

---

**Universidade Federal do Rio de Janeiro**  
**Pós-Graduação em Informática**

**Microarquitecturas de Alto  
Desempenho**

**Escalonamento  
de Instruções**

**Gabriel P. Silva**

# Introdução

---

- **Otimização de código é necessária como forma de diminuir o tempo de execução e/ou tamanho do programa.**
- **Existem várias técnicas de otimização de código utilizadas normalmente pelos compiladores:**
  - **Eliminação de expressões comuns**
  - **Eliminação de código morto**
  - **Eliminação de desvios desnecessários**
  - **Eliminação de código redundante**
  - **Movimentação de código**
- **As técnicas de otimização de código podem ser classificadas como dependentes ou não da arquitetura do processador.**

# Introdução

---

- **Exemplos de técnicas simples de escalonamento empregadas em arquiteturas com pipeline:**
  - Escalonamento de código para uso do “delay slot”
  - Escalonamento de código para evitar “stalls” devido a dependência direta de dados
- **Em nosso estudo estaremos abordando algumas técnicas de escalonamento de código, dependentes de máquina, para uso tanto em arquiteturas VLIW como em superescalares.**
- **Estaremos estudando também técnicas de escalonamento dinâmico que são empregadas em arquiteturas superescalares.**

# Introdução

---

- **Escalonamento Estático:**
  - Escalonamento de instruções em blocos básicos.
  - Escalonamento além das instruções de desvio:
    - Trace Scheduling
    - Loop Enrolling
    - Software Pipeline
- **Escalonamento Dinâmico**
  - Scoreboarding
  - Algoritmo de Tomasulo

# Compactação Local de Código

---

- **Consiste de três passos fundamentais:**
  - **Eliminação de dependências de controle: divisão do código em blocos básicos;**
  - **Eliminação de dependências de dados falsas com a técnica de renomeação de registradores;**
  - **Escalonamento de instruções a serem executadas em paralelo, tendo em vista as restrições de recurso e as dependências de dados ainda existentes: algoritmo de “list scheduling”.**
- **O escalonamento deve preservar a ordem semântica das instruções especificada no código original.**

# Blocos Básicos

---

- **Definição:**
  - É um trecho de código seqüencial, onde só existe um ponto de entrada e um ponto de saída
- **Conseqüências:**
  - Qualquer instrução de desvio encerra um bloco básico;
  - Qualquer instrução que possa ser o destino de uma instrução de desvio deve iniciar um bloco básico;
  - Todas as instruções de um bloco básico são executadas seqüencialmente do início até o final do bloco;
  - Qualquer alteração na ordem da execução das instruções de um bloco básico, que respeite as dependências de dados, não afeta o resto do programa.

# Divisão em Blocos Básicos

---

<b>I1:</b>	<b>dadd</b>	<b>R1, R2, R3</b>
<b>I2:</b>	<b>dsub</b>	<b>R3, R4, R20</b>
<b>I3:</b>	<b>dsubi</b>	<b>R3, R6, #1</b>
<b>I4:</b>	<b>daddu</b>	<b>R20, R6, R7</b>
<b>I5:</b>	<b>sllv</b>	<b>R7, R3, R8</b>
<b>I6:</b>	<b>srli</b>	<b>R9, R10, #1</b>
<b>I7:</b>	<b>daddi</b>	<b>R9, R11 , #3</b>
<b>I8:</b>	<b>bnez</b>	<b>R9, I17</b>
<b>I9:</b>	<b>mul</b>	<b>R12, R13</b>
<b>I10:</b>	<b>srai</b>	<b>R10, R15 , #2</b>
<b>I11:</b>	<b>or</b>	<b>R13, R14, R16</b>
<b>I12:</b>	<b>and</b>	<b>R16, R11, R17</b>
<b>I13:</b>	<b>daddu</b>	<b>R17, R5, R18</b>
<b>I14:</b>	<b>j</b>	<b>I18</b>
<b>I15:</b>	<b>div</b>	<b>R3, R4</b>
<b>I16:</b>	<b>bne</b>	<b>R1, R2, I6</b>
<b>I17:</b>	<b>dadd</b>	<b>R13, 3, R12</b>
<b>I18:</b>	<b>dsub</b>	<b>R5, R4, R2</b>

# Divisão em Blocos Básicos

---

<b>I1:</b>	<b>dadd</b>	<b>R1, R2, R3</b>
<b>I2:</b>	<b>dsub</b>	<b>R3, R4, R20</b>
<b>I3:</b>	<b>dsubi</b>	<b>R3, R6, #1</b>
<b>I4:</b>	<b>daddu</b>	<b>R20, R6, R7</b>
<b>I5:</b>	<b>sllv</b>	<b>R7, R3, R8</b>
<b>I6:</b>	<b>srli</b>	<b>R9, R10, #1</b>
<b>I7:</b>	<b>daddi</b>	<b>R9, R11, #3</b>
<b>I8:</b>	<b>bnez</b>	<b>R9, I17</b>
<b>I9:</b>	<b>mul</b>	<b>R12, R13</b>
<b>I10:</b>	<b>srai</b>	<b>R10, R15, #2</b>
<b>I11:</b>	<b>or</b>	<b>R13, R14, R16</b>
<b>I12:</b>	<b>and</b>	<b>R16, R11, R17</b>
<b>I13:</b>	<b>daddu</b>	<b>R17, R5, R18</b>
<b>I14:</b>	<b>j</b>	<b>I18</b>
<b>I15:</b>	<b>div</b>	<b>R3, R4</b>
<b>I16:</b>	<b>bne</b>	<b>R1, R2, I6</b>
<b>I17:</b>	<b>dadd</b>	<b>R13, 3, R12</b>
<b>I18:</b>	<b>dsub</b>	<b>R5, R4, R2</b>



# Divisão em Blocos Básicos

---

- **Bloco B1: Inst 1 a Inst 5**
- **Bloco B2: Inst 6 a Inst 8**
- **Bloco B3: Inst 9 a Inst 14**
- **Bloco B4: Inst 15 a Inst16**
- **Bloco B5: Inst 17**
- **Bloco B6: Inst 18 a ...**

