

Pipeline

Otimização da Previsão de Desvios

Branch Prediction Optimization

Rene Pegoraro

Baseado em:
 Intel® 64 and IA-32 Architectures
 Optimization Reference Manual, Intel, 2015
 e
 T. Shanley, Pentium Pro and Pentium II System Architecture,
 Addison-Wesley Professional, 1998

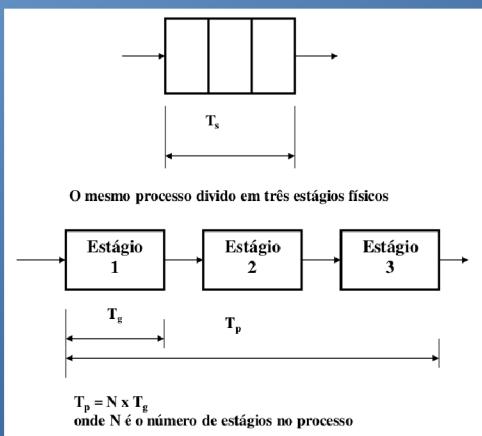
1

Pipelining

- Quase todos processos podem ser decomposto em estágios.
- Processamento serial é a execução de todos estágios de um processo antes de iniciar o primeiro estágio do próximo processo.
- Pipelining executa estágios de processos repetitivos de forma concorrente.

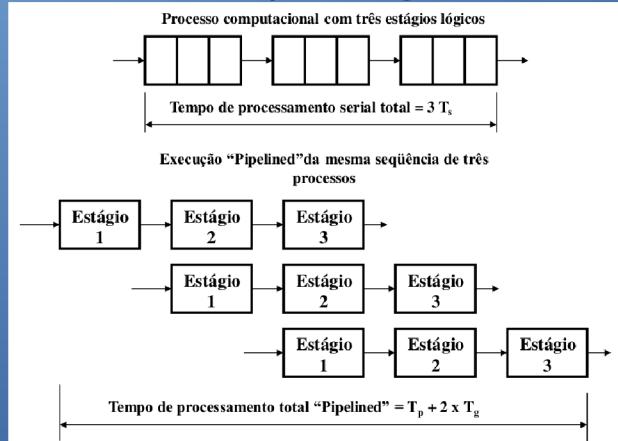
2

Pipelining



3

Pipelining



4

Pipelining

- Para demonstrar o funcionamento do Pipeline, consideramos uma arquitetura RISC com instruções
- Instruções tipo registro-registro (2 fases)
 - I - Busca da Instrução (Fetch)
 - E - Execução – Operação da ULA com entrada ou saída para registradores
- Operações de “Store” e “Load” (3 fases)
 - I - Busca de Instruções
 - E - Execução. Calcula endereço de memória
 - D - Memória. Operações registro → memória ou memória → registro

5

Pipelining

- Temporização de execução sequencial

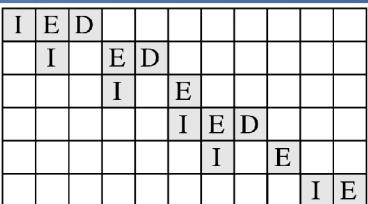
Load rA ← M	I	E	D							
Load rB ← M				I	E	D				
Add rC ← rA + rB							I	E		
Store M ← rC								I	E	
Branch X									I	E

6

Pipelining

- Temporização de um Pipeline de três estágios

Load rA \leftarrow M
 Load rB \leftarrow M
 Add rC \leftarrow rA + rB
 Store M \leftarrow rC
 Branch X



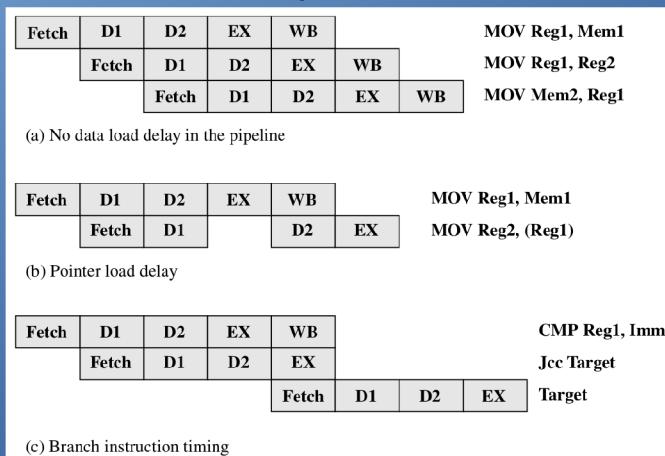
7

Intel 80486 Pipelining

- Cinco estágios:
 - Fetch (F): As instruções são obtidas do cache ou da memória externa e colocadas em um dos dois 16-byte prefetch buffers.
 - Decodifica 1 (D1): Toda a informação do código de operação e do modo de endereçamento é decodificada neste estágio.
 - Decodifica 2 (D2): Este estágio expande cada opcode em sinais de controle para a ALU.
 - Executa (EX): Esta etapa inclui operações de ALU, acesso a cache e atualização de registro.
 - Write Back (WB): Este estágio, se necessário, atualiza registros e flags de status modificados durante o estágio de execução anterior.

