UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO (UENF)

GABRIEL DE ALMEIDA
GABRIEL COSTA

ASPECTOS GERAIS DA MIC2 EM UMA MICROARQUITETURA: ESTUDO DO NÍVEL DE MICROARQUITETURA E CAMINHO DE DADOS

CAMPOS DOS GOYTACAZES
2023

1 Objetivo

Este artigo tem como objetivo abordar os aspectos gerais da Microarquitetura 2 (Mic2) de uma microarquitetura. Foi definido o subconjunto da JVM (Java Virtual Machine), que será a Mic1 do projeto. O subconjunto escolhido apresenta apenas operações com inteiros, por esse motivo, ela será chamada de IJVM. A microarquitetura que contém um microprograma cujo intuito é buscar, decodificar e executar as instruções atribuídas pela IJVM.

Palavras chaves: IJVM; Caminho de dados; MIC1; Microprograma; Busca de instruções; Execução de instruções

2 Introdução

Tanenbaum e Austin (2013) afirmam que o nível da microarquitetura é aquele que está acima do nível lógico digital, e sua função é explicitamente executar o nível ISA (Instruction Set Architecture - arquitetura do conjunto de instruções). O nível ISA funciona como um limiar entre o Software de um computador e o seu Hardware, definindo uma interface entre os dois. Ele é responsável por definir as instruções disponíveis, o formato delas, os endereçamentos, os registradores e outras características necessárias. A maioria das ISAs presentes nos computadores contemporâneos que apresentam projetos de arquitetura RISC apresentam instruções extremamente simples que costumam ser executadas em apenas um ciclo de clock, diferente das ISAs mais complexas, que podem exigir inúmeros ciclos de clock para executar uma única instrução.

De acordo com Tanenbaum e Austin (2013), as instruções da IJVM são extremamente simples de serem lidas e executadas, visto que são curtas e fáceis. O objetivo deste artigo é estudar essas instruções, em especial a Mic2, porém apresentando conceitos da Mic1, assim como da microarquitetura que deverá conter um microprograma cujo intuito é buscar, decodificar e executar as instruções atribuídas pela IJVM.

3 Metodologia

A metodologia utilizada no artigo baseou-se principalmente no livro "Organização Estruturada de Computadores" como fonte principal de informações e referências. Foi realizada uma revisão do livro, compreendendo a leitura atenta do capítulo 4. Isso permitiu uma compreensão aprofundada dos conceitos de microarquitetura e caminho de dados.

Foi feita uma organização lógica do conteúdo, estabelecendo uma estrutura coerente para o artigo. Isso incluiu a definição de seções e subseções que abordassem de forma clara e concisa os aspectos relevantes da microarquitetura. A escrita do artigo foi baseada em uma abordagem clara e objetiva, utilizando linguagem técnica adequada ao assunto.

Foi dada ênfase à explicação dos conceitos e à apresentação de exemplos ilustrativos sempre que apropriado, para facilitar a compreensão do leitor.

