Unidade 11: A Unidade de Controle

Prof. Daniel Caetano

Objetivo: Apresentar as funções o mecanismo de atuação da Unidade de Controle.

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

Na aula anterior foi apresentada a Unidade Lógica Aritmética (ULA). Como vimos, a ULA é responsável por executar o processamento "de fato" de um computador, mas a ULA realiza apenas operações individuais. Para que a ULA processe seqüências de instruções, é necessário que algum dispositivo forneça tais instruções, na ordem correta.

Assim, nesta aula será continuado o estudo da Unidade Central de Processamento (CPU), apresentando a Unidade de Controle (UC) e alguns outros registradores importantes.

1. A UNIDADE DE CONTROLE

Uma das melhores analogias existentes entre a ULA e a UC é a analogia da calculadora. Enquanto a ULA é como uma calculadora simples, que executa um pequeno número de operações, a UC é como o operador da calculadora, que sabe onde buscar informações para alimentar a calculadora e também em que ordem estas infomações devem ser repassadas.

Em outras palavras, enquanto a ULA faz "partes" de um trabalho, a UC gerencia a execução destas partes, de forma que um trabalho mais complexo seja executado.

1.1. Algumas Responsabilidades da Unidade de Controle

- Controlar a execução de instruções, na ordem correta: uma vez que a ULA só cuida de executar instruções individuais, a UC tem o papel de ir buscar a próxima instrução e trazê-la para a ULA, no momento correto.

- Leitura da memória principal: Na aula anterior foi visto que a ULA não pode acessar diretamente a memória principal da máquina. A ULA só faz operações sobre os registradores, sendo que as instruções devem ser comandadas diretamente a ela. Assim, a UC

tem o papel não só de buscar as instruções na memória, como também verificar se a instrução exige dados que estejam na memória. Se for o caso, a UC deve recuperar os dados na memória e colocá-los em registradores especiais e, finalmente, solicitar que ULA execute a operação sobre estes valores.

- Escrita na memória principal: Da mesma forma que a leitura, a ULA não pode escrever na memória principal da máquina. Assim, quando for necessário armazenar o resultado de uma operação na memória principal, é tarefa da UC transferir a informação de um registrador para a memória.

- Controlar os ciclos de interrupção: praticamente toda CPU atual aceita sinais de interrupção. Sinais de interrupção são sinais que indicam para a UC que ela deve parar, momentaneamente, o que está fazendo e ir executar uma outra tarefa. As razões para as interrupções são as mais diversas, como o disparo de um timer ou uma placa de rede / model solicitando um descarregamento de seu buffer.

1.2. Rotina de Operação da CPU

Em geral, é possível dizer que uma CPU tem uma seqüência de ações a executar; algumas delas são atividades da ULA, outras da UC. Esta seqüência está apresentada a seguir:

a) Busca de instrução: quando a CPU lê uma instrução na memória.
b) Interpretação de Instrução: quando a CPU decodifica a instrução para saber quais os passos seguintes necessários.
c) Busca de dados: caso seja determinado na interpretação que dados da memória ou periféricos são necessários, a CPU busca estes dados e os coloca em registradores.
d) Processamento de dados: quando a instrução requer uma operação lógica ou aritmética, ela é executada neste instante.
e) Escrita de dados: se o resultado da execução exigir uma escrita na memória ou periféricos, a CPU transfere o valor do registrador para o destino final.
f) Avaliação de Interrupções: após finalizar a execução de uma instrução, a CPU verifica se foi requisitada uma interrupção (Interrupt Request). Se sim, toma as providências necessárias. Se não, volta para **a)**.

Pelas responsabilidades da ULA e da UC, é possível perceber que a atividade **d** é executada pela ULA e todas as outras pela UC.

1.3. Registradores Usados pela UC

Assim como a ULA tem seus registradores especiais (*Acumulador* para armazenar os resultados e o *Flags* para indicar informações sobre a última operação executada), também a UC precisa de alguns registradores para funcionar correamente.

O primeiro deles vem da necessidade da UC saber onde está a próxima instrução a ser executada. Em outras aplavras, ela precisa de um registrador que indique a posição de

memória em que a próxima instrução do programa estará armazenado (para que ela possa realizar a busca de instrução). Este registrador sempre existe, em todos os computadores microprocessados, mas seu nome varia de uma arquitetura para outra. Normalmente este registrador é chamado de **PC**, de Program Counter (Contador de Programa).