8

Exemplo de instruções 80486 e o Pipeline



9

Otimizações de Desvios

- Otimizações de desvios têm um impacto significativo no desempenho.
- Compreender o fluxo de saltos e melhorar a sua previsibilidade, pode aumentar a velocidade de código de forma significativa.
- Otimizações que ajudam a previsão de desvio são:
 - Eliminar desvios sempre que possível
 - Organizar o código para ser coerente com o algoritmo de previsão de desvio estático
 - Funções inline e emparelhar chamadas e retornos.
 - Desenrole para que laços repetidamente executados tenham dezesseis ou menos iterações (a menos que provoque um aumento excessivo tamanho do código).

10

Previsão de Saltos

- Previsão de desvios
 - Prevê o destino de desvios e permite ao processador iniciar a executar instruções muito antes do fluxo de execução verdadeiro ser conhecido
 - Usa a "Branch Prediction Unit" (BPU)
 - Usa os "Branch Target Buffers" (BTB)

11

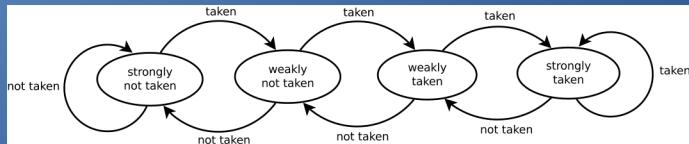
Branch Prediction Unity - BPU

- Previsão de desvios
 - Evita perdas no pipeline
- Tipos de previsões
 - Saltos (jmp) e chamadas (call) diretos
 - Saltos e chamadas indiretos
 - Saltos condicionais
- Reduc penalidades dos desvios
 - Mesmo assim, se possível, devem ser evitados

12

O Desvio Condicional Ocorrerá?

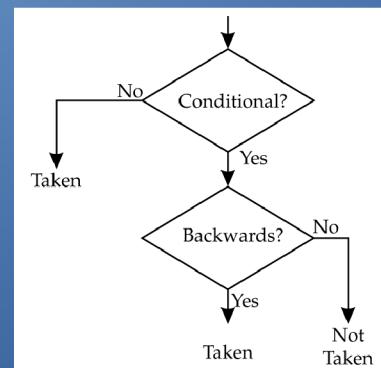
- Na primeira vez a previsão estática é utilizada
- Depois que o desvio condicional já teve sua condição avaliada, ele passa a ocupar uma entrada no BTB
- A previsão leva em consideração os últimos desvios realizados



13

Previsão Estática

- Desvios que não têm uma história na BTB são previstos utilizando um algoritmo de previsão estática



14

Eliminando Desvios

- Eliminar desvios melhora o desempenho
 - reduz a possibilidade de previsões erradas de desvios
 - reduz o número de entradas no BTB (branch target buffer)
 - desvios condicionais que nunca são realizados, não consomem recursos BTB.
- Principais formas de eliminar desvios
 - Organizar código para fazer blocos básicos contíguos
 - Desenrolar laços
 - Usar instruções condicionais CMOV e SETcc.

15

Otimizando com Instruções Condicionais

- Considere o código em C

$$x = (a < b) ? \text{CONST1} : \text{CONST2};$$
- Uma sequência equivalente em linguagem de montagem seria:

```

mov    eax, [a]
mov    ebx, const1
cmp    eax, [b]
jb     Rot
mov    ebx, const2
Rot:
mov    [x], ebx
  
```

Otimizando

```

mov    eax, [a]
mov    ebx, const1
cmp    eax, [b]
cmovae ebx, const2
mov    [x], ebx
  
```

16

Otimizando com Instruções Condicionais

- Considere o código em C

$$\text{if } (a > b) \text{ c++;}$$

e que a esteja em eax, b em ebx e c em ecx
- Uma sequência equivalente em linguagem de montagem seria:

```

cmp    eax, ebx
jle    Rot
inc    ecx
Rot:
  
```

Otimizando

```

cmp    eax, ebx
setg
add   edx, ecx
add   edx, edx
  
```

17