Arquitetura de Computadores

Arquitetura do Conjunto de Instruções

Marcelo Luiz Silva

2.1 - Introdução

Este capítulo refere-se a *Arquitetura do Conjunto de Instruções*, que é a parte da máquina visível ao programador ou ao implementador de um compilador. Serão apresentadas algumas alternativas de projetos para a arquitetura do conjunto de instruções. Particularmente, serão estudados os seguintes tópicos:

1. será apresentada a taxinomia das alternativas do conjunto de instruções, e uma avaliação qualitativa das vantagens e desvantagens das várias abordagens;
2. serão apresentadas, e analisadas, algumas medidas do conjunto de instruções, que são amplamente independentes de um conjunto de instruções específico;
3. será analisada as questões referentes a linguagens e compiladores, e seus comportamentos sobre a arquitetura do conjunto de instruções;
4. finalmente, será visto como estas idéias são refletidas no conjunto de instruções do DLX.

2.2 - Classificando as Arquiteturas do Conjunto de Instruções

A parte da arquitetura responsável pela diferenciação, entre outras arquiteturas, é o tipo de armazenamento interno da CPU. As classes consideradas são:

- *arquitetura de pilha*;
- *arquitetura de acumulador*;
- *arquitetura de registradores de propósito geral*.

Nestas arquiteturas, os operandos podes ser chamados *explicitamente* ou *implicitamente*, por exemplo:

- os operandos em uma *arquitetura de pilha* estão implicitamente no topo da pilha;
- em uma *arquitetura de acumulador*, um operando é implicitamente o acumulador;
- em uma *arquitetura de registradores de propósito geral* tem-se somente operandos explícitos, quer sejam locações de memórias ou de registradores. Os operandos explícitos, podem ser acessados diretamente a partir da memória, ou pode ser necessário carregá-los primeiro em uma memória temporária, dependendo da classe da instrução e da escolha da instrução específica.

A classe *arquitetura de registradores de propósito geral*, pode ser subdividida em:

- *arquitetura resgister-memory*: pode acessar a memória como parte de qualquer instrução;
- *arquitetura load-store* ou *arquitetura register-register*: pode acessar a memória somente com instruções de *load* e *store* .

Observação: existe uma terceira classe da arquitetura de registradores, a qual não é encontrada nas máquinas atuais, que é a *arquitetura memory-memory*, a qual mantém todos os operandos na memória.

EXEMPLO: como ficaria a seqüência de códigos $C = A + B$, nestas três classes de conjuntos de instruções:

Pilha	Acumulador	Registrador (register-memory)	Registrador (load-store)
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add, R3, R1, R2
Pop C			Store C, R3

Observação: assume-se que A, B, C, estejam na memória, e que os valores de A e B não podem ser destruídos.

Apesar do fato, das primeiras máquinas utilizarem arquiteturas de pilha ou de acumulador, virtualmente, toda máquina projetada depois de 1980 utiliza a arquitetura *load-store*. A principal razão para o aparecimento das máquinas de registradores de propósito geral (RPG) são:

- os registradores, como outras formas de armazenamento interno na CPU, são mais rápidos que a memória;
- registradores são facilmente utilizados pelo compilador, e podem ser utilizados mais efetivamente do que outras formas de armazenamento interno.

EXEMPLO: Seja a avaliação da expressão: (A*B) - (C*D) - (E*F)

Em uma máquina de registradores, as multiplicações podem ser feitas em qualquer ordem, o que acarreta eficiência, quer seja pela locação dos operandos, quer seja pela utilização de *pipeline*. Já em uma máquina de pilha a expressão deve ser avaliada da esquerda para a direita, a não ser que sejam utilizadas operações especiais ou trocas nas posições da pilha.

