Unidade 14: Arquiteturas CISC e RISC

Prof. Daniel Caetano

Objetivo: Apresentar os conceitos das arquiteturas CISC e RISC, confrontando seus desempenhos.

Bibliografia:

- STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.
- MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.

INTRODUÇÃO

Até o fim da década de 1970, praticamente todos os arquitetos de processadores e computadores acreditavam que melhorar os processadores estava diretamente relacionado ao aumento na complexidade das instruções (instruções que realizam tarefas cada vez maiores) e modos de endereçamento (acessos a estruturas de dados complexas diretamente pelas linguagens de máquina).

Sua crença não era descabida: como os tempos de acesso à memória eram bastante altos, reduzir o número de instruções a serem lidas era algo bastante positivo, já que reduzia a necessidade de comunicação com a memória. Entretanto, isso não perdurou: foi percebido que muitas destas instruções mais complexas praticamente nunca eram usadas, mas a existência das mesmas tornava outras instruções mais lentas, pela dificuldade em se implementar um pipeline adequado para processar igualmente instruções complexas e instruções simples.

Com o passar do tempo e o aumento da velocidade das memórias, os benefícios dos processadores com insturções complexas (CISC - Complex Instruction Set Computer) pareceu diminuir ainda mais, sendo que a tendência de novas arquiteturas acabou por ser uma eliminação destas instruções, resultando em processadores com um conjunto reduzido de instruções (RISC - Reduced Instruction Set Computer).

Nesta aula serão apresentadas algumas diferenças entre estas arquiteturas, vantanges e desvantagens, além da inerente controvérsia relacionada.

1. CISC - COMPLEX INSTRUCTION SET COMPUTER

Nos primórdios da era computacional, o maior gargalo de todo o processamento estava na leitura e escrita de dispositivos externos ao processador, como a memória, por exemplo. Como um dos acessos à memória indispensáveis para que o computador funcione é

a leitura de instruções (chamado de *busca de instrução*), os projetistas perceberam que se criassem instruções que executavam a tarefa de várias outras ao mesmo tempo seriam capazes de economizar o tempo de busca de instrução de praticamente todas elas.

Por exemplo, caso alguém desejasse copiar um bloco da memória de 500h bytes do endereço 1000h para o endereço 2000h, usando instruções simples teria de executar o seguinte código (em Assembly Z80):

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
COPIA:      LD    A, (HL)           ; Carrega byte da origem
            INC   HL                ; Aponta próximo byte de  origem em HL
            LD    (DE), A           ; Coloca byte no destino
            INC   DE                ; Aponta próximo byte de destino em HL
            DEC   BC                ; Decrementa 1 de BC
            LD    A,B               ; Coloca valor de B em A
            OR    C                 ; Soma os bits ligados de C em A
            ; Neste ponto A só vai conter zero se B e C eram zero
            JP    NZ,COPIA          ; Continua cópia enquanto A != zero
```

Isso é suficiente para realizar a tal cópia, mas foram gastas 8 instruções no processo, com um total de 10 bytes (de instruções e dados) a serem lidos da memória (fora as leituras e escritas de resultados de operações, como as realizadas pelas instruções LD A,(HL) e LD (DE),A). Ora, se o custo de leitura dos bytes da memória é muito alto, faz todo o sentido criar uma instrução que realiza este tipo de cópia, certo? No Z80, esta instrução se chama LDIR e o código acima ficaria da seguinte forma:

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
            LDIR                    ; Realiza cópia
```

A rotina de cópia passou de 8 instruções (com 10 bytes no total) para uma única instrução, que usa 2 bytes... definitivamente, um ganho substancial no que se refere à redução de tempo gasto com leitura de memória.

Como é possível ver, os projetistas tinham razão com relação a este aspecto. Entretanto, após algum tempo passou-se a se observar que estes ganhos eram limitados, já que o número de vezes que estas instruções eram usadas era mínimo. Um dos papas da arquitetura de computadores, Donald Knuth, fez uma análise das instruções mais usadas nos programas tradicionais. Os resultados foram:

Atribuição:	47%
If/Comparação:	23%
Chamadas de Função:	15%
Loops:	6%
Saltos simples:	3%
Outros:	7%

Observe que a rotina representada anteriormente se enquadra na categoria "Loops", que tem uma ocorrência que toma aproximadamente 6% de um programa. Considerando que é um loop bastante específico: um loop de cópia, provavelmente este porcentual é ainda menor. Possivelmente, substancialmente menor.

