MO615B - Implementação de Linguagens II
e
MC900A - Tópicos Especiais em Linguagem de Programação

Prof. Sandro Rigo
www.ic.unicamp.br/~sandro

Escalonamento de Instruções

Introdução

- Principal objetivo:
  - Melhorar o aproveitamento do pipeline
  - Melhorar o paralelismo entre as instruções
- *Branch Scheduling*
  - Preenchimento de *delay slots*
- *List Scheduling*
  - Escalonamento de instruções em BBs

Introdução

- *Trace Scheduling*
  - Escalonamento global (Múltiplos BBs)
- *Software Pipelining*
  - Escalonamento em laços

Dependências

- É uma relação que restringe a ordem de execução de duas instruções
- Dependência de controle
  - Surge do fluxo de controle do programa
- Dependência de dados
  - Surge do fluxo de dados entre as instruções

Dependência de Dados

- S1 < S2
  - significa que, na ordem dada, a execução da sentença S1 precede a de S2
- Quatro tipos de dependência
  - RAW (read after write)
  - RAR (read after read)
  - WAR (write after read)
  - WAW (write after write)

Dependência de Dados

- RAW (*Read After Write*)
  - Também conhecida como: *Flow dependence*, *data dependence*, *true dependende*

```
S1:  x := ...
...
S2:  ... := x
```

Dependência de Dados

- RAR (*Read After Read*)
  - Também conhecida como: *Input dependence*

```
S1:   ... := x + y
...
S2:   ... := x + z
```

Dependência de Dados

- WAR (*Write After Read*)
  - Também conhecida como: *Anti dependence*

```
S1:   ... := x
...
S2:   x := ...
```

Dependência de Dados

- WAW (*Write After Write*)
  - Também conhecida como: *Output dependence*

```
S1:  x := ...
...
S2:  x := ...
```

Grafo de dependências

- Nós representam as intruções
- Arestas são dependências entre as instruções
- Dependências de controle normalmente omitidas
  - A menos que seja a única que conecta dois nós
- Arestas são rotuladas com o tipo de dependência
  - Não aparece em todos os livros
  - Usado no livro do Muchnick

Exemplo: Grafo Dependências

- RAW: *flow* (f)
- WAW: *output* (o)
- WAR: *anti* (a)

```
1. a = b + c
2. if a > 10 got L1
3.    d = b * e
4.    e = d +1
5. L1: d = e/2
```

1

2

3

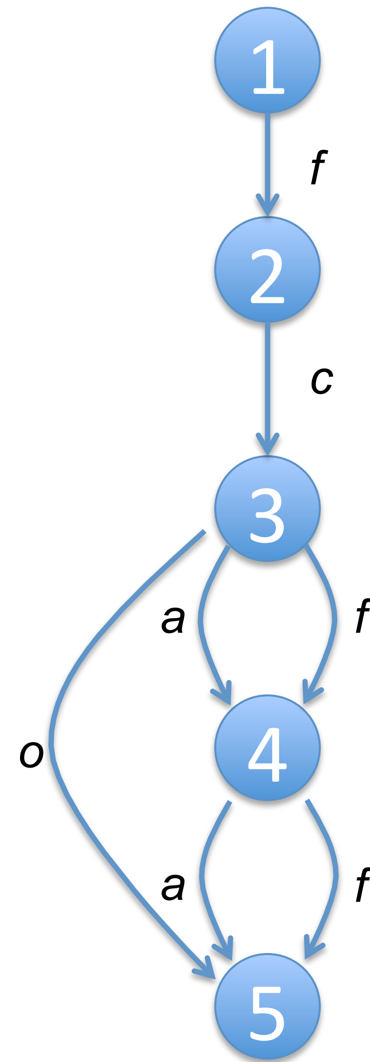
4

5

Exemplo: Grafo Dependências

- RAW: *flow* (f)
- WAW: *output* (o)
- WAR: *anti* (a)

```
1. a = b + c
2. if a > 10 got L1
3.     d = b * e
4.     e = d +1
5. L1: d = e/2
```



Grafo de dependências no Bloco Básico

- Representa dependências dentro de um bloco básico
- BBs não possuem laços
- O grafo é sempre acíclico
  - *Dependence DAG*

Grafo de dependências no Bloco Básico

- Aresta podem representar diversos tipos de dependência:
  - RAW, WAR, WAR,
  - Quando não conseguimos determinar se podemos mudar a ordem. Ex: acessos à memória
    - *load* seguido de *store* que usam registradores diferentes como endereço
    - Não sabemos se os endereços se sobrepõem
  - *Hazard* Estrutural. Ex:
    - restrições de tempo por causa da latência de certas instruções. Anotamos a aresta com a latência.