Outro fato importante é que os registradores podem ser utilizados para armazenar variáveis. Quando variáveis são alocadas para os registradores, tem-se a redução do tráfego de memória, o programa ganha velocidade (pois registradores são mais rápidos que memória), e o ocorre a melhoria na densidade de código (pois um registrador pode ser chamado com menos *bits* que uma locação de memória). Implementadores de compiladores preferem que todos os registradores sejam equivalentes e não reservados. As máquinas antigas não permitiam esta situação, destinado propósitos específicos aos registradores, decrementando efetivamente o número de registradores de propósito geral. Se o número real de registradores de propósito geral for pequeno, a tentativa de alocar-se variáveis em registradores não será produtiva. Ao invés disto, o compilador reservará todos os registradores não comprometidos para o uso em uma avaliação de expressão.

Surge a questão sobre qual a quantidade suficiente de registradores. A resposta dependerá de como os registradores são utilizados pelo compilador. A maioria dos compiladores reservam alguns registradores para avaliação de expressões, alguns para passagem de parâmetros, e permitem que os outros restantes sejam utilizados para armazenar variáveis.

Duas características principais do conjunto de instruções dividem as arquiteturas RPG. Ambas dizem respeito a natureza dos operandos para uma instrução típica de aritmética ou instrução lógica (instruções da ALU). As características são:

1. a primeira preocupa-se se uma instrução da ALU possui dois ou três operandos. No formato de três operandos, a instrução contém os operando de resultado e dois operandos fonte. No formato de dois operandos, um dos operandos é o operando fonte, e ao mesmo tempo, o operando resultado para a operação;
2. a segunda refere-se a quantidade de operandos que podem ser endereços de memória, nas instruções da ALU. O número de operandos de memória, suportados por uma instrução típica da ALU, pode variar de nenhum a três.

EXEMPLO: Combinações possíveis de operandos de memória, e total de operandos por uma instrução típica da ALU. Estas combinações servem para classificar quase todas as máquinas existentes:

- *LOAD-STORE (REGISTER-REGISTER)*: máquinas cujas instruções da ALU não fazem nenhuma referência à memória. Nesta máquina a memória só pode ser acessada com instruções *load* e *store*;
- *REGISTER*-MEMORY: máquinas cujas instruções da ALU possuem um operando de memória. Nesta máquina a memória pode ser acessada como parte de qualquer instrução;
- *MEMORY-MEMORY*: máquinas cujas instruções da ALU possuem mais de um operando de memória. Nesta máquina, todos os operandos são mantidos na memória.

Número de endereços de Memória	Número Máximo de Operandos Permitidos	Exemplos
0	3	SPARC, MIPS, Precision Architecture, PowerPC, ALPHA
1	2	Intel 80x86, Motorola 68000
2	2	VAX (também possui formato com três operandos)
3	3	VAX (também possui formato com dois operandos)

A seguir, são apresentadas as vantagens e desvantagens de cada uma destas alternativas. Entretanto, as comparações não são absolutas, elas são qualitativas e seus impactos reais dependem do compilador e estratégia de implementação. Uma máquina com operações *memory-memory*, pode ser facilmente dividida por um compilador, e utilizada como uma máquina *register-register*. deve-se observar que os fatores que mais causam impacto no projeto de uma nova arquitetura são a codificação das instruções, e o número de instruções necessárias para realizar uma tarefa.

Vantagens e Desvantagens dos Três Tipos Mais Comuns de Máquinas de Registradores de Propósito Geral

I) Register/Register - (0, 3)

VANTAGENS

- simples;
- codificação de comprimento fixo para instruções;
- modelo simples de geração de códigos;
- as instruções gastam um número de *clocks* semelhantes.

DESVANTAGENS

- para uma determinada tarefa, requer-se um número maior de instruções, com relação as arquiteturas cujas instruções possuem referências à memória;
- algumas instruções são pequenas, ocasionando desperdício de *bits* na codificação.

II) Register/Memory - (1, 2)

VANTAGENS
- pode-se acessar o dado sem o uso da instrução LOAD;
- o formato das instruções proporciona uma fácil decodificação e produz uma boa densidade.

DESVANTAGENS
- os operandos não são equivalentes, pois um operando fonte é destruído em uma operação binária;
- a codificação de um número de registro, e de um endereço de memória em cada instrução, pode restringir o número de registradores;
- os *clocks* por instrução variam pela localização do operando.

l) Memory/Memory - (3, 3)

VANTAGENS
- mais compacta;
- não gasta registadores para armazenamentos temporários.

