MIC-2

O nível de microarquitetura

Enzo Picanço Alberoni [1]

Mariana Cossetti Dalfior [2]

Resumo

Este artigo fornece uma visão geral abrangente da Microarquitetura 2 (Mic-2) de uma microarquitetura, que é um subconjunto da Máquina Virtual Java (JVM) chamada IJVM. A Mic-2 contém um microprograma que busca, decodifica e executa as instruções atribuídas pela IJVM. A arquitetura Mic-2 foi projetada para melhorar o desempenho em comparação com seu antecessor, o Mic-1, usando menos ciclos de clock para executar algumas operações e tendo um conjunto maior de instruções. O Mic-2 também tem um design modular que permite que os usuários personalizem seu sistema de acordo com suas necessidades e recursos disponíveis. Em resumo, a arquitetura Mic-2 oferece uma melhoria significativa no desempenho em comparação com sua antecessora, mantendo a alta capacidade de personalização.

Palavras-chave: Caminho_de_dados. Microprogramas. MIC-1. IJVM. MIC-2.

1 Introdução

A arquitetura de computadores é um ramo da ciência da computação que estuda a estrutura interna e o funcionamento dos sistemas de hardware. E dentro dessa área, a microarquitetura é responsável por detalhar como os processadores são projetados.

O nível da microarquitetura é responsável por detalhar como os processadores são projetados, e os microprogramas desempenham um papel fundamental para garantir o funcionamento adequado do processador. Esses programas são uma sequência de instruções básicas que permitem que o hardware execute tarefas complexas. O Mic-1 foi uma das primeiras unidades desenvolvidas para realizar esse trabalho no processo de execução de instruções. No entanto, com o passar dos anos, surgiu a necessidade de aprimorar a funcionalidade desses dispositivos, o que levou ao desenvolvimento de novos modelos, como o Mic-2, capaz de executar operações mais complexas com mais eficiência.

Ao analisar esses componentes e suas interações, podemos obter uma compreensão mais profunda de como os computadores funcionam e como eles podem ser otimizados para um melhor desempenho.

2 O nível da microarquitetura

No livro "Organização estruturada de computadores"(2013), os autores Andrew S. Tanenbaum e Todd Austin exploram a complexidade e a interconexão dos diferentes níveis de uma arquitetura de computador. Eles destacam o nível da microarquitetura, que é responsável pela execução da arquitetura do conjunto de instruções. A ISA funciona como uma ponte entre o software

[1]Ciência da Compuatação, Universidade Estadual do Norte Fluminense Darcy Ribeiro - UENF, 20211100053@pq.uenf.br

[2]Ciência da Compuatação, Universidade Estadual do Norte Fluminense Darcy Ribeiro - UENF, 20211100064@pq.uenf.br

e o hardware, definindo as instruções disponíveis, seus formatos, modos de endereçamento, registros e outras características necessárias para a operação do sistema.

Os ISAs de computadores modernos, especialmente aqueles com arquiteturas RISC, geralmente têm instruções simples que são executadas em um único ciclo de clock. Os autores também discutem as instruções IJVM, que se destacam por sua simplicidade e facilidade de leitura e execução. Este estudo se concentra especialmente em Mic2, embora também aborde conceitos de Mic-1 e a microarquitetura que contém um microprograma. Esse microprograma busca, decodifica e executa as instruções atribuídas pela IJVM.

2.1 Caminho de dados

Para começarmos a falar sobre o Mic-2 é necessário o entendimento do caminho de dados. O caminho de dados é um aspecto crucial da microarquitetura que determina como os dados trafegam pelo processador. Ele consiste em várias unidades funcionais, como registros, unidades lógicas aritméticas e multiplexadores, entre outros. Cada unidade no caminho de dados tem um papel e uma função específicos a desempenhar. Por exemplo, o registrador armazena dados temporariamente antes de serem processados por outros componentes. A unidade lógica aritmética realiza operações matemáticas como adição e subtração, enquanto o multiplexador seleciona entre várias entradas com base em sinais de controle. O tempo desempenha um papel fundamental para garantir que cada componente do caminho de dados execute sua tarefa com eficiência e precisão. Essas informações de tempo são expressas por meio de diagramas de tempo que ilustram o tempo que cada instrução leva para ser executada.

De acordo com Tanenbaum e Austin (2013), o caminho de dados é uma parte da CPU que contém a Unidade Lógica e Aritmética (ULA), bem como suas entradas e saídas, onde ocorrem as principais operações de processamento de dados. O caminho de dados contém vários registradores de 32 bits com nomes simbólicos, como PC, MAR e MDR. Nessa microarquitetura, existem inicialmente dois barramentos: B e C. A maioria dos registradores pode enviar seu conteúdo para o barramento B. A saída da ULA controla o deslocador e, em seguida, o barramento C, cujo valor pode ser gravado em um ou mais registradores simultaneamente. O barramento B é responsável por receber o conteúdo de alguns dos registros, enquanto o barramento C é responsável por receber a saída da ULA. Como podemos analisar na Figura 1.

