Arquitectura de Computadores

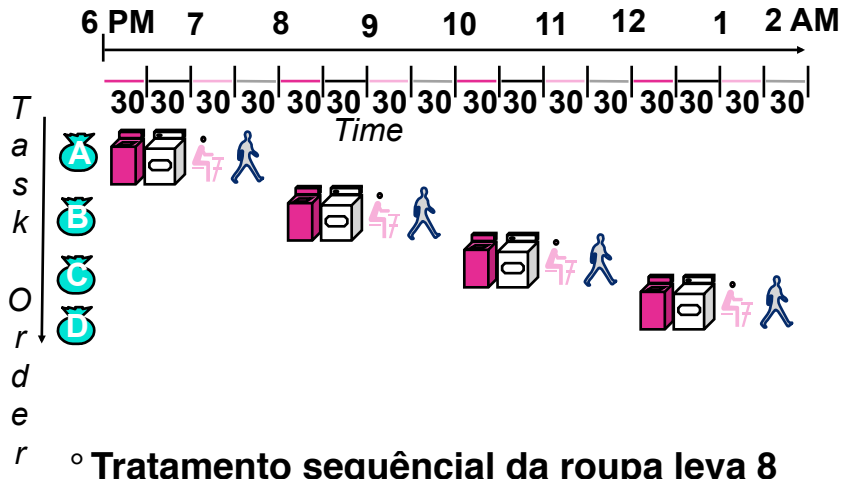
"Pipelining"

Docente: Pedro Sobral
http://www.ufp.pt/~pmsobral

Exemplo: Tratar da Roupa…

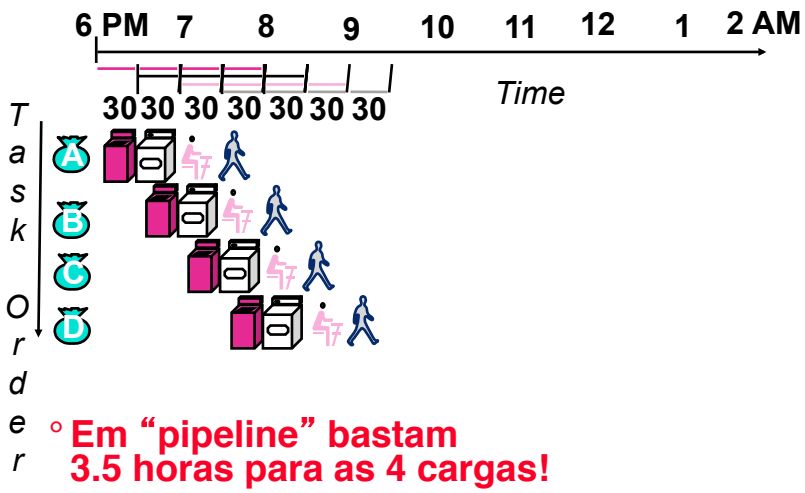
- Ana, Bruno, Carla, David têm, cada um,uma carga de roupa para lavar, secar, dobrar e guardar
- Lavagem leva 30 min.
- Secagem leva 30 min.
- "Dobragem" leva 30 min.
- "Arrumação" leva 30 min.

Tratamento sequêncial



- Tratamento sequêncial da roupa leva 8 horas para 4 cargas

Tratamento em "pipeline"

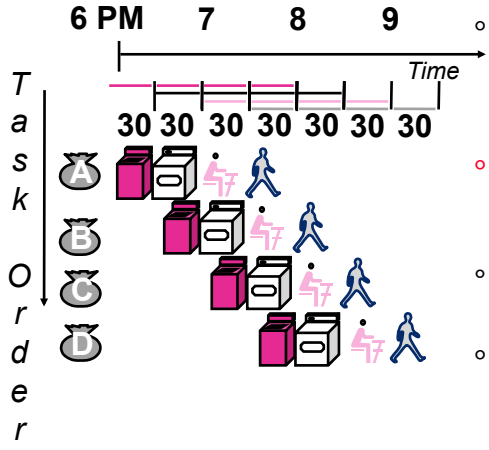


- Em "pipeline" bastam 3.5 horas para as 4 cargas!

Definições gerais

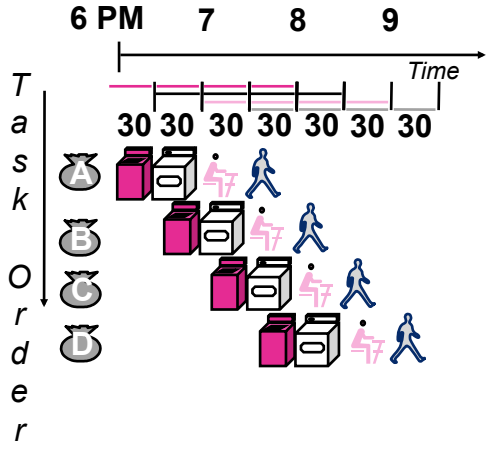
- **Latência**: tempo necessário para executar completamente uma tarefa
  - Por exemplo, o tempo de ler um sector de um disco é o tempo de acesso ao disco ou a sua latência
- **"Throughput"**: quantidade de trabalho executado num período temporal

Características do "pipeline" (1/2)