DESVANTAGENS
- grande variação do comprimento da instrução, especialmente para instruções de três operandos;
- grande variação no trabalho por instrução;
- geração de gargalo nos acessos à memória,

Observações:
- notação (m, n) : m representa a quantidade de operandos de memória, e n o total de operandos;
- as máquinas com poucas alternativas no formato da instrução, facilitam a implementação do compilador, pois o compilador deve realizar poucas decisões,
- as máquinas com uma grande variedade no formato das instruções, reduzem o número de *bits* necessários para codificar um programa,
- uma máquina que usa um número pequeno de *bits* para codificar um programa, é dita ter uma *boa densidade de instruções*,
- o número de registradores de uma máquina, também afeta o tamanho da instrução.

2.3 - Endereçamento de Memória

Independentemente da arquitetura ser *register-register* ou *register-memory*, deve-se ser definido dois pontos:
- como são interpretados os endereços de memória; e
- como eles são especificados.

Interpretando Endereços de Memória

A interpretação dos endereços de memória resultam em objetos, os quais são acessados em função de um endereço, e de um comprimento.

Os conjuntos de instruções discutidos neste texto, são endereçados por *byte* e provêem acesso para *bytes* (8 *bits*), *meia palavra* (16 *bits*), *palavra* (32 *bits*), e alguns casos *palavra dupla* (64 *bits*).

Tem-se duas convenções para ordenação dos *bytes* dentro de uma palavra:

1. *ORDENAÇÃO DE BYTES LITTLE ENDIAN*: coloca o *byte*, cujo o endereço é "x...x00", na posição menos significativa da palavra *(the little end)*;

2. *ORDENAÇÃO DE BYTES BIG ENDIAN*: coloca o *byte*, cujo o endereço é "x...x00", na posição mais significativa da palavra *(the big end)*.

No endereçamento *big endian*, o endereço de um dado é o endereço do byte mais significativo, enquanto no *little endian*, o endereço do dado é o endereço do byte menos significativo.

Quando se trabalha dentro de uma máquina, a ordem do *byte* não é freqüentemente perceptível, pois somente os programas que acessam as mesmas locações , ou como palavra ou como byte, podem perceber a diferença. Entretanto, a ordem do *byte* ocasiona problemas quando ocorre troca de dados entre máquinas com diferentes ordenações.

Em várias máquinas os acessos, a objetos maiores que um *byte*, devem ser alinhados. Um acesso a um objeto de tamanho *S bytes*, no endereço de *byte A*, é alinhado se *A mod S = 0*.

EXEMPLO: Endereços cujo acesso a um objeto é alinhado ou desalinhado, com relação ao *offset* do *byte* (os três *bits* de mais baixa ordem do endereço)

Objeto Endereçado	Alinhado no offset do byte	Desalinhado no offset do byte
byte	0, 1, 2, 3, 4, 5, 6, 7	nunca
meia palavra	0, 2, 4, 6	1, 3, 5, 7
palavra	0, 4	1, 2, 3, 5, 6, 7
palavra dupla	0	1, 2, 3, 4, 5, 6, 7

Qual a razão para se projetar uma máquina com restrições de alinhamento?

- acessos desalinhados causam complicações no hardware, pois a memória é tipicamente alinhada, nos limites da palavra ou palavra dupla;
- um acesso desalinhado na memória, irá requer múltiplos acessos alinhados na memória;
- mesmo nas máquinas que permitem acessos desalinhados, observa-se que os programas com acessos alinhados são executados mais rapidamente;
- mesmo se os dados estiverem alinhados, para suportar-se acessos a *byte* ou *meia palavra*, requer-se que sejam feitos alinhamentos, a fim de obter-se *bytes*, ou *meias palavras*, alinhados nos registradores;
- dependendo da instrução, a máquina pode requerer que seja estendido o sinal da quantidade;
- em algumas máquinas um *byte*, ou *meia palavra*, não afeta a porção superior de um registrador;
- com relação aos armazenamentos, somente os *bytes* afetados na memória podem ser alterados.

