Organização de Computadores

Prof. Robson de Souza

Nível de Microarquitetura - IJVM

Modelo de memória IJVM

Agora, estamos prontos para estudar a arquitetura da IJVM. Em essência, ela consiste em uma memória que pode ser vista de dois modos: um arranjo de 4.294.967.296 bytes (4 GB) ou um arranjo de 1.073.741.824 palavras, cada uma consistindo em 4 bytes. Diferente da maioria das ISAs, a Java Virtual Machine (Máquina Virtual Java) não deixa nenhum endereço absoluto de memória diretamente visível no nível ISA, mas há vários endereços implícitos que fornecem a base para um ponteiro. instruções IJVM só podem acessar a memória indexando a partir desses ponteiros. Em qualquer instante, as seguintes áreas de memória são definidas:

1. O conjunto de constantes. Essa área não pode ser escrita por um programa IJVM e consiste em constantes, cadeias e ponteiros para outras áreas da memória que podem ser referenciadas. Ele é carregado quando o programa é trazido para a memória e não é alterado depois. Há um registrador implícito, CPP, que contém o endereço da primeira palavra do conjunto de constantes.

2. O quadro de variáveis locais. Para cada invocação de um método é alocada uma área para armazenar variáveis durante o tempo de vida da invocação, denominada quadro de variáveis locais. O quadro de variáveis locais não inclui a pilha de operandos, que é separada. Há um registrador implícito que contém o endereço da primeira localização do quadro de variáveis locais. Nós o denominaremos LV. Os parâmetros passados na chamada do método são armazenados no início do quadro de variáveis locais.

3. A pilha de operandos. É garantido que o quadro não exceda certo tamanho, calculado com antecedência pelo compilador Java. O espaço da pilha de operandos é alocado diretamente acima do quadro de variáveis locais. De qualquer modo, há um registrador implícito que contém o endereço da palavra do topo da pilha. Note que, diferente do CPP e do LV, esse ponteiro, SP, muda durante a execução do método à medida que operandos são passados para a pilha ou retirados dela.

4. A área de método. Por fim, há uma região da memória que contém o programa, à qual nos referimos como a área de “texto” em um processo UNIX. Há um registrador implícito que contém o endereço da instrução a ser buscada em seguida. Esse ponteiro é denominado contador de programa (Program Counter) ou PC. Diferente das outras regiões da memória, a área de método é tratada como um vetor de bytes.

É preciso esclarecer uma questão em relação aos ponteiros. Os registradores CPP, LV e SP **são todos ponteiros para palavras**, não para bytes, e **são deslocados pelo número de palavras**. Para o subconjunto de inteiros que escolhemos, todas as referências a itens no conjunto de constantes, o quadro de variáveis locais e as pilhas são palavras, e todos os deslocamentos usados para indexar esses quadros são deslocamentos de palavras.

Por exemplo, LV, LV + 1 e LV + 2 se referem às primeiras três palavras do quadro de variáveis locais. Em comparação, LV, LV + 4 e LV + 8 se referem a palavras em intervalos de quatro palavras.

Ao contrário, PC contém um endereço de byte, e uma adição ou subtração ao PC altera o endereço por um número de bytes, e não por um número de palavras.

Conjunto de instruções da IJVM

O conjunto de instruções da IJVM é mostrado na figura que está no final desta nota de aula. Cada instrução consiste em um opcode e às vezes um operando, tal como um deslocamento de memória ou uma constante.

Na figura, A primeira coluna dá a codificação hexadecimal da instrução. A segunda dá seu mnemônico em linguagem de montagem. A terceira dá uma breve descrição de seu efeito.

São fornecidas instruções para passar para a pilha uma palavra que pode vir de diversas fontes. Entre essas fontes estão o conjunto de constantes (LDC_W), o quadro de variáveis locais (ILOAD) e a própria instrução (BIPUSH). Uma variável também pode ser retirada da pilha e armazenada no quadro de variáveis locais (ISTORE).

Duas operações aritméticas (IADD e ISUB), bem como duas operações lógicas booleanas (IAND e IOR), podem ser efetuadas usando as duas palavras de cima da pilha como operandos. Em todas as operações aritméticas e lógicas, duas palavras são retiradas da pilha e o resultado é devolvido a ela.

São fornecidas quatro instruções de desvio, uma incondicional (GOTO) e três condicionais (IFEQ, IFLT e IF_ICMPEQ). Todas as instruções de ramificação, se tomadas, ajustam o valor de PC conforme o tamanho de seus deslocamentos.