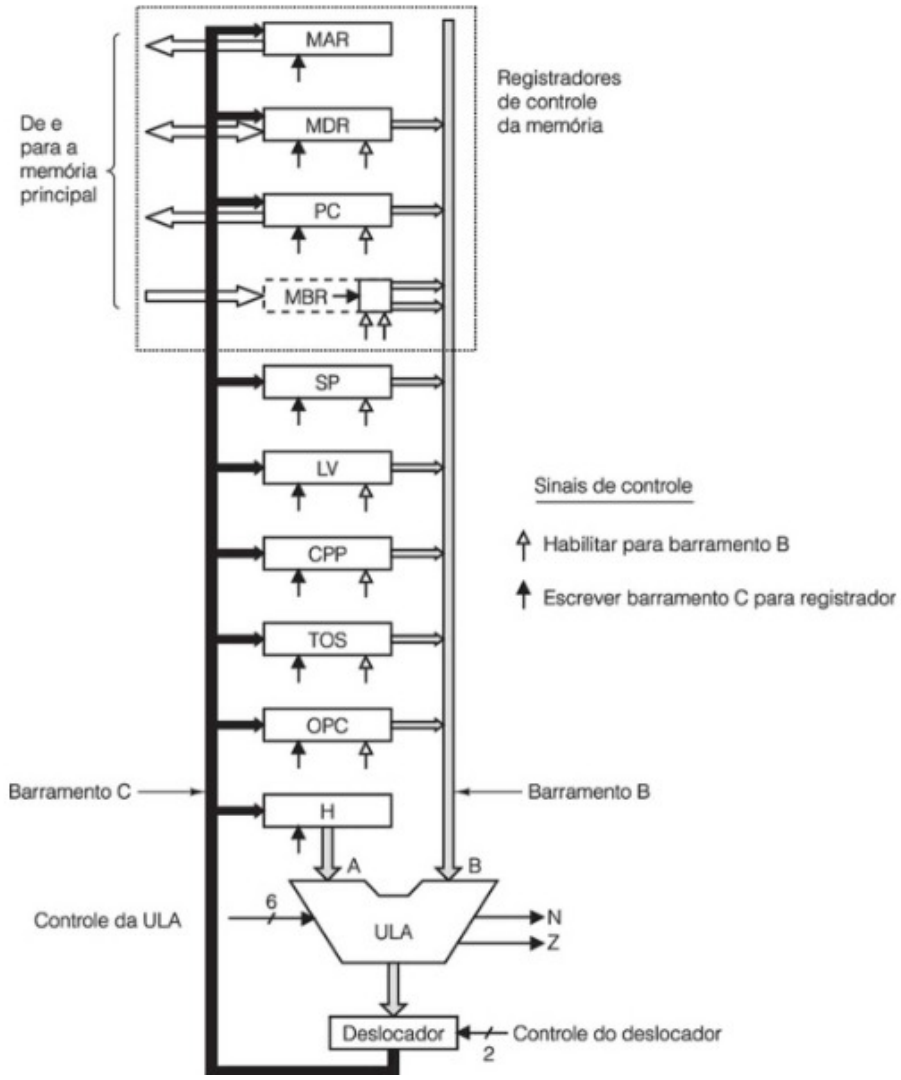
Por fim, o artigo passou por uma revisão para garantir a precisão e a consistência do conteúdo, bem como a correção gramatical e ortográfica.

4 Resultados

Antes de se iniciar o estudo real da Mic1 e Mic2, é necessário introduzir os conceitos do caminho de dados da microarquitetura aos quais essas microinstruções são aplicadas. Para isso, é preciso abordar o funcionamento e os principais aspectos relacionados ao caminho de dados.

Segundo Tanenbaum e Austin (2013), o caminho de dados é uma parte crucial da CPU, que contém a Unidade Lógica e Aritmética (ULA), bem como suas entradas e saídas, ou seja, é onde ocorrem as principais operações de processamento de dados. Ele contém diversos registradores de 32 bits com nomes simbólicos, como PC, MAR e MDR, por exemplo. 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 escrito em um ou mais registradores simultaneamente (TANENBAUM; AUSTIN,2013). Em outras palavras, o barramento B é responsável por receber o conteúdo de alguns dos registradores, e o barramento C é responsável por receber a saída da ULA.

Figure 1: Caminho de dados



No projeto da figura acima, a ULA requer a entrada de dois conjuntos de dados: uma entrada à esquerda, chamada A, e outra entrada à direita, chamada B. A entrada A está conectada a um registrador responsável pela retenção de conteúdo, chamado H, enquanto a entrada B está ligada ao barramento B, responsável por fornecer à ULA o conteúdo de qualquer um dos registradores conectados a ele.

Segundo Tanenbaum e Austin (2013), o processo de leitura e escrita de um registrador começa selecionando um registrador como entrada B da ULA. Quando isso ocorre, o conteúdo do registrador é passado para o barramento B no início do ciclo e permanece ali até o final do mesmo. Em seguida, a ULA realiza todos os processos necessários e retorna uma saída que é enviada para o deslocador e inserida no barramento C. Após isso, com o término do processo de saída de dados da ULA e a estabilidade do

deslocador, o conteúdo do barramento C é transferido para os registradores por meio de um sinal de clock. Vale lembrar que esse conteúdo pode ser armazenado inclusive no registrador que forneceu a entrada para o barramento B. A temporização precisa do caminho de dados permite ler e escrever no mesmo registrador em um único ciclo (TANENBAUM;AUSTIN,2013). Isso significa que o tempo de todo o processo de leitura e escrita é tão preciso que é possível ler e escrever no mesmo registrador em apenas um ciclo, sem produzir nenhum tipo de lixo.