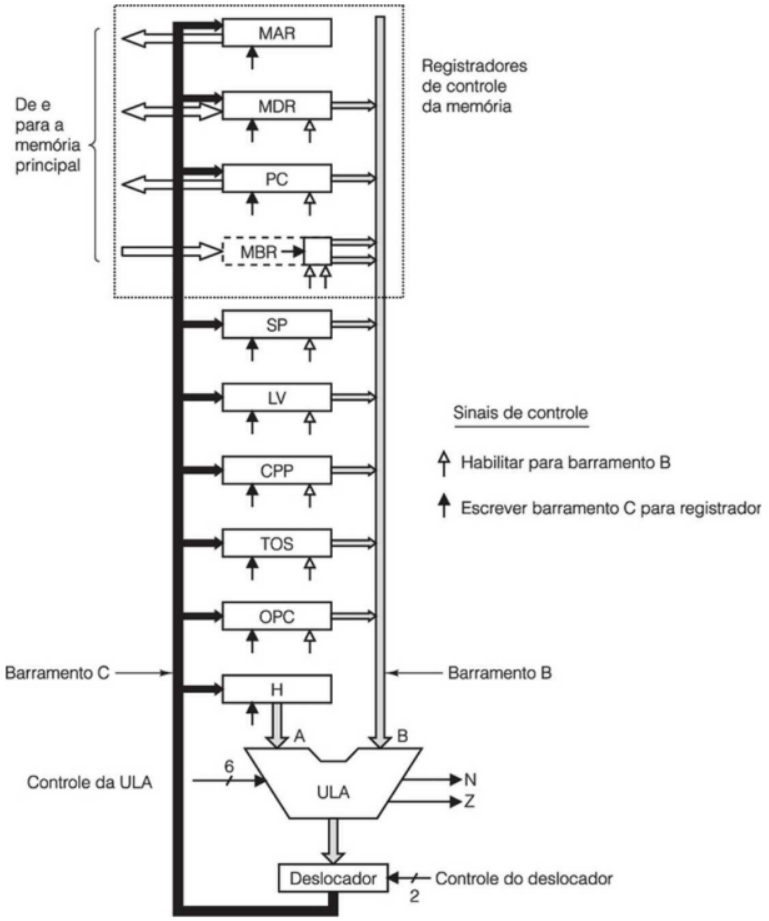

Figura 1: Caminho de dados

O processo de leitura e gravação de um registro começa com a seleção de um registro como entrada B da ULA. O conteúdo do registro é então passado para o barramento B no início do ciclo e permanece lá até o final do ciclo. Em seguida, a ULA executa todos os processos necessários e retorna uma saída que é enviada ao deslocador e inserida no barramento C. Após o processo de saída de dados da ULA e a estabilidade do deslocador, o conteúdo do barramento C é transferido para os registros por meio de um sinal de clock. A temporização exata do caminho de dados permite a leitura e a gravação no mesmo registrador em um único ciclo, sendo possível realizar isso sem produzir nenhum lixo.

Dessa forma, é possível perceber que um projeto eficiente de caminho de dados garante que as instruções sejam executadas rapidamente, sem gargalos ou atrasos. Ou seja, os projetistas devem considerar cuidadosamente fatores como a compatibilidade da arquitetura do conjunto de instruções ao projetar esse componente da microarquitetura para obter o desempenho ideal.

2.1.1 Temporização do caminho de dados

O tempo das operações do caminho de dados é fundamental para o desempenho geral de um processador. O processo de sincronização dessas operações é conhecido como "temporização do caminho de dados". Em termos simples, isso significa que cada instrução deve esperar que todas as suas entradas estejam disponíveis para que a execução possa começar.

Para evitar conflitos entre instruções, há várias técnicas usadas nos processadores modernos. De acordo com Tanenbaum e Austin (2013), uma dessas técnicas envolve a divisão de uma

instrução dividida em quatro subciclos: AW, AX, AY e AZ, que operam em diferentes partes do caminho de dados simultaneamente. Esses subciclos são sincronizados por um sinal de clock, o que garante que sejam executados em intervalos precisos.

O que faz cada subciclo?

O subciclo AW, por exemplo, é crucial para estabelecer os sinais elétricos corretos que irão comandar o caminho de dados. Sem essa preparação adequada, os subciclos subsequentes podem não ser capazes de realizar suas funções corretamente.

O subciclo AX, por sua vez, é responsável pela seleção e condução do registrador. Este é um passo crítico que determina quais dados serão manipulados nos subciclos seguintes. A precisão neste estágio é vital para garantir que os dados corretos sejam selecionados para processamento.

