

Arquitetura de Processadores

Prof. Júlio Carlos Balzano de Mattos



Departamento de Informática
Instituto de Física e Matemática
Universidade Federal de Pelotas



Pipelining

[Pipeline Hazards]

- São situações que impedem que a próxima instrução da seqüência de instruções seja executada durante o seu ciclo de relógio designado
- Três classes de Hazards:
 - Hazard Estrutural
 - Hazard de Dados
 - Hazard de Controle
- Os hazards no pipeline provocam a necessidade de uma parada (*stall*) no pipeline

[Hazard Estrutural]

- Provocados por conflitos de recursos quando o hardware não suporta todas as combinações de instruções em uma execução simultânea de instruções sobrepostas
- Algum recurso não em quantidade suficiente para permitir todas as combinações de instruções no pipeline

[Hazard Estrutural]

■ Exemplo 1:

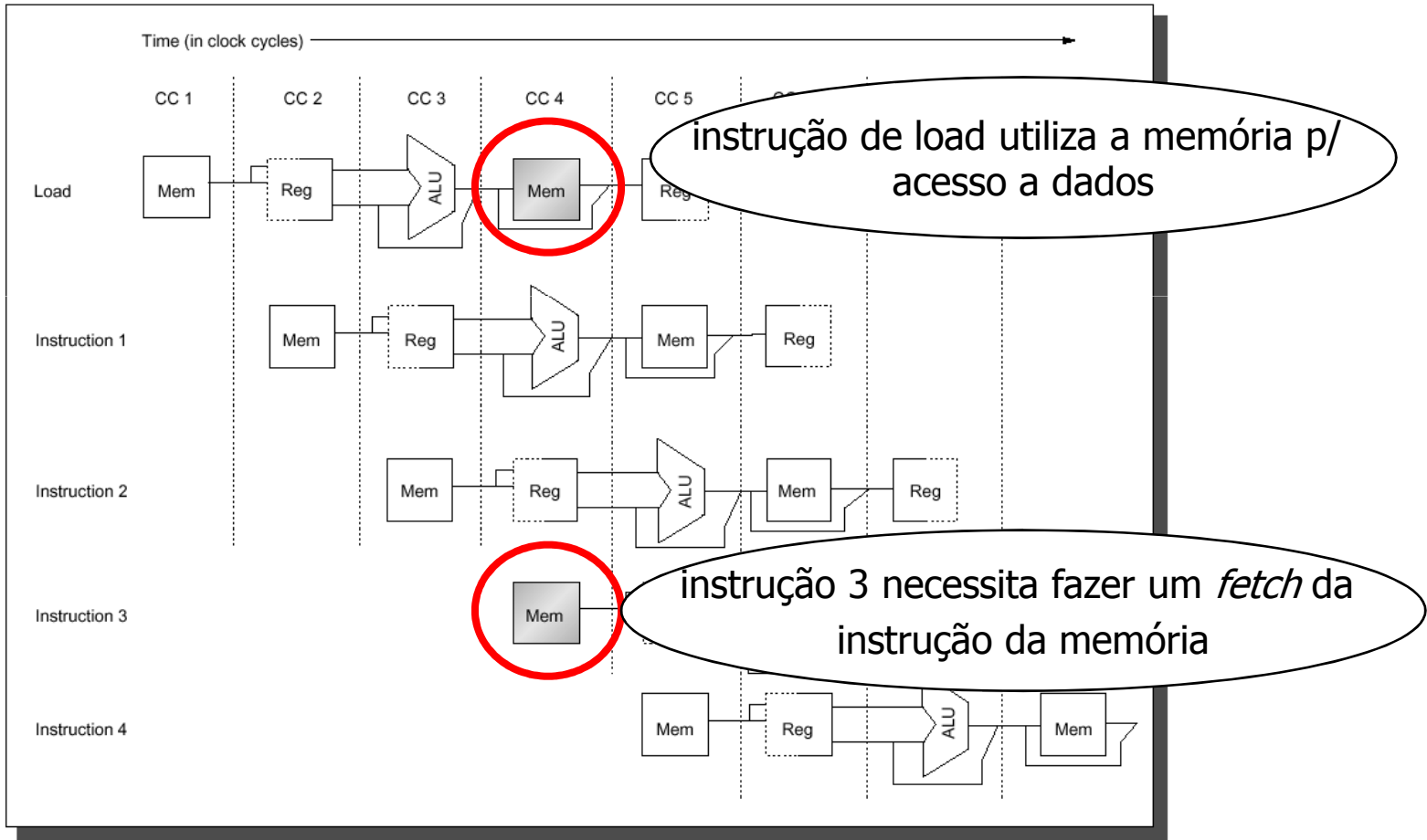
- Uma máquina possui somente uma porta de escrita no banco de registradores, porém as vezes é necessária duas escritas no mesmo ciclo

■ Exemplo 2:

- Máquinas que compartilham uma mesma memória para instruções e dados

[Hazard Estrutural]

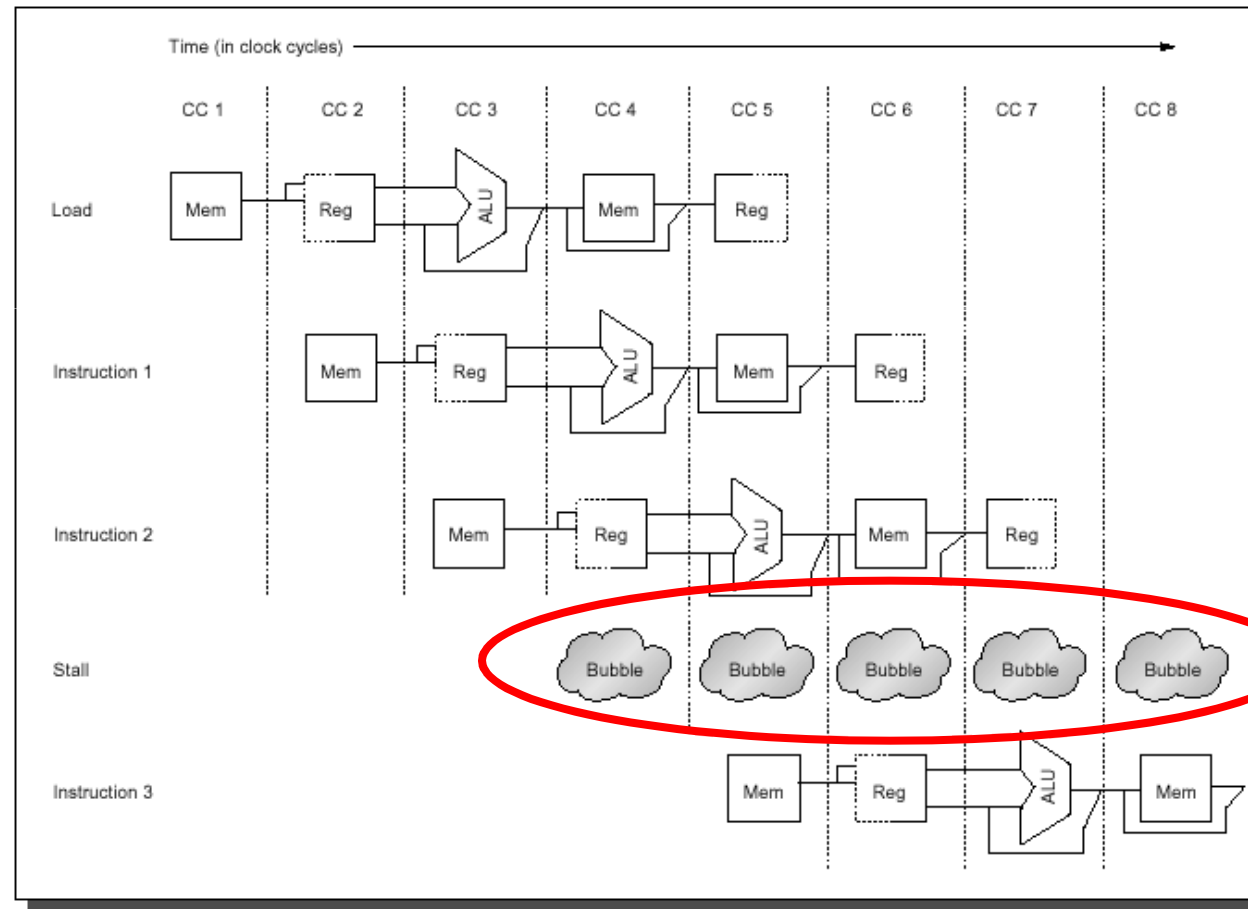
- Uma resolução com o mesmo nome posto de memória