4.1 Temporização de caminho de dados

Tanenbaum e Austin (2013) ressaltam que é possível ler e escrever no mesmo registrador em um único ciclo. Por exemplo, colocar o valor de SP no barramento B, desativar a entrada à esquerda da ULA, habilitar o sinal de incremento (INC) e armazenar o resultado de volta em SP. Para isso, a leitura e escrita são realizadas em diferentes momentos dentro do ciclo. Tanenbaum e Austin (2013) explica ainda que a temporização é dividida em 4 subciclos, sendo eles:

- **Subciclo AW**: É responsável por preparar os sinais elétricos na borda descendente para comandar o caminho de dados.
- **Subciclo AX**: É o tempo necessário para selecionar e conduzir o registrador.
- **Subciclo AY**: É o tempo em que a ULA e o deslocador começam a operar com os dados válidos.
- **Subciclo AZ**: É o tempo em que os resultados do barramento C são propagados para os registradores, podendo ser carregados na borda ascendente do próximo pulso de clock, enquanto o registrador que controla o barramento B é interrompido e se prepara para o próximo ciclo.

4.2 Operação memória

Tanenbaum e Austin (2013) afirmam que há dois modos diferentes de se comunicar com a memória: uma porta de memória de 32 bits endereçáveis por palavra e outra de 8 bits endereçáveis por byte. A porta de 32 bits é dividida em MAR (Memory Address Register) e MDR (Memory Data Register). Já a porta de 8 bits é controlada pelo registrador PC, que lê 1 byte para os 8 bits de ordem baixa do MBR, realizando apenas a leitura de dados da memória. Cada um desses registradores é controlado por um ou dois sinais de controle. O MAR não está conectado ao barramento B e não possui um sinal de

habilitação, enquanto o MBR não possui um sinal de escrita. Isso pode ser observado na imagem 1, em que a saída do registrador para o barramento B é indicada por uma seta clara, e a seta escura indica que um sinal de controle escreve o registrador a partir do barramento C.

O MAR contém endereços de palavras, de modo que os valores 0, 1, 2 etc. referem-se a palavras consecutivas. O PC contém endereços de bytes, portanto, os valores 0, 1, 2 etc. referem-se a bytes consecutivos. Assim, colocar o valor 2 em PC e iniciar uma leitura de memória irá ler o byte 2 da memória e colocá-lo nos 8 bits de ordem baixa do MBR. Colocar o valor 2 em MAR e iniciar uma leitura de memória irá ler os bytes 8-11 (ou seja, a palavra 2) da memória e colocá-los no MDR (TANENBAUM; AUSTIN, 2013). É importante destacar essa diferença porque o MAR e o PC referenciam duas partes diferentes da memória. Em resumo, o MAR/MDR é usado para ler e escrever palavras de dados da ISA, enquanto o PC/MBR é usado para ler o programa executável de nível ISA, que é uma sequência de bytes. Todos os outros registradores que possuem endereços utilizam endereços de palavras, como, por exemplo, o MAR.

Tanenbaum e Austin (2013) reiteram que há apenas uma memória real que funciona com bytes. Devido ao funcionamento da JVM, é permitido que o MAR conte palavras enquanto a memória física conta bytes. Para garantir que não ultrapasse a capacidade da máquina de 4 GB, são descartados os 2 bits superiores do MAR. Dessa forma, o bit 0 do MAR é ligado à linha 2 do barramento de endereço, o bit 1 é ligado à linha 3 do barramento de endereço, a linha 2 é ligada à linha 4 do barramento de endereço e assim por diante. Utilizando esse mapeamento, quando o MAR é 1, o endereço 4 é colocado no barramento, quando o MAR é 2, o endereço 8 é colocado no barramento e assim por diante.

O MBR pode ser copiado (gated) 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 (TANENBAUM; AUSTIN, 2013). Os valores sem sinal são eficazes para indexar em uma tabela ou quando um inteiro de 16 bits precisa ser montado a partir de 2 bytes consecutivos na sequência de instrução.

