

DCC007 – Organização de Computadores II

Aula 11 – Limite de Superescalar / VLIW

Prof. Omar Paranaíba Vilela Neto



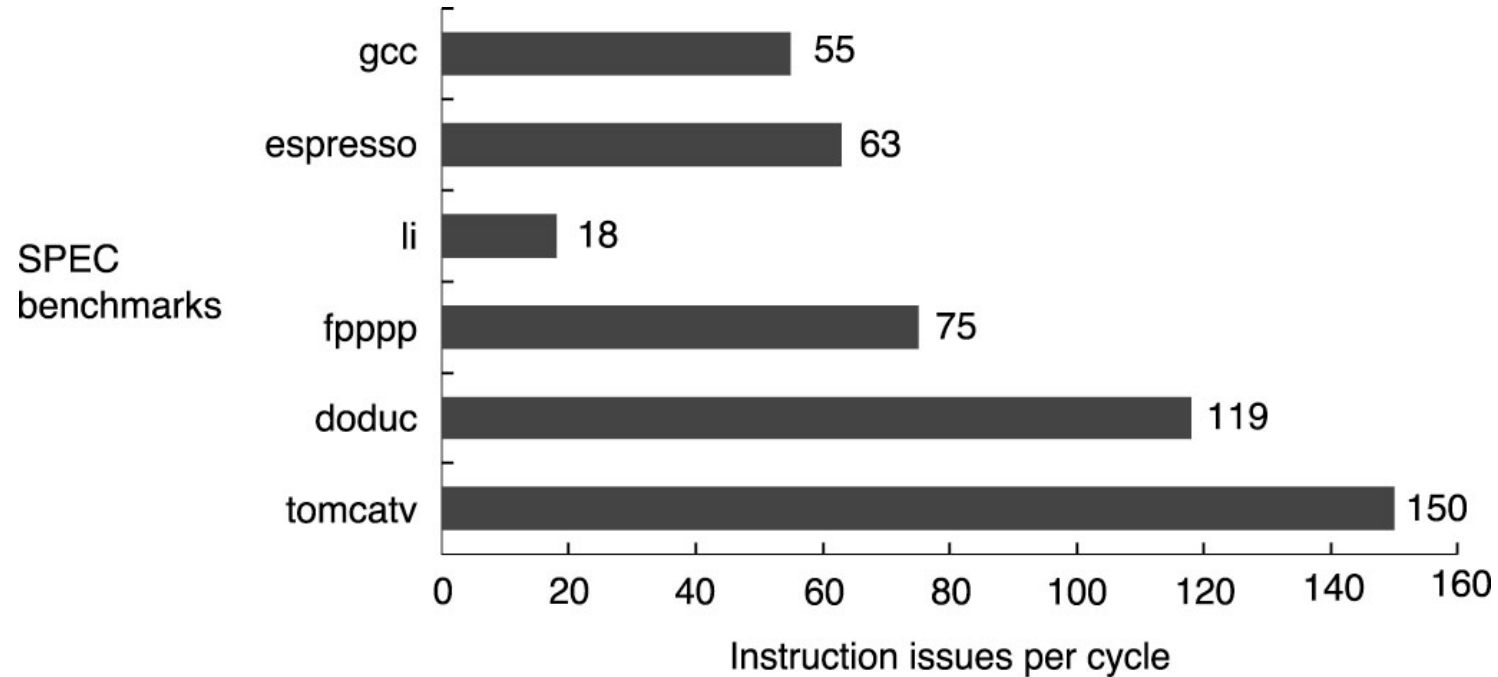
Limites para ILP

Modelo ideal de hardware; compiladores MIPS

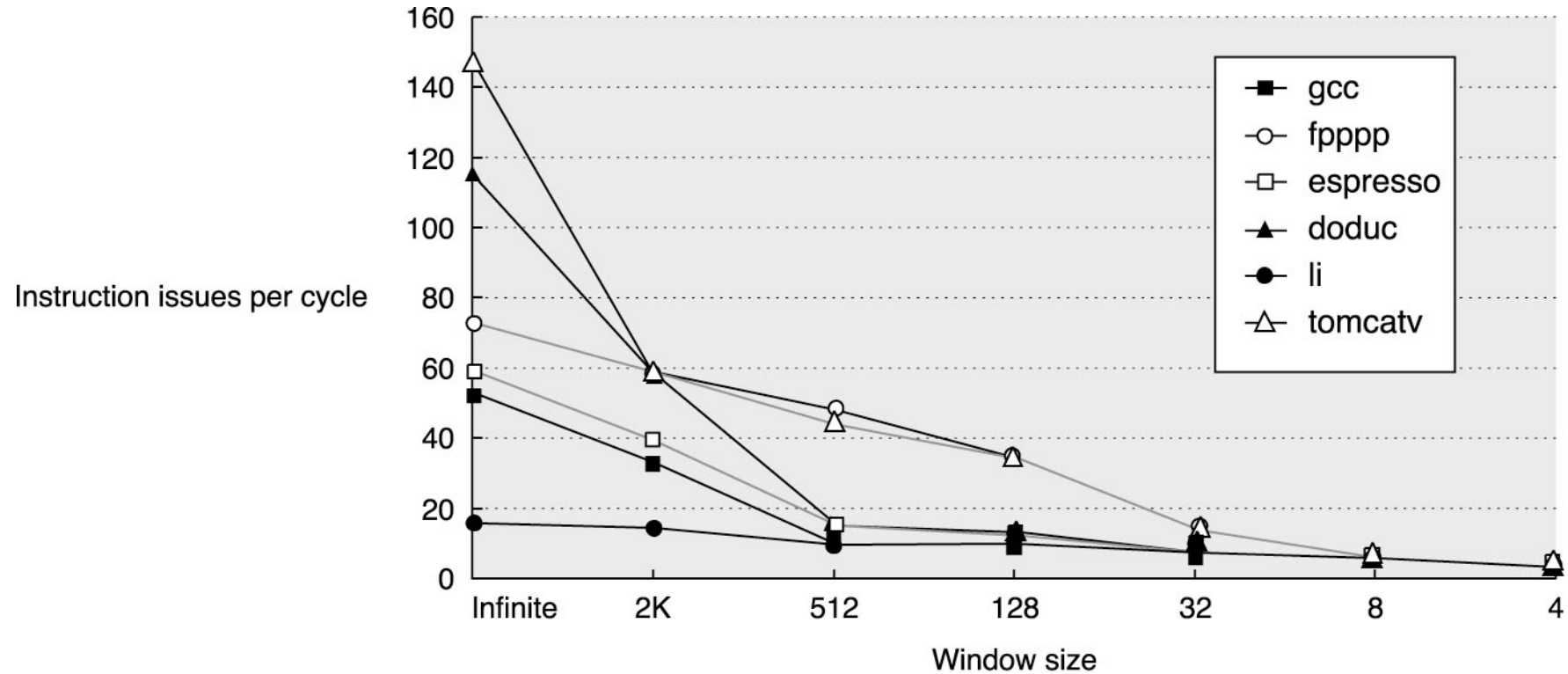
1. *Register renaming*—número infinito de registradores virtuais para evitar todos os hazards de WAW & WAR
2. *Branch prediction*—perfeito; sem previsões erradas
3. *Jump prediction*—previsões perfeitas => máquina com especulação perfeita & buffer de instruções ilimitado
4. *Análise de sinônimos de memória*—endereços são conhecidos e stores podem ser movidos antes de loads se os endereços não forem iguais