Com o surgimento dos pipelines, isso passou a causar algum problema; a existência destas instruções tornava a implementação do pipeline mais complexa e menos eficiente para todas as instruções (incluindo as de atribuição, que são a grande maioria de um programa!). Mas por que ocorre isso?

O que acontece é que instruções complexas como o LDIR apresentado representam uma severa complexidade para otimização por pipeline. Como visto, a implementação de um pipeline espera que todas as instruções possuam mais ou menos o mesmo número de estágios (que compõem o pipeline), estagios estes que normalmente são: *busca de instrução*, *interpretação de instrução*, *busca de operandos*, *operação na ULA* e *escrita de resultados*. Fica, claramente, difícil distribuir uma instrução como LDIR nestas etapas de uma maneira uniforme. Por conseqüência, um pipeline que otimize a operação de uma instrução LDIR (supondo que ele seja possíve), terá um certo número de estágios terão que *ignorar* as instruções simples, mas ainda assim consumirão tempo de processamento.

Este tipo de problema sempre depôs contra o uso instruções complexas; entretanto, se as memórias forem lentas o suficiente para justificar uma economia de tempo com a redução de leitura de instruções, o uso de arquitetura CISC (com instruções complexas) possa até mesmo ser justificado, em detrimento do uso de Pipeline (ou com o uso de um Pipeline "menos eficiente").

Mas o que tem ocorrido é um crescimento da velocidade das memórias, em especial com o uso dos diversos níveis de cache, o que tem tornado, pelos últimos 20 anos, um tanto discutível o ganho obtido com a redução do número de instruções obtidas por seguir uma arquitetura CISC.

2. RISC - REDUCED INSTRUCTION SET COMPUTER

Com a redução do "custo" da busca de instruções, os arquitetos de processadores trataram de buscar um esquema que otimizasse o desempenho do processador a partir daquelas operações que ele executava mais: as operações simples, como atribuições e comparações; o caminho para isso é simplificar estas operações - o que normalmente leva a CPUs mais simples e menores.

A principal medida escolhida para esta otimização foi a determinação de que leituras e escritas na memória só ocorreriam com instruções de **load** e **store**: instruções de operações lógicas e aritméticas direto na memória não seriam permitidas. Essa medida, além de simplificar a arquitetura interna da CPU, reduz a inter-dependência de algumas instruções e permite que um compilador seja capaz de perceber melhor esta inter-dependência. Como

conseqüência, essa redução da inter-dependência de instruções possibilita que o compilador as reorganize, buscando uma otimização do uso de um possível pipeline.

Adicionalmente, intruções mais simples podem ter sua execução mais facilmente encaixadas em um conjunto de estágios pequeno, como os cinco principais já vistos anteriormente:

1) Busca de Instrução
2) Decodificação
3) Busca de Operando
4) Operação na ULA
5) Escrita de resultado

Às arquiteturas construídas com base nestas linhas é dado o nome de **RISC: Reduced Instruction Set Computer**, pois normalmente o conjunto de instruções destas arquiteturas é menor, estando presentes apenas as de função mais simples.

Os processadores de arquitetura RISC seguem a seguinte filosofia de operação:

1) **Execução de prefetch**, para reduzir ainda mais o impacto da latência do ciclo de busca.
2) **Ausência de instruções complexas**, devendo o programador/compilador construir as ações complexas com diversas instruções simples.
3) **Minimização dos acessos a operandos em memória**, "exigindo" um maior número de registradores de propósito geral.
4) **Projeto baseado em pipeline**, devendo ser otimizado para este tipo de processamento.