O subciclo AY é o momento em que a Unidade Lógica Aritmética (ULA) e o deslocador começam a operar com os dados válidos. Este é o estágio em que a computação real ocorre, e é aqui que a ULA e o deslocador devem funcionar de forma eficiente para garantir que os cálculos sejam realizados corretamente.

Finalmente, o subciclo AZ é o tempo em que os resultados do barramento C são propagados para os registradores. Este é o estágio final do ciclo, onde os resultados são armazenados para uso futuro. É também durante este subciclo que o registrador que controla o barramento B é interrompido e se prepara para o próximo ciclo.

2.1.2 Operação de memória

A memória é um componente crucial de um sistema de computador, pois armazena dados e instruções para processamento pelo processador. A eficiência de um sistema de computador pode ser influenciada pela capacidade e velocidade da memória. A memória opera através de processos como leitura, gravação e acesso a dados ou instruções.

Existem duas formas de comunicação com a memória: uma porta de memória de 32 bits, endereçável por palavra, e outra de 8 bits, endereçável por byte. A primeira é controlada pelos registradores MAR (Memory Address Register) e MDR (Memory Data Register), enquanto a segunda é controlada pelo registrador PC, que lê 1 byte para os 8 bits de ordem baixa do MBR, sendo restrita apenas à leitura de dados da memória (Tanenbaum & Austin, 2013).

O MAR armazena endereços de palavras, enquanto o PC armazena endereços de bytes. Isso significa que ao inserir um valor em PC e iniciar uma leitura de memória, o byte correspondente da memória será lido e colocado nos 8 bits de ordem baixa do MBR. Por outro lado, ao inserir um valor em MAR e iniciar uma leitura de memória, os bytes correspondentes à palavra serão lidos e colocados no MDR.

Para permitir que o MAR conte palavras enquanto a memória física conta bytes, um mapeamento específico é realizado. Quando o MAR é colocado no barramento de endereço, seus bits não são mapeados diretamente para as linhas de endereço. Por exemplo, o bit 0 do MAR é ligado à linha 2 do barramento de endereço, o bit 1 do MAR é ligado à linha 3 do barramento de endereço, e assim por diante. Portanto, existe apenas uma memória real que opera com bytes.

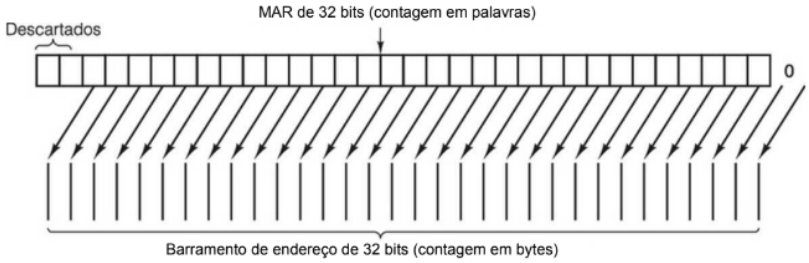

Figura 2: Mapeamento dos bits em MAR para o barramento de endereço.

O MBR, um registrador de 8 bits, pode ser transferido para o barramento B de duas maneiras: com ou sem sinal. Quando é necessário o valor sem sinal, a palavra de 32 bits colocada no barramento B contém o valor do MBR nos 8 bits de ordem baixa e zero nos 24 bits superiores. A opção de converter o MBR de 8 bits em um valor de 32 bits com sinal ou sem sinal no barramento B é determinada por qual dos dois sinais de controle é ativado.

2.2 Microinstruções

Microinstruções são uma parte fundamental da arquitetura de um processador. Eles são responsáveis por executar as tarefas de processamento mais básicas, como buscar dados na memória ou realizar operações matemáticas simples. Essas instruções estão presentes em todas as CPUs modernas e são usadas para controlar o fluxo de dados no processador. Eles são especialmente importantes em sistemas que utilizam microprogramação, como o IJVM.

Na prática, as microinstruções funcionam como um conjunto de comandos que podem ser executados pela CPU. Cada comando é composto por um opcode (código operacional) e um conjunto de operandos (dados que serão manipulados). Quando a CPU recebe uma instrução, ela procura as microinstruções associadas ao opcode correspondente e as executa sequencialmente.

O principal objetivo das microinstruções é permitir que as instruções sejam executadas em ciclos menores, reduzindo o tempo necessário para concluir uma tarefa específica. Sem eles, a CPU teria que realizar várias operações diferentes para concluir uma única tarefa. As microinstruções são armazenadas no nível de microprogramação do processador e são responsáveis por controlar o fluxo de dados pelo caminho de dados. Isso permite que os programadores criem programas mais eficientes, economizando recursos valiosos do sistema.