4.3 Microinstruções

Tanenbaum e Austin (2013) afirmam que existem 29 sinais de controle para um ciclo do caminho de dados

- 9 Sinais para controlar escrita de dados do barramento C para registradores

- 9 Sinais para controlar habilitação de registradores dirigidos ao barramento
- 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

Os sinais de controle são gerados apenas quando MAR e PC são carregados na borda de subida do clock. É possível fazer leituras seguidas em dois ciclos consecutivos, e ambas podem funcionar ao mesmo tempo. Entretanto, escrever e ler o mesmo byte pode gerar resultados indeterminados. Outro fator importante é que é recomendável habilitar apenas um registrador por vez no barramento B.

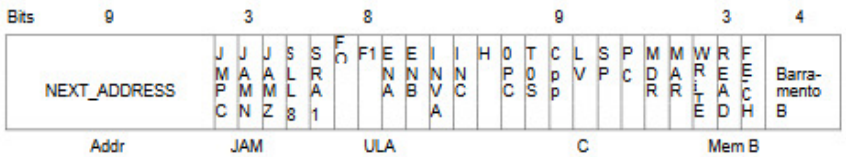
Nesse caminho de dados, existem 9 registradores. Portanto, podemos usar 4 bits para codificar as informações do barramento B e utilizar um decodificador para gerar 16 sinais de controle, sendo que 7 sinais não são necessários.

4.4 MIC-1

De acordo com Tanenbaum e Austin (2013), um dos principais componentes da parte de controle da máquina é a memória de armazenamento de controle. Essa memória contém exclusivamente microinstruções, em vez das instruções ISA. Na arquitetura MIC-1, essa memória possui 512 palavras, cada uma com microinstruções de 36 bits, conforme ilustrado na imagem 2. Diferentemente da memória principal, cada microinstrução especifica explicitamente a próxima instrução a ser executada.

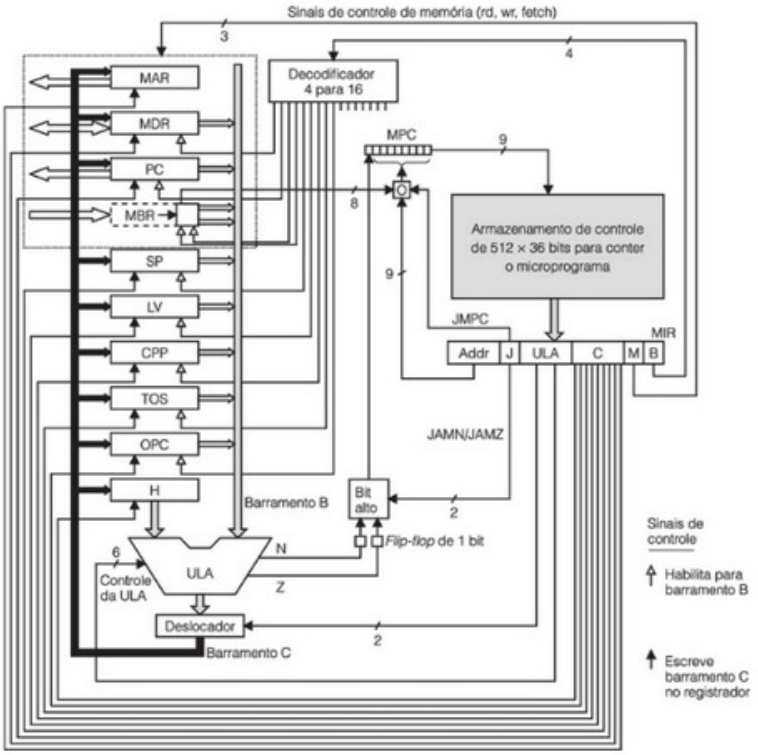
Em termos funcionais, a memória de armazenamento de controle pode ser considerada como uma memória somente leitura. Portanto, ela requer seu próprio registrador de endereço de memória, denominado MPC (Memory Program Counter), e seu próprio registrador de dados de memória, denominado MIR (Memory Instruction Register). As localizações na MPC não seguem uma ordem específica, enquanto o MIR tem a função de armazenar a microinstrução atual.