1 ciclo de latência para todas as instruções

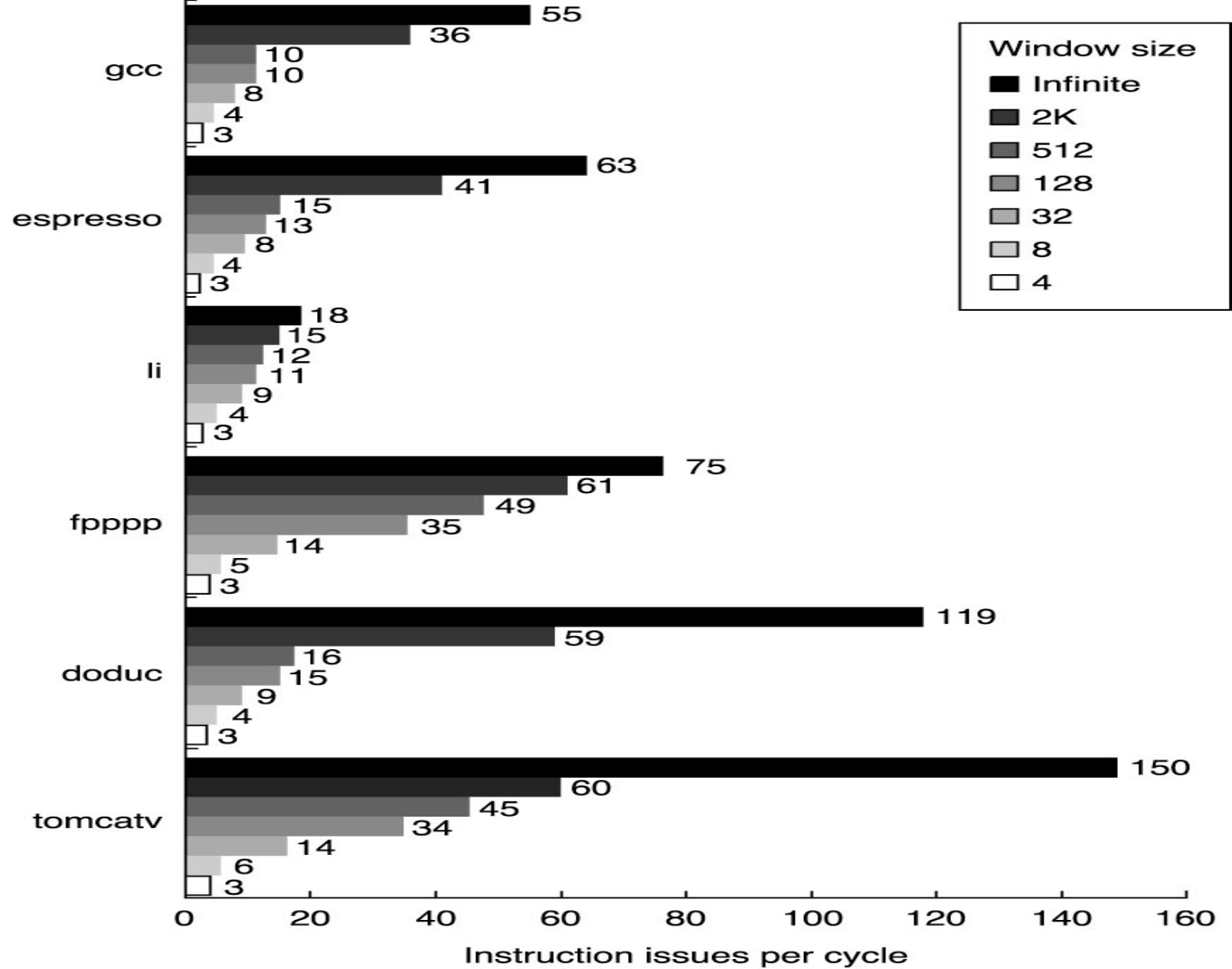
Limite Superior para ILP



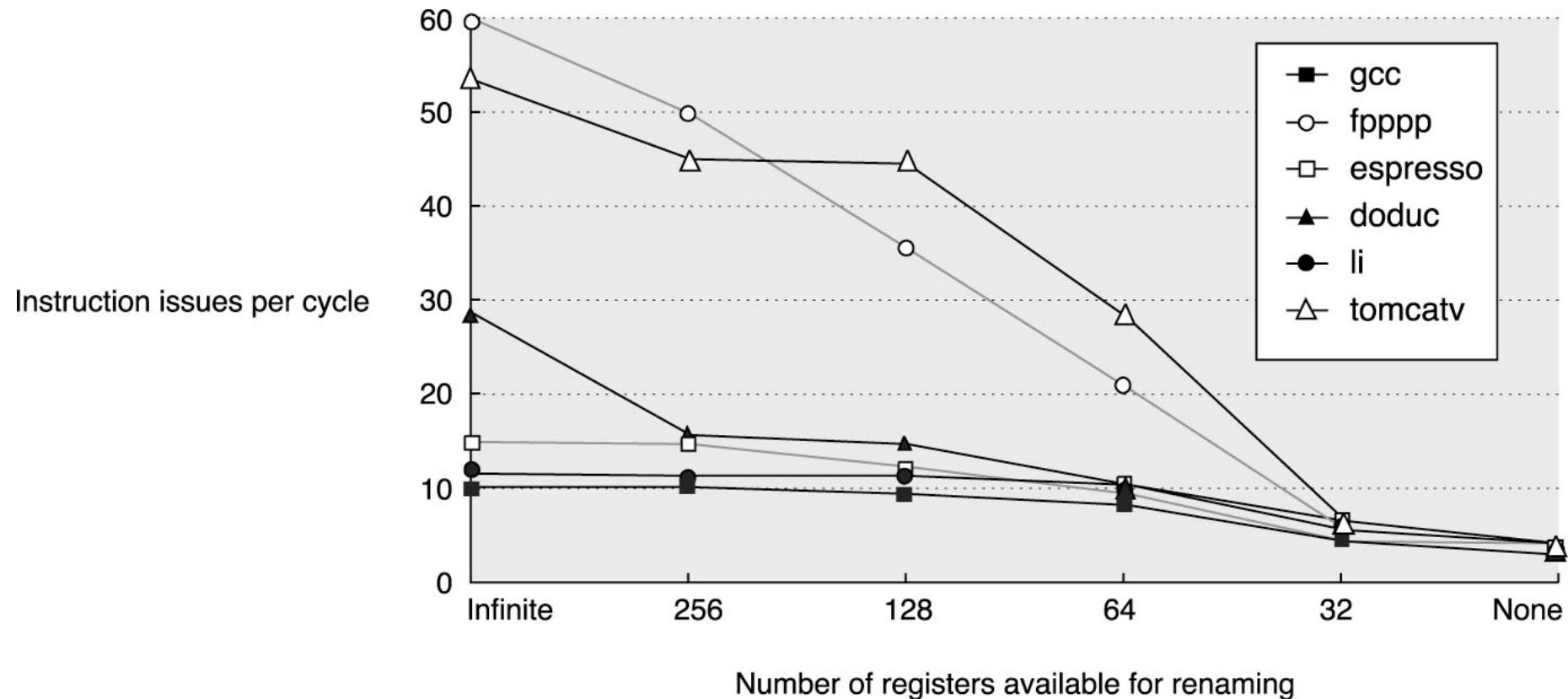
Impacto da Janela