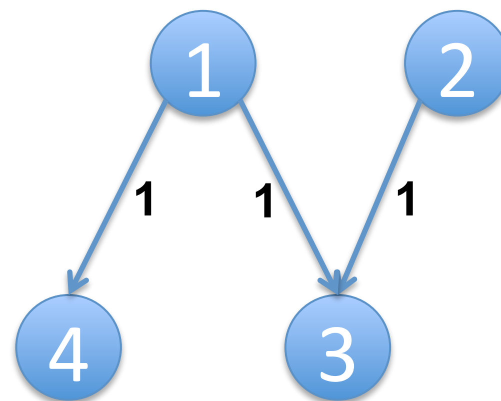
Grafo de dependências no Bloco Básico

- Rotular arestas com latência
  - Latência necessária entre duas instruções
  - Latência = atraso entre início de I1 e I2 menos tempo de execução requerido por I1 antes que qualquer outra instrução possa ser iniciada (normalmente 1 ciclo)
  - Se I2 pode iniciar no ciclo seguinte a I1 => Latência = 0

Grafo de dependências no Bloco Básico

Supondo que *loads* tenham uma latência de 1 ciclo, mas requer dois ciclos para para terminar.

```
1  r2  := [r1]
2  r2  := [r1 + 4]
3  r4  := r2 + r3
4  r5  := r2 - 1
```



Grafo de dependências no Bloco Básico

Supondo que *loads* tenham uma latência de 1 ciclo, mas requer dois ciclos para para terminar.

```
1  r3  := [r15]
2  r4  := [r15+4]
3  r2  := r3-r4
4  r5  := [r12]
5  r12  := r12+4
6  r6  := r3*r5
7  [r15+4] := r3
8  r5  := r6+2
```

1 2 4

3 7 5

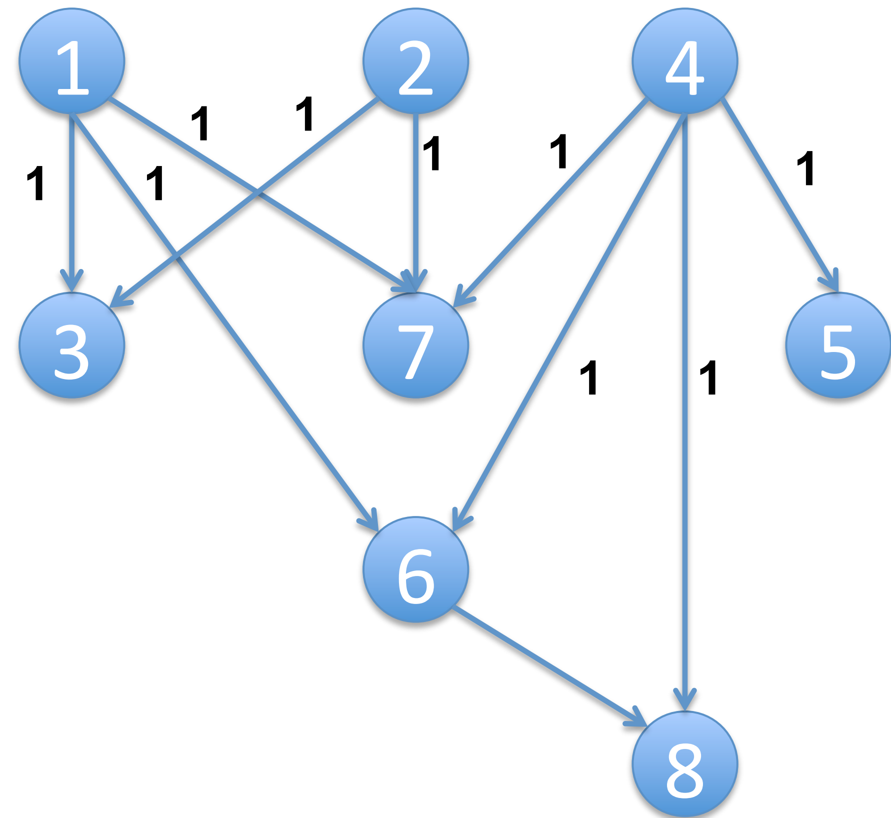
6

8

Grafo de dependências no Bloco Básico

Supondo que *loads* tenham uma latência de 1 ciclo, requerendo dois ciclos para terminar.

```
1  r3 := [r15]
2  r4 := [r15+4]
3  r2 := r3-r4
4  r5 := [r12]
5  r12 := r12+4
6  r6 := r3*r5
7  [r15+4] := r3
8  r5 := r6+2
```