Hazard Estrutural

Devido ao

pendências



Hazard Estrutural

	Número do Ciclo de Relógio							
Instrução	1	2	3	4	5	6	7	8
load	IF	ID	EX	MEM	WB			
Instr. 1		IF	ID	EX	MEM	WB		
Instr. 2			IF	ID	EX	MEM	WB	
Instr. 3				stall	IF	ID	EX	MEM
Instr. 4						IF	ID	EX
Instr. 5							IF	ID

[Hazard de Dados]

- O maior efeito do pipeline é alterar o tempo relativo das instruções através da sobreposição de instruções na execução, isto introduz:
 - hazard de dados
 - hazard de controle
- Hazard de Dados:
 - Acontece quando uma instrução depende do resultado da instrução previa, que por sua vez ainda não foi concluída

[Hazard de Dados]

■ Hazard de Dados:

- Ocorre quando o pipeline altera a ordem de leitura/escrita dos operandos (diferentemente da ordem seqüencial em uma máquina não pipeline)

ADD R1, R2, R3

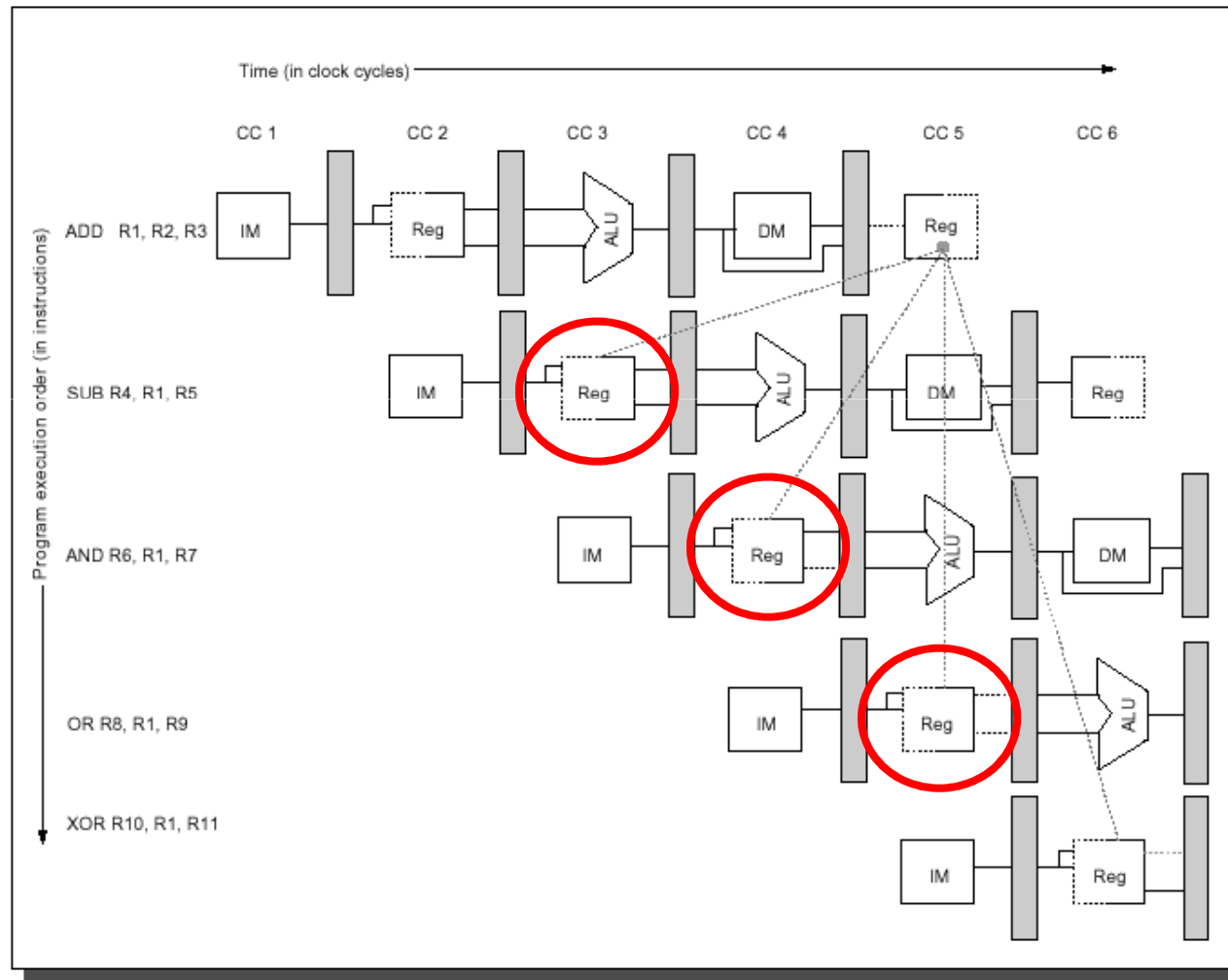
SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11

Hazard de Datos



[Hazard de Dados]

■ Classificação dos hazard de dados:

- (Considere duas instruções i e j, com i ocorrendo antes de j)
- RAW (read after write)
 - j tenta ler o fonte antes que i o escreva
 - Dependência de dados (direta ou verdadeira)
 - Exemplo:

ADD R1, R2, R3

SUB R4, R1, R5

AND R6, R1, R7

Hazard de Dados

- WAW (write after write)
 - j tenta escrever um operando antes que i o escreva
 - Dependência de dados (saída)
 - Exemplo:

LW R1, 0(R2)

ADD R1, R2, R3

Instrução	Número do Ciclo de Relógio					
	1	2	3	4	5	6
LW R1, 0(R2)	IF	ID	EX	MEM1	MEM2	WB
ADD R1, R2, R3		IF	ID	EX	WB	

Hazard de Dados

- WAR (write after read)
 - j tenta escrever em um destino antes que esse seja lido por i
 - Dependência de dados (anti-dependência)
 - Exemplo:

SW 0(R1), R2
ADD R2, R3, R4

Instrução	Número do Ciclo de Relógio					
	1	2	3	4	5	6
SW 0(R1), R2	IF	ID	EX	MEM1	MEM2	WB
ADD R2, R3, R4		IF	ID	EX	WB	

[Hazard de Controle]

- Surgem no pipelining de desvios e outras instruções que alteram o PC
- Podem provocar paradas no pipeline maiores que os hazard de dados
- Quando um desvio é executado:
 - Desvio Tomado (taken) – se um desvio alterar o PC para o seu destino
 - Desvio Não Tomado (not taken) – se falhar (PC não é alterado)



Principais Técnicas

Para redução ou eliminação das
dependências

[Forwarding]

■ Forwarding

- Técnica simples de hardware de solução do problema de dependências (também conhecida por *bypassing* ou *short-circuit*)
- Técnica que visa minimizar o problema das dependências de dados

■ Funciona da seguinte maneira:

- O resultado da ALU (registrador EX/MEM) sempre volta para os latches de entrada da ALU
- Se o hardware detecta que uma operação anterior escreveu em um registrador fonte da operação corrente da ALU, uma lógica de controle seleciona o resultado “forwarding” como entrada da ALU ao invés do valor lido do banco de registradores

[Forwarding]

- Voltando ao exemplo:

ADD R1, R2, R3

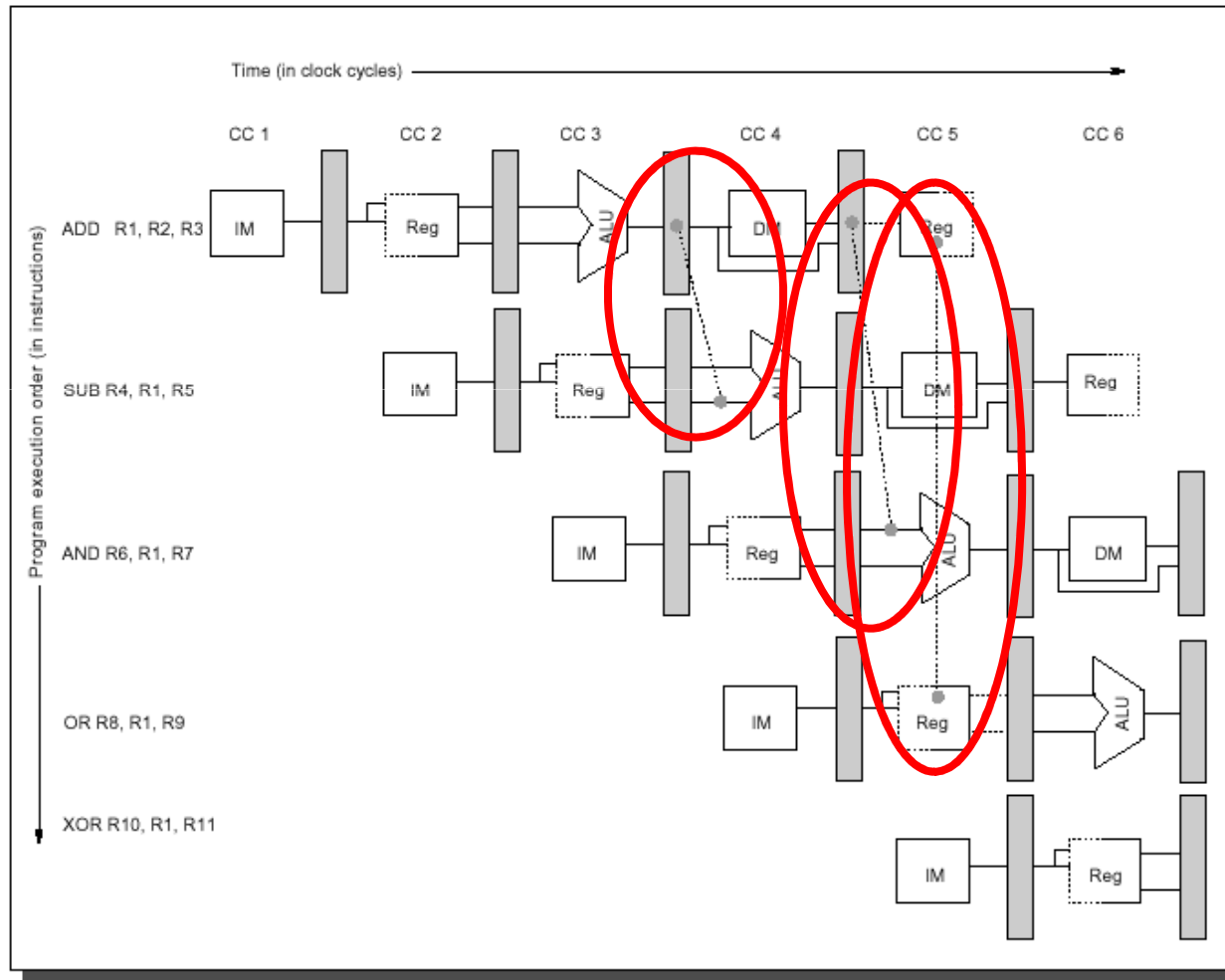
SUB R4, R1, R5

AND R6, R1, R7

OR R8, R1, R9

XOR R10, R1, R11

Forwarding



[Escalonamento Estático]

- Também conhecido como “pipeline scheduling” ou “instruction scheduling”
- Técnica utilizada pelo compilador para reduzir as penalidades provocadas pelos hazard de dados
- Para manter o pipeline cheio (sem stalls)
 - O compilador tenta escalonar o pipeline evitando os stalls
 - Como: através do rearranjo da seqüência de código tentando eliminar o hazard
- Primeiro uso da técnica: 1960

Escalonamento Estático

■ Exemplo de um código: $A = B + C$

- Assuma que A, B, e C são variáveis na memória

LW R1, B

LW R2, C

ADD R3, R1, R2

SW A, R3

LW R1, B	IF	ID	EX	MEM	WB				
LW R2, C		IF	ID	EX	MEM	WB			
ADD R3, R1, R2			IF	ID	stall	EX	MEM	WB	
SW A, R3				IF	stall	ID	EX	MEM	WB

- Assumindo que se utiliza forwarding

[Escalonamento Estático]

- Exemplo utilizando escalonamento estático para evitar stalls da seguinte seqüência:

$A = B + C;$

$D = E - F;$

- Assumindo que o load possui a latência igual a 1

LW Rb, B

LW Rc, C

LW Re, E

ADD Ra, Rb, Rc

LW Rf, F

SW A, Ra

ADD Rd, Re, Rf

SW D, Rd

[Escalonamento Estático]

- No exemplo anterior as instruções foram reordenadas visando retirar as dependências
- A maioria dos compiladores atuais utilizam escalonamento de instruções para aumentar o desempenho do pipeline
- Normalmente, os compiladores utilizam algoritmos simples que escalonam instruções do mesmo “basic block”
 - Basic Block - sequência de código que possui somente um entrada e uma saída
 - Exemplo: o código dentro de um then de um if

[Delayed Branch]


- Técnica muito utilizada em unidades de controle microprogramadas
- Consiste de um branch-delay slot
 - É uma instrução ou instruções (depende do tamanho do slot) que estão localizadas após a instrução de desvio
 - Estas instruções são executadas se o desvio é ou não tomado (taken ou not taken)
- É realiza pelo compilador
 - O compilador possui como objetivo que as instruções sucessoras de um desvio sejam válida e úteis mesmo na presença de um desvio

[Delayed Branch]

■ Exemplo:

- Máquina com um delayed branch de apenas um ciclo:
- Antes de aplicar a técnica (código original):

```
ADD R1, R2, R3  
if R2 = 0 then  
    Delay slot
```



- Após aplicar a técnica:

```
if R2 = 0 then  
    ADD R1, R2, R3
```

[Loop Unrolling]

- Técnica que melhora o problema com as dependências de controle
- A técnica
 - Realiza a replicação do corpo loop múltiplas vezes, e ajusta o código de terminação do loop
 - Realizada pelo compilador (estaticamente)
- O loop unrolling também pode ser utilizado para melhorar o escalonamento
 - Já que elimina os desvios permitindo que instruções de diferentes iterações sejam escalonadas juntas

[Loop Unrolling]

- O exemplo utiliza as técnicas de loop unrolling, escalonamento e delay branch
- Exemplo:
 - Utiliza um simples loop que adiciona um valor escalar para um array na memória
 - Exemplo de código fonte:

```
for (i=1; i<=1000; i++)  
    x[i] = x[i] + s;
```
 - O código em assembly (sem escalonamento e loop unrolling)

Loop: LD F0, 0(R1)	; F0 = elemento do array
ADDD F4, F0, F2	; adiciona um escalar (F2)
SD 0(R1), F4	; armazena o resultado
SUBI R1, R1, #8	; decrementa o ponteiro (8 bytes)
BNEZ R1, Loop	; desvia R1 != zero

[Loop Unrolling]

- O exemplo **sem** escalonamento, incluindo os stalls (paradas)

```
Loop: LD F0, 0(R1)
      stall
      ADDD F4, F0, F2
      stall
      stall
      SD 0(R1), F4
      SUBI R1, R1, #8
      stall
      BNEZ R1, Loop
      stall
```

- Este código requer 10 ciclos por interação

[Loop Unrolling]

- O exemplo **com** escalonamento, incluindo os stalls (paradas)

```
Loop: LD F0, 0(R1)
      SUBI R1, R1, #8
      ADDD F4, F0, F2
      stall
      BNEZ R1, Loop          ; delay branch
      SD 8(R1), F4           ; alterado e trocado com SUBI
```

- Note que foi utilizado delay branch
- Este código requer 6 ciclos por interação

[Loop Unrolling]

- Exemplo desenrolando o loop por quatro vezes:

Loop: LD F0, 0(R1)
 ADDD F4, F0, F2
 SD 0(R1), F4
 LD F6, -8(R1)
 ADDD F8, F6, F2
 SD -8(R1), F8
 LD F10, -16(R1)
 ADDD F12, F10, F2
 SD -16(R1), F12
 LD F14, -24(R1)
 ADDD F16, F14, F2
 SD -24(R1), F16
 SUBI R1, R1, #32
 BNEZ R1, Loop

[Loop Unrolling]

- No exemplo anterior foram eliminados três desvios e três decrementos de R1
- Sem escalonamento, todas as operações são seguidas por dependência causando um stall
- O loop do exemplo anterior levaria 28 ciclos
 - (cada LD provoca um stall, cada AD DD 2 stalls, cada SUBI 1 stall, ...)
- Assim, esta versão é mais lenta do que a versão original do loop escalonada

[Loop Unrolling]

- Exemplo desenrolando o loop por quatro vezes e utilizando escalonamento e delay branch:

```
Loop: LD F0, 0(R1)
      LD F6, -8(R1)
      LD F10, -16(R1)
      LD F14, -24(R1)
      ADDD F4, F0, F2
      ADDD F8, F6, F2
      ADDD F12, F10, F2
      ADDD F16, F14, F2
      SD 0(R1), F4
      SD -8(R1), F8
      SUBI R1, R1, #32
      SD -16(R1), F12
      BNEZ R1, Loop
      SD 8(R1), F16
```


[Loop Unrolling]

- O tempo de execução do exemplo anterior é de 14 ciclos de relógio
 - Com 3.5 ($14/4=3.5$) ciclos de relógio por iteração
 - Comparando com o exemplo que utiliza loop unrolling sem escalonamento que possui um tempo de execução de 28 ciclos, possui 7 ($28/4=7$) ciclos de relógio por iteração
 - Comparando com o exemplo que utiliza **sem** loop unrolling com escalonamento que possui um tempo de execução de 24 ciclos, possui 6 ($24/4=6$) ciclos de relógio por iteração

