Universidade do Sul de Santa Catarina – Unisul
UC Sistemas Computacionais e Segurança
Professora Márcia Giraldi

Arquiteturas em Pipeline

1. Organização de uma Máquina Pipeline

A maioria das arquiteturas e processadores projetados depois de 1990 possuem uma organização chamada pipeline. A tradução literal de "pipeline" seria "linha de dutos", uma tradução mais aproximada do significado que este termo assume em arquiteturas de computadores seria "linha de montagem". Organizar a execução de uma tarefa em pipeline significa dividir a execução desta tarefa em estágios sucessivos, exatamente como ocorre na produção de um produto em uma linha de montagem.

Em um microprocessador, a execução de uma instrução é tipicamente dividida em cinco estágios: *busca da instrução*, *decodificação da instrução*, *execução da instrução*, *acesso à memória* e *escrita de resultados*. Esta divisão da execução em múltiplos estágios aumenta o desempenho do processador pois é possível ter uma instrução diferente executando em cada estágio ao mesmo tempo.

A passagem de uma instrução através dos diferentes estágios do pipeline é similar à produção de um carro em uma linha de montagem. Existem algumas regras muito importantes que devem ser seguidas quando se projeta uma arquitetura em pipeline:

1. Todos os estágios devem ter a mesma duração de tempo.
2. Deve-se procurar manter o pipeline cheio a maior parte do tempo.
3. O intervalo mínimo entre o término de execução de duas instruções consecutivas é igual ao tempo de execução do estágio que leva mais tempo.
4. Dadas duas arquiteturas implementadas com a mesma tecnologia, a arquitetura que é construída usando pipeline não reduz o tempo de execução de instruções, mas aumenta a freqüência de execução de instruções (*throughput*).

A execução que ocorre em cada um dos estágios é descrita a seguir:

1) *Busca de Instrução*: O contador de programa é usado para buscar a próxima instrução. As instruções são usualmente armazenadas em uma memória cache que é lida durante o estágio de busca.

2) *Decodificação de Instruções e Busca de Operandos*: O código da instrução é buscado, os campos são analisados e os sinais de controle são gerados. Os campos das instruções referentes aos registradores são usados para ler os operandos no banco de registradores.

3) ***Execução de Instruções****:* A operação especificada pelo código de operação é executada. Para uma instrução de acesso à memória, o endereço efetivo da memória é calculado.

4) ***Acesso à Memória****:* Dados são carregados da memória ou escritos na memória. Uma cache de dados é tipicamente utilizada.

5) ***Escrita de Resultados****:* O resultado da operação é escrito de volta no banco de registradores.

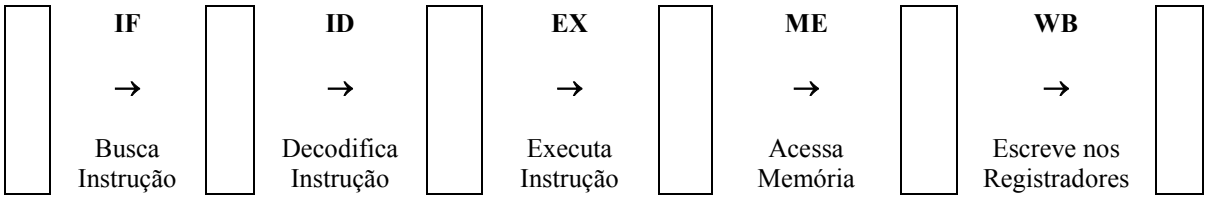

Fig. 4.1. Estrutura de uma máquina pipeline.

Cada um dos retângulos mostrados na fig. 4.1 representa um banco de *"flip-flops"* que são elementos de memória utilizados para armazenar os resultados no final de cada estágio do pipeline. Os pipelines utilizados em microprocessadores são síncronos, portanto um sinal de relógio não mostrado na fig. 4.1 habilita o elemento de memória a "passar" seus resultados para o estágio seguinte. Como existe um único relógio para comandar o pipeline, o tempo gasto em todos os estágios do pipeline é idêntico e não pode ser menor o que o tempo gasto no estágio mais lento. Este tempo gasto em cada estágio é o período do *clock* utilizado para comandar o pipeline e ele determina a velocidade de execução das instruções.