Branch Scheduling

- Dois objetivos:
  - Preencher *delay slots*
  - Preencher *delay* entre cálculo da condição e o desvio
- Arquiteturas RISC
  - MIPS, SPARC
  - Um *delay slot* após desvios
  - Esses *slots* devem ser preenchidos
    - Instruções úteis
    - NOPs

Branch Scheduling

- Power, PowerPC, etc
  - Requerem um certo # de ciclos entre uma instrução que calcula a condição e a instrução de desvio
  - Causa *stalls* na instrução de desvio

Branch Hazard

	i	i+1	i+2	i+3	i+4
1	IF				
2	ID	IF			
3	EX	ID	IF		
4	MEM	EX	ID	IF	
5	WB	MEM	EX	ID	IF
6		WB	MEM	EX	ID
7			WB	MEM	EX
8				WB	MEM
9					WB

Instrução de salto determina o próximo endereço no estágio EX

Exemplo

- 1 *delay slot* para o *goto* e um ciclo para o valor do *load* estar disponível
- Este código executa em quantos ciclos?

```
1. r2 = [r1]
2. r3 = [r1+4]
3. r4 = r2 + r3
4. r5 = r2 -1
5. goto L1
6. nop
```

Exemplo

- 1 *delay slot* para o *goto* e um ciclo para o valor do *load* estar disponível
- Este código executa em quantos ciclos?

```
1. r2 = [r1]
2. r3 = [r1+4]
3. r4 = r2 + r3
4. r5 = r2 -1
5. goto L1
6. nop
```

```
1. r2 = [r1]
2. r3 = [r1+4]
3. r5 = r2 -1
4. goto L1
5. r4 = r2 + r3
```

Nullifying / Canceling

- Algumas arquiteturas permitem cancelamento de instruções
  - SPARC
  - ARM
- Instruções condicionais
  - Uso de predicados

Alternativas de Escalonamento

Delay slot é sempre executado: escalonar instrução do próprio BB

```
1. r1 = r2 + r3
2. if r2 = 0 goto 6
3.
```

```
4. r4 = r2 + r3
5. r5 = r2 – 1
```

```
6. r4 = r5 – r6
...
```

```

2. if r2 = 0 goto 6
1. r1 = r2 + r3
```

```
4. r4 = r2 + r3
5. r5 = r2 – 1
```

```
6. r4 = r5 – r6
```

Alternativas de Escalonamento

Delay slot só é executado quando o salto é tomado. Escalonamos uma instrução do bloco alvo.

```
1. r1 = r2 + r3
2. if r2 = 0 goto 6
3.
```

```
4. r4 = r2 + r3
5. r5 = r2 – 1
```

```
6. r4 = r5 – r6
7. ...
```

```
1. r1 = r2 + r3
2. if r2 = 0 goto 7
6. r4 = r5 – r6
```

```
4. r4 = r2 + r3
5. r5 = r2 – 1
```

```
7. ...
```

Alternativas de Escalonamento

Delay slot só é executado quando o salto não é tomado. Escalonamos uma instrução do bloco *fall-through*.

```
1. r1 = r2 + r3
2. if r2 = 0 goto 6
3.
```
```
4. r4 = r2 + r3
5. r5 = r2 – 1
```
```
6. r4 = r5 – r6
7. ...
```

```
1. r1 = r2 + r3
2. if r2 = 0 goto 7
4. r4 = r2 + r3
```
```
5. r5 = r2 – 1
```
```
6. r4 = r5 – r6
7. ...
```

List Scheduling

List Scheduling

- Escalonamento de instruções em um bloco básico para melhorar o desempenho
- Idéia: gerar um ordem topológica do DAG que
  - Não altere os resultados
  - Minimize o tempo de execução do bloco básico
- É um problema NP-Difícil
  - Buscamos uma heurística efetiva
  - $O(n^2)$, mas geralmente linear na prática

List Scheduling

- Desempenho dominado pela construção do grafo
  - Também $O(n^2)$ no pior caso mas normalmente linear na prática