[Loop Unrolling]

- O ganho com escalonamento do loop unrolling é bem maior do que o código original
 - Porquê o loop unrolling expõe mais computação que pode ser escalonada para minimizar os stalls
- O loop unrolling é simples porém é um método muito útil para o aumento do tamanho dos fragmentos de código que podem ser escalonados eficientemente

Principais Técnicas (resumo)

- Forwarding
 - Reduz paradas (stalls) provocadas por dependência de dados (**Dados - Direta**)
- Escalonamento Estático
 - Reduz paradas (stalls) provocadas por dependência de dados (**Dados - Direta**)
- Delayed Branch
 - Melhora o desempenho no caso de desvios (**Controle**)
- Loop Unrolling
 - Reduz paradas (stalls) provocados por desvios (**Controle**)
 - Auxilia o escalonamento estático

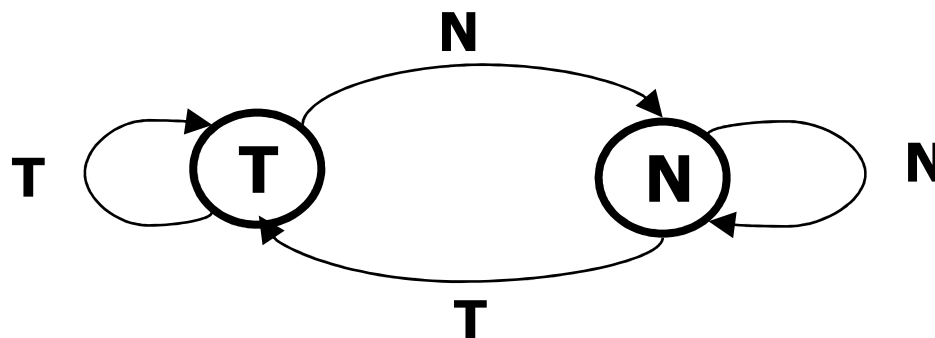
[Previsão de Desvios]

- Técnica utilizada para minimizar as dependências de controle
- Idéia: tentar prever qual será o desvio (taken ou not taken)
- Dois tipos de previsão:
 - Estática
 - Dinâmica
- Previsão estática
 - Na previsão estática o resultado previsto sempre será o mesmo
 - Três possibilidades:
 - O desvio sempre ocorrerá
 - O desvio nunca ocorrerá
 - Código (opcode) da operação determina a previsão

[Previsão de Desvios]

■ Previsão Dinâmica

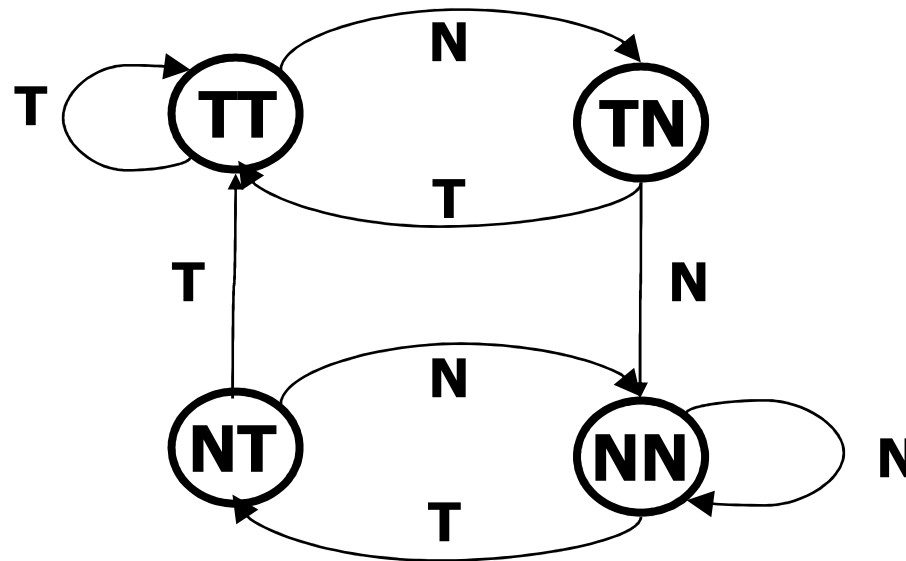
- É uma previsão baseada em história (utiliza as informações sobre os desvios anteriores)
- Previsão de 1 bit de história
 - T – taken, N – not taken



[Previsão de Desvios]

■ Previsão Dinâmica

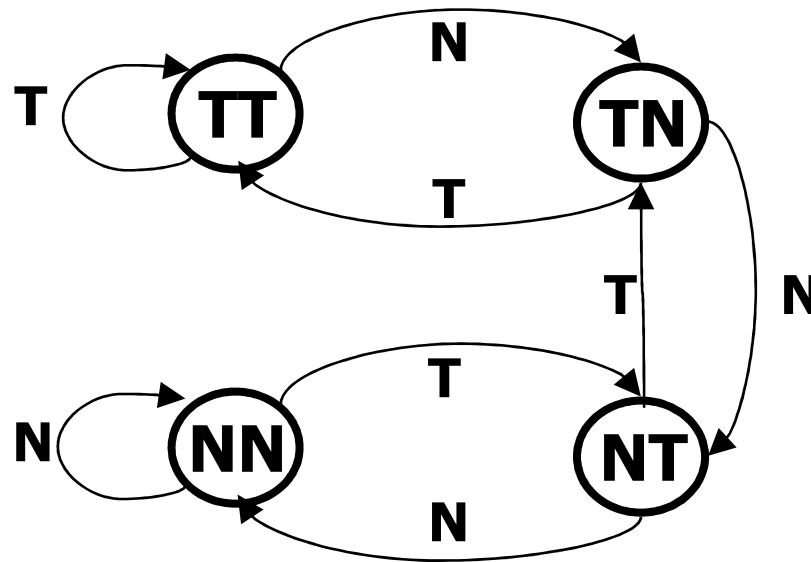
- Previsão de 2 bit de história



Previsão de Desvios

■ Previsão Dinâmica

- Previsão de 2 bit de história – contador saturado



[Execução Especulativa]

- Técnica que possui com idéia executar uma instrução antes que o processador conheça se a instrução deva ser executada
- Evita principalmente paradas por dependências de controle, porém também permite evitar todos os tipos de dependência de dados
- Exemplo: executar os dois trechos de um teste

```
if R2 = 0 then  
    ADD R1, R2, R3  
else  
    ADD R4, R1, R0
```




Superescalaridade

[Introdução]

■ Princípios

- Várias unidades de execução
- Várias instruções completadas simultaneamente em cada ciclo de relógio
- hardware é responsável pela extração de paralelismo

■ Na prática

- IPC pouco maior do que 2 (limitação do ILP)

■ Problemas

- Execução simultânea de instruções
- Conflitos de acesso a recursos comuns (ex: memória)
- Dependência de Dados
- Dependência de controle

[Introdução]

■ Características

- Pipelines ou unidades funcionais podem possuir latências diferentes
- Término das instruções em seqüência diferente do programa original

■ Processador com capacidade de “look-ahead”

- se há conflito que impede execução da instrução atual, processador
- examina instruções além do ponto atual do programa
- procura instruções que sejam independentes
- executa estas instruções

■ Possibilidade de execução fora de ordem

- cuidado para manter a correção dos resultados do programa

[Introdução]

■ Despacho de instruções

- refere-se ao fornecimento de instruções para as unidades funcionais

■ Terminação de instruções

- refere-se à escrita de resultados (em registradores, no caso de processadores RIS)

■ Alternativas

- despacho em ordem, terminação em ordem
- despacho em ordem, terminação fora de ordem
- despacho fora de ordem, terminação fora de ordem

[Register Renaming]

- Técnica que elimina as dependências de nome (anti-dependência e dependências de saída)
- O conceito
 - É que instruções envolvidas em dependências de nome podem ser executadas simultaneamente ou podem ser reordenadas, se o nome (número do registrador ou localização na memória) utilizados nas instruções podem ser alterados tornando as instruções sem conflito
- Pode ser realizado estaticamente (pelo compilador) ou dinamicamente (pelo hardware)

[Register Renaming]

- Emprega o conceito de registradores lógico e físicos
 - Registradores físicos – disponíveis no hardware (invisíveis ao usuário)
 - Registradores lógicos – conjunto menor (utilizados pelo programador)
- Quando identifica uma dependência
 - O registrador lógico é substituído por um registrador físico, ocorrendo a renomeação na instrução dependente, eliminando a dependência