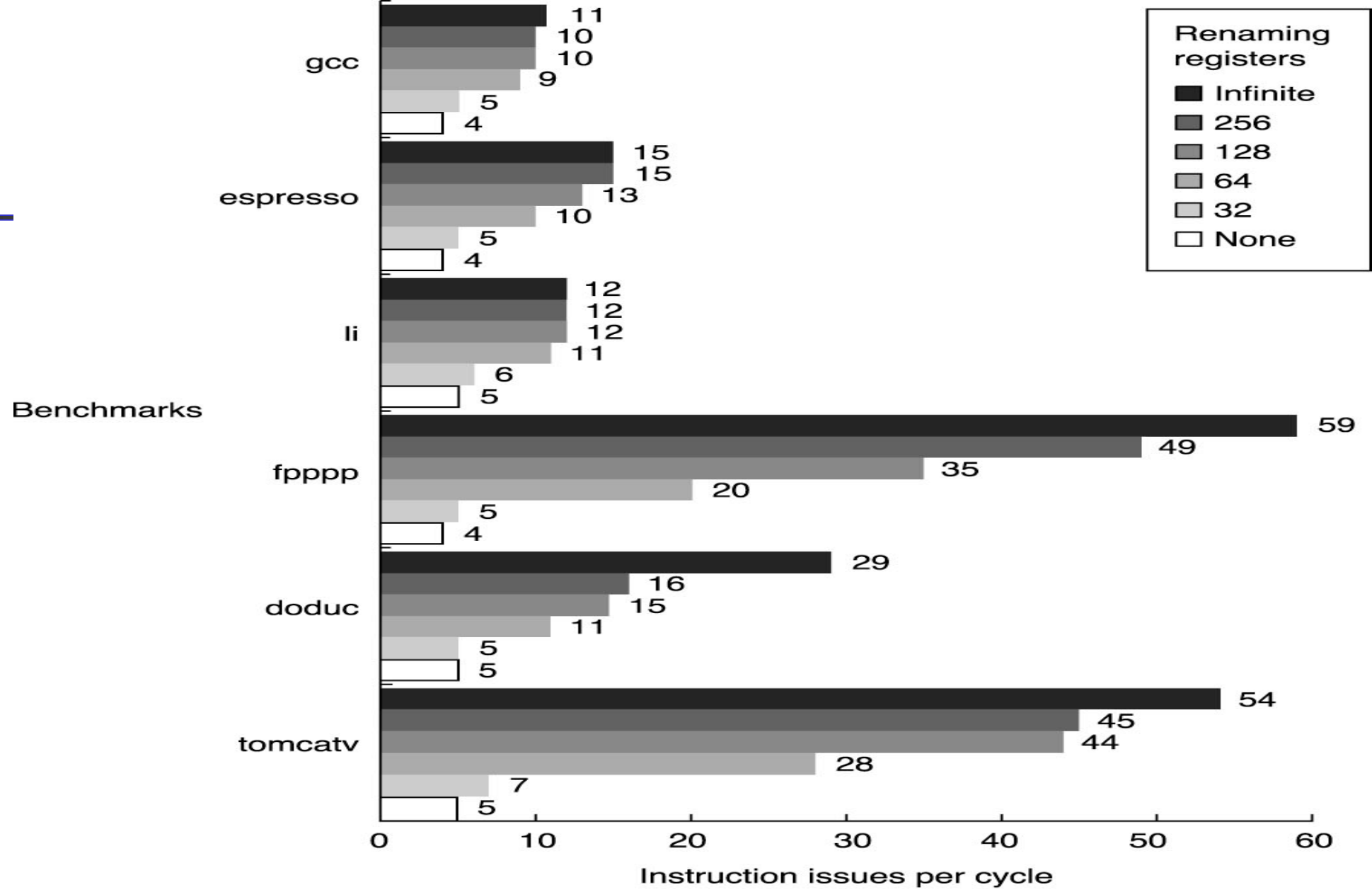


Benchmarks

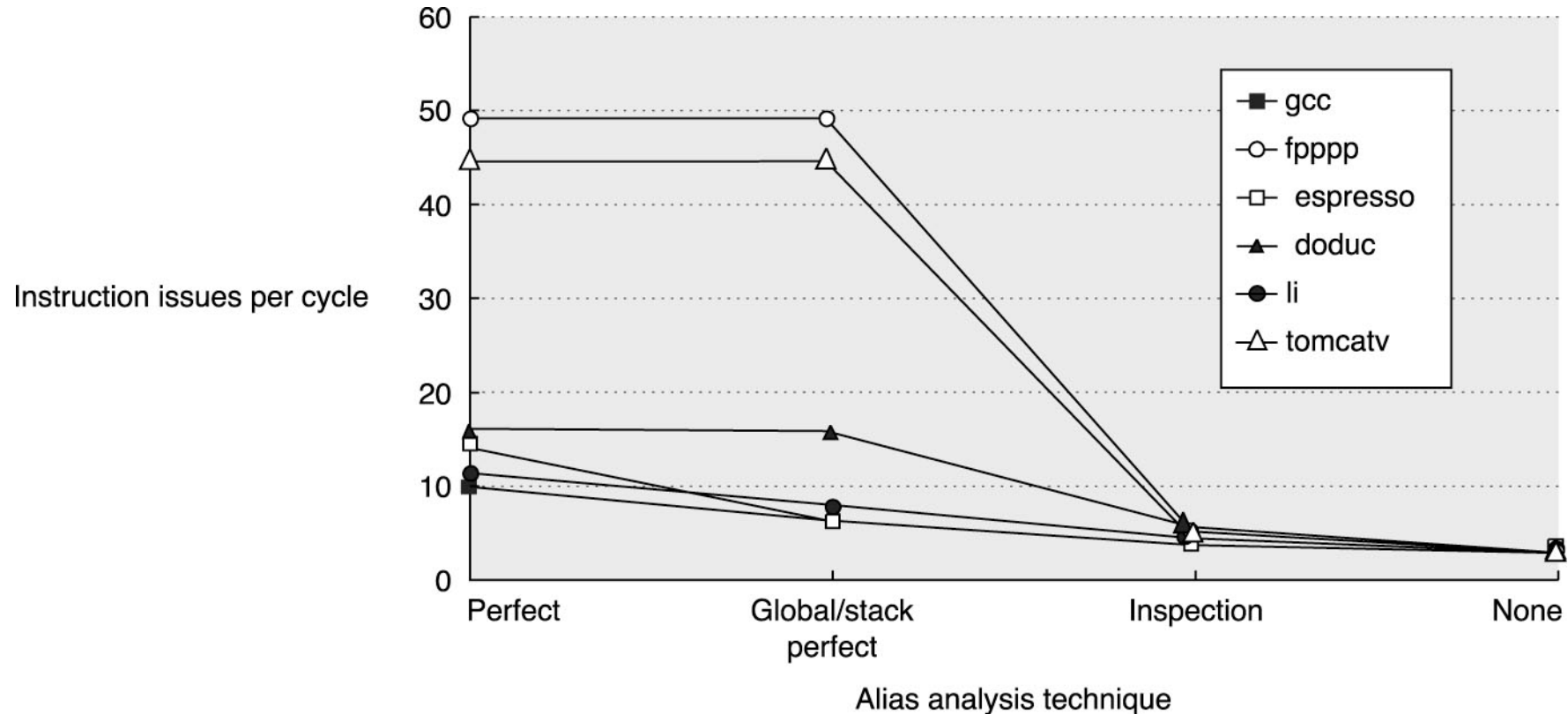


Efeito do Número de Registradores – Renom.

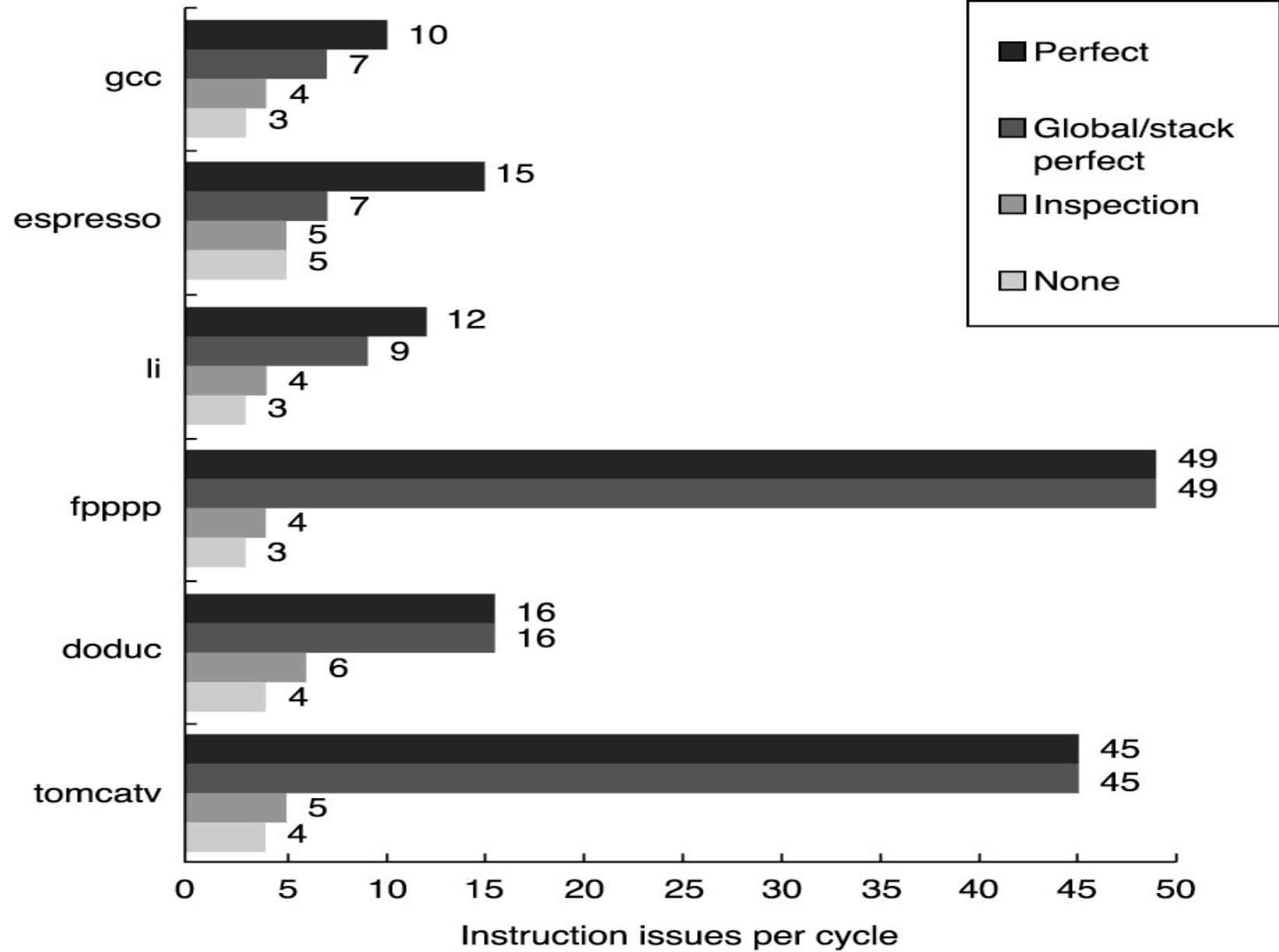




Impacto de Alias (Memória)