No livro "Organização estruturada de computadores"de Tanenbaum e Austin, há a descrição do controle do caminho de dados, que requer 29 sinais divididos em cinco grupos funcionais. Esses sinais especificam as operações para um ciclo do caminho de dados, que consiste em copiar valores dos registradores para o barramento B, propagar os sinais através da ALU e shifter, direcioná-los para o barramento C e, finalmente, escrever os resultados no registrador ou registros.

- 9 sinais para controlar escrita de dados do barramento C para registradores.
- 9 sinais para controlar habilitação de registradores dirigidos ao barramento B para a entrada da ULA.
- 8 sinais para controlar as funções da ULA e do deslocador.
- 2 sinais para indicar leitura/escrita na memória via MAR/MDR.

- 1 sinal para indicar busca na memória via PC/MBR.

2.3 MIC-1

A memória de armazenamento de controle, como descrita por Tanenbaum e Austin (2013), é um componente crucial na arquitetura MIC-1. Ela é responsável por armazenar microinstruções, que são instruções mais detalhadas e específicas do que as instruções ISA (Instruction Set Architecture) comumente encontradas na memória principal. Cada uma dessas microinstruções é armazenada em uma das 512 palavras da memória de armazenamento de controle, cada uma contendo 36 bits.

Essa memória de armazenamento de controle funciona de maneira semelhante a uma memória somente leitura (ROM), o que significa que as informações armazenadas nela não são facilmente alteradas ou apagadas. Isso requer que ela tenha seu próprio registrador de endereço de memória, conhecido como MPC (Memory Program Counter), e seu próprio registrador de dados de memória, conhecido como MIR (Memory Instruction Register). O MPC não segue uma ordem específica, enquanto o MIR é responsável por armazenar a microinstrução atual.

Tanenbaum e Austin (2013) explicam que o MIR é responsável por controlar a seleção da próxima microinstrução, enquanto a ULA contém 8 bits que selecionam a função da mesma e comandam o deslocador. Os bits C fazem os registradores individuais carregarem a saída da ULA vinda do barramento C, enquanto os bits M controlam operações de memória. Por fim, os últimos 4 bits comandam o decodificador que determina o que entra no barramento B. Nesse caso, foi usado um decodificador padrão 4 para 16, mesmo que sejam requeridas apenas 9 possibilidades.

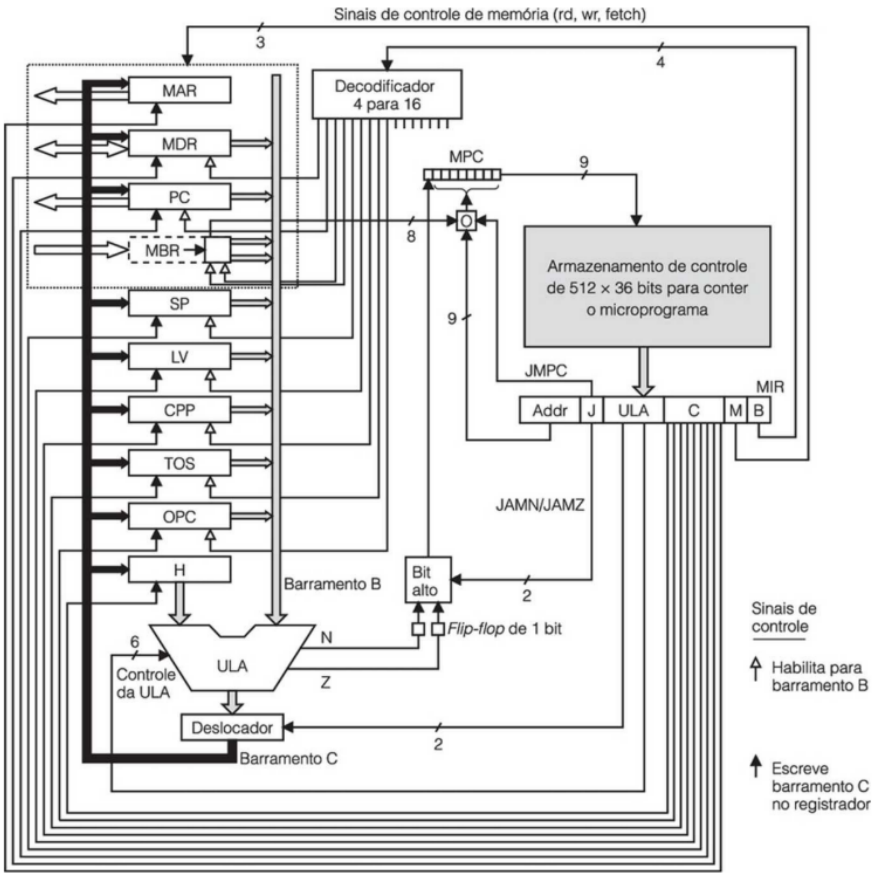

Figura 3: Diagrama de blocos completo da microarquitetura do Mic-1.