List Scheduling

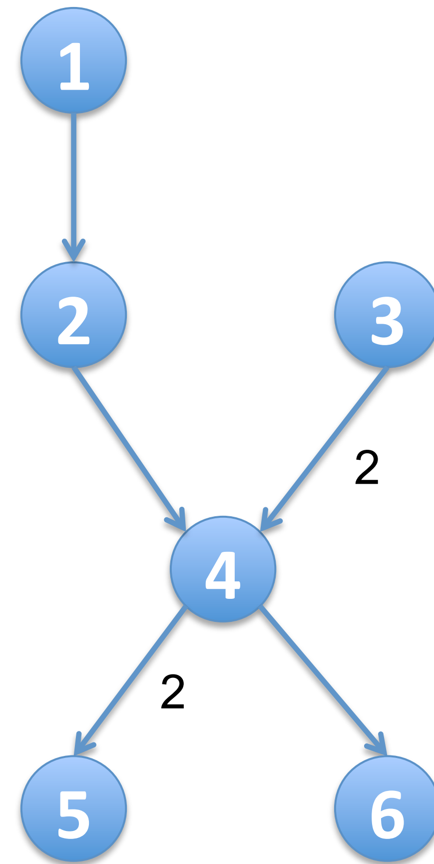
- Intuição por trás do algoritmo:
  - Enquanto houver nós para escalonar
    1. Escolha um nó que:
       - os predecessores já tenham sido escalonados; e
       - já tenha passado "tempo suficiente" (latência) desde que eles foram escalonado.
    2. Escalone o nó escolhido!

List Scheduling

- ... **tenha passado “tempo suficiente”** ...
- Suponha que um nó n tem um predecessor p_i
- n e p_i são
  - escalonados em S(n) e S(p_i),
  - executam em T(n) e T(p_i),
  - a latência de p_i->n é L(p_i)
- Então, n **não deve** ser escalonado antes de
  - max(S(p_i) + T(p_i) + L(p_i)), para todo predecessor p_i de n.
- Se for, vai causar um stall no pipeline.
  - Utilizaremos o termo ETime(n) para denotar o tempo mais cedo que um nó n pode ser executado sem causar um *stall* no pipeline.

List Scheduling

- Exemplo:



- Escalonamento possível:
  - 3, 1, 2, 4, 6, *stall*, 5
- Outro escalonamento:
  - 1, 2, 3, *stall*, *stall*, 4, 6, *stall*, 5

List Scheduling

- O método tem 2 fases:
  - Atravessar o DAG das folhas às raizes
  - Atravessar o DAG das raizes às folhas

List Scheduling

- Fase 1: Das folhas para as raízes
  - Rotule cada nó com o maior atraso possível a partir daquele nó até o final do bloco
  - ExecTime(n): # ciclos necessários para executar n
  - Maior atraso possível: *Delay(n)*

$$Delay(n) = \begin{cases} ExecTime(n), \text{ para n folha} \\ \max\limits_{m \in DAGSucc(n,DAG)} Late_Delay(n,m), \text{ caso contrário} \end{cases}$$

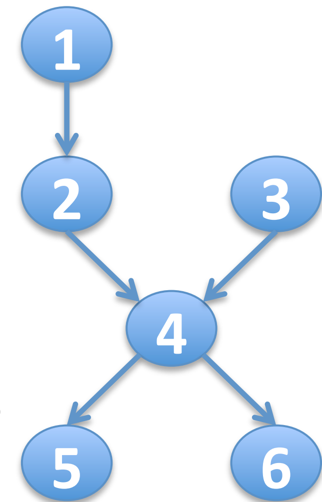
  - Late_Delay(n,m) = Latência(n,m) + Delay(m)

List Scheduling

- Fase 1: Exemplo
  - Dado o grafo de dependências, compute o *Delay(n)* para cada nó *n*.
  - ExecTime(6) = 2,
  - ExecTime(1,2,3,4,5) = 1
  - Latência(n,m) = 1 para todos os pares (n,m).

Nó	*Delay*
1	
2	
3	
4	
5	
6	

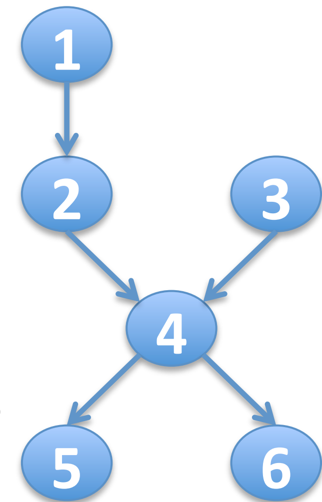
$$Delay(n) = \begin{cases} ExecTime(n), & \text{para n folha} \\ \max_{m \in DAGSucc(n,DAG)} Late_Delay(n,m), & \text{caso contrário} \end{cases}$$



List Scheduling

- Fase 1: Exemplo
  - Dado o grafo de dependências, compute o *Delay(n)* para cada nó *n*.
  - ExecTime(6) = 2,
  - ExecTime(1,2,3,4,5) = 1
  - Latência(n,m) = 1 para todos os pares (n,m).