Há também instruções IJVM para trocar as duas palavras do topo da pilha uma pela outra (SWAP).

Exemplo de compilação de Java para IJVM

Para uma melhor compreensão do assunto, vamos utilizar como exemplo um código Java simples que será convertido para uma linguagem de montagem correspondente. Na linguagem de montagem já fazemos uso das instruções. Por fim, o código na linguagem de montagem deve ser convertido para o formato da IJVM em hexadecimal. Utilizaremos o formato hexadecimal pois ele será escrito de forma mais sucinta que os valores binários puros, mas vale ressaltar que se quiser saber o valor em binário, é só converter os números da base hexadecimal para binário.

A figura abaixo mostra um exemplo de um código escrito em Java convertido para linguagem de montagem e para o programa IJVM em hexadecimal. Note que, como já foi explicado, cada instrução em linguagem de montagem corresponde a uma instrução em hexadecimal.

(a) Fragmento em Java. (b) Linguagem de montagem Java correspondente. (c) Programa IJVM em hexadecimal.

(a)

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

(b)

```
1        ILOAD  j          // i = j + k
2        ILOAD  k
3        IADD
4        ISTORE i
5        ILOAD  i          // if (i == 3)
6        BIPUSH 3
7        IF_ICMPEQ L1
8        ILOAD  j          // j = j – 1
9        BIPUSH 1
10       ISUB
11       ISTORE j
12       GOTO   L2
13  L1:  BIPUSH  0         // k = 0
14       ISTORE  k
15  L2:
```

(c)

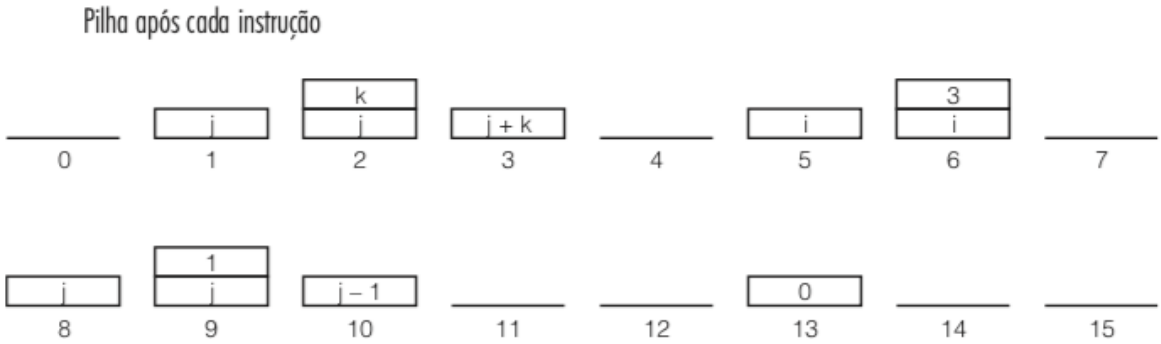
```
0x15 0x02
0x15 0x03
0x60
0x36 0x01
0x15 0x01
0x10 0x03
0x9F 0x00 0x0D
0x15 0x02
0x10 0x01
0x64
0x36 0x02
0xA7 0x00 0x07
0x10 0x00
0x36 0x03
```

Fonte: (Tanenbaum e Austin, 2013)

O código compilado é direto. Primeiro, j e k são passadas para a pilha, somadas e o resultado é armazenado em i. Então, i e a constante 3 são passadas para a pilha e comparadas. Se forem iguais, é tomado um desvio para L1, onde k é ajustada para 0. Se forem diferentes, a comparação falha e o código logo após IF_ICMPEQ é executado. Feito isso, ele desvia para L2, onde as partes then e else se fundem.

A pilha de operandos para o programa IJVM acima é mostrada na figura abaixo. Antes de o código começar a executar, ela está vazia, o que é indicado pela linha horizontal acima do 0. Após a primeira ILOAD, j está na pilha, como indicado por j no retângulo acima de 1 (o que significa que a instrução 1 foi executada). Depois da segunda ILOAD, duas palavras estão na pilha, como mostrado acima de 2. Após a IADD, há somente uma palavra na pilha, que contém a soma j + k. Quando a palavra do topo é retirada e armazenada em i, a pilha está vazia, como mostrado acima do 4.

A instrução 5 (ILOAD) inicia a declaração if passando i para a pilha (em 5). Em seguida, vem a constante 3 (em 6). Após a comparação, a pilha está novamente vazia (7). A instrução 8 é o início da parte else do fragmento de programa Java. A parte else continua até a instrução 12, quando então desvia para a parte then e vai para o rótulo L2.