Modos de Endereçamento

O modo de endereçamento é a especificação da arquitetura para o endereço de um objeto, o qual ele irá acessar.

Em uma máquina RPG, um modo de endereçamento pode especificar uma constante, um registrador, ou uma locação na memória. Quando uma locação na memória é utilizada, o endereço de memória absoluto, especificado pelo modo de endereçamento, é denominado *endereço efetivo*.

No exemplo a seguir, tem-se os modos de endereçamentos de dados utilizados nas máquinas mais recentes. Os literais, ou *immediates*, são considerados modos de endereçamento na memória (mesmo que o valor que eles acessam esteja dentro da

instrução), ainda que os registradores estejam freqüentemente separados. Foram mantidos separados os modos de endereçamento que dependem do contador de programa *(PC-relavite addressing)*. O endereçamento relativo ao PC, é utilizado primariamente para especificar os endereços de código nas instruções de controle.
EXEMPLO

Modo de Endereçamento	Exemplo de Instrução	Denotação	Utilização
registrador	Add R4, R3	Regs[R4] ← Regs[R4] + Regs [R3]	quando um valor está em um registrador
imediato	Add R4, #3	Regs[R4] ← Regs[R4] + 3	para constantes
deslocamento	Add R4, 100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	acesso à variáveis locais
indireto no registrador	Add R4, (R1)	Regs[R4] ← Regs[R4] + Mem[Regs[R1]]	acesso utilizando um ponteiro ou um endereço calculado
indexado	Add R3, (R1+R2)	Regs[R3] ← Regs[R3] + Mem[Regs[R1]+ Regs[R2]]	as vezes é útil em endereçametno de um array: R1=base do array; R2=quantidade indexada
direto ou absoluto	Add R1, (1001)	Regs[R1] ← Regs[R1] + Mem[1001]	as vezes é útil para acessar dados estáticos; pode ser necessário que o endereço constante seja grande
indireto na memória	Add R1, @ (R3)	Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]	se R3 é o endereço de um ponteiro p, o modo resulta em *p
auto-incremento	Add R1, (R2)+	Regs[R1] ← Regs[R1] + Mem[Regs[R2]] Regs[R2]←Regs[R2]+d	útil para varrer-se um array dentro de um loop. R2 aponta para o início do array; cada referência incrementa R2 de um tamanho de um elemento (d)
auto-decremento	Add R1, - (R2)	Regs[R2]←Regs[R2]-d Regs[R1] ← Regs[R1] + Mem[Regs[R2]]	mesmo uso que o auto-incremento; auto-incremento e auto-decremento podem ser utilizados para implementar as operações push e pop de uma pilha
escalado	Add R1, 100 (R2) [R3]	Regs[R1] ← Regs[R1] +Mem[100+Regs[R2]+ Regs[R3]*d]	utilizado para indexar array. Em algumas máquinas, pode ser utilizado em qualquer modo de endereçamento indexado.

Observações:
- o símbolo ←, representa atribuição;
- os array Mem e Regs representam a memória principal e os registradores, respectivamente;
- Mem[Regs[R2]] representa o conteúdo da memória, cujo endereço é dado pelo conteúdo do registrador 1 (R1).

Algumas propriedades dos modos de endereçamento:

- permitem a redução do total de instruções para realizar uma operação;
- podem aumentar a complexidade de se construir uma máquina;
- pode incrementar o CPI médio da máquina.

Consequentemente, os modos de endereçamento possuem um papel importante na definição da arquitetura do conjunto de instruções da máquina, pois um benefício é adicionado em detrimento de outro, logo o projetista deve optar pela situação que melhor atenda os seu propósito.

Modos de Endereçamento por Deslocamento

A questão principal deste modo de endereçamento, refere-se ao alcance do deslocamento a ser utilizado. A escolha do tamanho do campo de deslocamento é importante, pois ele afeta diretamente o comprimento da instrução.

Modo de endereçamento Imediato ou Literal

Imediatos, podem ser utilizados nas seguintes situações:

a) em operações aritméticas;
b) em comparações, principalmente nos branches;
c) em operações de *mov*, quando uma constante for requerida em um registrador.