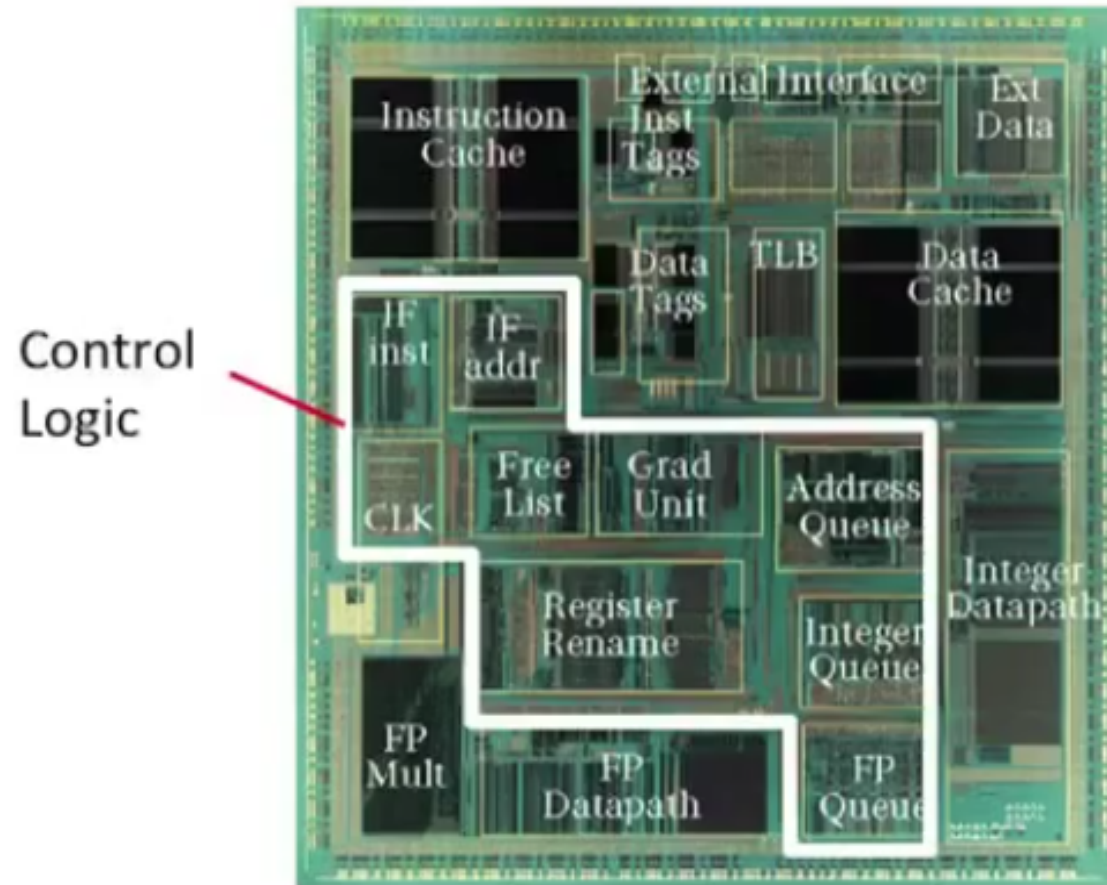


Benchmarks



Controle ocupa muito espaço

MIPS R10000 Microprocessador Superescalar



Agenda

- Dependências
- *Loop unrolling*
- VLIW
- Software pipelining
- Trace scheduling

Loop c/ FP

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```

Loop c/ FP: Onde Estão os Hazards?

```
Loop:  L.D      F0,0(R1) ;F0=vector element
        ADD.D   F4,F0,F2 ;add scalar in F2
        S.D     F4,0(R1) ;store result
        DADDUI  R1,R1,#-8;decrement pointer 8B
        BNE     R1,R2,Loop ;branch R1!=zero
```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Stalls in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Branch	1

Hazards no Loop c/ FP

```
Loop:  L.D    F0, 0(R1)           ;F0=vector element
        ADD.D F4, F0, F2         ;add scalar in F2
        S.D    F4, 0(R1)         ;store result
        DADDUI R1, R1, #-8       ;decrement pointer 8B (DW)
        BNE    R1, R2, Loop      ;branch R1!=zero
```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Stalls in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Branch	1

■ Onde estão os stalls?

Loop c/ FP Mostrando Stalls

```
1 Loop:    L.D      F0,0(R1)    ;F0=vector element
2          stall
3          ADD.D    F4,F0,F2    ;add scalar in F2
4          stall
5          stall
6          S.D      F4,0(R1)    ;store result
7          DADDUI   R1,R1,#-8    ;decrement pointer 8B (DW)
8          stall
9          BNE      R1,R2,Loop  ;branch R1!=zero
10         stall
```

Como podemos reescrever código para minimizar stalls?

Código Revisado para Minimizar Stalls

```
1 Loop:      L.D      F0, 0(R1)
2            DADDUI   R1, R1, #-8
3            ADD.D    F4, F0, F2
4            stall
5            BNE      R1, R2, Loop ;delayed branch
6            S.D      F4, 8(R1)   ;altered when move past SUBI
```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Stalls in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

E se expandirmos o loop 4 vezes?

Expansão do Loop

```
1 Loop: L.D    F0, 0(R1)
2      ADD.D   F4, F0, F2
3      S.D     F4, 0(R1)      ;drop SUBI & BNEZ
4      L.D     F6, -8(R1)
5      ADD.D   F8, F6, F2
6      S.D     F8, -8(R1)     ;drop SUBI & BNEZ
7      L.D     F10, -16(R1)
8      ADD.D   F12, F10, F2
9      S.D     F12, -16(R1)   ;drop SUBI & BNEZ
10     L.D     F14, -24(R1)
11     ADD.D   F16, F14, F2
12     S.D     F16, -24(R1)
13     DADDUI  R1, R1, #-32    ;alter to 4*8
14     BNE     R1, R2, LOOP
```

Podemos
reescrever
loop para
minimizar
stalls?

$14 + 2 + 4 \times (1+2) = 28$ ciclos, ou 7 por iteração

Assume R1 é múltiplo de 4

Minimização dos Stalls

```
1 Loop: L.D      F0, 0(R1)
2       L.D      F6, -8(R1)
3       L.D      F10, -16(R1)
4       L.D      F14, -24(R1)
5       ADD.D    F4, F0, F2
6       ADD.D    F8, F6, F2
7       ADD.D    F12, F10, F2
8       ADD.D    F16, F14, F2
9       S.D      F4, 0(R1)
10      S.D      F8, -8(R1)
11      DADDUI   R1, R1, #-32
12      S.D      F12, 16(R1) ; 16-32 = -16
13      BNEZ     R1, LOOP
14      S.D      F16, 8(R1) ; 8-32 = -24
```

14 ciclos, ou 3.5 por iteração

- Quais pressuposições foram feitas?
 - Pode mover store depois de DADDUI mesmo que F16 seja modificado
 - Pode mover loads antes de stores: mantém dados de memória corretos?
 - Quando é seguro fazer tais modificações?

Perspectiva do Compilador na Movimentação do Código

- **Definição:** compilador está preocupado com **dependências no programa**, sejam elas hazards ou não, o que depende do pipeline
- **Data dependency** (RAW se for hazard em HW)
 - Instrução i produz resultado usado por j, ou
 - Instrução j possui dependência de dados com k, e instrução k possui dependência de dados com instrução i, em outras palavras, **dependência de dados é transitiva.**
- Fácil determinar para registradores (nomes fixos)

Perspectiva do Compilador na Movimentação do Código

- Outro tipo de dependência é chamada *name dependence*: duas instruções usam mesmo nome, mas não trocam dados
- *Antidependence* (WAR se for hazard em HW)
 - Instrução j escreve um registrador ou localidade de memória que instrução i lê, mas instrução i é executada antes de j.
- *Output dependence* (WAW se for hazard em HW)
 - Instrução i e instrução j escrevem no mesmo registrador ou posição de memória; a ordem de escrita deve ser preservada.

Perspectiva do Compilador na Movimentação do Código

- Difícil para acessos de memória
 - $100(R4) = 20(R6)?$
 - Em iterações diferentes de loops, $20(R6) = 20(R6)?$
- No nosso exemplo, precisávamos saber se R1 não modificasse, então:

$0(R1) \neq -8(R1) \neq -16(R1) \neq -24(R1)$

Não existiriam dependências entre alguns loads e stores e o código poderia ser movido

Perspectiva do Compilador na Movimentação do Código

- Dependência final chamada de *control dependence*

- Exemplo

```
if p1 { S1 ; } ;
```

```
if p2 { S2 ; }
```

S1 possui dependência de controle em p1 e S2 possui dependência de controle em p2, mas não em p1.

