um registrador) ou usar 4 bits para o *opcode* e 12 bits para um endereço de memória (como em MARIE, considerando uma memória de tamanho 4K). Qualquer referência à memória requer 12 bits, deixando somente 4 bits para outras finalidades. Entretanto, se todo o dado na memória é primeiro carregado em um registrador deste conjunto de registradores, a instrução pode selecionar um elemento particular de dado usando somente 4 bits (considerando 16 registradores). Estas duas opções são ilustradas na Figura 5.2.

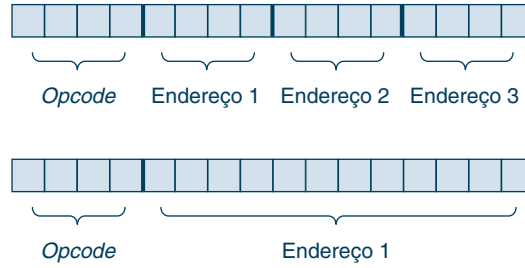

FIGURA 5.2 Duas possibilidades para um formato de instrução de 16 bits.