A *latência* de um pipeline é o tempo necessário para uma instrução atravessar todo o pipeline. Portanto para calcular o tempo de latência de uma máquina com pipeline basta multiplicar o período do *clock* pelo número de estágios no pipeline. A latência é importante apenas para se determinar quanto tempo a execução da primeira instrução leva para ser completada.

	Estágio do pipeline onde a instrução se encontra							
Ciclo de Clock	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Instrução 1	IF	ID	EX	ME	WB			
Instrução 2		IF	ID	EX	ME	WB		
Instrução 3			IF	ID	EX	ME	WB	
Instrução 4				IF	ID	EX	ME	WB

Fig. 4.2. Execução de uma seqüência de instruções num pipeline.

2. Geração de Bolhas no Pipeline

Uma "bolha" em um pipeline consiste em uma seqüência de um ou mais períodos de *clock* em que um estágio do pipeline está vazio. Se um estágio do pipeline estiver vazio no ciclo de *clock* n, consequentemente o estágio seguinte estará vazio no ciclo de *clock* n+1.

Desta forma bolhas formadas na entrada de um pipeline propagam-se através do pipeline até desaparecerem no último estágio. Situações que geram bolhas em pipelines incluem:

1) a execução de instruções de desvio,

2) o atendimento de interrupções,

3) o tratamento de exceções,

4) o retardo na execução de instruções devido a dependências existentes com instruções que a precedem.

No caso de atendimento de exceções e interrupções não existem muitas técnicas efetivas para minorar o problema da formação de bolhas no pipeline, pois estas ocorrências são bastante imprevisíveis.

No caso de execução de desvios condicionais a formação de bolhas pode ser reduzida através da utilização de predição de ocorrência e desvio.

No caso de dependências, a formação de bolhas pode ser minorada através do reordenamento de instruções. Vamos examinar estas técnicas nas próximas seções.

3 Previsão de Desvios

O problema introduzido por desvios condicionais em arquiteturas organizadas em pipeline é que quando o desvio é decodificado na unidade de decodificação de instruções é impossível decidir se o desvio será executado ou não. Isto é, não se pode determinar *a priori* qual a próxima instrução a ser executada. Existem duas possibilidades:

a) a condição que determina o desvio é falsa e o desvio não é executada, neste caso, a próxima instrução a ser executada é a instrução seguinte à instrução de desvio no programa;

b) condição é verdadeira e o endereço da próxima instrução a ser executada é determinado pela instrução de desvio.

Como determinar qual a próxima instrução a ser buscada quando uma instrução de desvio condicional é decodificada?

A forma mais simples e menos eficaz de tratar um desvio condicional consiste em simplesmente paralisar a busca de instruções quando uma instrução de desvio condicional é decodificada. Com esta forma de tratamento de desvio garantimos que todo desvio condicional gerará uma bolha no pipeline, pois a busca de instrução ficará paralisada até que se possa decidir se o desvio será executado ou não. Esta técnica só deve ser utilizada quando o custo de buscar e depois ter que descartar a instrução errada é muito alto.

Uma outra forma é tentar prever o que vai acontecer com o desvio. Neste caso a previsão pode ser estática ou dinâmica. A previsão estática é aquela em que dada uma instrução de desvio em um programa nós vamos sempre fazer a mesma previsão para aquela instrução. Observe que podemos fazer previsões diferentes para diferentes instruções de desvio no mesmo programa. Na previsão dinâmica podemos mudar a previsão para uma mesma instrução de desvio à medida que o programa é executado.