Estas situações ocorrem quando as constantes são escritas no código (as quais tendem a ser pequenas), ou para endereços constantes (os quais podem ser grandes).

Ao utilizar-se imediatos, é importante saber se eles serão necessários em todas operações ou em apenas um subconjunto delas.

Outra medida importante do conjunto de instruções é o intervalo dos valores para os imediatos. Da mesma forma que os valores dos deslocamentos, o tamanho dos valores dos imediatos afetam o comprimento das instruções.

2.4 - Operações do Conjunto de Instruções

Os operadores suportados pela maioria das arquiteturas são:

TIPO DO OPERADOR	EXEMPLO
aritmético e lógico	aritmética de inteiros e operações lógicas: ADD, SUB, and, or
transferência de dados	transferência de dados entre memória e registradores: Load/Store
controle	branchs, jumps, chamada e retorno de procedimento, traps
sistema	chamadas do sistema operacional, instruções de gerenciamento da memória virtual
ponto flutuante	operações de ponto flutuante: ADD, MULTIPLY
decimal	soma e multiplicação decimal, conversão de decimais para caracteres
string	comparação, busca, copias de strings
gráfico	operações de pixel, de compressão e de descompressão

O implementador da arquitetura do conjunto de instruções, deve ter em mente a seguinte regra prática: *"as operações mais simples de um conjunto de instruções, são as mais executadas"*. Logo, ele deve fazer o possível para torná-las mais rápidas.

Instruções para o Fluxo de Controle

Pode-se distinguir quatro tipos diferentes de mudanças do fluxo de controle:
1) branch (condicional);
2) jump (incondicional);
3) chamada de procedimentos;
4) retorno de procedimentos.

O endereço de destino de uma instrução de controle de fluxo sempre deve ser especificado. Na maioria dos casos o destino é especificado explicitamente na instrução. A exceção principal é o retorno de um procedimento, pois o retorno não é conhecido em tempo de compilação. A forma mais comum de especificar-se o endereço de destino, é provendo um deslocamento que é adicionado ao contador de programa (PC). As instruções de controle deste tipo são chamadas: relativas ao PC (*PC-relative*). Os branches ou jumps, relativos ao PC, são vantajosos porque o alvo é freqüentemente próximo da instrução corrente, e requer-se poucos bits para especificar a posição relativa ao PC corrente.

A utilização do endereçamento relativo ao PC, também permite que o código seja executado independentemente do local que ele é carregado. Esta propriedade, denominada *independência de posição*, pode eliminar algum trabalho quando faz-se o link do programa, e também é útil em programas que sofrem links em tempo de execução.

Para implementar os retornos e jumps indiretos, os quais o alvo não é conhecido em tempo de compilação, requer-se outro método de endereçamento diferente do PC-relative. Aqui, deve existir uma forma para especificar-se o alvo dinamicamente, a fim de que ele possa ser alterado em tempo de execução. Este endereçamento dinâmico pode ser tão simples quanto chamar um registrador que contém o endereço alvo. De forma alternativa, o jump pode permitir qualquer modo de endereçamento a ser usado para suprir o endereço alvo. Estes jumps indiretos pelo registrador, também são úteis para:
a) instruções case ou switch das linguagens de programação, as quais seleciona uma dentre várias alternativas;
b) bibliotecas compartilhadas dinamicamente, onde é permitido que uma biblioteca seja carregada somente quando ele for realmente chamada pelo programa;
c) funções virtuais das linguagens orientadas por objetos, as quais permitem que rotinas diferentes sejam chamadas, dependendo do tipo do dado.

Em todos estes casos, o endereço alvo não é conhecido em tempo de compilação, e desta forma, é usualmente carregado a partir da memória para um registrador, antes do jump indireto pelo registrador.

Devido os branches usarem endereçamento relativo ao PC para especificar seus alvos, ocorre uma questão importante : o quanto estão longe dos branches, os alvos dos branches? Conhecendo-se a distribuição destes deslocamentos, ajudará na escolha do provimento do offset do branch, e consequentemente, ela irá afetar a codificação e o comprimento da instrução.