# Dependência de Dados

---

- **Dependência Direta**

dadd        **R1**, R2, R3

dsub        R4, **R1**, R5

- **Anti-Dependência:**

dadd        R1, **R2**, R3

dsub        **R2**, R4, R5

- **Dependência de Saída:**

dadd        **R1**, R2, R3

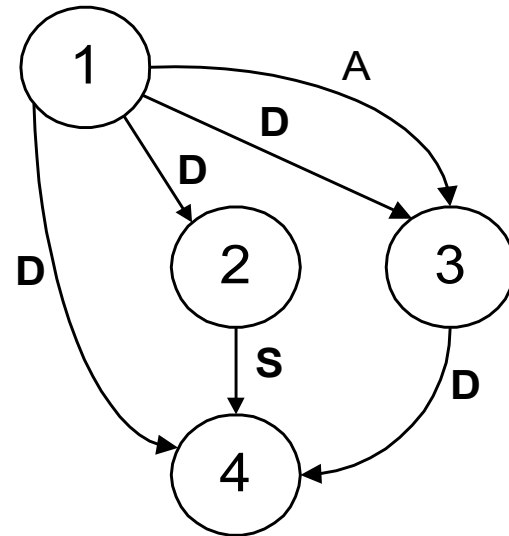
dsub        **R1**, R4, R5

# Grafo de Dependências

---

- Trecho de Código

I1: dadd     **R1**, **R2**, R3  
I2: dsub     **R4**, R0, **R1**  
I3: dadd     **R2**, **R1**, R5  
I4: dadd     **R4**, **R2**, **R1**



**D -> Dependência Direta ou Verdadeira**

**S -> Dependência de Saída**

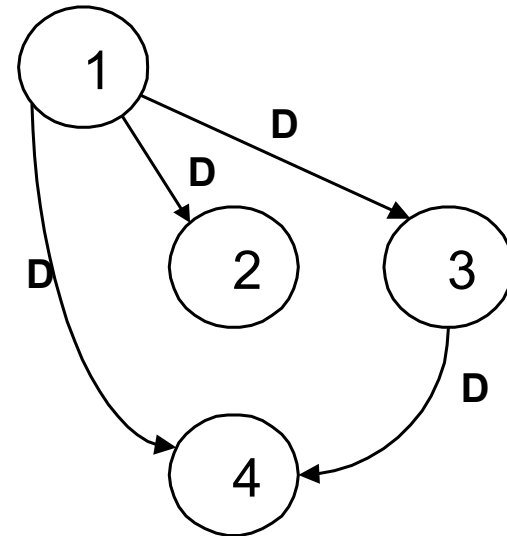
**A -> Anti-Dependência**

# Grafo de Dependências

---

- Trecho de Código

I1: dadd     **R1**, **R2**, R3  
I2: dsub     **R4**, R0, **R1**  
I3: dadd     **R12**, **R1**, R5  
I4: dadd     **R14**, **R12**, **R1**



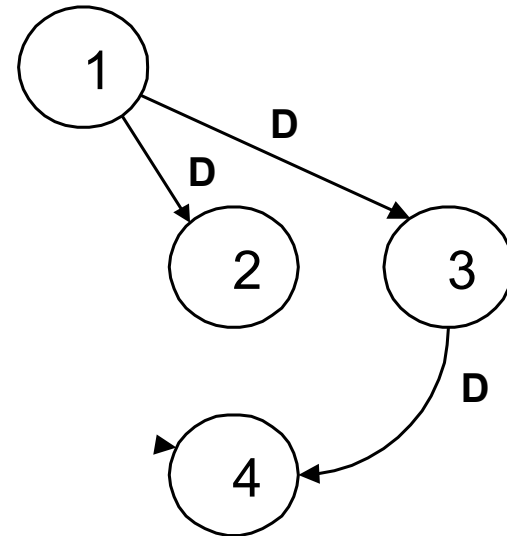
**Grafo depois de eliminadas as dependências falsas e as dependências redundantes. O grafo final não pode conter ciclos. Para eliminação das dependências falsas utilizamos a renomeação.**

# Grafo de Dependências

---

- Trecho de Código

I1: dadd    **R1**, **R2**, R3  
I2: dsub    **R4**, R0, **R1**  
I3: dadd    **R12**, **R1**, R5  
I4: dadd    **R14**, **R12**, **R1**



**Grafo depois de eliminadas as dependências falsas e as dependências redundantes. O grafo final não pode conter ciclos. Para eliminação das dependências falsas utilizamos a renomeação.**

# Renomeação de Registradores

---

- **Objetivo:** reduzir o número de dependências falsas, trocando os registradores utilizados como operandos nas instruções.
- Um registrador é dito *ativo* quando o valor por ele armazenado durante a execução do bloco é utilizado posteriormente no decorrer do programa
- Qualquer registrador *inativo* pode ser utilizado para renomear operandos e eliminar dependências falsas
- **Ex:**

## ORIGINAL

I1: dadd R1, R2, R3  
I2: dsub R4, R0, R1  
I3: dadd R2, R1, R5  
I4: dadd R4, R2, R1  
I5: dsub R6, R4, R2

## RENOMEADO

I1: dadd R1, R2, R3  
I2: dsub R4, R0, R1  
I3: dadd R7, R1, R5  
I4: dadd R8, R7, R1  
I5: dsub R6, R8, R7

# Renomeação de Registradores

---

## ORIGINAL

```
L.D    F0 , x (R0)
L.D    F2 , x (R0)
L.D    F4 , s (R0)
DADDI  R1 , R0 , #2
L.D    F2 , (0) R1
MUL.D  F4 , F6 , F2
ADD.D  F4 , F4 , F8
MUL.D  F2 , F0 , F2
DADDI  R1 , R1 , #1
SLTI   R2 , R1 , #22
```

## RENOMEADO

```
L.D    F0 , x (R0)
L.D    F2 , x (R0)
L.D    F4 , s (R0)
DADDI  R1 , R0 , #2
L.D    F12 , (0) R1
MUL.D  F14 , F6 , F12
ADD.D  F24 , F14 , F8
MUL.D  F22 , F0 , F12
DADDI  R11 , R1 , #1
SLTI   R2 , R11 , #22
```

# Escalonamento de Instruções

---

- **Realizados os passos de divisão em blocos básicos e renomeação, o próximo passo é escalonar as instruções dentro de cada bloco básico.**
- **O algoritmo de “List Scheduling” é o de uso mais freqüente nesses casos.**
- **Para realização do algoritmo é necessário que tenhamos, além do grafo de dependências acíclico, as seguintes informações:**
  - **Recursos de hardware utilizados por cada instrução, ou seja, a unidade funcional para qual ela se destina.**
  - **A latência de cada instrução, ou seja, o número de ciclos que leva para ser executada na unidade funcional correspondente.**
  - **O total de recursos disponíveis no processador, ou seja, o número de unidades funcionais de cada tipo.**