Fonte: (Tanenbaum e Austin, 2013)

Por fim, para mapear esse programa para uma execução física, é necessário utilizar os registradores como as unidades de memória, bem como unidades de armazenamento de informações. A essa altura a escolha de nomes para a maioria dos registradores já deve ser óbvia: CPP, LV e SP são usados para conter os ponteiros para o conjunto de constantes, variáveis locais e o topo da pilha, enquanto PC contém o endereço do próximo byte a ser buscado no fluxo de instruções. MBR é um registrador de 1 byte que contém os bytes da sequência de instrução, à medida que eles chegam da memória para ser interpretados. Além disso, cada projeto terá os registradores necessários, tais como registradores adicionais, porém isso vai depender de cada microarquitetura.

Projeto do nível de microarquitetura

Como quase tudo na ciência da computação, o projeto da microarquitetura está repleto de compromissos. Computadores têm muitas características desejáveis, entre elas velocidade, custo, confiabilidade, facilidade de utilização, requisitos de energia e tamanho físico. Contudo, um compromisso comanda as decisões mais importantes que os projetistas de CPU devem tomar: velocidade versus custo.

A velocidade pode ser medida de várias maneiras, mas dadas uma tecnologia de circuitos e uma ISA, há três abordagens básicas para aumentar a velocidade de execução:

1. Reduzir o número de ciclos de clock necessários para executar uma instrução.

2. Simplificar a organização de modo que o ciclo de clock possa ser mais curto.

3. Sobrepor a execução de instruções.

As duas primeiras são óbvias, mas há uma surpreendente variedade de oportunidades de projeto que pode afetar drasticamente o número de ciclos de clock, o período de clock, ou – em grande parte das vezes – ambos.

O número de ciclos de clock necessários para executar um conjunto de operações é conhecido como

comprimento do caminho. Às vezes, o comprimento do caminho pode ser encurtado adicionando-se hardware especializado. Por exemplo, adicionando um incrementador ao PC, nesse caso, não precisamos mais usar a ULA para fazer avançar o PC.

Obviamente que tudo isso tem um preço a ser pago, logo, sobrepor a execução de instruções (pipeline) é, de longe, o mais interessante e oferece a melhor oportunidade para drásticos aumentos de velocidade. A simples sobreposição da busca e execução de instruções dá um resultado surpreendentemente efetivo. Entretanto, técnicas mais sofisticadas avançam muito mais, sobrepondo a execução de muitas instruções. Na verdade, essa ideia está no coração do projeto de computadores modernos.

Referências bibliográficas:

TANENBAUM, Andrew S. Organização Estruturada de Computadores, 2007, 5ª Edição.

TANENBAUM, Andrew S. AUSTIN, Todd; Organização Estruturada de Computadores, 2013, 6ª Edição.

Conjunto de instruções da IJVM:

Hexa	Mnemônico	Significado
0x10	BIPUSH *byte*	Carregue o byte para a pilha
0x59	DUP	Copie a palavra do topo da pilha e passe-a para a pilha
0xA7	GOTO *offset*	Desvio incondicional
0x60	IADD	Retire duas palavras da pilha; carregue sua soma
0x7E	IAND	Retire duas palavras da pilha; carregue AND booleano
0x99	IFEQ *offset*	Retire palavra da pilha e desvie se for zero
0x9B	IFLT *offset*	Retire palavra da pilha e desvie se for menor do que zero
0x9F	IF_ICMPEQ *offset*	Retire duas palavras da pilha; desvie se iguais
0x84	IINC *varnum const*	Some uma constante a uma variável local
0x15	ILOAD *varnum*	Carregue variável local para pilha
0xB6	INVOKEVIRTUAL *disp*	Invoque um método
0x80	IOR	Retire duas palavras da pilha; carregue OR booleano
0xAC	IRETURN	Retorne do método com valor inteiro
0x36	ISTORE *varnum*	Retire palavra da pilha e armazene em variável local
0x64	ISUB	Retire duas palavras da pilha; carregue sua diferença
0x13	LDC_W *index*	Carregue constante do conjunto de constantes para pilha
0x00	NOP	Não faça nada
0x57	POP	Apague palavra no topo da pilha
0x5F	SWAP	Troque as duas palavras do topo da pilha uma pela outra
0xC4	WIDE	Instrução prefixada; instrução seguinte tem um índice de 16 bits

Fonte: (Tanenbaum e Austin, 2013)