Nó	*Delay*
1	5
2	4
3	4
4	3
5	1
6	2



$$Delay(n) = \begin{cases} ExecTime(n), & \text{para n folha} \\ \max\limits_{m \in DAGSucc(n,DAG)} Late_Delay(n,m), & \text{caso contrário} \end{cases}$$

List Scheduling

- Fase 2: Das raízes para as folhas
  - Escolha dos nós a serem escalonados
  - Tempo corrente (atual): inicia com zero
  - ETime[n] (*earliest time*): Menor tempo em que cada nó deve ser escalonado para evitar stalls
  - *Sched*: sequência de nós que já foram escalonados

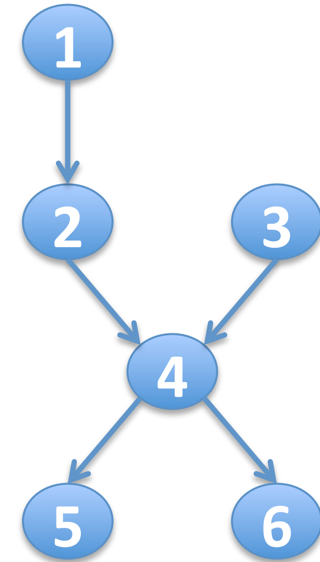
List Scheduling

- Fase 2: Das raízes para as folhas
  - *Cands*: nós que ainda não foram escalonados, mas cujos predecessores já foram
    - MCands: nós com o máximo atraso até o fim do bloco
    - ECands: nós em MCands cujo menor tempo de início é menor ou igual ao tempo corrente. Não causam *stall* se começarem a execução agora!

List Scheduling

- Fase 2: Das raízes para as folhas
  - Pegue de ECands o nó com o menor Etime
    - Heurística !!!
  - Coloque-o em *Sched*
  - Repita até que *Sched* contenha todos os nós do DAG

List Scheduling



- Fase 2: Exemplo
  - ExecTime(6) = 2,
  - ExecTime(1,2,3,4,5) = 1
  - Latência(n,m) = 1 para todos os pares (n,m).
- Sched:
- Cands:
  - MCands:
  - ECands:
- CurrTime:

Nó	*Delay*	*ETime(n)*
1	5	
2	4	
3	4	
4	3	
5	1	
6	2	

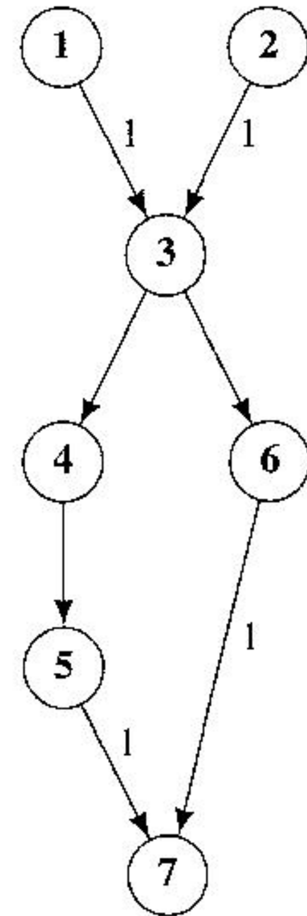
ETime[i] = max(ETime[i], CurTime+latency(n,i))

Alocação e Escalonamento

- Interação entre alocador de registradores e escalonador
  - Pode trazer sérios problemas

```
1      r1 ← [r12+0](4)
2      r2 ← [r12+4](4)
3      r1 ← r1 + r2
4      [r12,0](4) ← r1
5      r1 ← [r12+8](4)
6      r2 ← [r12+12](4)
7      r2 ← r1 + r2
```

(a)



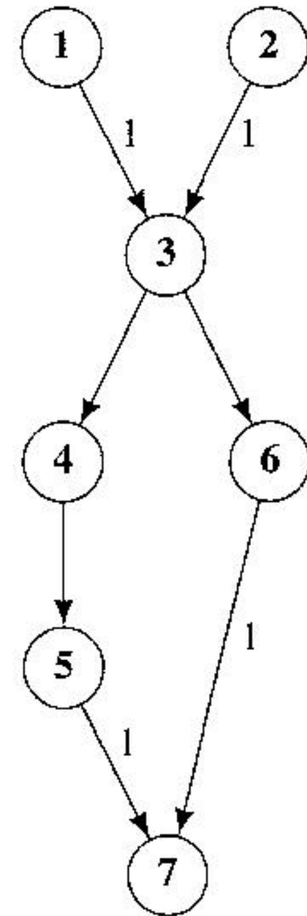
Alocação e Escalonamento

- Interação entre alocador de registradores e escalonador
  - Pode trazer sérios problemas

⟶ *stall*

```
1    r1 ← [r12+0](4)
2    r2 ← [r12+4](4)
3    r1 ← r1 + r2
4    [r12,0](4) ← r1
5    r1 ← [r12+8](4)
6    r2 ← [r12+12](4)
7    r2 ← r1 + r2
```