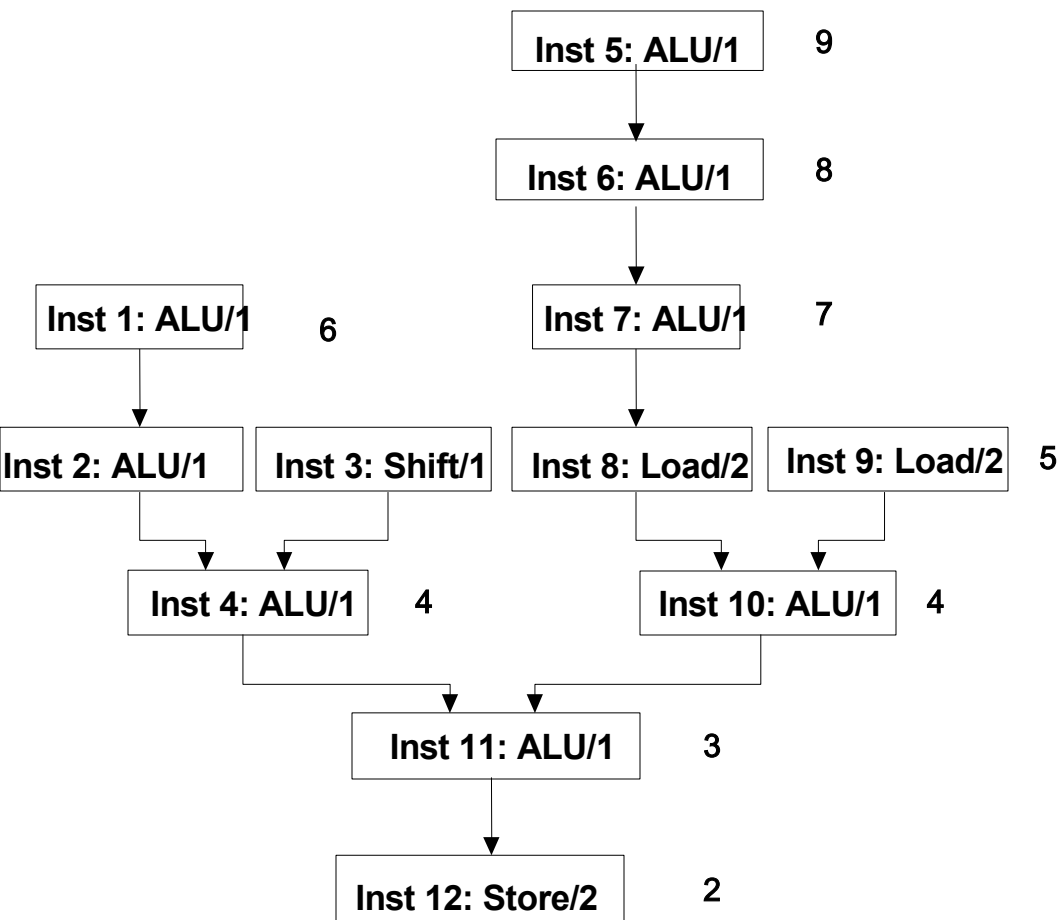


# Escalonamento “Top Down”

---

- No percurso de **cima para baixo**, o escalonamento é feito do primeiro para o último ciclo. Ou seja, as instruções que vão ser executadas no último ciclo são escolhidas primeiro.
- A prioridade de cada instrução para o escalonamento é dada pela latência acumulada no percurso do grafo de dependências de baixo para cima até essa instrução. Uma instrução é escalonada para execução no primeiro ciclo, em contagem crescente, que satisfaça as seguintes condições:
  - Todos os seus operandos já estão prontos;
  - Não há instrução mais prioritária pronta com conflito de recursos.

# Algoritmo List Scheduling Top Down



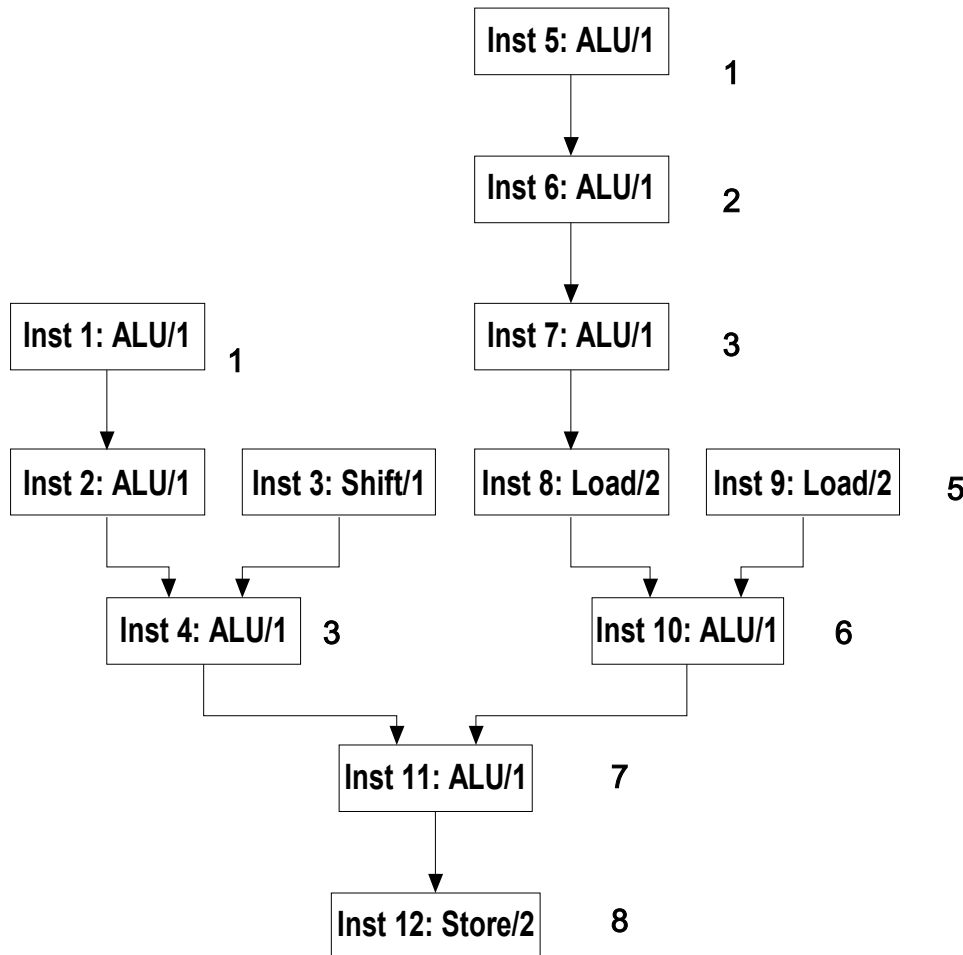
Ciclo	ALU 1	ALU 2	LD/ STO	SHF
1	I5	I1	I9	I3
2	I6	I2		
3	I7	I4		
4			I8	
5				
6	I10			
7		I11		
8			I12	