3.1 Previsão Estática

A forma mais simples de previsão estática é aquela em que a mesma previsão é feita para um dado desvio condicional. Esta técnica de previsão é simples de implementar e pode tomar duas formas:

- ***Os desvios condicionais nunca ocorrem:*** Assumindo que os desvios condicionais nunca ocorrem, simplesmente se continua o processamento normal de instruções incrementando o PC e buscando a instrução seguinte. Se for determinado mais tarde que o programa deve desviar as instruções buscadas terão que ser descartadas e os efeitos de quaisquer operações realizadas por elas devem ser anulados.

- ***Os desvios condicionais sempre ocorrem:*** Assumindo que os desvios condicionais sempre ocorrem é necessário calcular o endereço de desvio muito rapidamente já na Unidade de Decodificação de Instrução para dar tempo de buscar a nova instrução no endereço especificado pela instrução de desvio.

Algumas observações feitas por projetistas de computadores após analisar diversos programas indicam que um grande número de desvios ocorre no final de laços de programa. Se um laço de programa for executado *n* vezes, o desvio que se encontra no final do laço irá ocorrer *n* vezes e não ocorrer uma vez no final do laço. Alguns programadores também observaram que desvios condicionais originados por comandos IF em linguagens de ao nível tendem a não ocorrer. Portanto, existem evidências de que seria desejável possuir-se a capacidade de fazer previsão estática diferenciada para diferentes instruções de desvio em um mesmo programa (ex: *beq* sempre ocorre, *bne* nunca ocorre, etc...).

Uma solução para esta situação consiste em adicionar um bit no código de instruções de desvio condicional para informar o hardware se aquele desvio será provavelmente executado ou provavelmente não executado. Este bit deverá ser especificado pelo compilador. Como a determinação de que a previsão será de execução ou não do desvio é feita em tempo de compilação, este método de previsão também é estático.

3.2 Previsão Dinâmica

Na previsão dinâmica de desvios condicionais uma mesma instrução de desvio pode receber uma previsão de ser ou não executada em diferentes momentos do programa. Uma solução comum é criar uma tabela de desvios em hardware. Esta tabela pode ser gerenciada na forma de uma cache (Content Addressable Memory - CAM). A cada vez que uma instrução de desvio é executada ela é adicionada à tabela e um bit é setado ou resetado para indicar se o desvio foi executado ou não. Na tabela também é registrado o endereço para qual o desvio foi realizado. Desta forma na próxima vez que a mesma instrução de desvio for decodificada, esta tabela é consultada e a previsão feita é de que a mesma coisa (execução ou não execução do desvio) vai ocorrer de novo. Se a previsão for de execução do desvio o endereço no qual a nova instrução deve ser buscada já se encontra nesta tabela.

4 Processamento de Exceções

A ocorrência de exceções em um computador é imprevisível e, portanto, numa máquina em pipeline uma exceção normalmente resulta na formação de bolhas. Uma complicação adicional é que como existem várias instruções em diferentes estágios de execução ao mesmo tempo, é difícil decidir qual foi a instrução que causou a geração da exceção. Uma solução para minorar este problema é categorizar as exceções de acordo com o estágio do pipeline em que elas ocorrem. Por exemplo, uma instrução ilegal só pode ocorrer na unidade de decodificação de instruções (ID) e uma divisão por zero ou uma exceção devida a overflow só pode ocorrer na unidade de execução (EX).

Algumas máquinas com arquitetura em pipeline são ditas ter exceções imprecisas. Nestas máquinas não é possível determinar qual a instrução que causou a exceção.

Uma outra complicação nas arquiteturas em pipeline é o aparecimento de múltiplas exceções no mesmo ciclo de *clock*. Por exemplo, uma instrução que causa erro aritmético pode ser seguida de uma instrução ilegal. Neste caso a unidade de execução (EX) gerará uma exceção por erro aritmético e a unidade

de decodificação (ID) gerará uma exceção por instrução ilegal, ambas no mesmo ciclo de clock. A solução para este problema consiste em criar uma tabela de prioridade para o processamento de exceções.