Figure 2: Formato de microinstrução para MIC-1



Tanenbaum e Austin (2013) explicam que na imagem 3 observamos o MIR que possui os grupos ADDR e J que controlam a seleção da próxima microinstrução. A ULA contém 8 bits que selecionam a função da mesma e comandam o deslocador. Já os bits C fazem os registradores individuais carregarem a saída da ULA vinda do barramento C, ainda, os bits M controlam operações de memória. Por fim, os últimos 4 bits controlam o decodificador que determina a entrada no barramento B. Na MIC-1, foi escolhido um decodificador de 4 para 16.

Figure 3: Caminho de dados MIC-1



4.5 IJVM

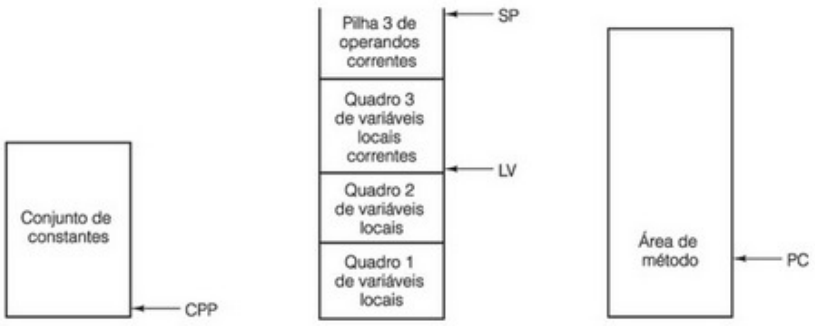
A implementação da MIC-1 é uma IJVM de 4GB de memória. Java Virtual Machine (máquina virtual Java) não deixa nenhum endereço absoluto de memória diretamente visível no nível ISA, mas há vários endereços implícitos que fornecem a base para um ponteiro. Instruções IJVM só podem acessar a memória indexando a partir desses ponteiros (TANENBAUM; AUSTIN, 2013). São definidas as seguintes áreas de memória:

- **O conjunto de constantes**: Não é escrita por um programa IJVM, possui constantes, cadeias e ponteiros para outras áreas da memória para serem referenciadas. É

carregado quando o programa é trazido para a memória, e não é alterado posteriormente. Possui o registrador CPP, que contém endereço da primeira palavra das constantes

- **O quadro de variáveis**: A cada invocação de um método é alocado uma área para armazenar variáveis temporariamente. No início, estão os parâmetros com os quais foram invocados pelo método. Nossa IJVM, executa a pilha de operandos logo acima do quadro de variáveis. Há um registrador que contém o endereço da primeira localização do quadro de variáveis locais chamado de LV.
- **A pilha de operandos**: O espaço da pilha de operandos é alocado diretamente acima do quadro de variáveis locais, em nossa IJVM, ele é imaginado como parte do quadro de variáveis locais.
- **A área de método**: Há uma região da memória que contém o programa, a qual chamamos como área de texto. Possui um registrador implícito que contém o endereço da instrução a ser buscado a seguir. O chamamos de PC.

Figure 4: Partes da IJVM



Os registradores CPP, LV e SP são ponteiros para palavras, enquanto o PC é para byte.

4.6 O conjunto de instruções

Tanenbaum e Austin (2013) citam todos os conjuntos de instruções da IJVM na imagem 5. É citado, seu código em Hexadecimal, seu mnemônico e suas descrição.

Figure 5: Instrucoes IJVM

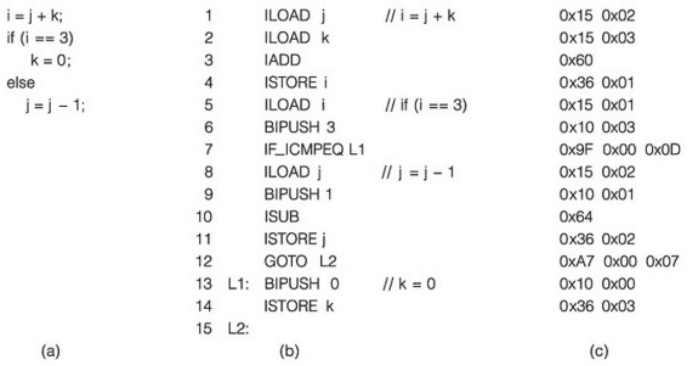
Hexa	Mnemônico	Significado
0x10	BIPUSH *byte*	Carregue o byte para a pilha
0x59	DUP	Copie a palavra do topo da pilha e passe-a para a pilha
0xA7	GOTO *offset*	Desvio incondicional
0x60	IADD	Retire duas palavras da pilha; carregue sua soma
0x7E	IAND	Retire duas palavras da pilha; carregue AND booleano
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	IFJCMPEQ *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
OxAC	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