Register Renaming

- Esquema de renomeação de registradores:
 - Registradores Lógicos: A, B, C, ...
 - Registradores Físicos: R0, R1, R2, R3 ...
 - Possui uma tabela de mapeamentos: que possui a associação dos registradores lógicos com os registradores físicos
 - Possui uma lista de registradores ativos: que quando caracterizada uma dependência, ocorre uma renomeação no registrador que originou a dependência e este registrador é colocado em uma lista de ativos (que não podem ser liberados, pois estão sendo utilizados por instruções que ainda não terminaram a sua execução)

Register Renaming

- Código Original

```
LW B, 1
LW C, 2
LW E, 3
LW F, 4
LW G, 5
ADD A, B, C
ADD D, A, E
ADD E, F, G
ADD B, F, G
ADD B, 1, C
```

Lista de Ativos

R2	R0	R8			
-----------	-----------	-----------	--	--	--

Código Renomeado

```
LW R0, 1
LW R1, 2
LW R2, 3
LW R3, 4
LW R4, 5
ADD R5, R0, R1
ADD R6, R5, R2
ADD R7, R3, R4
ADD R8, R3, R4
ADD R9, 1, R1
```

Tabela de Mapeamento

B	R0,R8,R9
C	R1
E	R2,R7
F	R3
G	R4
A	R5
D	R6

[Escalonamento Dinâmico]

- Estratégia bastante utilizada no tratamento de dependência de dados (realizada pelo hardware)
- Vantagens sobre o escalonamento estático:
 - Resolver dependências desconhecidas em tempo de compilação
 - Simplifica o compilador
 - Permite que um código compilado para um pipeline seja executado eficientemente em outro pipeline diferente
- Duas técnicas mais utilizadas:
 - Scoreboard
 - Algoritmo de Tomazulo

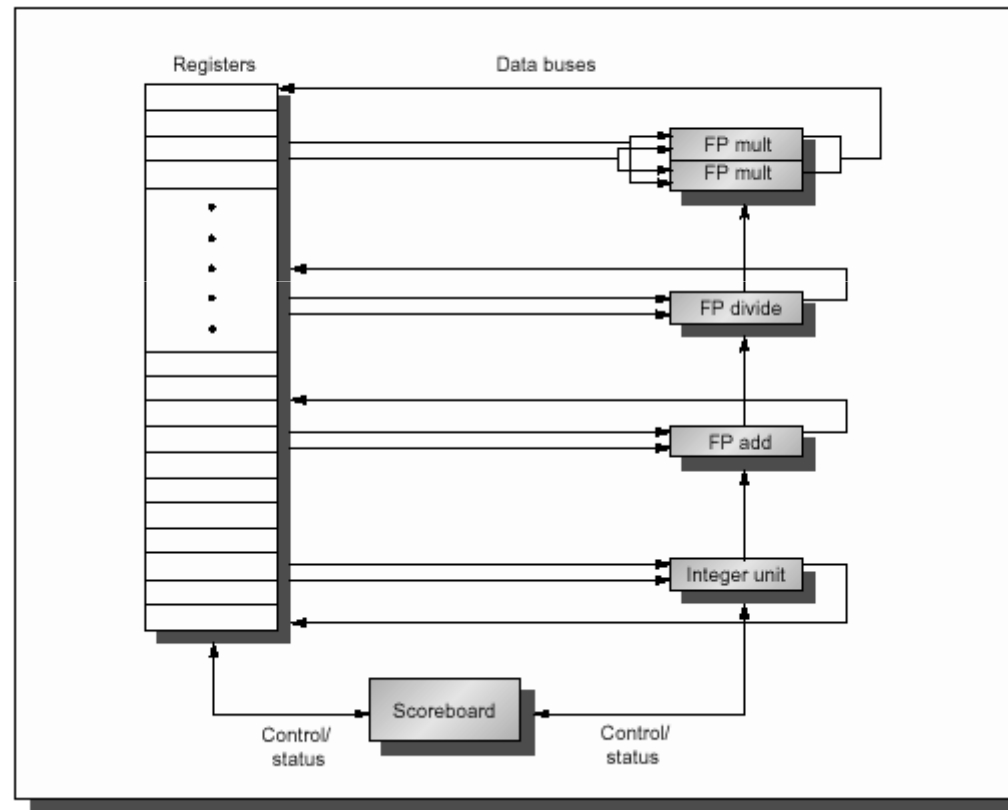
[Escalonamento Dinâmico]

■ Scoreboard

- Técnica que permite que instruções sejam executadas fora de ordem quando existem recursos suficientes e não exista dependência de dados
- Utilizado primeiramente no CDC 6600
- Possui três partes principais:
 - Tabela de status de instrução – indica o estágio que a instrução está
 - Tabela de status das unidades funcionais – indica o status da unidade funcional (Ocupada ou não, operação que esta realizando, registrador destino, registradores fontes, ...)
 - Tabela de status dos registradores – indica que unidade funcional irá escrever em cada registrador

Escalonamento Dinâmico

■ Scoreboard: exemplo de estrutura básica



Escalonamento Dinâmico

■ Scoreboard: exemplo de tabelas

Instruction		Instruction status			
		Issue	Read operands	Execution complete	Write result
LD	F6, 34 (R2)	√	√	√	√
LD	F2, 45 (R3)	√	√	√	
MULTD	F0, F2, F4	√			
SUBD	F8, F6, F2	√			
DIVD	F10, F0, F6	√			
ADDD	F6, F8, F2				

Functional unit status									
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status									
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Sub	Divide			

[Escalonamento Dinâmico]

- Algoritmo de Tomazulo
 - Também permite a execução de instruções fora de ordem
 - Além de atacar os problemas das dependência verdadeiras, também permite evitar as anti-dependências e dependências de saída (Scoreboard – somente dependências diretas)
- Utilizado no IBM 360/91 (unidade de ponto Flutuante)
- Implementada em unidade de ponto flutuante

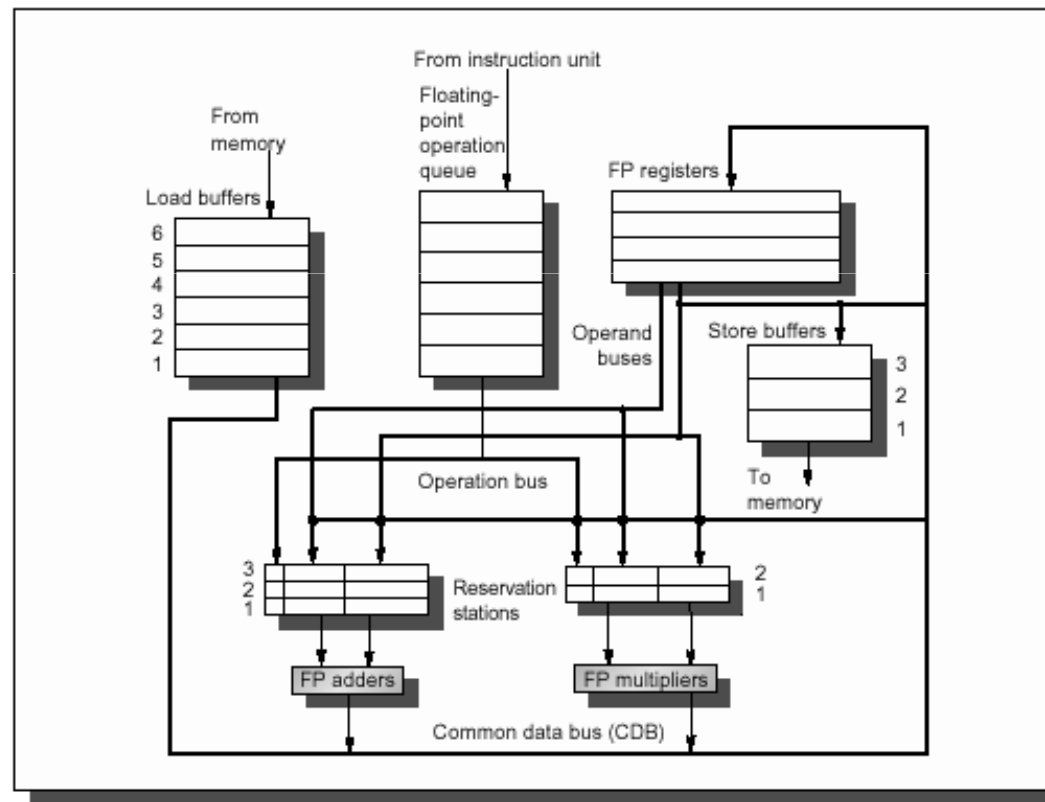
[Escalonamento Dinâmico]