# Escalonamento de Instruções

---

- No percurso de **baixo para cima**, o escalonamento é feito do último para o primeiro ciclo. A prioridade de cada instrução para o escalonamento é dada pela latência acumulada do grafo de dependências de cima para baixo. Uma instrução é escalonada para execução no primeiro ciclo, em contagem decrescente, que satisfaça as seguintes condições:
  - Todas as instruções sucessoras já foram escalonadas
  - A distância entre esse ciclo e o menor ciclo onde foi escalonada uma instrução sucessora é igual ou maior que a latência da instrução
  - Não há instrução mais prioritária pronta com conflito de recursos

# Algoritmo List Scheduling Bottom-Up



Ciclo	ALU 1	ALU 2	LD/ STO	SHF
8			I12	
7		I11		
6	I10	I4		
5	I2			I3
4		I1	I8	
3	I7		I9	
2		I6		
1	I5			

# Trace Scheduling

---

- **Algoritmo proposto por Fischer em 1979 para compactação de operações em microinstruções horizontais**
- **Principal técnica usada para escalonamento de operações por software em arquiteturas VLIW ou superescalares**
- **Técnica baseada no conceito de traces, caminhos possíveis que, em geral, atravessam vários blocos básicos**
- **Um trace é determinado assumindo direções a serem tomadas a cada desvio existente no caminho**
- **O algoritmo utiliza o conceito de escalonamento de instruções, aplicado anteriormente a apenas um bloco básico, para otimizar um trace completo, permitindo que o processo de otimização vá além das instruções de desvio**

# Trace Scheduling

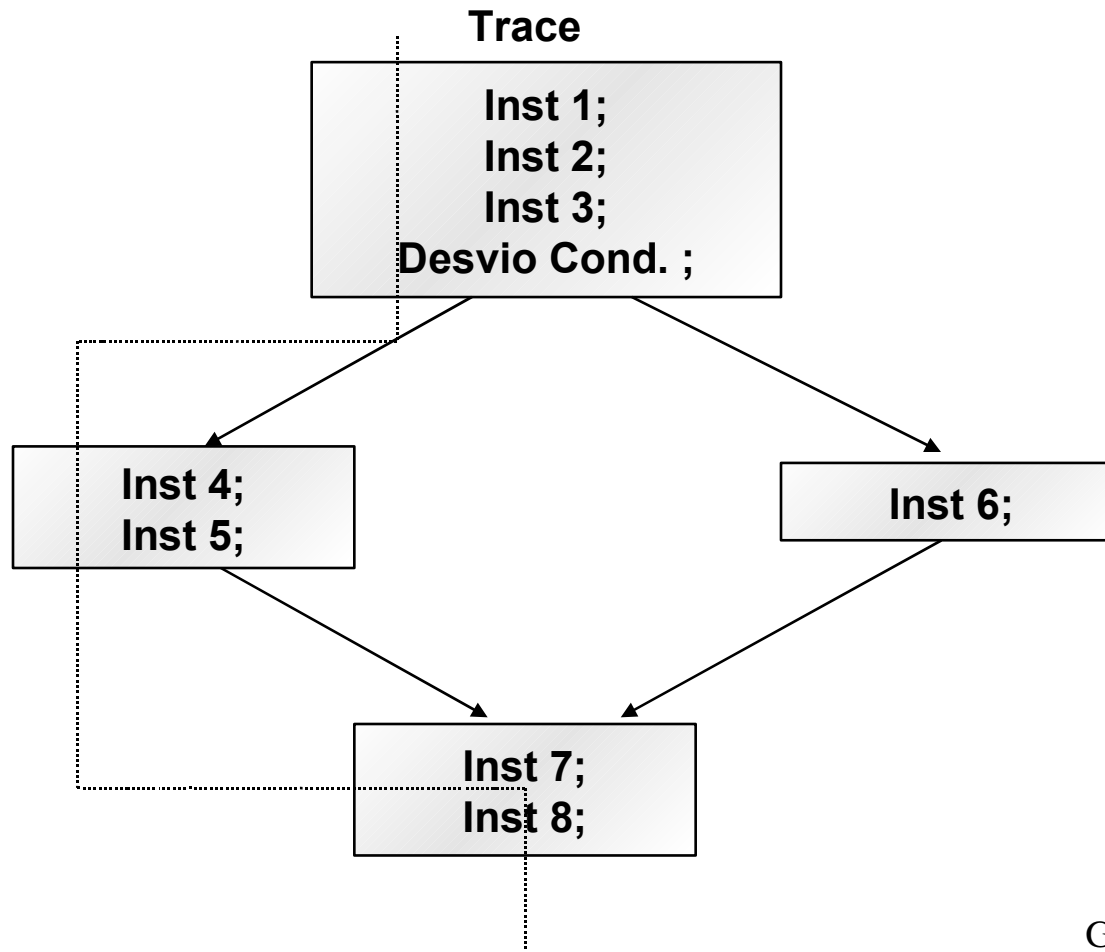
---

- **O algoritmo utiliza profiles de execuções anteriores do programa ou predições estáticas de desvio para ordenar os traces segundo a probabilidade de serem executados.**
- **Ao ser feita a otimização de um dado trace, é necessário inserir código de reparo ou de compensação em blocos básicos situados fora do trace para corrigir o efeito de eventuais predições erradas.**
- **Na otimização dos traces subseqüentes, os eventuais códigos de reparo inseridos são também considerados no escalonamento otimizado das instruções.**