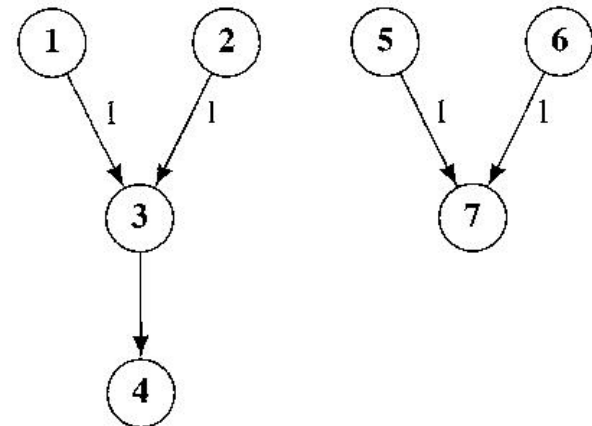
(a)



Alocação e Escalonamento

- Renomeando os registradores

	(a)	(b)
1	r1 ← [r12+0](4)	r1 ← [r12+0](4)
2	r2 ← [r12+4](4)	r2 ← [r12+4](4)
3	r1 ← r1 + r2	r3 ← r1 + r2
4	[r12,0](4) ← r1	[r12,0](4) ← r3
5	r1 ← [r12+8](4)	r4 ← [r12+8](4)
6	r2 ← [r12+12](4)	r5 ← [r12+12](4)
7	r2 ← r1 + r2	r6 ← r4 + r5



Alocação e Escalonamento

- Novo escalonamento
  - Não gera *stalls*

```
r1 ← [r12+0](4)
r2 ← [r12+4](4)
r4 ← [r12+8](4)
r5 ← [r12+12](4)
r3 ← r1 + r2
[r12+0](4) ← r3
r6 ← r4 + r5
```

Quando fazer Escalonamento?

- No mundo real:
  - Feito imediatamente antes da alocação
  - Repetido depois caso código de spill seja gerado
  - Tática usada por
    - IBM: Compiladores POWER e PowerPC
    - Sun: SPARC
    - HP: PA-RISC

Software Pipelining

Software Pipelining

- Melhorar desempenho de laços
- Explorar ILP
  - VLIW
  - Superescalar
  - Single-scalar que permitem instruções de inteiros e FP serem executadas ao mesmo tempo
- Partes de várias iterações são executadas ao mesmo tempo
  - Explora paralelismo no corpo do laço

Software Pipelining

	Issue latency	Result latency
`L: ldf [r1],f0`	1	1
`fadds f0,f1,f2`	1	7
`stf f2,[r1]`	6	3
`sub r1,4,r1`	1	1
`cmp r1,0`	1	1
`bg L`	1	2
`nop`	1	1

FIG. 17.15 A simple SPARC loop with assumed issue and result latencies.

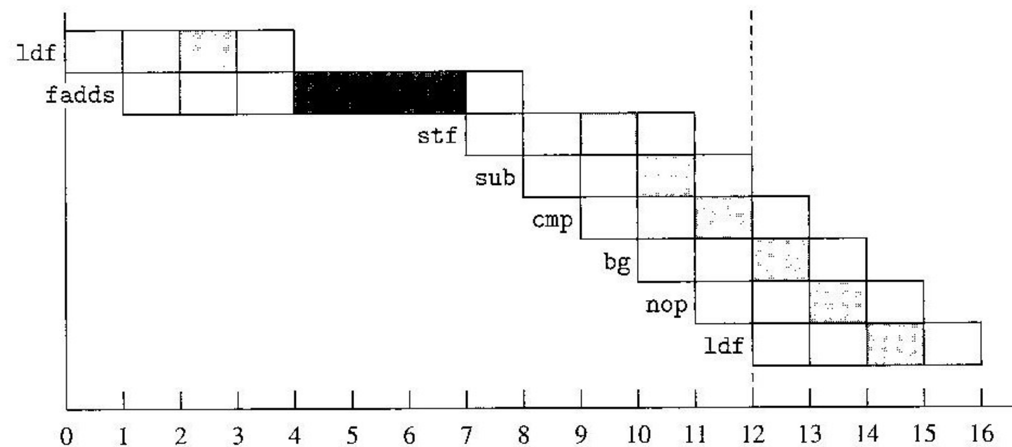


FIG. 17.16 Pipeline diagram for the loop body in Figure 17.15.