■ Algoritmo de Tomazulo

- Possui idéias semelhantes ao Scoreboard
- Possui as estações de reserva (reservation stations) que permite realizar register renaming nestas estações
- Principais diferenças:
 - Tomazulo: realiza register renaming, Scoreboard: não realiza
 - Scoreboard: controle centralizado, Tomazulo: controle mais distribuído (unidades de detecção de hazard e execução são distribuídas)
 - Tomazulo: os resultados das operações passam diretamente as unidades funcionais para as estações de reserva sem a necessidade de passar pelos registradores

Escalonamento Dinâmico

■ Tomazulo: exemplo de estrutura básica



Principais Técnicas (resumo)

- Renomeação de Registradores
 - Eliminar as dependências de nome (anti-dependência e dependência de saída)
- Escalonamento Dinâmico
 - Scoreboard: reduzir dependência de dados (direta)
 - Tomazulo: reduzir dependência de dados (todas)
- Previsão de Desvios
 - Reduzir os efeitos provocados pelas dependências de controle
- Execução Especulativa
 - Reduzir os efeitos das dependências de dados e controle



Arquiteturas VLIW

[Introdução]

■ VLIW - Very Long Instruction Word

- Máquinas que exploram o paralelismo no nível de instruções
- São arquiteturas distintas das tradicionais arquiteturas CISC e RISC
- Arquiteturas VLIW e superescalar compartilham algumas características:
 - Múltiplas unidades de execução
 - Capacidade de executar múltiplas operações em paralelo
 - As técnicas para atingir um alto desempenho são diferentes

[Comparação]

VLIW

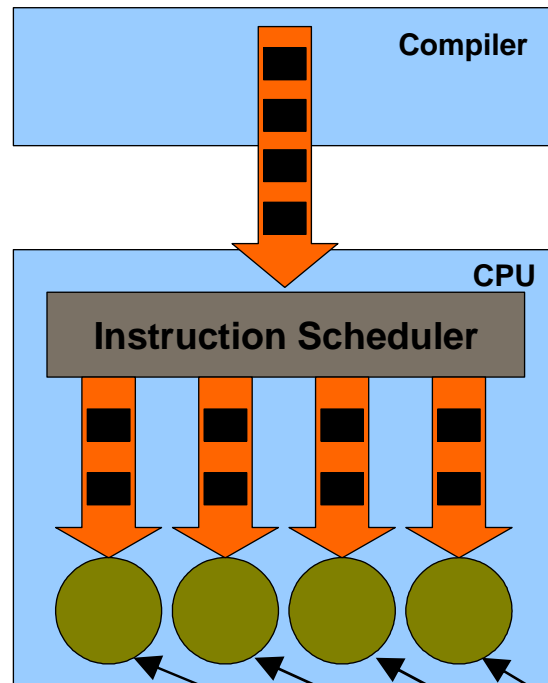
- Descoberta do paralelismo
 - em tempo de compilação
 - pelo compilador
- HW mais simples
- Maior suporte pelo compilador (mais complexo)

Superescalar

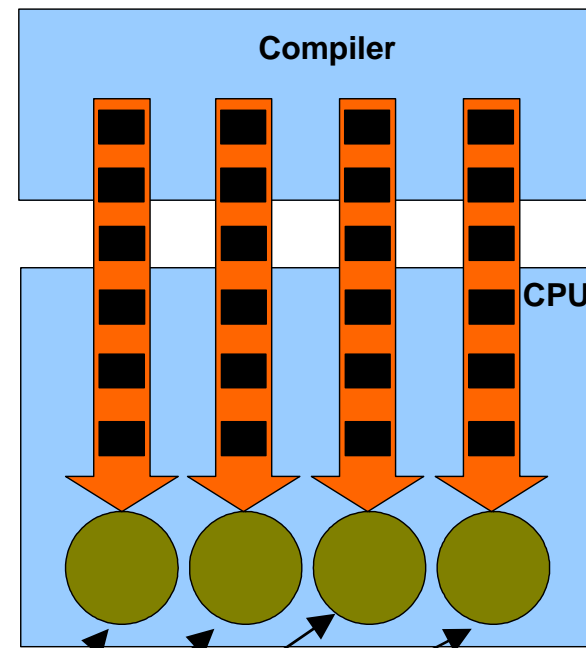
- Descoberta do paralelismo
 - em tempo de execução
 - por um HW dedicado
- HW mais complexo
- O compilador não é complexo

VLIW x Superscalar

■ Superscalar



■ VLIW



Unidades
Funcionais

[Introdução]

■ CISC, RISC e VLIW

- Pode-se dizer que as arquiteturas RISC permitem implementações mais simples e de mais alto desempenho que as arquiteturas CISC
- Já as arquiteturas VLIW são mais simples e baratas que as RISC (HW é bem mais simples)
- Contudo, as arquiteturas VLIW necessitam um maior suporte do compilador

■ Principal diferença

- No hardware de despacho

[Instruções]

■ Instruções VLIW

- Um instrução VLIW especifica diversas operações, assim as instruções VLIW são necessariamente mais longas que RISC e CISC (daí vem o nome !)
- Várias operações (instruções de uma máquina RISC) são codificadas na mesma instrução
- Cada posição dentro da palavra especifica a unidade funcional que será utilizada
- O HW de despacho é bem mais simples que de um superescalar

[Implementações]

■ Implementação VLIW

- Capacidades muito similares aos superescalares
 - Despachar e completar mais de uma instrução por ciclo
 - Importante exceção: o HW não é responsável por descobrir instruções que deverão ser executadas concorrentemente
- Uma instrução VLIW já codifica as operações que podem ser executadas concorrentemente
- Essa codificação explícita torna enormemente simplificada a implementação do hardware
- Implementações mais simples que RISC e CISC
- “Maneira simplificada de construir um processador RISC”

[Comparações]

■ Diferenças Arquiteturais

- Em geral, as arquiteturas RISC e CISC possuem mais similaridades que diferenças das arquiteturas VLIW
- Porém as diferenças que existem provocam profundas diferenças nas implementações
- O funcionamento é o mesmo:
 - O hardware busca e executa as instruções sequencialmente até que uma instrução de desvio provoque uma mudança no fluxo de controle


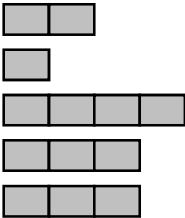
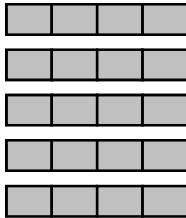
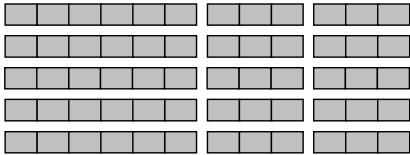
[Comparações]

■ Comparação Arquitetural (RISC, CISC e VLIW)

Característica Arquitetural	CISC	RISC	VLIW
Tamanho instrução	variável	tamanho único (32 bits)	Tamanho único
Formato instrução	campos variáveis	regular	regular
Semântica instrução	varia de simples a complexa	operação simples	muitas simples independentes
Registradores	poucos, específicos	muitos, propósito geral	muitos, propósito geral

[Comparações]

■ Comparação Arquitetural (RISC, CISC e VLIW)

Característica Arquitetural	CISC	RISC	VLIW
Referência a memória	varias instruções acessando	arquitetura load/store	arquitetura load/store
Projeto de Hardware	microcódigo	um pipeline sem microcódigo	vários pipeline sem microcódigo despacho simples
Figura com 5 instruções  = 1 byte			

[Comparações]

■ Comparando instruções

- CISC

add 4[r1], r2

addMR	4	r1	r2
-------	---	----	----

- RISC

load r5, 4[r1]

add r5, r5, r2

store 4[r1], r5

load	r5	r1	4
------	----	----	---

add	r5	r5	r2
-----	----	----	----

store	r5	r1	4
-------	----	----	---

[Comparações]

■ Comparando instruções (cont.)