5 Bolhas Causadas por Dependências

Uma outra causa de formação de bolhas em arquiteturas com pipeline são as dependências existentes entre instruções num programa. Para ilustrar este problema, vamos considerar o exemplo de pseudo programa assembler apresentado a seguir. Este pseudocódigo foi escrito com uma simbologia muito similar à simbologia utilizada pela Motorola para os processadores da família MC68000. O trecho de programa abaixo permuta os valores armazenados nas posições $800 e $1000 da memória.

inst.:

```
A    MOVE  $800, D0;     copia o valor do endereço $800 no registrador D0
B    MOVE  $1000, D1;    copia o valor do endereço $1000 no registrador D1
C    MOVE  D1, $800;     copia o valor do registrador D1 no endereço $800
D    MOVE  D0, $1000;    copia o valor do registrador D0 no endereço $1000
```

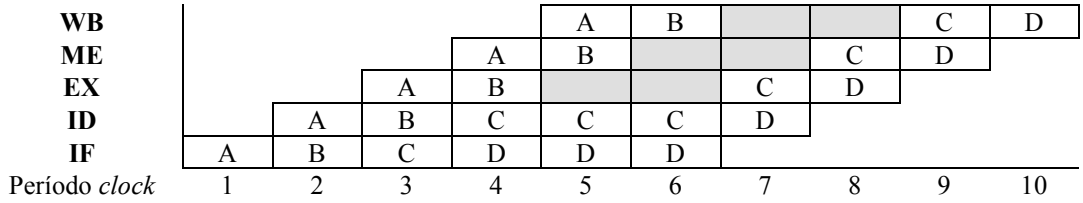

Fig. 4.3. Formação de bolha devido à dependência entre instruções.

Observe na fig. 4.3 que no período de *clock* 6 quando a instrução C deveria executar no estágio de escrita na memória (ME) para escrever o valor de D1 no endereço $800, o valor lido do endereço $1000 ainda não está disponível em D1 pois ele só será escrito quando a instrução B tiver executado no estágio de escrita nos registradores (WB), o que ocorre também no período de *clock* 6. Portanto a execução da instrução C necessita ser atrasada por dois ciclos de relógio, gerando uma bolha no pipeline.

Uma forma simples de resolver este problema consiste em reordenar as instruções no programa, conforme ilustrado a seguir.

inst.:

```
B    MOVE  $1000, D1;    copia o valor do endereço $1000 no registrador D1
A    MOVE  $800, D0;     copia o valor do endereço $800 no registrador D0
C    MOVE  D1, $800;     copia o valor do registrador D1 no endereço $800
D    MOVE  D0, $1000;    copia o valor do registrador D0 no endereço $1000
```

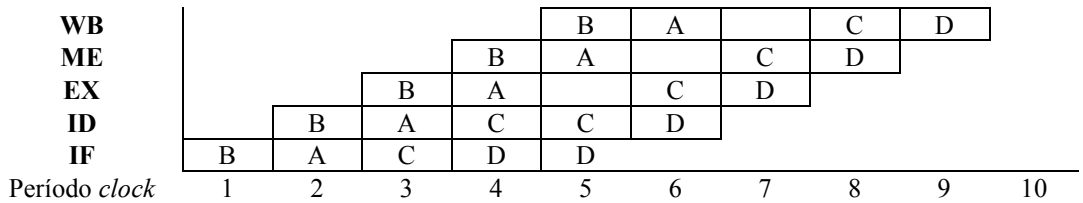

Fig. 4.4. Minimização de bolhas por reordenamento de instruções.

A fig. 4.4 ilustra que um simples reordenamento de instruções é suficiente neste caso para minimizar a bolha gerada pela dependência. Observe que o tempo total para a execução da seqüência de instruções foi reduzido de 1 ciclo de *clock*.