Várias instruções de diversas fontes diferentes são fornecidas para passar uma palavra para a pilha. Dentre essas instruções destacam-se o conjunto de constantes (LDC-W), o quadro de variáveis locais (ILOAD) e a instrução (BIPUSH). Também existem instruções para retirar uma variável da pilha e colocá-la no quadro de variáveis locais (ISTORE). Também existem instruções para operações aritméticas (IADD e ISUB), e booleanas (IAND e IOR). É importante citar ainda a instrução (INVOKEVIRTUAL) usada para realizar a chamada de outro método, e em conjunto existe a instrução (IRETURN) que retorna o controle ao método que o invocou.

4.7 Compilando em IJVM

Nesse tópico será abordado como um código Java é compilado para IJVM, para isso acompanhe a imagem 6 Nessa figura, é mostrado em um simples código feito em Java, quando esse código passa por um compilador Java, de acordo com Tanenbaum e Austin (2013); ele será transformado para a linguagem de montagem da IJVM representado pela imagem (b). Com isso, o código (b) passará pelo assembler Java, que o transformará no código binário (c).

Figure 6: linguagem Montagem



Seguindo a lógica das instruções apresentadas na imagem Q, j e k são passados para a pilha, é realizada a soma entre os dois, e o resultado é salvo em i. Com isso, i e 3 são passados para a pilha e comparados, se a comparação for verdadeira, o caminho é desviado e k valerá 0. Se a comparação for falsa e todo o código após o 7 é executado.

4.8 MIC-2

Tanenbaum e Austin (2013) argumentam que a MIC-2 seria a melhoria da MIC-1 com a utilização do IFU (Instruction Fetch Unit). A IFU é uma unidade independente para buscar e processar instruções. Seria uma tática para reduzir o comprimento do caminho. Há dois modos de fazer

- Interpretada a cada opcode, determinando a quantidade de campos adicionais a serem buscados, e montar em um registrador pronto para a utilização.
- A IFU pode aproveitar a natureza sequencial das instruções e disponibilizar os próximos fragmentos de 8 e 16 bits todas as vezes, que isso tenha ou não algum sentido. Então, a unidade de execução principal pode requisitar o que precisar. (TANENBAUM; AUSTIN, 2013).

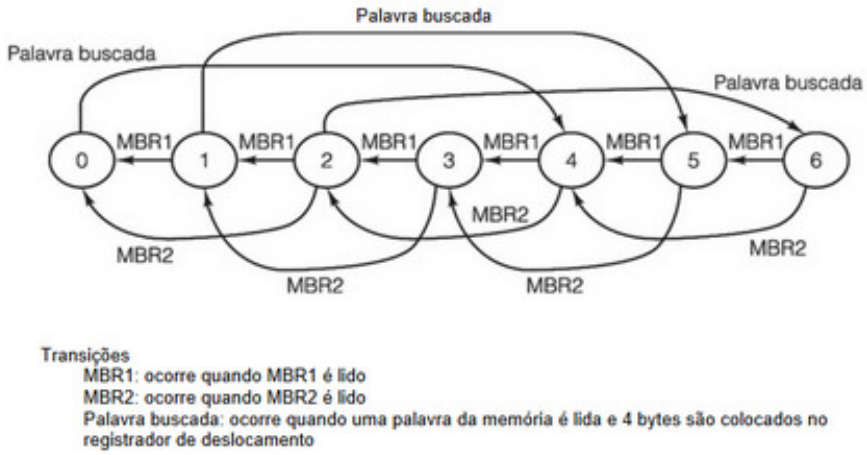
Tanenbaum e Austin (2013) dizem que há dois MBRS: O MBR1 de 8 bits e o MBR2 de 16 bits. A IFU monitora os bytes mais consumidos recentemente pela unidade de execução principal. Além disso, disponibiliza o próximo byte em MBR1, e percebe automaticamente quando MBR1 é lido, busca o próximo byte e o carrega em MBR1 de imediato. O MBR2 é semelhante, mas contém os próximos 2 bytes. Ainda, Tanenbaum e Austin (2013), descrevem que o MBR1 contém o byte mais antigo no registrador de deslocamento, enquanto o MBR2 contém os 2 bytes mais antigos. Sempre que MBR1 é lido, é deslocado