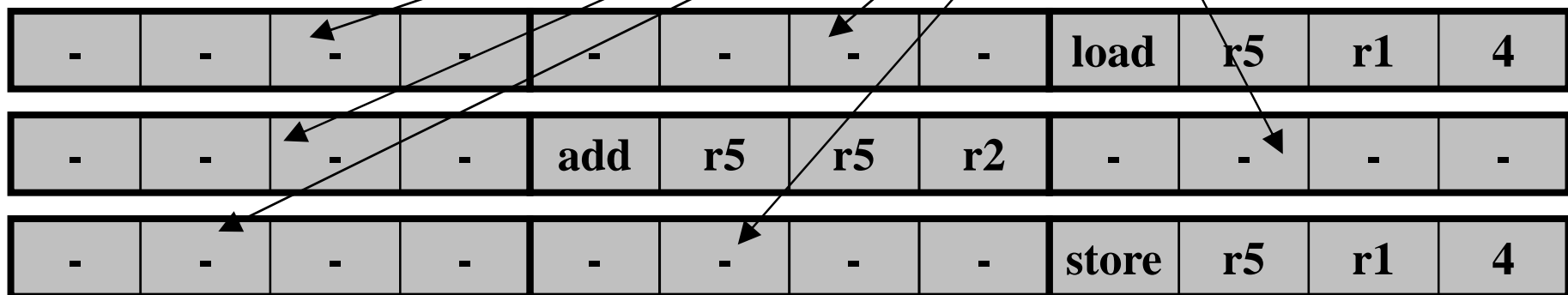
- VLIW

load r5, 4[r1]

add r5, r5, r2

store 4[r1], r5

Poderiam ser alocadas
outras instruções



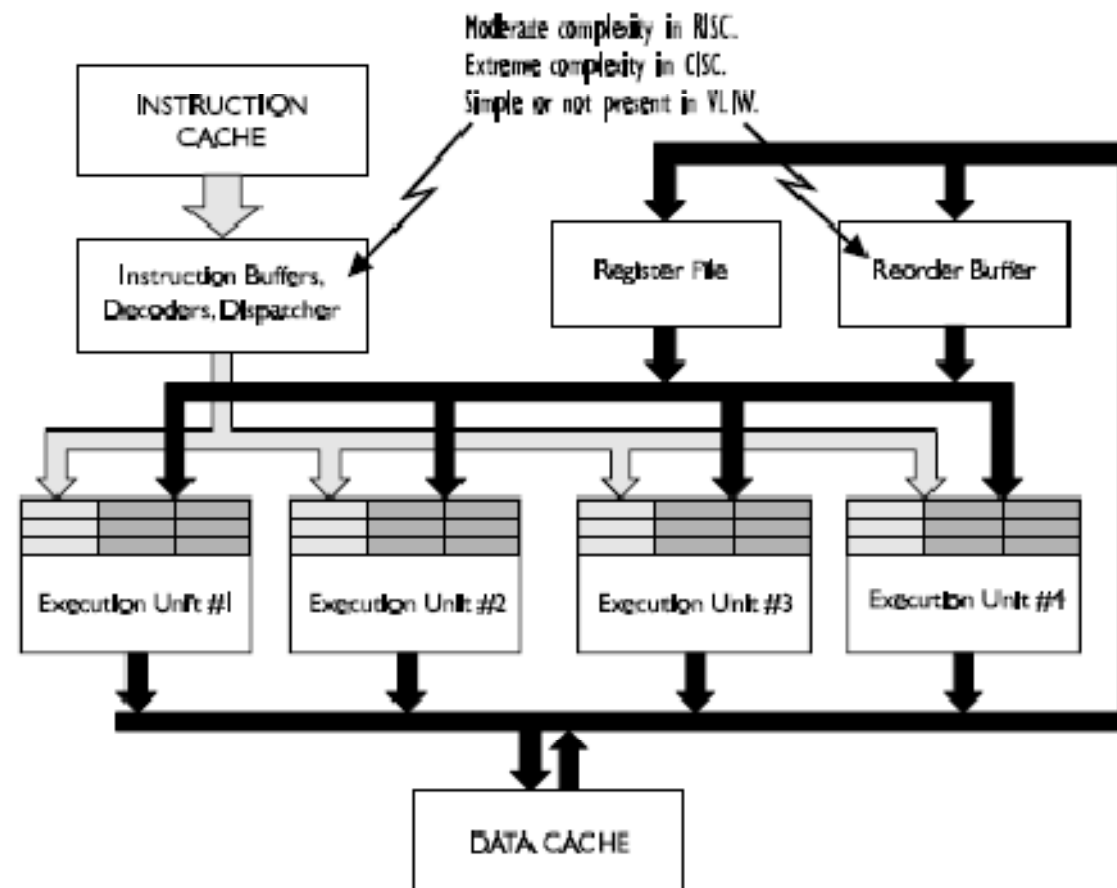
[Comparações]

■ Comprando Arquitetura VLIW com Supescalar

- No seu todo, a arquitetura VLIW é bem menos complexa
- A unidade de despacho é bem mais simples
- A parte de decodificação também é simplificada
- Não existe necessidade de estações de reserva antes das unidades de execução
- Reoder buffer (buffer de reordenação) é muito simplificado ou inexistente no VLIW