Sempre que é iniciado um ciclo de processamento (descrito na seção anterior), uma a UC busca a próxima instrução na memória, na posição indicada pelo PC. Em seguida, o PC é atualizado para apontar para a próxima posição da memória (logo após a instrução), que deve indicar a instrução seguinte.

Bem, como foi visto anteriormente, a UC precisa analisar esta instrução antes de decidir o que fazer em seguida. Por esta razão, costuma existir um registrador especial para armazenar a última instrução lida, chamado **IR**, de Instruction Register (Registrador de Instruções).

Para conseguir ler e escrever dados em memórias e periféricos, a UC também precisa de um contato com o barramento, o que é feito através de registradores especiais, de armazenamento temporário, chamados **MAR**, de Memory Address Register (Registro de Endereço de Memória) e o **MBR**, de Memory Buffer Register (Registro de Buffer de Memória). Assim, quando é preciso escrever na memória (ou em um periférico), a UC coloca o endereço no registrador MAR, o dado no registrador MBR e comanda a transferência pelo barramento de controle. Quando for preciso ler da memória (ou do periférico), a UC coloca o endereço no MAR, comanda a leitura pelo barramento de controle e então recupera o valor lido pelo MBR.

Adicionalmente a estes registradores, as CPUs costumam ter outros registradores que podem facilitar sua operação e mesmo sua programação. Alguns destes são os **registradores de propósito geral**, que servem para armazenar resultados intermediários de processamento, evitando a necessidade de muitas escritas e leituras da memória quando várias operações precisarem ser executadas em seqüência, a fim de transformar os dados de entrada nos dados de saída desejados. O nome destes registradores costuma ser letras diversas como B, C, D...

Existem também os **registradores de pilha**, normalmente com nomes como **SP**, de Stack Pointer (Ponteiro da Pilha) ou **BP** (Base da Pilha), que servem para que uma pilha seja usada pelo processador, na memória. Em essência, é onde o endereço de retorno é armazenado, quando um desvio é feito em linguagem de máquina; afinal, é preciso saber para onde voltar após a realização de uma chamada de subrotina. A pilha que o processador fornece pode ser usada com outros objetivos, como passagem de parâmetros etc.

Quase todas as arquiteturas fornecem os **registradores de índices**, que são registradores que permitem acessar, por exemplo, posições de uma matriz. Ele guarda uma posição de memória específica e existem instruções que permitem acessar o n-ésimo elemento a partir daquela posição. Seus nomes variam muito de uma arquitetura para outra, como **IX**, de IndeX, **SI**, de Source Index (Índice Fonte) ou ainda **DI**, de Destination Index (Índice Destino).

Arquiteturas com segumento possuem ainda os **registradores de segmento**, que definem o endereço "zero" da memória para um determinado tipo de informação. A arquitetura x86, por exemplo, possui diversos registradores deste tipo: **CS**, de Code Segment (Segmento de Código), **DS**, de Data Segment (Segmento de Dados), **SS**, de Stack Segment (Segmento de Pilha) e **ES**, de Extra data Segment (Segmento de Dados Extra). Quando estes segmentos existem, os endereços usados nos índices, contador de programa e outros são "somados" com os endereços do segmento para que a posição real na memória seja calculada. Por exemplo:

SS = 10000h => Endereço do segmento da pilha
SP = 1500h => Endereço da pilha (dentro do segmento)
Endereço real da pilha = SS + SP = 10000h + 1500h = 11500h

Isso permite que um programa possa rodar em qualquer parte da memória, mesmo que ele tenha sido criado para ser executado no endereço 0h: basta eu indicar no registrador CS o endereço inicial de carregamento deste programa e, para todas as instruções deste programa, vai ser como se ele estivesse no endereço 0h. A CPU, com os registradores de sergento, são responsáveis pela tradução do endereço "virtual" para o endereço real.