Software Pipelining

- 1 unidade de inteiro + 1 FP
  - Ciclos de execução sombreados
  - Load e stores de FP são feitos pela unidade de inteiros
- 6 ciclos entre fadds e stf
  - Idéia
    - Sobrepor o stf da iteração anterior com o fadds da iteração corrente
  - Ajustamos para iniciar o laço com o stf da iteração 1 e o fadds da iteração 2

Software Pipelining

- Adiciona
  - 3 instruções antes do laço
  - 5 instruções após o laço
    - Completa as 2 últimas iterações
- Ciclos para o corpo: 7
- Reduz o tempo por iteração por 5/12
  - ≈ 42%
- O laço deve iterar pelo menos 3 vezes

Software Pipelining

	Issue latency	Result latency
`ldf [r1],f0`		
`fadds f0,f1,f2`		
`ldf [r1-4],f0`		
`L: stf f2,[r1]`	1	3
`fadds f0,f1,f2`	1	7
`ldf [r1-8],f0`	1	1
`cmp r1,8`	1	1
`bg L`	1	2
`sub r1,4,r1`	1	1
`stf f2,[r1]`		
`sub r1,4,r1`		
`fadds f0,f1,f2`		
`stf f2,[r1]`		

FIG. 17.17 The result of software pipelining the loop in Figure 17.15, with issue and result latencies.

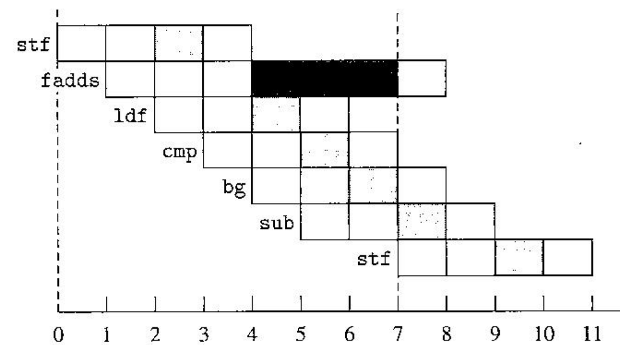


FIG. 17.18 Pipeline diagram for the loop body in Figure 17.17.

Software Pipelining

- Move iterações para fora do laço
  - No. Mínimo de iterações
    - Saber de antemão ou
    - Gerar código para testar
      - Manter 2 versões de laço
- Estratégia de implementação:
  - Window scheduling

Window Scheduling

- Faz duas cópias conectadas do grafo de dependência do corpo do laço
  - Corpo precisa ser um único BB
- Move uma janela pelas cópias
  - A cada ponto a janela contém uma cópia completa do corpo
  - Instruções acima e abaixo da janela
    - Prólogo e Epílogo do laço
- Movendo a janela tentamos vários escalonamentos
  - Busca por um que reduza a latência total do laço

Window Scheduling

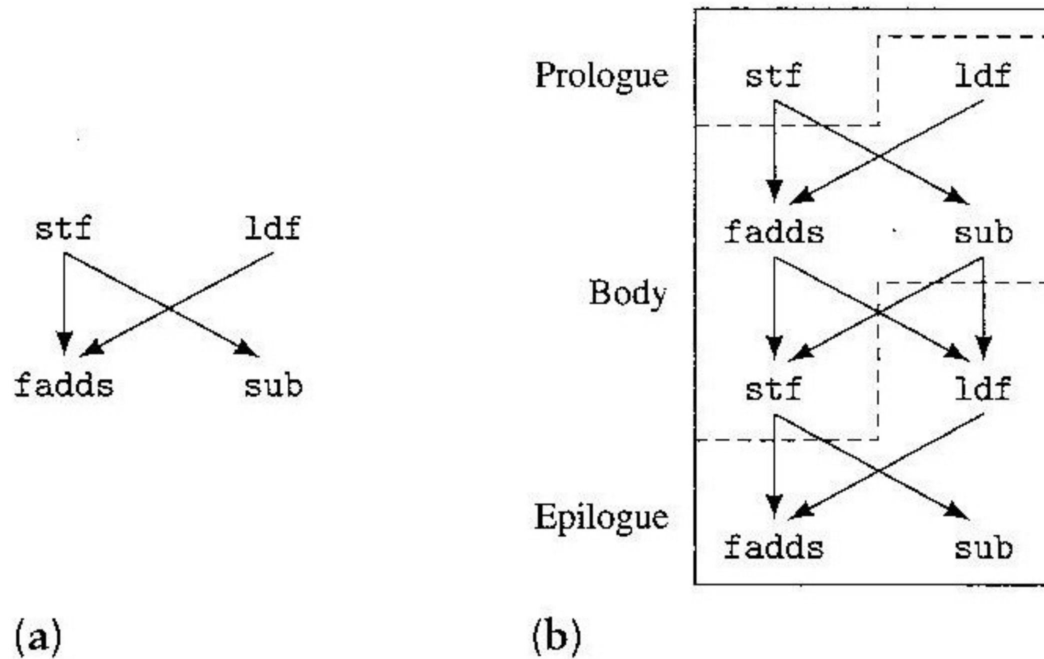


FIG. 17.19 (a) Dependence DAG for a sample loop body, and (b) double version of it used in window scheduling, with dashed lines showing a possible window.