# Código de Reparo

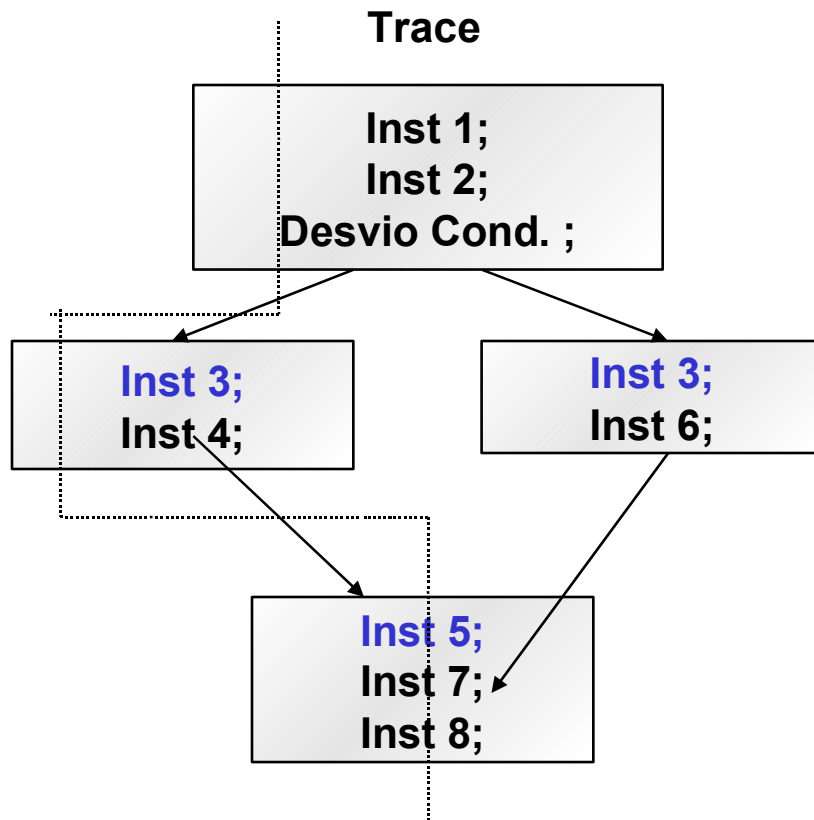
---

- **Código Original:**

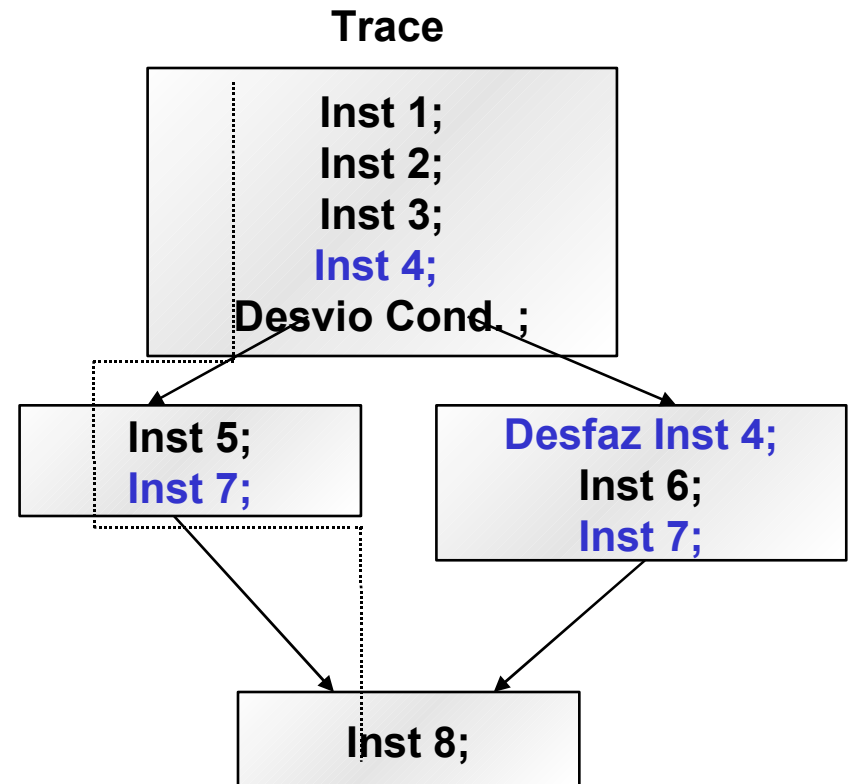


# Código de Reparo

- Escalonamento com Movimentação para Blocos Posteriores



- Escalonamento com Movimentação para Blocos Anteriores





# Código de Reparo

---

- **Quando uma instrução é movida para um bloco abaixo de um desvio, essa instrução deve ser copiada no bloco que é executado quando o desvio produz um resultado diferente do assumido no trace.**
- **Quando uma instrução é movida para um bloco abaixo, que pode ser alcançado por outros caminhos que não trace, os desvios para esse bloco, nesses caminhos, devem ser ajustados para que a junção com o trace se dê num ponto abaixo da instrução movimentada. Se para isso, alguma instrução for indevidamente saltada, essa instrução deve ser copiada nesses outros caminhos em algum ponto anterior ao desvio ajustado.**

# Código de Reparo

---

- **Quando uma instrução é movida para cima, a partir de um bloco que pode ser atingido por outros caminhos que não o do trace, essa instrução deve ser copiada em todos outros caminhos , assegurando sua execução, independente do caminho seguido pelo programa.**
- **Quando uma instrução é movida para antes de um desvio, qualquer mudança de estado provocada por essa instrução deve ser desfeita no bloco que é executado quando o resultado do desvio é diferente daquele assumido pelo trace.**
- **Para desfazer o efeito de uma instrução, o escalonador pode usar dois artifícios: realizar uma operação inversa (solução nem sempre aplicável) ou inserir código para explicitamente guardar o estado anterior.**