Vale lembrar que cada arquitetura tem nomes distintos para estes resigtradores e em algumas delas existem ainda outros; além disso, alguns destes registradores até existem em algumas arquiteturas, mas não são visíveis para o programador, isto é, o registrador está lá e é usado pela CPU, mas o programador não tem acesso direto a eles (embora muitas vezes tenha acesso indireto, como sempre ocorre com os registradores MAR e MBR).

2. CICLO DE INSTRUÇÃO

Nas seções anteriores já foi descrito o ciclo de instrução, que é a seqüência de passos que a UC segue até que uma instrução seja executada. Nesta parte será apresentado um diagrama genérico que mostra todos os passos que um processador comum executa para executar suas instruções. Os principais subciclos são os de busca de instruções, busca de dados, execução e interrupção.

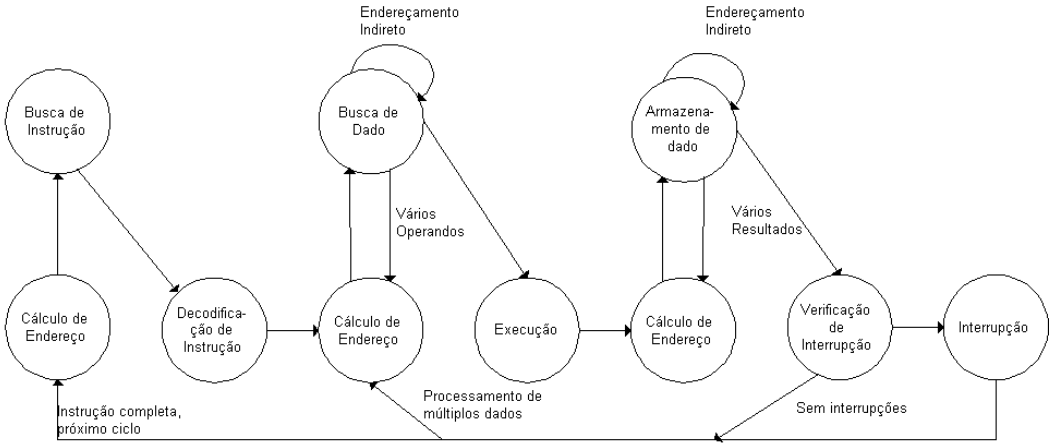

Figura 1 - Diagram de transição de estados do ciclo de instrução (STALLINGS, 2003)

Como é possível ver, o primeiro passo é a busca de instrução, onde a UC coloca o valor do PC no MAR, comanda leitura da memória e recebe o dado (que neste caso é uma instrução) pelo MBR, que em seguida copia para o IR.

Em seguida, a UC decodifica a instrução, avaliando se há a necessidade de busca de dados adicionais (por exemplo, se for uma instrução do tipo "ADD A,B", nenhum dado precisa ser buscado. Se houver a necessidade, o próximo passo é a busca do dado, onde a UC realiza o mesmo processo da leitura da instrução, mas agora para a leitura do dado. Esse processo é repetido até que todos os dados necessários tenham sido colocados em registradores.

O passo seguinte é a execução, onde a UC meramente comanda a ULA para executar a operação relevante. A ULA devolve um resultado em um registrador. Se este dado precisar ser armazenado externamente à CPU, a UC cloca este dado na MBR e o endereço destino na MAR e comanda a escrita, usando o barramento de controle. Se houver mais de um dado a armazenar, este ciclo é repetido.

Finalmente, a UC verifica se há *requisição de interrupção pendente*. Se houver, ela executa o ciclo da interrupção (que varia de arquitetura para arquitetura). Caso contrário, o funcionamento prossegue, com o cálculo do novo endereço de instrução e o ciclo recomeça.

3. A PIPELINE

A idéia de pipeline é a mesma da produção em série em uma fábrica: "quebrar" a produção de alguma tarefa em pequenas tarefas que podem ser executadas paralelamente. Isso significa que vários componentes contribuem para o resultado final, cada um executando sua parte.

Como foi possível ver pela seção anterior, existem vários passos em que a execução de uma instrução pode ser dividida. A idéia, então, é que cada um destes passos seja executado independentemente e por uma parte diferente da CPU, de forma que o processamento ocorra mais rapidamente.