Trace Scheduling

- Problema com *list scheduling*: Transições entre blocos básicos!
- Deve inserir instruções suficientes no fim do bloco básico para garantir que os resultados estarão disponíveis na entrada do próximo bloco básico.
  - *Overhead* significativo!

Trace Scheduling

- Solução: escalonamento entre blocos básicos.
  - Ex: *Trace Scheduling*
- Traço: sequência de instruções que incluem saltos, mas não laços. Formam um único caminho no programa.
- *Trace Scheduling*: escalonar instruções no trace inteiro de uma só vez.
- Traço são escolhidos pela sua frequência de execução
  - Detalhe: não pode escalonar grafos cíclicos. Laços devem ser desenrolados

Trace Scheduling

- Desenvolvido por Fisher(1981)
- Utilizado para escalonar código em
  - Máquinas VLIW
  - Superescalares com alta capacidade

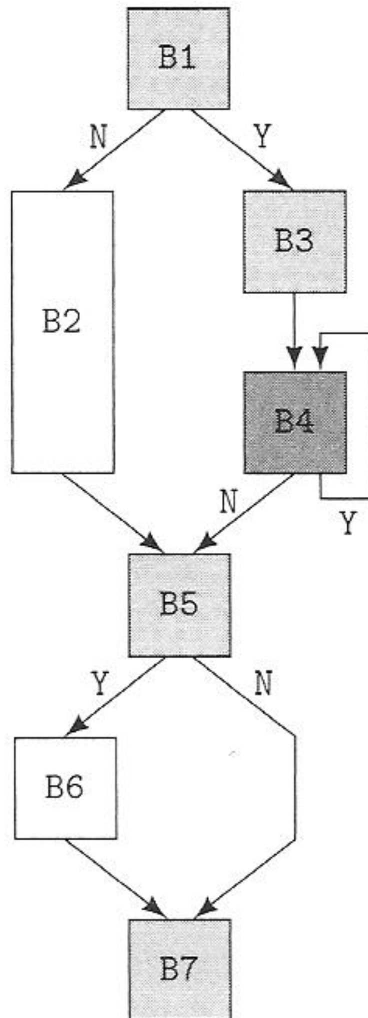
Trace Scheduling

- Escalonamento do traço
  - Usa-se um método de escalonamento de bloco básico
  - Escalona-se o traço todo
  - Iniciando pelo mais freqüentemente executado
- Pode causar execução fora de ordem
  - Necessita de inserção de códido de compensação
    - Entrada e saída dos traços
- Laços são normalmente desenrolados

Trace Scheduling

- Escalonamento e Compensação
  - Repetidos para todos os traços
  - Ou até que um limite de freqüência de execução seja atingido
    - EX: Traços executados pelo menos 20% das vezes
  - Traços que sobrarem são escalonados da maneira usual
- *Profiling*
  - Pode ser estimativa ou *feedback*
  - O segundo funciona bem melhor

Exemplo



- Caminho mais freqüente:
  - B1, B3, B4, B5 e B7
- Traços:
  - B1 e B3
  - B4 (Corpo do laço)
  - B5 e B7
  - Escalonados independentemente
- B2 e B6 escalonados separadamente
- Compensação:
  - Saída N do bloco B1
  - Vejamos exemplo ...

Exemplo

```
B1: x ← x + 1
    y ← x - y
    if x < 5 goto B3

B2: z ← x * z
    x ← x + 1
    goto B5

B3: y ← 2 * y
    x ← x - 2
    .
    .
    .
```

```
B1: x ← x + 1

    if x < 5 goto B3
B2: y ← x - y
    z ← x * z
    x ← x + 1
    goto B5
B3: y ← x - y
    y ← 2 * y
    x ← x - 2
     .
     .
     .
```

- Escalonador moveu x ← x - y para o bloco B3
  - Após um desvio
- Inclui uma cópia no início de B2
  - Compensação

Trace Scheduling

- Atinge bom desempenho
  - VLIW
  - Superescalares de alto grau
    - 8 instruções ou mais (Muchnick)
- Pode aumentar muito o tamanho do código
- Se o comportamento da aplicação depende da entrada
  - Desempenho pode variar muito entre execuções