Perspectiva do Compilador na Movimentação do Código

- Duas (óbvias) restrições para dependências de controle:
 - Uma instrução que possui dependência de controle com um branch **não** pode ser movida para **antes** do branch.
 - Uma instrução que não possui dependência de controle com um branch **não** pode ser movida para **depois** do branch.
- Dependências de controle são relaxadas para conseguir paralelismo; mesmo efeito é conseguido se preservarmos ordem das exceções e fluxo de dados

Escalonamento Estático com Processadores VLIW

- VLIW: Very Long Instruction Word
 - IA-64 ou Itanium da Intel
 - Trimedia da Philips
 - TMS320C62X da Texas Instrument

Escalonamento Estático com Processadores VLIW

- VLIW: decodificação vs. tamanho da instrução
 - Há espaço no código da instr. para **diversas FUs**
 - **Operações** definidas pelo compilador para executar **na mesma palavra podem executar em paralelo**
 - Ex., 2 operações inteiras, 2 operações FP, 2 refs. memória, 1 branch
 - 16 a 24 bits por campo => 112 bits a 168 bits de tamanho
 - **Precisa escalonar código através de branches** para ser efetivo

Loop Unrolling em VLIW

***Memória 1
Clock***

Memória 2

FP 1

FP 2

Int/

branch

Loop Unrolling em VLIW

<i>Memória 1</i> <i>Clock</i>	<i>Memória 2</i>	<i>FP 1</i>	<i>FP 2</i>	<i>Int/</i> <i>branch</i>	
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)	L.D F30,-56(R1)	ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
	ADD.D F24,F22,F2		ADD.D F20,F18,F2		5
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2	ADD.D F32,F30,F2	DADDUI R1,R1,#-56	6
S.D F12,40(R1)	S.D F16,32(R1)				7
S.D F20,24(R1)	S.D F24,16(R1)			BNE R1,R2,LOOP	8
S.D F28,8(R1)	S.D F28,0(R1)				9

Loop Unrolling em VLIW

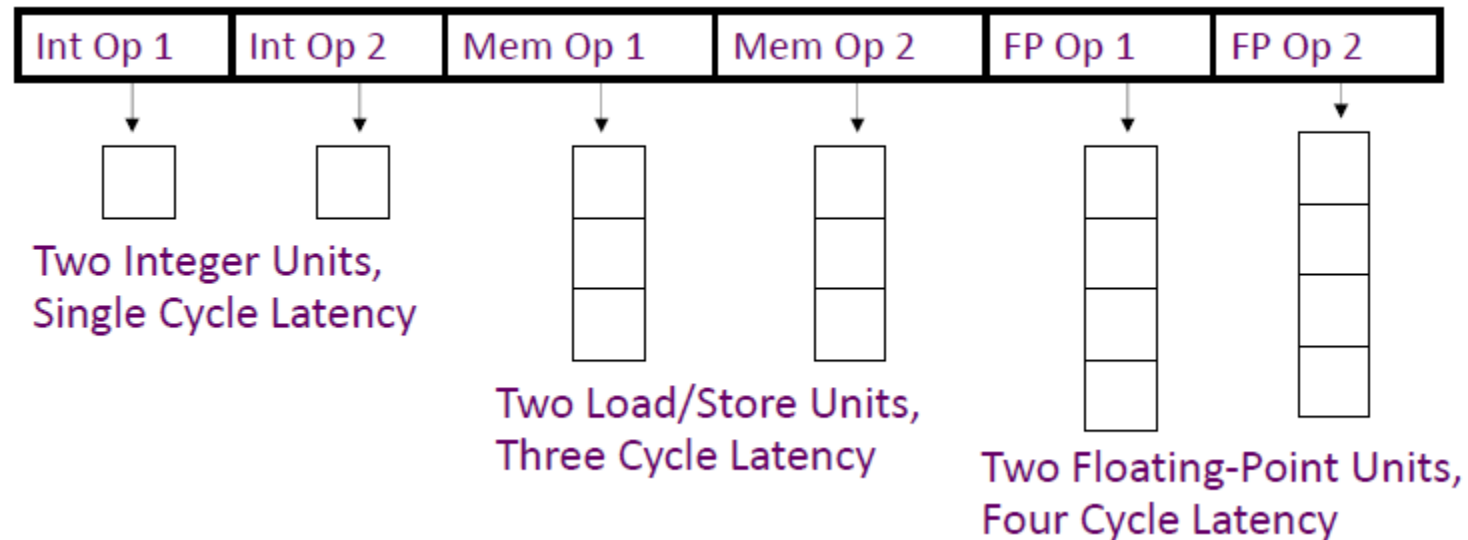
<i>Memória 1</i> <i>Clock</i>	<i>Memória 2</i>	<i>FP 1</i>	<i>FP 2</i>	<i>Int/</i> <i>branch</i>	
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)	L.D F30,-56(R1)	ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
	ADD.D F24,F22,F2		ADD.D F20,F18,F2		5
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2	ADD.D F32,F30,F2	DADDUI R1,R1,#-56	6
S.D F12,40(R1)	S.D F16,32(R1)				7
S.D F20,24(R1)	S.D F24,16(R1)			BNE R1,R2,LOOP	8
S.D F28,8(R1)	S.D F28,0(R1)				9

Unrolled 8 vezes para evitar atrasos

- 8 executadas em 9 ciclos, ou 1.125 ciclos por iteração
- Precisa de mais registradores em VLIW

Escalonamento Estático com Processadores VLIW

■ Outro Exemplo



Escalonamento Estático com Processadores VLIW

■ Outro Exemplo

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Compile

loop:

```
loop:  lw F1, 0(R1)
      addiu R1, R1, 4
      add.s F2, F0, F1
      sw F2, 0(R2)
      addiu R2, R2, 4
      bne R1, R3, loop
```

Schedule

[illegible]

Escalonamento Estático com Processadores VLIW

■ Outro Exemplo

```
for (i=0; i<N; i++)  
  B[i] = A[i] + C;
```

Compile

```
loop: lw F1, 0(R1)  
      addiu R1, R1, 4  
      add.s F2, F0, F1  
      sw F2, 0(R2)  
      addiu R2, R2, 4  
      bne R1, R3, loop
```

loop:

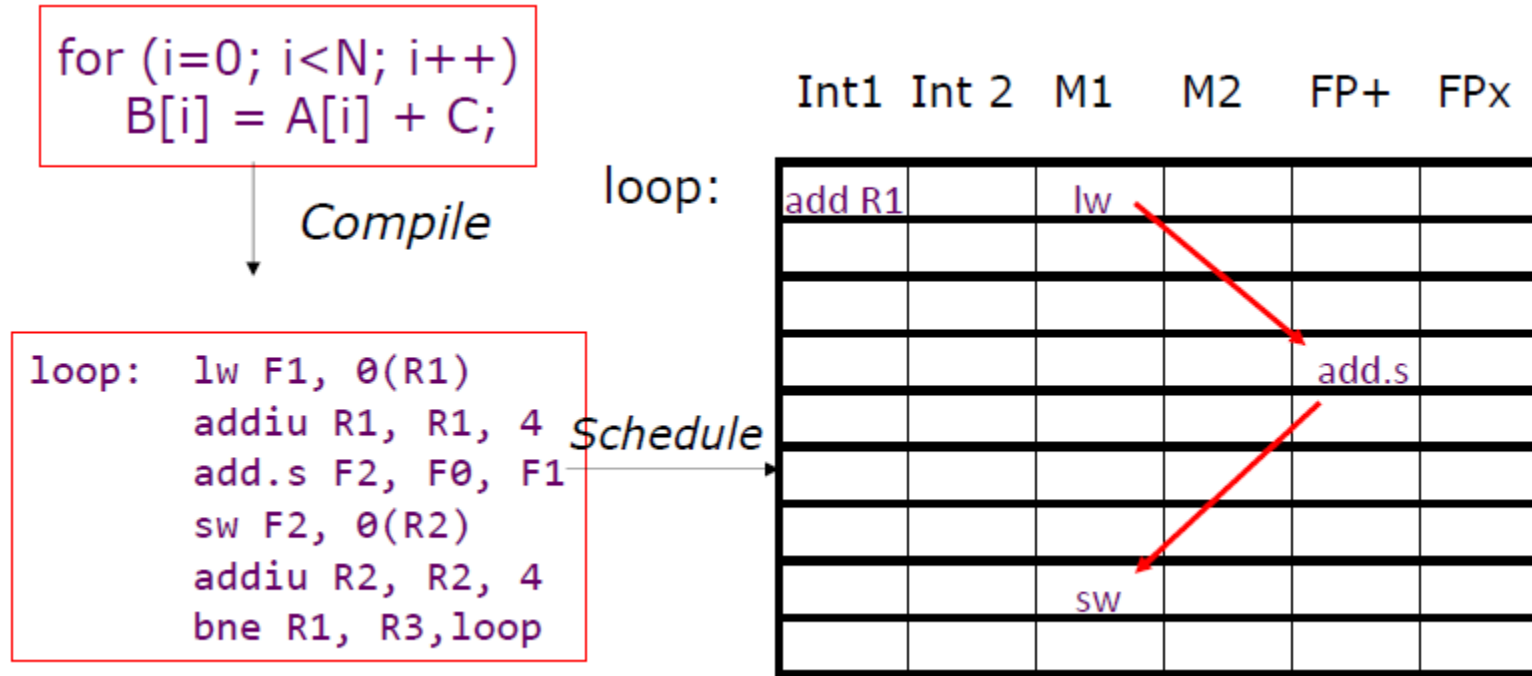
Schedule

Int1 Int 2 M1 M2 FP+ FPx

add R1		lw			
				add.s	

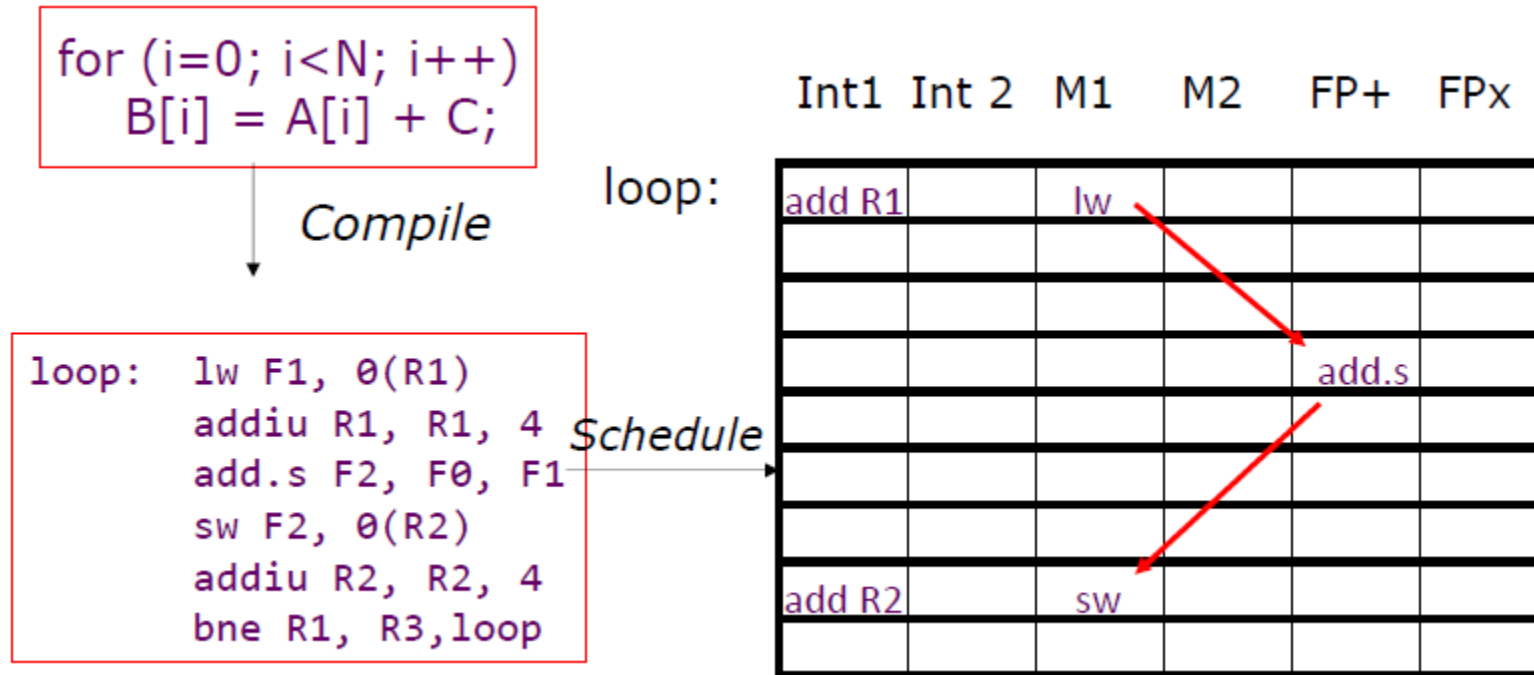
Escalonamento Estático com Processadores VLIW

■ Outro Exemplo



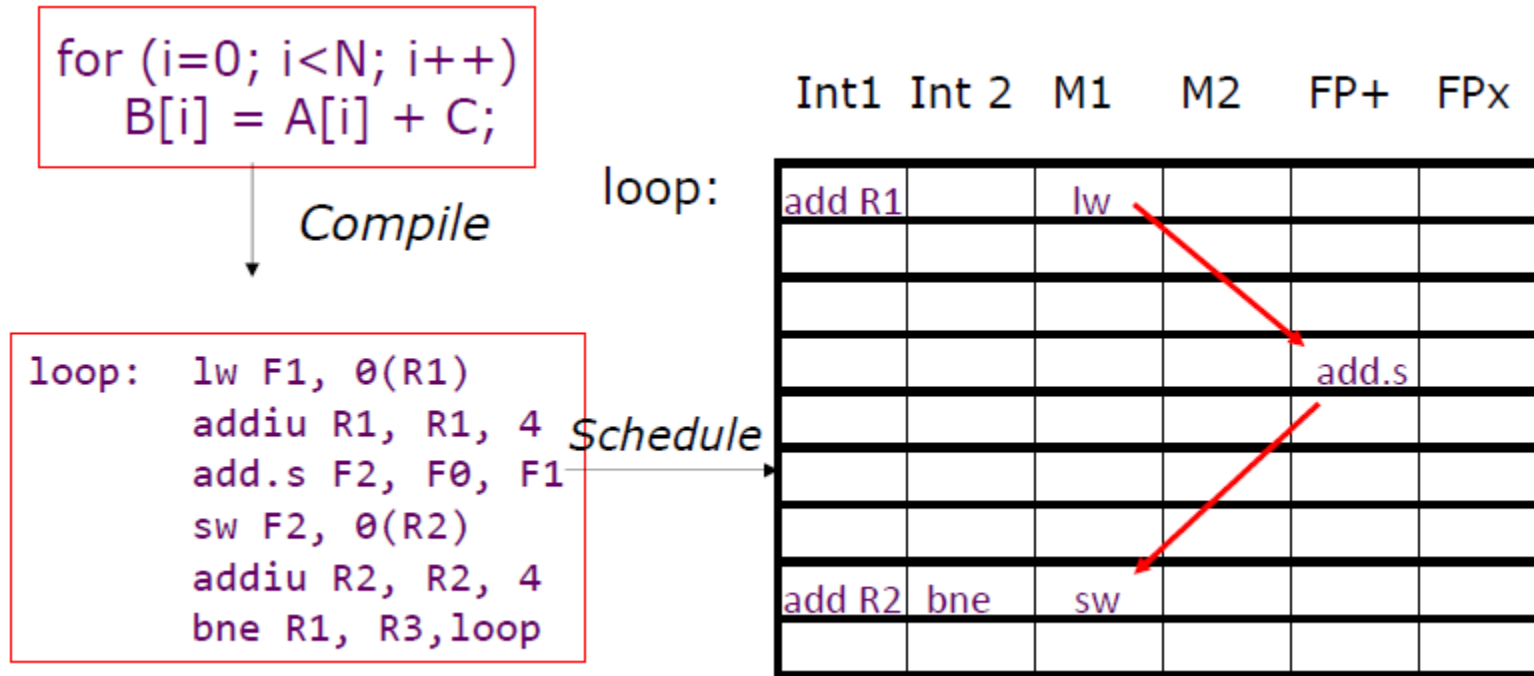
Escalonamento Estático com Processadores VLIW

■ Outro Exemplo



Escalonamento Estático com Processadores VLIW

■ Outro Exemplo



Escalonamento Estático com Processadores VLIW

- Outro Exemplo – Agora com loop unrolling

Unroll 4 ways

```

loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop

```

Int1 Int 2 M1 M2 FP+ FPx

```
loop:
```

Schedule

[illegible]

Escalonamento Estático com Processadores VLIW

- Outro Exemplo – Agora com loop unrolling

Unroll 4 ways

```

loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop

```

[illegible]

Escalonamento Estático com Processadores VLIW

■ Outro Exemplo – Agora com loop unrolling

Unroll 4 ways

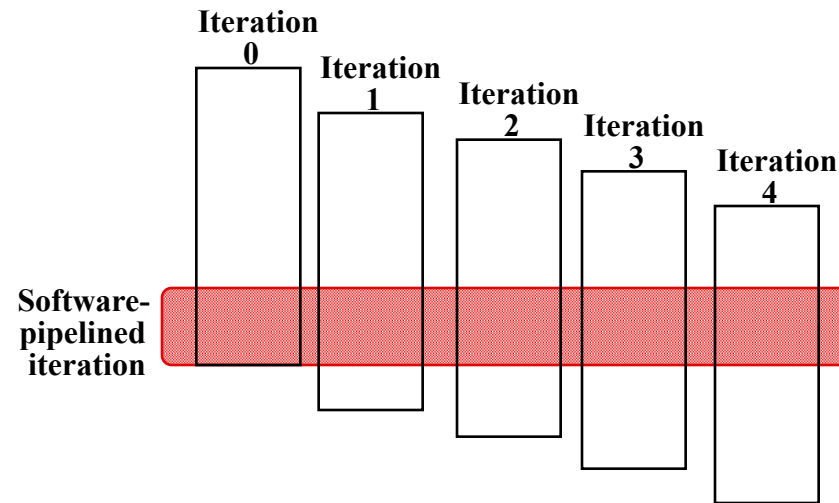
```
loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
```

Schedule

	Int1	Int 2	M1	M2	FP+	FPx
loop:			lw F1			
			lw F2			
			lw F3			
add R1			lw F4		add.s F5	
					add.s F6	
					add.s F7	
					add.s F8	
			sw F5			
			sw F6			
			sw F7			
add R2	bne	sw F8				

Software Pipelining

- **Observação:** se iterações de loops são independentes, então podemos ganhar ILP executando instruções de diferentes iterações de cada vez
- **Software pipelining:** reorganiza loops de forma que cada iteração executada é realizada por instruções escolhidas das iterações diferentes do loop original (isto é, Tomasulo em SW)