Mas como isso ocorre? Imagine o processo explicado na seção 2. Nos momentos em que a comunicação com a memória é feita, por exemplo, a ULA fica ociosa. Nos momentos em que a ULA trabalha, a comunicação com a memória fica ociosa. Certamente o processamento linear não é a melhor forma de aproveitar os recursos.

Imagine então que emos duas grandes etapas (simplificando o processo explicado anteriormente): a etapa de busca e a etapa de execução. Se tivermos duas unidades na UC, uma para cuidar da busca e outra para cuidar da execução, enquanto a execução de uma instrução está sendo feita, a seguinte já pode estar sendo buscada! Observe as seqüênciasa seguir.

	SEM pipeline		COM pipeline	
Seqüência no Tempo	Busca	Execução	Busca	Execução
0	I1	-	I1	-
1	-	I1	I2	I1
2	I2	-	I3	I2
3	-	I2	I4	I3
4	I3	-	I5	I4

Observe que no tempo que foram executadas 2 instruções sem pipeline, com o pipeline de 2 níveis foram executadas 4 instruções. Entretanto, isso é uma aproximação grosseira, pois os tempos de execução de cada um destes estários é muito diferente, sendo que o aproveitamento ainda não é perfeito. Para um bom aproveitamente, precisamos dividir as tarefas em blocos que tomem mais ou menos a mesma fatia de tempo. Este tipo específico de pipeline é chamado de "Prefetch" (leitura antecipada).

Se quebrarmos, por exemplo, a execução em 6 etapas: Busca de Instrução (BI), Decodificação de Instrução (DI), Cálculo de Operandos (CO), Busca de Operandos (BO), Execução da Instrução (EI) e Escrita de Operando (EO), termos etapas mais balanceadas com relação ao tempo gasto. Observe na tabela abaixo o que ocorre:

	SEM Pipeline						COM Pipeline					
T	BI	DI	CO	BO	EI	EO	BI	DI	CO	BO	EI	EO
0	I1	-	-	-	-	-	I1	-	-	-	-	-
1	-	I1	-	-	-	-	I2	I1	-	-	-	-
2	-	-	I1	-	-	-	I3	I2	I1	-	-	-
3	-	-	-	I1	-	-	I4	I3	I2	I1	-	-
4	-	-	-	-	I1	-	I5	I4	I3	I2	I1	-
5	-	-	-	-	-	I1	I6	I5	I4	I3	I2	I1
6	I2	-	-	-	-	-	I7	I6	I5	I4	I3	I2
7	-	I2	-	-	-	-	I8	I7	I6	I5	I4	I3
8	-	-	I2	-	-	-	I9	I8	I7	I6	I5	I4
9	-	-	-	I2	-	-	I10	I9	I8	I7	I6	I5
10	-	-	-	-	I2	-	I11	I10	I9	I8	I7	I6
11	-	-	-	-	-	I2	I12	I11	I10	I9	I8	I7
12	I3	-	-	-	-	-	I13	I12	I11	I10	I9	I8

Basicamente, no tempo que foram executadas 2 instruções sem pipeline, foram executadas 8 instruções com pipeline. É claro que o tempo de execução de uma instrução sem pipeline neste caso de 6 estágios é aproximadamente o mesmo tempo de execução da mesma instrução sem pipeline no caso com 2 estágios; Isso ocorre porque, obviamente, cada um dos 6 estágios deste caso toma um tempo muito menor que cada um dos dois estágios do modelo anterior.

Bem, mas se o número de estágios aumenta o desempenho, porque não usar o máximo possível? Por algumas razões. Uma delas é que, a partir de um determinado número de estágios a quebra pode acabar fazendo com que dois estágios passem a gastar mais tempo de

execução do que o estágio original que foi dividido. Mas esta não é a razão fundamental: existe um gargalo mais evidente no sistema de pipelines: ele pressupõe que as instruções são independentes entre si; assim, para que o pipeline tenha o desempenho apresentado, uma instrução a ser executada não pode depender do resultado das instruções anteriores.