Como resultado, algumas características das arquiteturas RISC passam a ser marcantes:

1) **Instruções de tamanho fixo**: uma palavra.
2) **Execução em UM ciclo de clock**: com instruções simples, o pipeline consegue executar uma instrução a cada tick de clock.
3) **As instruções de processamento só operam em registradores**: para usá-las, é sempre necessário usar operações de load previamente e store posteriormente. (Arquitetura LOAD-STORE)
4) **Não há modos de endereçamentos complexos**: nada de registradores de índice e muito menos de segmento. Os cálculos de endereços são realizados por instruções normais.
5) **Grande número de registradores de propósito geral**: para evitar acessos à memória em cálculos intermediários.

3. COMPARAÇÃO DE DESEMPENHO

A forma mais comum de cálculo para comparação de desempenho é a chamada de "cálculo de speedup". O speedup é dado pela seguinte fórmula:

$$S = 100 * (T_S - T_C) / T_C$$

Onde T_S é o "**Tempo Sem Melhoria**" (ou sem otimização) e o T_C é o "**Tempo Com Melhoria**" (ou com otimização). O resultado será uma porcentagem, indicando o quão mais rápido o software ficou devido à melhoria introduzida.

Esta fórmula pode ser expandida, pois é possível calcular os tempos de execução por outra fórmula:

$$T = N_I * C_{PI} * P$$

Tornando claro que N_I é o **Número de Instruções** de um programa, C_{PI} é o número de **Ciclos de clock médio Por Instrução** e P é o **Período**, que é o tempo de cada ciclo de clock (usualmente dado em nanossegundos), sendo o P = 1/freqüência.

Com isso, é possível estimar, teoricamente, o ganho de uma arquitetura RISC frente à uma arquitetura CISC. Suponhamos, por exemplo, um processador com o o Z80 rodando a 3.57MHz, com um C_{PI} de algo em torno de 10 Ciclos Por Instrução e um período de 280ns. Comparemos este com um processador RISC ARM, rodando nos mesmos 3.57MHz (baixíssimo consumo), com 1,25 Ciclos por Instrução (considerando uma perda de desempenho de pipeline em saltos condicionais) e um período de 280ns.

Considerando os programas analisados no item 1, de cópia de trechos de memória, temos que o primeiro (considerado um programa compatível com uma arquitetura RISC), tem um total de:

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
COPIA:      LD    A, (HL)           ; Carrega byte da origem
            INC   HL                ; Aponta próximo byte de  origem em HL
            LD    (DE), A           ; Coloca byte no destino
            INC   DE                ; Aponta próximo byte de destino em HL
            DEC   BC                ; Decrementa 1 de BC
            LD    A,B               ; Coloca valor de B em A
            OR    C                 ; Soma os bits ligados de C em A
            ; Neste ponto A só vai conter zero se B e C eram zero
            JP    NZ,COPIA          ; Continua cópia enquanto A != zero
```

- 3 instruções fixas, mais 8 instruções que se repetirão 500h (1280) vezes, num total de 10243 instruções. O tempo RISC $T_R = N_I * C_{PI} * P = 10243 * 1,25 * 280 = 3585050$ ns, que é aproximadamente 0,003585 segundos.

Já o segundo, compatível com uma arquitetura CISC, tem um total de:

```
LD    BC, 0500h         ; Número de bytes
LD    HL, 01000h        ; Origem
LD    DE, 02000h        ; Destino
; Aqui começa a rotina de cópia
LDIR                    ; Realiza cópia
```

- 3 instruções fixas e 1 que se repetirá 1280 vezes, num total de 1283 instruções. O tempo CISC $T_C = N_I * C_{PI} * P = 1283 * 10 * 280 = 3592400$ ns, que é aproximadamente 0,003592 segundos.

Comparando os dois tempos, $S = 100 * (T_C - T_R) / T_R$, temos que S = 0,21%, que representa praticamente um empate técnico. Entretanto, é importante ressaltar que a CPU RISC que faz este trabalho é significativamente menor, consome significativamente menos energia e esquenta significativamente menos: o que significa que ela custa significativamente MENOS que a CPU CISC. Por outro lado, significa também que é mais simples e barato aumentar a freqüência de processamento da RISC, para o dobro, por exemplo; isso faria com que, possivelmente, seu custo energético e financeiro fossem similares ao da CPU CISC, mas o aumento de desempenho seria sensível.

Recalculando o TR para o dobro do clock (metade do período), ou seja, P = 140ns, $T_R = N_I * C_{PI} * P = 10243 * 1,25 * 140 = 1792525$ ns, que é aproximadamente 0,001793 segundos. Com este valor, o speedup $S = 100 * (T_C - T_R) / T_R$ se torna: S = 100,41%, ou seja, um ganho de mais de 100% ao dobrar a velocidade.

3.1. Comparação Controversa

Apesar do que foi apresentado anteriormente, esta não passa de uma avaliação teórica. Na prática, este ganho pressupõe as considerações de custo e consumo feitas anteriormente e, para piorar, é dependente de qual programa está sendo avaliado.

Existe um consenso de que, em geral, uma arquitetura RISC é mais eficiente que uma arquitetura CISC; entretanto, é reconhecido que isso não é uma regra e, em alguns casos, uma arquitetura CISC pode ser mais eficiente. Isso porque pouco se escreve em linguagem de máquina nos tempos atuais e o desempenho de um software estará, então, ligado à eficiência de otimizaçã do compilador.

Arquiteturas RISC parecem melhores para os compiladores; programas gerados por bons compiladores para processadores RISC deixam muito pouco espaço para otimização manual. Com relação aos processadores CISC, em geral a programação manual acaba sendo mais eficiente.

Talvez também por esta razão, mas certamente pela compatibilidade reversa, a família de processadores mais famosa da atualidade - a família x86, seja uma arquitetura mista. Tanto os processadores da Intel quanto de sua concorrente AMD são processadores com núcleo

RISC mas com uma "camada de tradução" CISC: o programador vê um processador CISC, mas o processador interpreta um código RISC. O processo é representado muito simplificadamente abaixo:

1) Assembly x86 (CISC) Software

---

2) Controle Microprogramado (Tradutor CISC => RISC) CPU
3) Microprograma RISC
4) Controle Microprogramado/Superscalar (Tradutor RISC => sinais elétricos)
5) Core com operações RISC

O que ocorre é que um programa escrito como:

```
LD    BC, 0500h         ; Número de bytes
LD    HL, 01000h        ; Origem
LD    DE, 02000h        ; Destino
; Aqui começa a rotina de cópia
LDIR                    ; Realiza cópia
```

Ao passar da camada 2 para a 3 seria convertido para algo mais parecido com isso:

```
            LD    BC, 0500h         ; Número de bytes
            LD    HL, 01000h        ; Origem
            LD    DE, 02000h        ; Destino
            ; Aqui começa a rotina de cópia
COPIA:      LD    A, (HL)           ; Carrega byte da origem
            INC   HL                ; Aponta próximo byte de  origem em HL
            LD    (DE), A           ; Coloca byte no destino
            INC   DE                ; Aponta próximo byte de destino em HL
            DEC   BC                ; Decrementa 1 de BC
            LD    A,B               ; Coloca valor de B em A
            OR    C                 ; Soma os bits ligados de C em A
            ; Neste ponto A só vai conter zero se B e C eram zero
            JP    NZ,COPIA          ; Continua cópia enquanto A != zero
```

Esta arquitetura permite que prorgramas antigos (que usam uma linguagem de máquina CISC) sejam capazes de ser executados em um processador RISC; a vantagem disto para a Intel e a AMD é que possibilitou um aumento expressivo de freqüência de operação do core (clock) a um custo mais baixo, além de permitir um uso mais eficiente dos pipelines. O único custo é um delay de uma camada de interpretação/tradução mas que, na prática, pode ser descrito, muito simplificadamente, como se fosse mais um nível de pipeline.

4. BIBLIOGRAFIA

STALLINGS, W. **Arquitetura e organização de computadores**. 5ed. São Paulo: Ed. Pearson Prentice Hall, 2003.

MURDOCCA, M. J; HEURING, V.P. **Introdução à arquitetura de computadores**. S.I.: Ed. Campus, 2000.