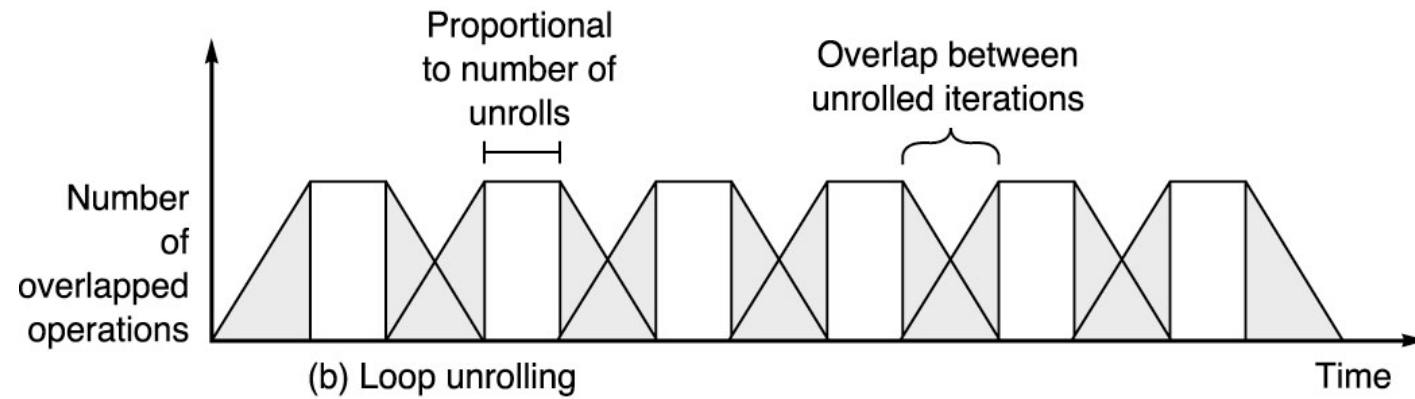
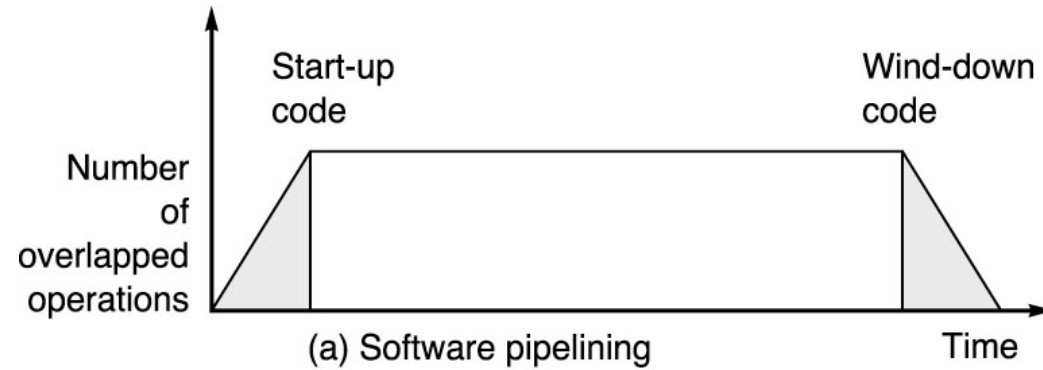


Exemplo de SW Pipelining

Antes: Unrolled 3 vezes

```
1  L.D  F0,0(R1)
2  ADD.D  F4,F0,F2
3  S.D  F4,0(R1)
4  L.D  F6,-8(R1)
5  ADD.D  F8,F6,F2
6  S.D  F8,-8(R1)
7  L.D  F10,-16(R1)
8  ADD.D  F12,F10,F2
9  S.D  F12,-16(R1)
10 DADDUI R1,R1,#-24
11 BNE R1,R2,LOOP
```

Software Pipelining



Software Pipelining / VLIW

Unroll 4 ways first

```
loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
```

Int1	Int 2	M1	M2	FP+	FPx
		lw F1			
		lw F2			
		lw F3			
add R1		lw F4			
				add.s F5	
				add.s F6	
				add.s F7	
				add.s F8	
			sw F5		
			sw F6		
	add R2		sw F7		
	bne		sw F8		

Software Pipelining / VLIW

Unroll 4 ways first

```
loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
```

Int1	Int 2	M1	M2	FP+	FPx
		lw F1			
		lw F2			
		lw F3			
add R1		lw F4			
		lw F1		add.s F5	
		lw F2		add.s F6	
		lw F3		add.s F7	
add R1		lw F4		add.s F8	
			sw F5	add.s F5	
			sw F6	add.s F6	
	add R2		sw F7	add.s F7	
	bne		sw F8	add.s F8	
			sw F5		
			sw F6		
	add R2		sw F7		
	bne		sw F8		

Software Pipelining / VLIW

Unroll 4 ways first

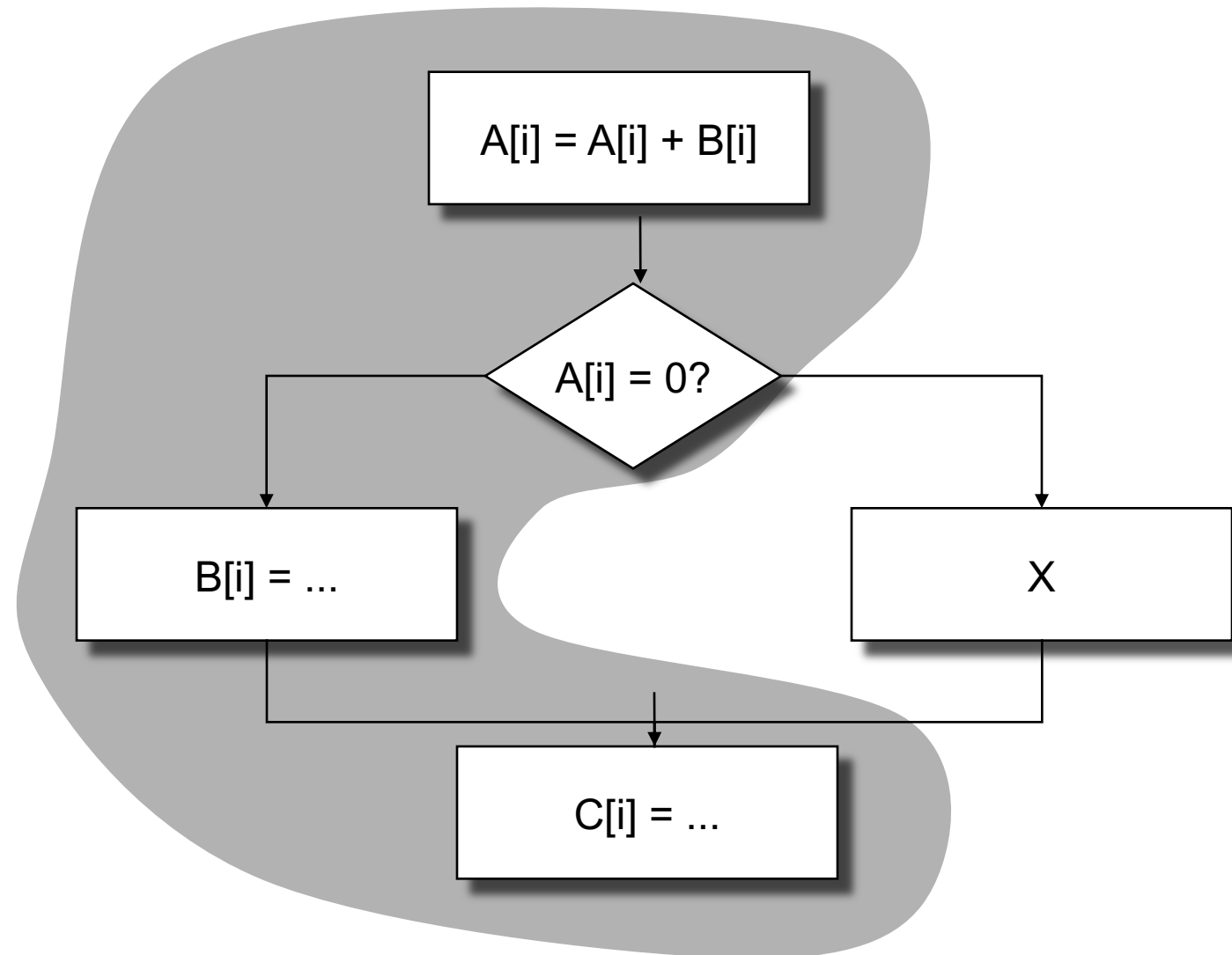
```
loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
```