2.4 IJMV

A arquitetura da IJVM é composta por uma memória que pode ser interpretada de duas maneiras: como um conjunto de 4 GB ou como um conjunto de 1.073.741.824 palavras, cada uma com 4 bytes. Ao contrário de muitas ISAs, a Máquina Virtual Java (JVM) não permite a visibilidade direta de endereços de memória absolutos no nível ISA. No entanto, existem vários endereços implícitos que servem como base para um ponteiro. A IJVM é uma implemetação da MIC-1 e as instruções dela só podem acessar a memória indexando a partir desses ponteiros. Existem áreas de memória a ser definidas:

- O conjunto de constantes: não pode ser modificada por um programa IJVM. Esta área contém constantes, cadeias e ponteiros para outras áreas de memória. Ela é carregada quando o programa é introduzido na memória e permanece inalterada. Existe um registrador implícito, o CPP, que contém o endereço da primeira palavra do conjunto de constantes.

- O quadro de variáveis locais: Para cada invocação de um método na IJVM, é alocada uma área chamada quadro de variáveis locais para armazenar variáveis durante a duração da invocação. Os parâmetros do método são armazenados no início deste quadro. A pilha de operandos é separada do quadro de variáveis locais, mas por eficiência, é executada logo acima dele. Existe um registrador implícito, LV, que contém o endereço inicial do quadro de variáveis locais.

- A pilha de operandos: O tamanho do quadro de variáveis locais na IJVM é garantido para não exceder um limite pré-calculado pelo compilador Java. O espaço para a pilha de operandos é alocado acima do quadro de variáveis locais. Na implementação, é útil pensar na pilha de operandos como parte do quadro de variáveis locais. Existe um registrador implícito, SP, que contém o endereço do topo da pilha. Este ponteiro, ao contrário do CPP e do LV, muda durante a execução do método conforme operandos são adicionados ou removidos da pilha.

- A área de método: Na IJVM, existe uma região de memória que contém o programa, conhecida como a área de "texto". Um registrador implícito, chamado contador de programa (Program Counter) ou PC, contém o endereço da próxima instrução a ser buscada. Ao contrário das outras áreas de memória, a área de método é tratada como um vetor de bytes.

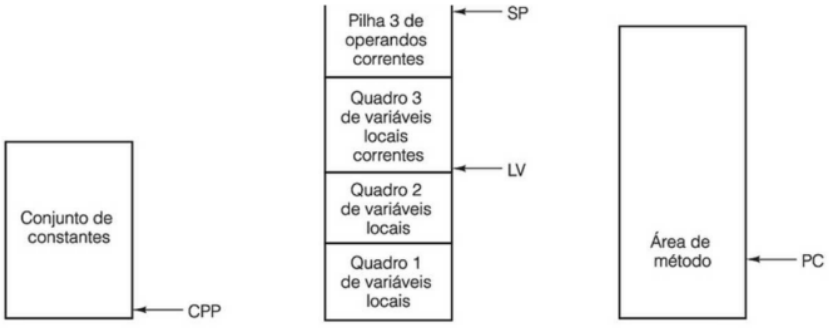

Figura 4: As partes da memória IJVM.

Os registradores CPP, LV e SP são todos ponteiros para palavras, não para bytes, e são deslocados pelo número de palavras. Ao contrário, PC contém um endereço de byte.

2.4.1 Compilador de IJVM

Para que o código Java seja compilado para IJVM é necessário conhecer algumas instruções que são utilizadas para isso. Existem várias instruções notáveis na IJVM, incluindo o conjunto de constantes (LDC-W), o quadro de variáveis locais (ILOAD) e a instrução (BIPUSH). Há instruções para remover uma variável da pilha e colocá-la no quadro de variáveis locais (ISTORE), bem como instruções para operações aritméticas (IADD e ISUB) e booleanas (IAND e IOR). Além disso, a instrução (INVOKEVIRTUAL) é usada para chamar outro método, e a instrução (IRETURN) é usada para devolver o controle ao método que fez a chamada. A seguir podemos observar um exemplo:

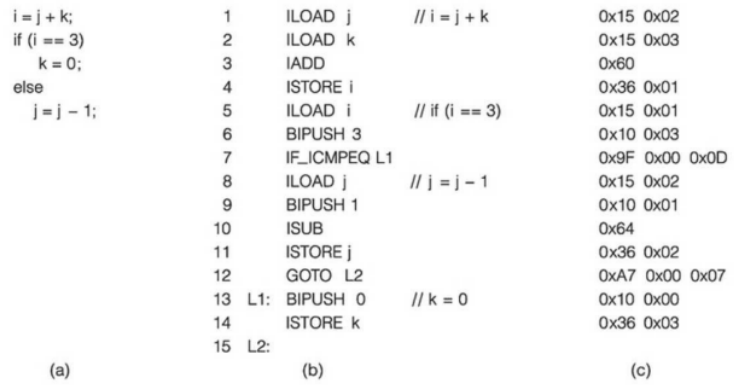

Figura 5: Exemplo do assembler Java traduzido para o programa de montagem e então para o programa IJVM em hexadecimal

De acordo com as instruções mostradas na imagem, j e k são empurrados para a pilha, onde são somados e o resultado é armazenado em i. Em seguida, i e 3 são colocados na pilha e comparados. Se a comparação for verdadeira, o fluxo do programa é desviado e k é definido como 0. Se a comparação for falsa, todo o código após o 7 é executado.