# Aspectos Relevantes

---

- **Se a estimativa dos traces que têm maior probabilidade de serem executados não for precisa, é possível que o algoritmo insira muito código de reparo nos traces que são de fato os mais executados. Nesse caso o desempenho final pode até piorar: a predição dos desvios deve ser muito precisa.**
- **O algoritmo é mais eficiente quando o código de reparo é fácil de construir por ser pequeno e simples: maior eficiência nas arquiteturas RISC.**
- **O algoritmo otimiza um trace em detrimento de outros: maior chance de dar bons resultados em programas com baixa frequência de instruções de desvio.**
- **A otimização produzida pelo algoritmo pode mover instruções “guardadas” por um desvio condicional para antes do desvio: possibilidade de geração de exceções falsas. A adoção de uma política mais segura de movimentação de instruções para antes de um desvio é necessária, o que pode reduzir a eficiência dos algoritmos**

# Renomeação de Registradores

---



**No exemplo a seguir consideramos uma arquitetura com um número ilimitado de unidades funcionais, com as latências acima, e também com um número ilimitado de registradores disponíveis para renomeação.**

# Renomeação de Registradores

---

O código à esquerda em linguagem “C” gera o código em linguagem de montagem do MIPS64 à direita.

```
For (j=0; j<n; j++)  
{  
    C(j) = A(j)+B(j)  
}
```

Laço Original

```
L1:    l.d      F2, A (R1)      (a)  
      l.d      F3, B (R1)      (b)  
      add.d    F4, F2, F3      (c)  
      s.d      C (R1), F4      (d)  
      dadd    R1, R1, # 8      (e)  
      bne     R1, R5, L1      (f)
```

Código em Linguagem  
de Montagem MIPS64

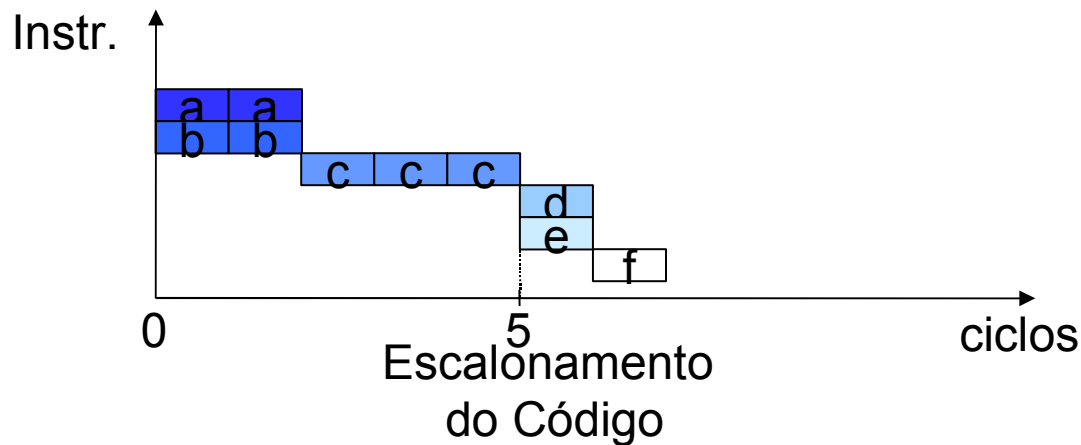
# Renomeação de Registradores

```
For (j=0; j<n; j++)  
{  
  C(j) = A(j)+B(j)  
}
```

laço Original

```
L1: l.d    F2, A (R1)    (a)  
    l.d    F3, B (R1)    (b)  
    add.d  F4, F2, F3    (c)  
    s.d    C (R1), F4    (d)  
    dadd   R1, R1, # 8    (e)  
    bne    R1, R5, L1    (f)
```

Código em Linguagem  
de Máquina MIPS64



**Desempenho: 7 ciclos / 1 iteração**

# Desenrolamento do laço

---

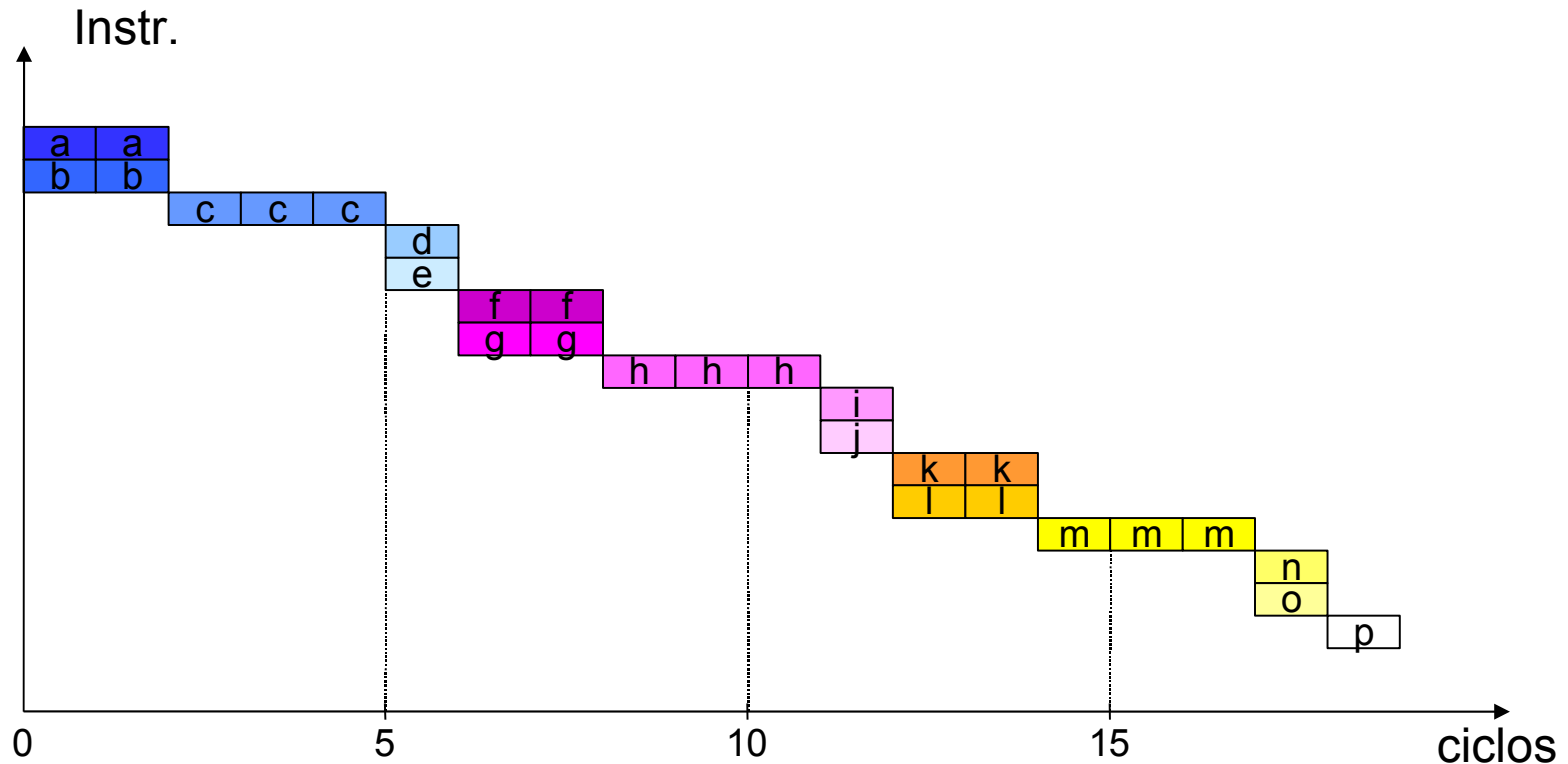
```
L1: l.d    F2, A (R1) (a)
     l.d    F3, B (R1) (b)
     add.d  F4, F2, F3 (c)
     s.d    C (R1), F4 (d)
     dadd   R1, R1, # 8 (e)
     bne   R1, R5, L1 (f)
```