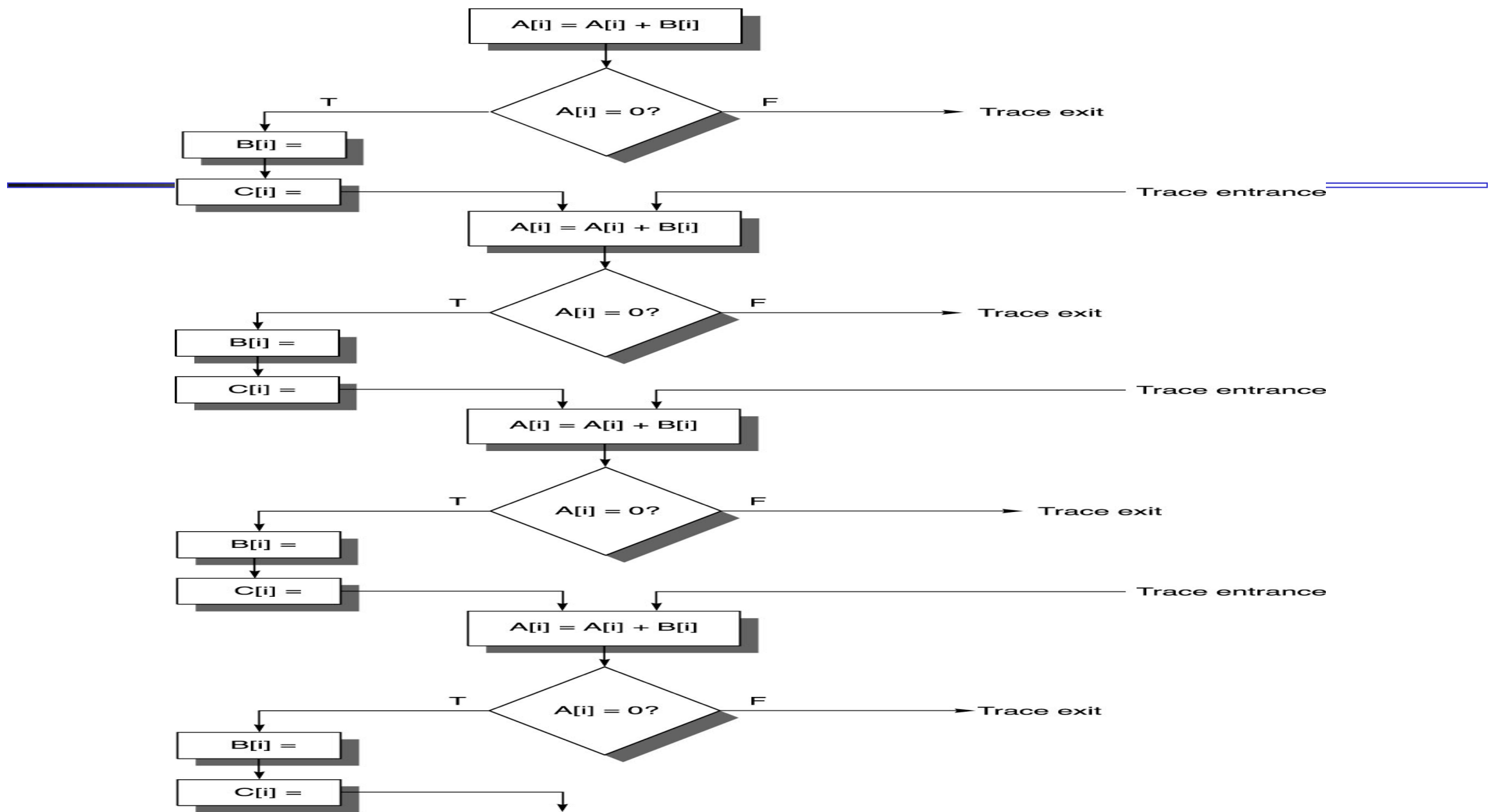
Int1	Int 2	M1	M2	FP+	FPx
		lw F1			
		lw F2			
		lw F3			
add R1		lw F4			
		lw F1		add.s F5	
		lw F2		add.s F6	
		lw F3		add.s F7	
add R1		lw F4		add.s F8	
		lw F1	sw F5	add.s F5	
		lw F2	sw F6	add.s F6	
	add R2	lw F3	sw F7	add.s F7	
add R1 bne		lw F4	sw F8	add.s F8	
			sw F5	add.s F5	
			sw F6	add.s F6	
	add R2		sw F7	add.s F7	
	bne		sw F8	add.s F8	
			sw F5		

Trace Scheduling

- Permite encontrar paralelismo cruzando branches de IFs vs. branches de LOOPS
- Dois passos:
 - Trace Selection
 - Encontre sequência mais provável de blocos básicos (trace) a partir de sequência estatisticamente prevista de código sequencial
 - Trace Compaction
 - Comprima trace no menor número de instruções VLIW o possível
 - Necessita de manter informações para recuperação em caso de previsão errônea

Seleção de *Trace*



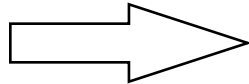


Extraindo Mais Paralelismo

- Execução condicional de instruções
(*conditional or predicated instructions*)

`if (a == 0) s=t`

`bnez r1,L`
`addu r2,r3,r0`
`L: ...`



`cmovz r2,r3,r1`

- Mas ainda leva tempo se instrução não for executada
- Popular nas máquinas modernas
 - MIPS, Alpha, PowerPC e SPARC possuem move condicional
 - HPPA permite instr. R-R cancelar condicionalmente a instrução seguinte
 - IA-64 permite instruções para execução condicional

Detectando e Explorando Paralelismo em Loops

- Exemplo: Onde estão as dependências de dados?
(A,B,C distintos & sem overlap)

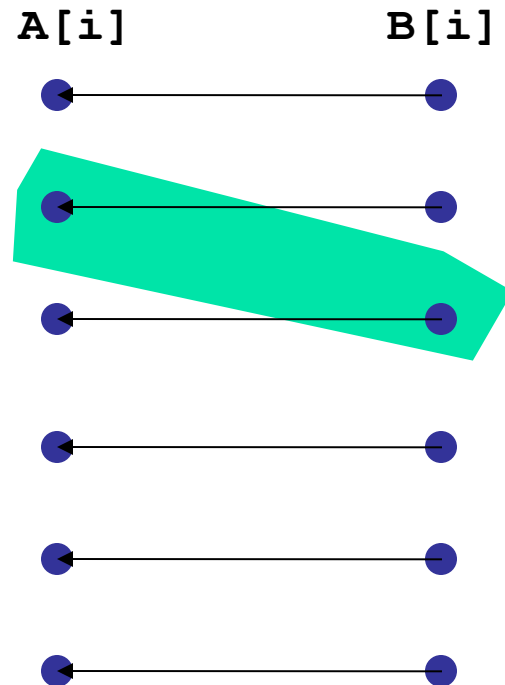
```
for (i=1; i<=100; i=i+1) {  
    A[i+1] = A[i] + C[i];      /* S1 */  
    B[i+1] = B[i] + A[i+1]; } /* S2 */
```

1. S2 usa o valor A[i+1] calculado por S1 na mesma iteração.
2. S1 usa valor calculado por S1 em uma iteração anterior, já que cálculo de A[i+1] usa A[i]. O mesmo acontece para S2 com B[i] e B[i+1]. Isto é chamado “**loop-carried dependence**” entre iterações

- Iterações são dependentes, e **não podem ser executadas em paralelo**
- No nosso caso, cada iteração era independente

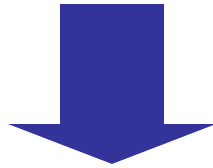
Detectando e Explorando Paralelismo em Loops

```
for (i=1; i<=100; i=i+1) {  
    A[i] = A[i] + B[i];      /* S1 */  
    B[i+1] = C[i] + D[i];    /* S2 */  
}
```



Detectando e Expandindo Loops

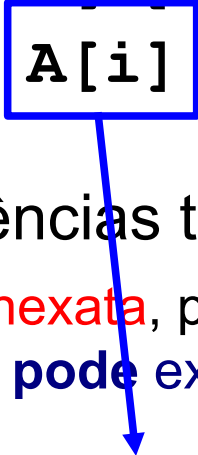
```
for (i=1; i<=100; i=i+1) {  
    A[i] = A[i] + B[i];          /* S1 */  
    B[i+1] = C[i] + D[i]; }     /* S2 */
```



```
A[1] = A[1] + B[1];  
for (i=1; i<=99; i=i+1) {  
    B[i+1] = C[i] + D[i];      /* S2 */  
    A[i+1] = A[i+1] + B[i+1]; /* S1 */  
}  
B[101] = C[100] + D[100]
```

Detectando e Explorando Paralelismo em Loops

```
for (i=1; i<=100; i=i+1) {  
  A[i] = B[i] + C[i];  
  D[i] = A[i] * E[i];}
```



Detecção de dependências transportadas por loop.

- **Detecção é inexata**, pois **somente informa se** dependência **pode** existir, e não se ela vai existir

Olhar para o registrador e não para a memória

Identificando *loop-carried dependencies*

- Elas aparecem na forma de equações de recorrência

```
for (i=6; i<=100; i++)  
    Y[i]=Y[i-5]+Y[i];
```

- Loop possui dependência de distância 5
 - Quanto maior a distância, maior o potencial para paralelismo
 - Nesse caso, podemos desdobrar 5 iterações independentes, que podem ser remanejadas a vontade do otimizador de código

Resumo

- Tecnologia de compiladores pode ajudar extração de paralelismo para ajudar hardware
 - Loop unrolling e detecção de paralelismo em loops
 - Escalonamento de código em VLIW
 - Software pipelining
 - Trace scheduling