2.5 MIC-2

A Mic-2 é uma arquitetura de microprograma usada em processadores. Ela foi projetada para melhorar a eficiência e o desempenho dos sistemas, oferecendo um nível mais alto de microinstruções do que sua antecessora, a Mic-1. Com a Mic-2, os desenvolvedores podem criar instruções complexas usando microprogramação. Dessa forma, eles conseguem otimizar o funcionamento do sistema e garantir que ele execute tarefas com maior rapidez e precisão. Além disso, a Mic-2 permite que as instruções sejam executadas em paralelo. Isso significa que várias operações podem ser realizadas ao mesmo tempo, tornando o processador muito mais rápido do que antes.

Tanenbaum e Austin (2013) dizem que a utilização da IFU na MIC-2 seria uma maneira de melhorar a performance em relação à MIC-1. A IFU é responsável por buscar e processar as instruções, o que ajuda a reduzir o comprimento do caminho percorrido pelo processo. Existem duas formas possíveis de fazer isso: interpretando cada opcode para determinar quantos campos

adicionais devem ser buscados ou aproveitando a natureza sequencial das instruções para disponibilizar os próximos fragmentos de 8 e 16 bits sempre que possível.. Dessa forma, é possível otimizar o desempenho da arquitetura proposta pelos autores.

Além disso existem dois tipos de MBRS: o MBR1, que possui 8 bits, e o MBR2, que tem 16 bits. A IFU acompanha os bytes mais usados recentemente pela unidade principal de execução e disponibiliza automaticamente o próximo byte em MBR1. Além de que, a IFU reconhece quando MBR1 é lido e carrega imediatamente o próximo byte. O MBR2 funciona de maneira semelhante ao MBR1, porém contém os próximos 2 bytes. Em relação aos dados armazenados em cada registrador do MBRS, Tanenbaum e Austin (2013) explicam que no caso do MBR1 está sempre localizado no registrador deslocado à direita, enquanto no caso do MBR2 são preservados os dois últimos bytes utilizados. Se houver capacidade adicional no registrador de deslocamento para outra palavra, a IFU inicia um ciclo de memória para realizar a leitura.

A Unidade de Busca de Instruções (IFU) utiliza o IMAR como seu próprio registrador para endereçar a memória. Esse sistema possui um incrementador exclusivo, evitando a necessidade da ULA principal. Ao realizar esse processo, é necessário que o IFU esteja atento ao barramento C para que o valor do PC seja copiado corretamente para o IMAR durante a busca da nova palavra. Além disso, cabe à IFU manter atualizado o valor do PC.

A seguir é possível observar o microcódigo para a máquina melhorada:

Rótulo	Operações	Comentários
nop1	goto (MBR)	Desvie para a próxima instrução
iaddl	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iadd2	H = TOS	H = topo da pilha
iadd3	MDR = TOS = MDR + H; wr; goto (MBR1)	Some duas palavras do topo; escreva para novo topo da pilha
isubl	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
isub2	H = TOS	H = topo da pilha
isub3	MDR = TOS = MDR - H; wr; goto (MBR1)	Subtraia TOS da palavra anterior na pilha
iandl	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iand2	H = TOS	H = topo da pilha
iand3	MDR = TOS = MDR AND H; wr; goto (MBR1)	AND palavra anterior da pilha com TOS
ior1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
ior2	H = TOS	H = topo da pilha
ior3	MDR = TOS = MDR OR H; wr; goto (MBR1)	OR palavra anterior da pilha com TOS
dup1	MAR = SP = SP + 1	Incremente SP; copie para MAR
dup2	MDR = TOS; wr; goto (MBR1)	Escreva nova palavra da pilha
pop1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
P°P2		Espere pela leitura
pop3	TOS = MDR; goto (MBR1)	Copie nova palavra para TOS
swapl	MAR = SP - 1; rd	Leia a segunda palavra da pilha; ajuste MAR para SP
swap2	MAR = SP	Prepare para escrever nova 2ª palavra
swap3	H = MDR; wr	Salve novo TOS; escreva 2ª palavra para pilha
swap4	MDR = TOS	Copie antigo TOS para MDR
swap5	MAR = SP - 1; wr	Escreva antigo TOS para 2º lugar na pilha
swap6	TOS = H; goto (MBR1)	Atualize TOS
bipushl	SP = MAR = SP + 1	Ajuste MAR para escrever para novo topo da pilha
bipush2	MDR = TOS = MBR1; wr; goto (MBR1)	Atualize pilha em TOS e memória
iloadl	MAR = LV + MBR1U; rd	Passe LV + índice para MAR; leia operando
iload2	MAR = SP = SP + 1	Incremente SP; passe novo SP para MAR
iload3	TOS = MDR; wr; goto (MBR1)	Atualize pilha em TOS e memória
istorel	MAR = LV + MBR1U	Ajuste MAR para LV + índice
istore2	MDR = TOS; wr	Copie TOS para armazenamento
istore3	MAR = SP = SP - 1; rd	Decremente SP; leia novo TOS
istore4		Espere por leitura
istoreõ	TOS = MDR; goto (MBR1)	Atualize TOS