Existem situações em que um simples reordenamento de instruções não é suficiente para minimizar (ou eliminar) bolhas causadas por dependências. Considere o programa abaixo que realiza a inversão de uma tabela de palavras localizada entre os endereços $1000 e $9000 da memória.

```
A           MOVEA  #$1000, A0;     inicializa A0
B           MOVEA  #$9000, A1;     inicializa A1
C     LOOP  MOVE (A0), D0;         copia em D0 dado apontado por AO
D           MOVE  (A1), D1;        copia em D1 dado apontado por A1
E           MOVE  D0, (A1);        copia valor em D0 no end. indicado em A1
F           MOVE  D1, (A0);        copia valor em D1 no end. indicado em A0
G           ADDQ  #2, A0;          incrementa o valor de A0
H           SUBQ  #2, A1;          decrementa o valor de A1
I           CMPA  A0, A1;          compara os endereços em A0 e A1
J           BGT  LOOP;             enquanto A1 é maior que A0, continua
K           MOVE #$500, A0;        inicia outro procedimento
L           MOVE #$17000, A1;      inicia outro procedimento
```

WB						-	-	**C**	**D**		-	-	**G**	**H**			-	-
ME					-	-	**C**	**D**		**E**	**F**	**G**	**H**				-	-	
EX				A	B	**C**	**D**		**E**	**F**	**G**	**H**				**I**	**J**		
ID			A	B	**C**	**D**	**E**	**E**	**F**	**G**	**H**	**I**	**I**	**I**	**J**	K		C
IF		A	B	**C**	**D**	**E**	**F**	**F**	**G**	**H**	**I**	**J**	**J**	**J**	K	L	C	D
Clk	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	

Fig. 4.5. Execução do programa com loop.

Conforme ilustrado na fig. 4.5, o laço do programa que efetivamente transfere os dados para fazer a inversão da tabela de palavras necessita de 12 ciclos de *clock* para executar. Para inverter uma tabela com *n* palavras é necessário executar este laço *n/2* vezes. Portanto o número de ciclos de *clock* necessário para realizar a inversão da tabela é *6n* (supondo que cada instrução é executada em apenas 1 ciclo de *clock*).

A fig. 4.5 indica a formação de 3 bolhas dentro da execução do loop:

1) a primeira bolha aparece no ciclo de *clock* 7 quando a instrução E tem que esperar até o fim do WB da instrução C. Em outras palavras, ela tem que aguardar até que o novo valor do registrador D0 produzido pela instrução C seja escrito no estágio de escrita em registradores WB, o que só vai ocorrer no ciclo de *clock* 6. Esta bolha tem o tamanho de 1 ciclo de *clock*.

2) a segunda bolha, com tamanho de 2 ciclos de *clock*, aparece no ciclo de *clock* 12 quando a instrução I tem que esperar até o fim do WB da instrução H.

3) a terceira bolha aparece no final do loop devido ao uso de uma previsão de que o desvio não será executado, o que faz com que o processador busque as instruções K e L que não serão executadas. Quando é

determinado que estas instruções não serão executadas, elas são eliminadas do pipe, gerando uma bolha de dois ciclos de *clock*.

O programa foi reescrito para que fosse eliminada a dependência entre as instruções de atualização dos endereços em A0 e A1 e a instrução de comparação. Para que isto ocorresse a atualização dos valores dos registradores foi feita no início do laço e a inicialização de A0 e A1 foi alterada apropriadamente. O novo programa é apresentado a seguir.

```
A           MOVEA  #$0FFE, A0;    inicializa A0
B           MOVEA  #$9002, A1;    inicializa A1
C    LOOP   ADDQ  #2, A0;    incrementa o valor de A0
D           SUBQ  #2, A1;    decrementa o valor de A1
E           MOVE  (A0), D0;    copia em D0 dado apontado por AO
F           MOVE  (A1), D1;    copia em D1 dado apontado por A1
G           MOVE  D0, (A1);    copia valor em D0 no end. indicado em A1
H           MOVE  D1, (A0);    copia valor em D1 no end. indicado em A0
I           CMPA  A0, A1;    compara os endereços em A0 e A1
J           BGT  LOOP;    enquanto A1 é maior que A0, continua
K           MOVE  #$500, A0;    inicia outro procedimento
L           MOVE  #$17000, A1;    inicia outro procedimento
```

Para eliminar a bolha no final do loop, as instruções foram reordenadas dentro do laço e a previsão estática do desvio foi alterada para uma previsão de que o desvio sempre ocorre. Assim obtemos a execução mostrada na fig. 4.6.