Devido ao fato da maioria das alterações do fluxo de controle serem realizadas por branches, é importante decidir-se como especificar o branch. A seguir tem-se três técnicas utilizadas:
a) Condition Code (CC)

Como a condição é testada: as operações da ALU setam bits especiais, possivelmente controlados pelo programa;

Vantagem: às vezes a condição é setada livremente;

Desvantagem: CC é um estado extra; CC força a ordenação da instrução, pois ela passa informação de uma instrução para um branch.

b) Condition Register

Como a condição é testada: testa um registro arbitrário com o resultado de uma comparação;

Vantagem: simples;

Desvantagem: utiliza um registrador.

c) Compare and Branch

Como a condição é testada: a comparação é parte de um branch. Freqüentemente a comparação é limitada a um subconjunto;

Vantagem: uma instrução, ao invés de duas para um branch;

Desvantagem: podem ocorrer muitos cálculos por instruções.

Uma das mais notáveis propriedades dos branches é que um grande número de comparações são testes simples de igualdade ou desigualdade, e dentre estas, a maioria são comparações com zero. Consequentemente, algumas arquiteturas tratam isto como um caso especial, principalmente se uma instrução do tipo compare and branch estiver sendo usada.

As chamadas de procedimentos e retornos incluem transferência de controle, e possivelmente, o salvamento de algum estado. No mínimo, deve-se salvar o endereço de retorno. Algumas arquiteturas possuem um mecanismo para salvar os registradores, enquanto outras requerem que o compilador gere as instruções para salvar o estado. Existem duas convenções importantes para salvar os registradores:

a) CALLER SAVING: significa que o procedimento que chama deve salvar os registradores, os quais ele quer preservados para acessos após a chamada;
b) CALLEE SAVING: significa que o procedimento chamado deve salvar os registradores que ele quer usar.

Existem situações que o caller saving deve ser usado, por causa dos padrões de acesso para variáveis globais visíveis em dois procedimentos diferentes. Por exemplo, suponha um procedimento P1 que chama P2, e ambos manipulam a variável global x. Se P1 tiver alocado x para um registrador, ele deve salvar x em uma locação conhecida por P2, antes da chamada para P2. A habilidade do compilador em descobrir quando um procedimento chamado pode acessar as quantidades alocadas em um registrador, é complicada pela possibilidade da compilação separada, e situações onde P2 pode não utilizar x, mas pode chamar outro procedimento, P3, que acessa x. Por causa destas complicações, muitos compiladores utilizam a técnica caller saving.

Nos casos que ambas convenções podem ser utilizadas, alguns programas serão melhor executados com callee saving, e outros com o caller saving. Os compiladores mais sofisticados utilizam uma combinação destas duas técnicas, e o alocador de registros pode escolher qual registro usar para uma variável, baseado na técnica escolhida.

2.5 - Tamanho e Tipo dos Operandos

Existem duas alternativas para especificar o tipo de um operando:

1) o tipo pode ser especificado, codificando-o no opcode (mais utilizado);
2) o tipo pode ser anotado com o dado, tag, o qual será interpretado pelo hardware (não utilizado atualmente).

Usualmente, o tipo de um operando fornece efetivamente seu tamanho. Por exemplo, os tipos mais comuns são: caracter (1 byte), meia palavra (16 bits), palavra (32 bits), ponto flutuante de precisão simples (32 bits), e ponto flutuante de precisão dupla (64 bits). Geralmente, utiliza-se o código ASCII para caracteres, e os números inteiros são representados por binários em complemento de dois.

Algumas arquiteturas provêem operações sobre cadeias de caracteres, embora tais operações sejam usualmente limitadas, e tratam cada byte da string como um único caracter. As operações típicas são comparações e cópias.

Também tem-se arquiteturas que suportam um formato decimal para aplicações comerciais, chamado binary-coded decimal (BCD), onde quatro bits são utilizados para codificar os valores de 0 a 9, e 2 dígitos decimais são embutidos em cada byte. Por exemplo o código BCD 0001 1001, refere-se ao decimal 19.