Figura 6: Microprograma para a Mic-2.

Rótulo	Operações	Comentários
widel	goto (MBR1 OR 0x100)	Próximo endereço é 0x100 com operação OR efetuada com *opcode*
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Idêntica a iloadl mas usando índice de 2 bytes
widejstorel	MAR = LV + MBR2U; goto istore2	Idêntica a istorel mas usando índice de 2 bytes
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	O mesmo que widejloadl mas indexando a partir de CPP
iincl	MAR = LV + MBR1U; rd	Ajuste MAR para LV + índice para leitura
iinc2	H = MBR1	Ajuste H para constante
iinc3	MDR = MDR + H; wr; goto (MBR1)	Incremente por constante e atualize
gotol	H = PC - 1	Copie PC para H
goto2	PC = H + MBR2	Some deslocamento e atualize PC
goto3		Tem de esperar que IFU busque novo *opcode*
goto4	goto (MBR1)	Despache para a próxima instrução
ifltl	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iflt2	OPC = TOS	Salve TOS em OPC temporariamente
iflt3	TOS = MDR	Ponha novo topo da pilha em TOS
iflt4	N = OPC; if (N) goto T; else goto F	Desvie no bit N
ifeql	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
ifeq2	OPC = TOS	Salve TOS em OPC temporariamente
ifeq3	TOS = MDR	Ponha novo topo da pilha em TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Desvie no bit Z
ifjcmpeql	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
if_icmpeq2	MAR = SP = SP - 1	Ajuste MAR para ler novo topo da pilha
if_icmpeq3	H = MDR; rd	Copie segunda palavra da pilha para H
if_icmpeq4	OPC = TOS	Salve TOS em OPC temporariamente
if_icmpeq5	TOS = MDR	Ponha novo topo da pilha em TOS
if_icmpeq6	Z = H - OPC; if (Z) goto T; else goto F	Se 2 palavras do topo forem iguais, vá para T, senão vá para F
T	H = PC - 1; goto goto2	O mesmo que gotol
F	H = MBR2	Toque bytes em MBR2 para descartar
F2	goto (MBR1)	
invokevirtuaM	MAR = CPP + MBR2U; rd	Ponha endereço de ponteiro de método em MAR
invokevirtual2	OPC = PC	Salve Return PC em OPC
invokevirtual3	PC = MDR	Ajuste PC para 1ª byte do código de método
invokevirtual4	TOS = SP - MBR2U	TOS = endereço de OBJREF - 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = endereço de OBJREF
invokevirtual6	MDR = SP + MBR2U + 1 ; wr	Sobrescreva OBJREF com ponteiro de ligação
invokevirtual7	MAR = SP = MDR	Ajuste SP, MAR à localização para conter PC antigo

Figura 7: Microprograma para a Mic-2.