1 byte para a direita. Sempre que MBR2 é lido, ele desloca 2 bytes para a direita. Se restar espaço suficiente no registrador de deslocamento para mais uma palavra, a IFU ativa um ciclo de memória para lê-la.

A figura 7 demonstra o projeto da IFU modelada por uma FSM (Finite State Machine). Tanenbaum e Austin (2013) ressaltam que há duas partes da FSM:

- **Estado**: Representa uma possível situação na qual FSM pode estar. É possível estar em 7 estados possíveis, os quais representam a quantidade de bytes naquele registrador no momento
- **Arco**: Representa um dos três eventos diferentes que podem ocorrer. O primeiro é a leitura do 1 byte do MBR1, é ativado o registrador de deslocamento e 1 byte é deslocado para fora da extremidade direita, reduzindo o estado por um fator de 1. O segundo evento é a leitura de 2 bytes do MBR2, é reduzido o estado por um fator de dois. Quando o FSM para os estados 0, 1, ou 2, é iniciada uma referência à memória para buscar uma nova palavra.

Figure 7: MBR



A IFU tem seu próprio registrador de endereço de memória, denominado IMAR, que é usado para endereçar a memória quando uma nova palavra tem de ser buscada. Esse registrador tem seu próprio incrementador dedicado, de modo que a ULA principal não é necessária para incrementá-lo e nem para buscar a próxima palavra (TANENBAUM; AUSTIN, 2013). A IFU deve monitorar o barramento C, para sempre que o PC for carregado, o novo valor de PC ser copiado para o IMAR. Além disso, o IFU mantém o PC atualizado.

A seguir, está o microcódigo para a máquina melhorada (MIC-2) com o IFU.

Figure 8: Códigos MIC-2

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

Figure 9: Códigos MIC-2

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

Figure 10: Códigos 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

Figure 11: Códigos MIC-2

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 1º 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

Figure 12: Códigos 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
ireturnô	MAR = SP	
ireturn7	LV = MDR	Restaure LV
ireturn8	MDR = TOS; wr; goto (MBR1)	Salve valor de retorno no topo da pilha original

A MIC-2 melhora algumas instruções em relação a MIC-1. Tanenbaum e Austin (2013) citam algumas como: O LDC_W passa a ter apenas três microinstruções, o SWAP reduz de 8 para 6 microinstruções, ILOAD reduz de 4 para 3, IFJCMPEQ reduz de 13 para 10.

5 Conclusão

Neste artigo foram abordados os principais aspectos da microarquitetura, com o principal foco nos níveis da Mic1 e 2. Para isso foi necessário abordar o papel do nível ISA como uma forma de criar uma interface entre o Hardware e o Software. Além disso, foram apresentados conceitos sobre o caminho de dados da microarquitetura, assim como o funcionamento das operações de memória.

Para compreender o funcionamento da Mic1 e 2, foram discutidas as microinstruções, responsáveis por gerenciar/comandar o caminho de dados de uma microarquitetura. E com isso, foi apresentado a Mic1, usando a IJVM como modelo de exemplo, abordando as suas áreas de memória, instruções, pilhas, registros e outras características, para assim abordar os principais aspectos da Mic2.

No geral, o objetivo deste artigo foi abordar e passar uma visão geral da Mic1 e 2, no entanto, para esse fim foi necessário abordar conceitos como o caminho de dados, operações de memória e microinstruções. Porém, é necessário ressaltar novamente que cada microarquitetura de uma máquina é única, apresentando suas próprias características de acordo com o seu projeto. Apesar disso, o estudo desses aspectos seguindo um exemplo é extremamente essencial para compreender o funcionamento do processador e da arquitetura de um computador.

6 Referências

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