WB					-	-	**C**	**D**		**E**	**F**		**-**	**-**	**-**	-	C
ME				-	-	**C**	**D**		**E**	**F**		**G**	**H**	**-**	**-**	C	D
EX			A	B	**C**	**D**		**E**	**F**		**G**	**H**	**I**	**J**	C	D	
ID		A	B	**C**	**D**	**E**	**E**	**F**	**G**	**G**	**H**	**I**	**J**	C	D	E	E
IF	A	B	**C**	**D**	**E**	**F**	**F**	**G**	**H**	**H**	**I**	**J**	C	D	E	F	F
Clk	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**

Fig. 4.6. Eliminação de bolhas dentro do loop.

Conforme mostrado na fig. 4.6, a modificação da previsão estática de desvio para desvio executado e a eliminação de bolha por reordenação de instruções dentro do laço causaram uma redução de 30 no número de ciclos de *clock* necessários para executar o laço. Com um laço de 8 ciclos de *clock*, uma tabela com *n* palavras pode ser invertida em 4n ciclos de *clock*. Observe que a instrução C não faz nada e foi inserida apenas como uma reserva de espaço de tempo para que o laço pudesse começar ordenadamente. A inserção da instrução C é necessária por causa da dependência existente entre a instrução B e a instrução D.

6. Máquinas Superpipeline, Superescalar

O período de clock de uma máquina pipeline é determinado pelo estágio que leva maior tempo para ser executado. Uma técnica utilizada para acelerar uma máquina pipeline é subdividir os estágios mais lentos

em subestágios de menor duração. Uma máquina com um número de estágios maior do que cinco é chamada de superpipeline. A fig. 4.7 ilustra uma organização superpipeline obtida pela divisão do estágio de busca de instruções (IF) em dois subestágios e do estágio de acesso à memória em três subestágios.

IF	**IF**	**ID**	**EX**	**ME**	**ME**	**ME**	**WB**	
	IF	**IF**	**ID**	**EX**	**ME**	**ME**	**ME**	**WB**

Fig. 4.7. Arquitetura Superpipeline.

Uma outra técnica para acelerar o processamento em máquinas pipeline é a utilização de múltiplas unidades funcionais que podem operar concorrentemente. Numa arquitetura deste tipo múltiplas instruções podem ser executadas no mesmo ciclo de clock. É necessário realizar análise de dependências para determinar se as instruções começadas ao mesmo tempo não possuem interdependências.

IF	**ID**	**EX**	**ME**	**WB**
IF	**ID**	**EX**	**ME**	**WB**

Fig. 4.8. Arquitetura Superescalar.

A fig. 4.8. ilustra uma arquitetura superescalar com dois pipelines que podem operar em paralelo. Uma preocupação que surge com a implementação de máquinas superescalar é a possibilidade de término de execução *fora de ordem*. Isto ocorreria quando suas instruções são iniciadas ao mesmo tempo em dois pipelines e uma bolha surge em um deles causando que uma instrução leve mais tempo do que a outra. Uma solução adotada para este problema consiste em criar um *buffer de reordenação* que armazena o resultado das instruções até que os resultados possam ser escritos na ordem correta.

A maioria dos processadores RISC são arquiteturas superescalar e superpipelined. Isto é eles possuem mais de cinco estágios no pipeline devido à subdivisão de alguns estágios e eles possuem múltiplas unidades funcionais em cada estágio. Uma configuração típica é o uso de quatro unidades funcionais em cada estágio.

Referência:

PATTERSON, David A. & Hennessy, John L. **Organização e Projeto de Computadores: a interface Hardware/Software.** Rio de Janeiro: Editora LTC, 2 ed. 2000