Rótulo	Operações	Comentários
widel	goto (MBR1 OR 0x100)	Próximo endereço é 0x100 com operação OR efetuada com *opcode*
wide_iload1	MAR = LV + MBR2U; rd; goto iload2	Idêntica a iloadl mas usando índice de 2 bytes
widejstorel	MAR = LV + MBR2U; goto istore2	Idêntica a istorel mas usando índice de 2 bytes
ldc_w1	MAR = CPP + MBR2U; rd; goto iload2	O mesmo que widejloadl mas indexando a partir de CPP
iincl	MAR = LV + MBR1U; rd	Ajuste MAR para LV + índice para leitura
iinc2	H = MBR1	Ajuste H para constante
iinc3	MDR = MDR + H; wr; goto (MBR1)	Incremente por constante e atualize
gotol	H = PC - 1	Copie PC para H
goto2	PC = H + MBR2	Some deslocamento e atualize PC
goto3		Tem de esperar que IFU busque novo *opcode*
goto4	goto (MBR1)	Despache para a próxima instrução
ifltl	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iflt2	OPC = TOS	Salve TOS em OPC temporariamente
iflt3	TOS = MDR	Ponha novo topo da pilha em TOS
iflt4	N = OPC; if (N) goto T; else goto F	Desvie no bit N
ifeql	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
ifeq2	OPC = TOS	Salve TOS em OPC temporariamente
ifeq3	TOS = MDR	Ponha novo topo da pilha em TOS
ifeq4	Z = OPC; if (Z) goto T; else goto F	Desvie no bit Z
ifjcmpeql	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
if_icmpeq2	MAR = SP = SP - 1	Ajuste MAR para ler novo topo da pilha
if_icmpeq3	H = MDR; rd	Copie segunda palavra da pilha para H
if_icmpeq4	OPC = TOS	Salve TOS em OPC temporariamente
if_icmpeq5	TOS = MDR	Ponha novo topo da pilha em TOS
if_icmpeq6	Z = H - OPC; if (Z) goto T; else goto F	Se 2 palavras do topo forem iguais, vá para T, senão vá para F
T	H = PC - 1; goto goto2	O mesmo que gotol
F	H = MBR2	Toque bytes em MBR2 para descartar
F2	goto (MBR1)	
invokevirtuaM	MAR = CPP + MBR2U; rd	Ponha endereço de ponteiro de método em MAR
invokevirtual2	OPC = PC	Salve Return PC em OPC
invokevirtual3	PC = MDR	Ajuste PC para 1ª byte do código de método
invokevirtual4	TOS = SP - MBR2U	TOS = endereço de OBJREF - 1
invokevirtual5	TOS = MAR = H = TOS + 1	TOS = endereço de OBJREF
invokevirtual6	MDR = SP + MBR2U + 1 ; wr	Sobrescreva OBJREF com ponteiro de ligação
invokevirtual7	MAR = SP = MDR	Ajuste SP, MAR à localização para conter PC antigo

Figura 8: Microprograma para a Mic-2.

Rótulo	Operações	Comentários
invokevirtual8	MDR = OPC; wr	Prepare para salvar PC antigo
invokevirtual9	MAR = SP = SP + 1	Incremente SP para apontar para a localização para conter LV antigo
invokevirtuaMO	MDR = LV; wr	Salve LV antigo
invokevirtual11	LV = TOS; goto (MBR1)	Ajuste LV para apontar para o parâmetro de ordem zero
ireturnl	MAR = SP = LV; rd	Reajuste SP, MAR para ler ponteiro de ligação
ireturn2		Espere por ponteiro de ligação
ireturn3	LV = MAR = MDR; rd	Ajuste LV, MAR para ponteiro de ligação; leia PC antigo
ireturn4	MAR = LV + 1	Ajuste MAR para apontar para LV antigo; leia LV antigo
ireturnõ	PC = MDR; rd	Restaure PC
ireturnõ	MAR = SP	
ireturn7	LV = MDR	Restaure LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Salve valor de retorno no topo da pilha original

Figura 9: Microprograma para a Mic-2.

A Mic-2 apresenta melhorias variáveis em diferentes instruções em relação a Mic-1. Por exemplo, a instrução LDC_W foi otimizada de nove para apenas três microinstruções, reduzindo seu tempo de execução em um terço. Em contraste, a instrução SWAP viu uma melhoria mais modesta, passando de oito para seis microinstruções. No entanto, o que realmente impacta o desempenho geral são as melhorias nas instruções mais frequentemente utilizadas. Entre elas, podemos destacar a ILOAD, que foi otimizada de seis para três microinstruções, a IADD, que passou de quatro para três, e a IFJCMPEQ, que agora requer 10 microinstruções para o caso de desvio tomado (anteriormente eram 13) e 8 para o caso de desvio não tomado (anteriormente eram 10). Essas otimizações nas instruções mais comuns contribuem significativamente para o aumento geral da eficiência.

3 Conclusões

Concluímos assim nossa exploração detalhada da Mic-2, a evolução da microarquitetura que seguiu a Mic-1. Ao longo deste artigo, destacamos a importância vital das microinstruções para a operação eficaz da unidade central de processamento de um computador. Utilizamos o IJVM como um modelo de referência para ilustrar esses conceitos complexos de maneira mais tangível.

Ao nos aprofundarmos na Mic-2, pudemos apreciar as inovações e melhorias que ela trouxe em relação à sua antecessora. Essas melhorias permitiram um processamento mais eficiente e acelerado, demonstrando o progresso contínuo na busca por desempenho computacional otimizado.

Além disso, esperamos que este artigo tenha proporcionado uma compreensão mais profunda da interação entre hardware e software em um nível microarquitetônico. Através do estudo da Mic-2, podemos apreciar a complexidade e a beleza da engenharia de computadores, bem como a importância da evolução constante nesta área dinâmica e em rápida evolução.

Referências

[1] Tanenbaum, A. S., & Austin, T. Organização estruturada de computadores. 6ª Edição. Editora Pearson, 30 maio, 2013.