**Código Original**

```
L1: l.d    F2, A (R1) (a)
     l.d    F3, B (R1) (b)
     add.d  F4, F2, F3 (c)
     s.d    C (R1), F4 (d)
     dadd   R1, R1, # 8 (e)
     l.d    F2, A (R1) (f)
     l.d    F3, B (R1) (g)
     add.d  F4, F2, F3 (h)
     s.d    C (R1), F4 (i)
     dadd   R1, R1, # 8 (j)
     l.d    F2, A (R1) (k)
     l.d    F3, B (R1) (l)
     add.d  F4, F2, F3 (m)
     s.d    C (R1), F4 (n)
     dadd   R1, R1, # 8 (o)
     bne   R1, R5, L1 (p)
```

**Depois do Desenrolamento  
do laço**

# Desenrolamento do laço



Escalonamento do Código

Depois:  $19 \text{ ciclos} / 3 \text{ iterações} = 6.3 \text{ ciclos / iteração}$



# Renomeação de Registradores

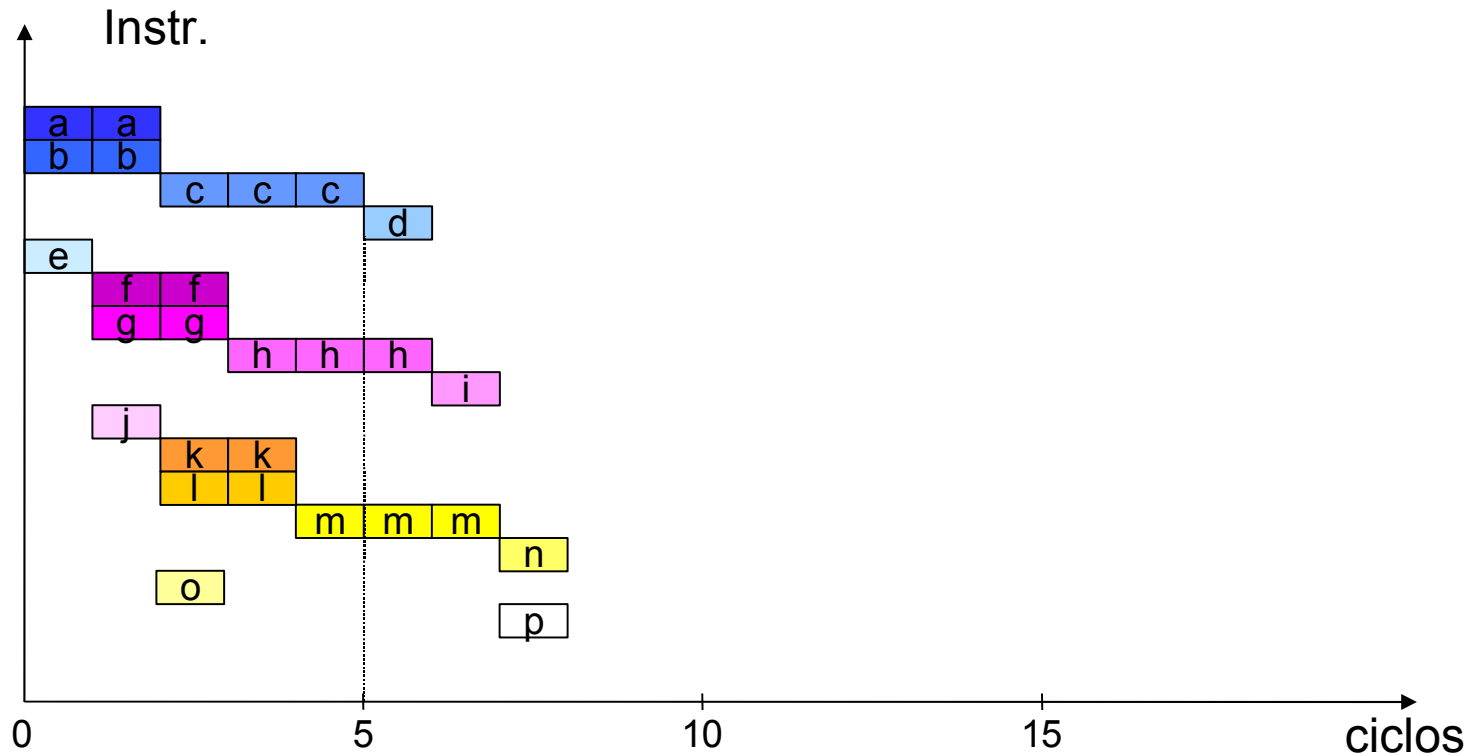
L1:	l.d	F2, A (R1)	(a)
	l.d	F3, B (R1)	(b)
	add.d	F4, F2, F3	(c)
	s.d	F4, C (R1)	(d)
	dadd	R1, R1, # 8	(e)
	l.d	F2, A (R1)	(f)
	l.d	F3, B (R1)	(g)
	add.d	F4, F2, F3	(h)
	s.d	F4, C (R1)	(i)
	dadd	R1, R1, # 8	(j)
	l.d	F2, A (R1)	(k)
	l.d	F3, B (R1)	(l)
	add.d	F4, F2, F3	(m)
	s.d	F4, C (R1)	(n)
	dadd	R1, R1, # 8	(o)
	bne	R1, R5, L1	(p)