Quando uma instrução depende do resultado das anteriores, teremos alguns estágios "esperando" a execução da outra instrução terminar para que a "cadeia de montagme possa ter prosseguimento. Vamos dar um exemplo. Imagine que A instrução I2 dependa do resultado da instrução I1 para ser executada. Então, o que vai acontecer no pipeline é descrito a seguir:

	COM Pipeline					
T	BI	DI	CO	BO	EI	EO
0	I1	-	-	-	-	-
1	I2	I1	-	-	-	-
2	I3	I2	I1	-	-	-
3	I4	I3	I2	I1	-	-
4	I5	I4	I3	I2	I1	-
5	I5	I4	I3	I2	-	I1
6	I6	I5	I4	I3	I2	-
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10	I10	I9	I8	I7	I6	I5
11	I11	I10	I9	I8	I7	I6
12	I12	I11	I10	I9	I8	I7

Observe que, comparando com o quadro anterior, uma instrução a menos foi processada. Isso é pior ainda quando uma instrução do tipo "desvio condicional" precisa ser interpretada; isso porque a posição de leituras da próxima instrução vai depender da execução de uma instrução. Neste caso, quando ocorre este tipo de desvio, o pipeline é esvaziado e perde-se uma boa parte do desempenho.

Ocorre que a chance destes "problemas" acontecerem e os atrasos causados por eles aumentam com o número de níveis do pipeline. Desta forma, um número excessivo de níveis de pipeline podem acabar por degradas o desempenho, além de fazer com que uma CPU aqueça mais e mais. Para entender isso (o aquecimento) pense em uma fábrica: quanto mais funcionários, mais confusa é a movimentação dentro da fábrica. Nos circuitos, os níveis de pipeline fazem o papel dos funcionários da linha de montagem e, quanto maior número de movimentações internas de sinais, maior é o calor gerado.

A subdivisão excessiva dos pipelines foi o que matou a linha Intel Pentium IV, que foi abandonada. A Intel precisou retroceder sua tecnologia à do Pentium M (Pentium III móvel) e continuar o projeto em outra direção, com menos níveis de pipeline, o que deu origem aos processadores Pentium D, Core 2 Duo e os mais atuais Core i.

4. MICROPROGRAMAÇÃO

Apesar de os estágios de execução de uma instrução - como a busca de instrução, decodificação de instrução etc. - parecerem simples, sua implementação com circuitos lógicos é bastante complexa e, em geral, impõe uma grande dificuldade para modificações e expansões no conjunto de instruções.

Uma proposta para solucionar este problema foi a **microprogramação**. A microprogramação foi proposta ainda na década de 1950, mas, devido ao custo da tecnologia, que exigia memória rápida e barata, só na década de 1960 é que foi implementada em sistemas comerciais, tendo sido o IBM Série 360 o primeiro a usar um processador com essa característica.

Mas o que é a microprogramação?

Vamos relembrar, primeiramente, o que ocorre dentro do estágio de execução de uma instrução. Cada estágio de execução de busca de instrução é executado por várias tarefas e, assim que elas são terminadas, passa-se ao próximo estágio de execução.

Por exemplo: o estágio de busca de instrução pressupõe:

a) Configurar o barramento de endereços (MAR) com o valor do registrador PC;
b) Configurar o barramento de controle para leitura de memória;
c) Aguardar um intervalo referente ao tempo de resposta da RAM
d) Ler o valor do barramento de dados (MBR) para o IR
e) Remover os sinais do barramento de endereços e controle

Só depois disso é que a CPU parte para o próximo estágio. É possível, ainda, quebrar cada uma dessas tarefas em várias outras que envolvem apenas sinais elétricos e que podem ser executadas paralelamente. **Essas** pequenas tarefas que envolvem apenas sinais elétricos e que podem ser executasdas paralelamente são chamadas de **microinstruções**, e são compartilhadas para a execução de diversas (macro)instruções diferentes.

No fundo, é como se existisse uma espécie de Unidade de Controle dentro da Unidade de Controle, que "quebra" as (macro)instruções em tarefas muito menores e mais simples - como já dito, envolvendo apenas sinais elétricos e decisões lógicas mais elementares - e as executa para que, no conjunto, resultem no resultado da macroinstrução. Assim, cada instrução que a UC tem que executar é, na prática, o resultado de um programa gravado na CPU, chamado de **microprograma**. O conjunto de todos os microprogramas que formam o conjunto de instruções é o **firmware** da CPU.

5. 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.