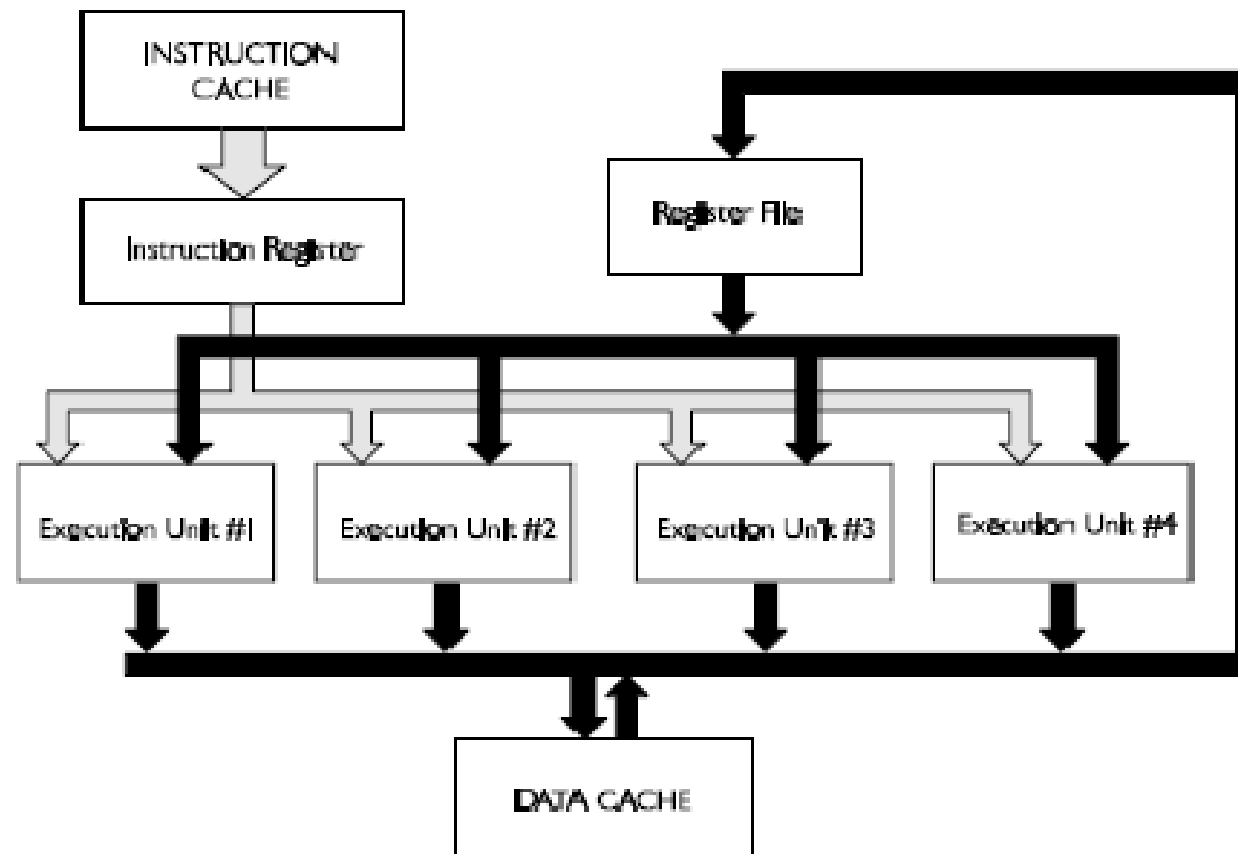
Comparações

■ Arquitetura RISC



Comparações

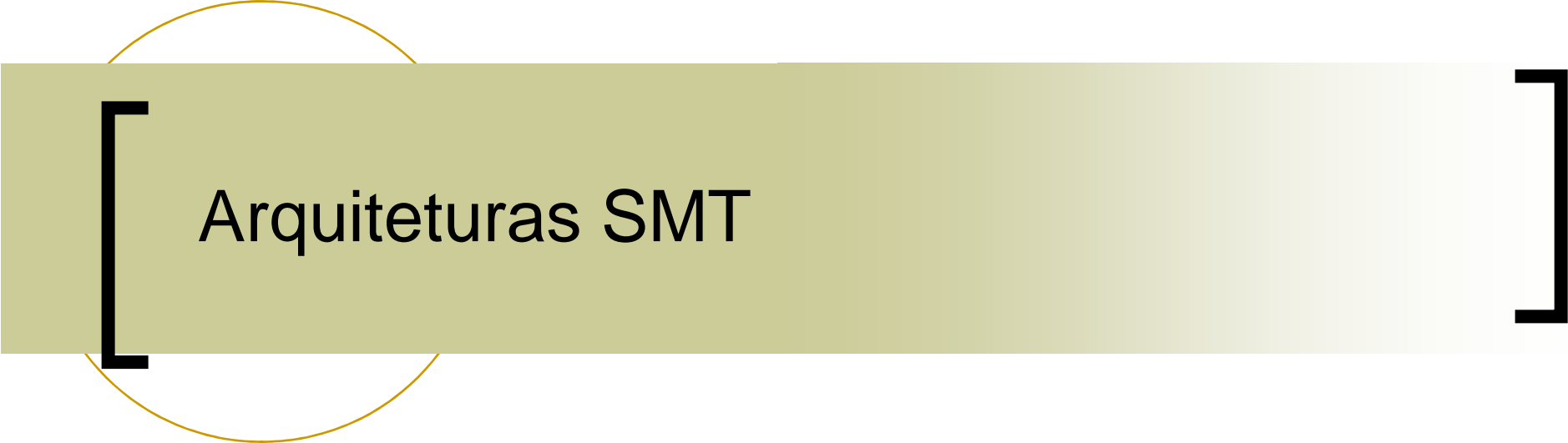
■ Arquitetura VLIW



Exemplos de Processadores

■ Exemplos

- Processador Itanium (Intel)
 - Processador projetado com a tecnologia IA64
 - IA64 é uma arquitetura VLIW de 64 bits da Intel
- Processador Crusoe (Transmeta)
 - Compatível com o conjunto de instruções x86
 - Realiza tradução dinâmica de código (binary translation) das instruções
 - Realiza a tradução binária do código x86 para o código do Crusoe (que é uma arquitetura VLIW)



Arquiteturas SMT

[Introdução]

■ Arquiteturas Superescalares

- Execução de várias instruções por ciclo (4, 8 ou mais)
- Na prática apenas são executadas uma ou duas (depende da aplicação)
- Devido ao baixo nível de paralelismo no nível das instruções - ILP (*Instruction Level Parallelism*)

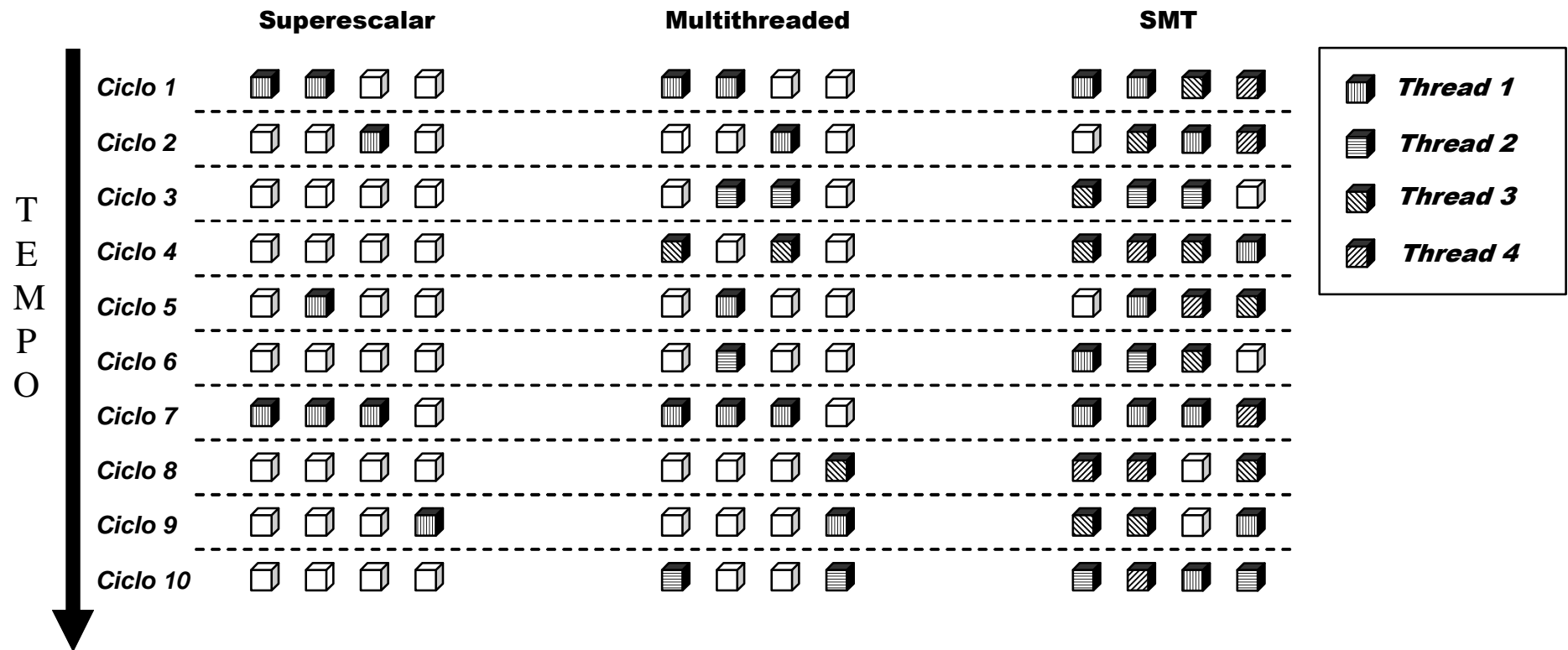
■ Arquiteturas SMT

- Solução de um processador que pode atacar vários tipos de paralelismo
- Paralelismo no nível das *threads* (TLP – *Thread Level Parallelism*)
- A técnica de SMT – *Simultaneous Multithreading* ataca tanto o paralelismo no nível das *threads* quanto no nível das instruções

[Introdução]

- Diferença entre o funcionamento de processadores
 - Superescalares
 - *Multithread*
 - *Simultaneous multithreading*
- Explicação da Figura
 - Cada cubo representa uma unidade funcional do processador
 - Cada linha (conjunto horizontal de unidades funcionais) representa o estado do processador a cada ciclo
 - Um cubo que esteja preenchido significa que a unidade funcional correspondente está executando uma instrução proveniente de uma *thread*
 - Quando o cubo está vazio, significa que a unidade funcional correspondente não está sendo utilizada

Diferenças



Diferenças

■ Unidades Funcionais

- O não uso de unidades funcionais ao decorrer do tempo pode ser caracterizado como desperdício horizontal ou vertical

■ Superescalar

- Há a execução de apenas um processo – e uma *thread* – por vez
- A partir desta *thread*, várias instruções podem ser buscadas por ciclo para o preenchimento das unidades funcionais
- Quando, por algum motivo (como dependência de dados), não há instruções suficientes para preencher todas as unidades funcionais, ocorre desperdício horizontal
- E quando não há nenhuma instrução naquele ciclo que possa se encaixar em pelo menos uma unidade funcional (isso acontece, por exemplo, quando há uma falha na *cache* de instruções e a fila de instruções se esvazia), ocorre desperdício vertical

Diferenças

■ *Multithread*

- Estes processadores contêm *hardware* adicional para guardar os estados de várias *threads*
- Em um ciclo qualquer o processador executa instruções originárias de uma *thread*
- No próximo ciclo, há um chaveamento de contexto para a próxima *thread*, da qual as instruções serão buscadas, e assim sucessivamente
- Principal vantagem deste tipo de processador é a melhor tolerância com operações de longa latência
- Exemplo: *cache miss* durante a execução de uma *thread*, ao invés do processador esperar pelos dados e ficar inativo, simplesmente outra *thread* recomeça a ser executada

Diferenças

■ SMT - *Simultaneous multithreading*

- É permitida a seleção de instruções originárias de diferentes *threads* durante o mesmo ciclo
- Ataca o problema do ILP citado anteriormente, selecionando instruções de diferentes *threads*, e reduzindo drasticamente o problema de desperdício horizontal
- O processador dinamicamente faz o agendamento das instruções de diferentes *threads* para preencherem suas unidades funcionais em um mesmo ciclo
- Se um suposto processador possuir 8 unidades funcionais, e em um determinado ciclo, há uma *thread* com ILP suficiente para preencher 5 destas unidades, pode-se tentar alocar as 3 unidades restantes utilizando instruções originárias de outras *threads*

[Referências]

- [1] Patterson, D. & Hennessy, J. **Computer Architecture: A Quantative Approach**. San Francisco: Morgan Kaufmann, 1996.
- [2] Philips Semiconductors. **An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture**.
- [3] Susan Eggers and et al. **Simultaneous Multithreading: A Platform for Next-generation Processors**. IEEE Micro, September/October 1997.