Depois do  
Desenrolamento

L1:	l.d	F21, A(R11)	(a)
	l.d	F31, B(R11)	(b)
	add.d	F41, F21, F31	(c)
	s.d	F41, C(R11)	(d)
	dadd	R12, R11, #4	(e)
	l.d	F22, A(R12)	(f)
	l.d	F32, B(R12)	(g)
	add.d	F42, f22, f32	(h)
	s.d	f42, C(R12)	(i)
	dadd	R13, R12, #4	(j)
	l.d	F23, A(R13)	(k)
	l.d	F33, B,(R13)	(l)
	add.d	F43, F23, F33	(m)
	s.d	F43, C(R13)	(n)
	dadd	R11, R13, #4	(o)
	bne	R11, R5, L1	(p)

Depois da  
Renomeação

# Desenrolamento de laço e Renomeação de Registradores



Escalonamento do Código

$$8 \text{ ciclos} / 3 \text{ iterações} = 2.7 \text{ ciclos} / \text{ iteração}$$

# Software Pipelining

---

- É uma técnica de escalonamento para processadores com múltiplas unidades funcionais, que busca otimizar o código de um laço explorando paralelismo entre instruções distintas do laço.

- Exemplo:

```
for i = 1 to 7 do
{
    b(i) = 2.0 * a(i);
}
```

# Software Pipelining

---

▪ Assumindo que a o processador em questão tenha unidades suficientes para a execução de até quatro instruções por ciclo e sendo o seguinte código em linguagem de montagem:

```
1 load r101, b(1) ; /* b(1) é carregado
2 fmul r101, 2.0, r101;
3 decr r200; /* decrementa o laço
4 nop /* Espera o resultado
5 store a(i)+, r101 /* armazena a(i),autoincrementa
```

▪ Neste código supõe-se que a unidade de multiplicação de ponto flutuante (“pipelined”) leva 3 ciclos para apresentar um resultado.

▪ Uma segunda iteração pode-se iniciar no ciclo 2, contudo, o registrador r101 precisa ser renomeado para, p. ex. , r102.

# Software Pipelining

- Assim sendo, teremos a seguinte execução no tempo:

Ciclo	Número da Iteração						
	1	2	3	4	5	6	7
1	load						
2	fmul	load					
3	decr	fmul	load				
4	nop	decr	fmul	load			
5	store	nop	decr	fmul	load		
6		store	nop	decr	fmul	load	
7			store	nop	decr	fmul	load
8				store	nop	decr	fmul
9						store	nop
10	decr						
11	nop						store
							store

# Software Pipelining

---

- No ciclo 5 as seguintes operações são realizadas em paralelo: store (iteração 1), nop (iter. 2), decr (iter. 3), fmu (iter, 4), load (iter. 5)
- Os ciclos de 5 a 7 tem um padrão repetitivo que pode ser re-escrito da seguinte maneira:

```
laço: store (i); decr (i+2); fmul (i+3); load(i+4); bc laço;  
/* Este laço tem que ser executado para i=1 a 3 */
```

- Depois deste escalonamento, o que se obtém é a execução “software-pipelined” do laço, que consiste basicamente de 3 partes: prólogo, novo corpo do “laço” e epílogo.

# Software Pipelining

---

- **No exemplo dado cada iteração do novo “laço” pode ser executada em apenas um ciclo.**
- **Determinar o melhor padrão repetitivo para compor o novo corpo do “laço” e que seja executado no menor número possível de ciclos é o grande problema na técnica de “software pipelining”.**
- **Esta técnica serve tanto para processadores superescalares e VLIW, como para arquiteturas SIMD. Neste caso, cada iteração do novo “laço” é alocada para um processador diferente.**
- **Essa técnica de escalonamento busca reduzir o intervalo de iniciação entre duas iterações e não a duração de cada iteração, criando um pipeline em software entre as iterações do laço.**

# Software Pipelining

---

```
for (i=n; i > 0; i--)  
    A(i) = A(i) + B;
```

- Neste exemplo, todos os elementos de um vetor são incrementados, a partir do endereço em R1, com o conteúdo de F2. O registrador R1 aponta inicialmente para a última posição do vetor.

```
LOOP:          L.D          F0, 0(R1)  
              ADD.D       F4, F0, F2  
              S.D         F4, 0(R1)  
              DADDUI      R1, R1, #-8  
              BNE        R1, R2, LOOP
```



# Software Pipelining

---

- Realizamos então o desenrolamento “simbólico” do laço, colocando três iterações sucessivas e escolhendo uma instrução de cada iteração.
- Como esta operação é simbólica, não precisamos das instruções para decrementar R1 e testar o fim do laço.

Iteração i:	L.D	F0, 0(R1)
	ADD.D	F4, F0, F2
	<b>S.D</b>	<b>F4, 0(R1)</b>
Iteração i+1:	L.D	F0, 0(R1)
	<b>ADD.D</b>	<b>F4, F0, F2</b>
	S.D	F4, 0(R1)
Iteração i+2:	<b>L.D</b>	<b>F0, 0(R1)</b>
	ADD.D	F4, F0, F2
	S.D	F4, 0(R1)

# Software Pipelining

---

- Laço final com “software pipeline” com código de reparo antes e depois da execução no novo laço.

	L.D	F0, 0(R1)	
	ADD.D	F4, F0, F2	
	DADDUI	R1, R1, #-8	; R1 = i-1
	L.D	F0, 0(R1)	
	DADDUI	R1, R1, #-8	; R1 = i-2
LOOP:	S.D	F4, 16(R1)	; armazena em M[i]
	ADD.D	F4, F0, F2	; soma para M[i-1]
	L.D	F0, 0(R1)	; carrega para M[i-2]
	DADDUI	R1, R1, #-8	; atualiza o valor de i
	BNE	R1, R2, LOOP	; fim no novo “loop”
	S.D	F4, 8(R1)	; armazena para M[i-1]
	ADD.D	F4, F0, F2	
	S.D	F4, 0(R1)	; armazena para M [i-2]