- O "pipeline" não altera a latência de uma tarefa, mas aumenta o "throughput" do sistema
- Multiplas tarefas em execução simultânea usando diferentes recursos
- "Speedup" potêncial = Número de fases do processo
- Tempo para "encher" e "esvaziar" o "pipeline" reduz o "speedup": 2.3X v. 4X neste exemplo

Características do "pipeline" (2/2)



- Suponha que a lavagem passava a levar 20min. Qual o aumento de desempenho do "pipeline"?
- A taxa do "pipeline" está limitada pela fase mais lenta
- Fases com tempos muito diferentes reduzem o aumento de desempenho-"speedup"

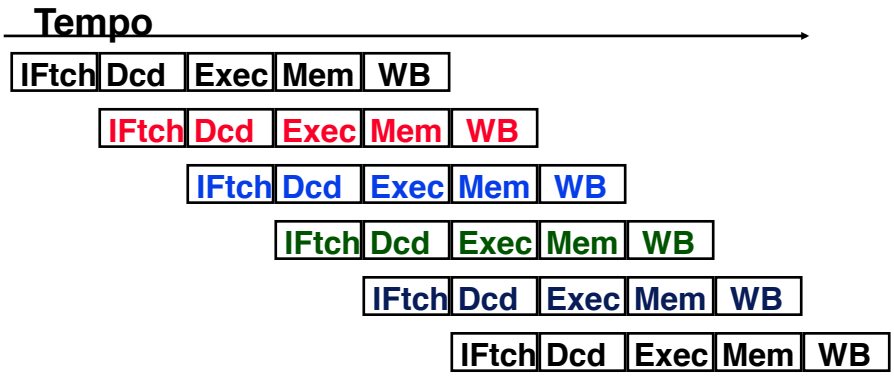
Fases na execução de instruções do MIPS

1) IFtch: "Instruction Fetch"- carregar a instrução, incrementar PC
2) Dcd: "Instruction Decode" – descodificar a instrução, ler os registos
3) Exec:"Execute" –executar a instrução

   Ref. Memória: Calcular endereço
   Operação Log. Ou arit. : executar a operação

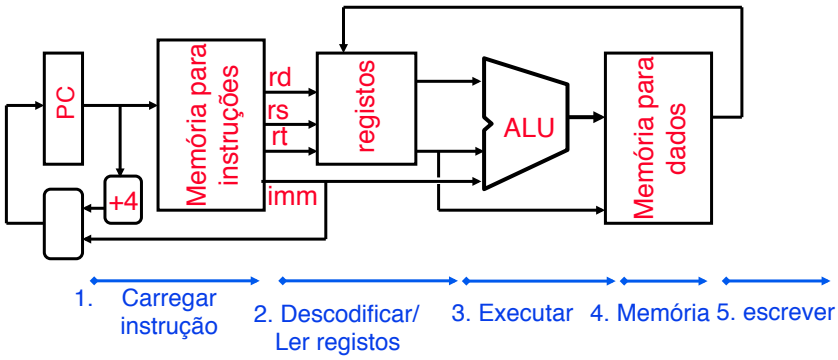
4) Mem: "Acesso à memória"
   Load: Ler dados da memória
   Store: Escrever para a memória
5) WB: "Write Back" – escrever resultados de volta para os registos

Representação da execução em "pipeline"

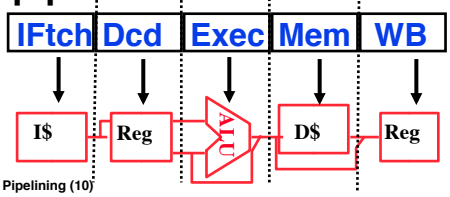


- Todas as instruções têm o mesmo número de passos, ou "fases" no pipeline, portanto, dependendo da instrução, durante algumas "fases" podem ficar em espera…

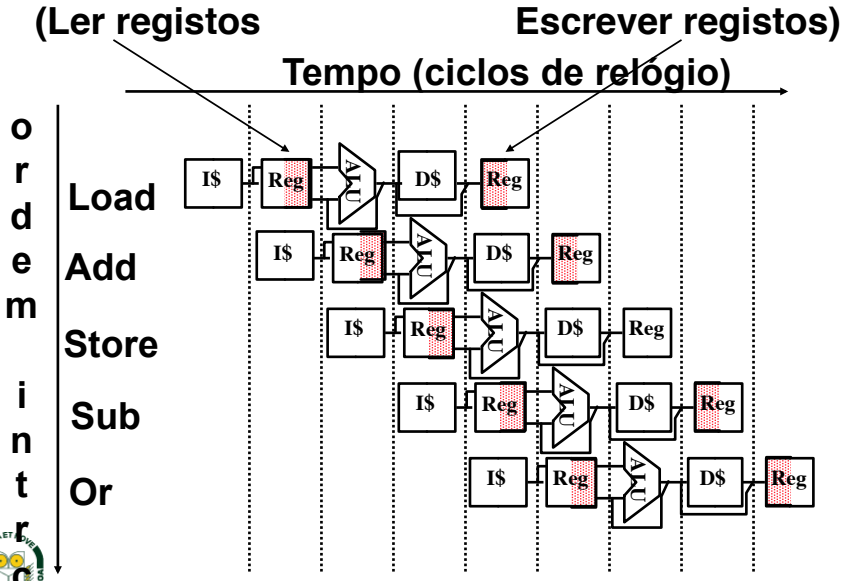
Circuito de dados e o "pipeline"



- Podemos usar o circuito de dados para representar o pipeline…



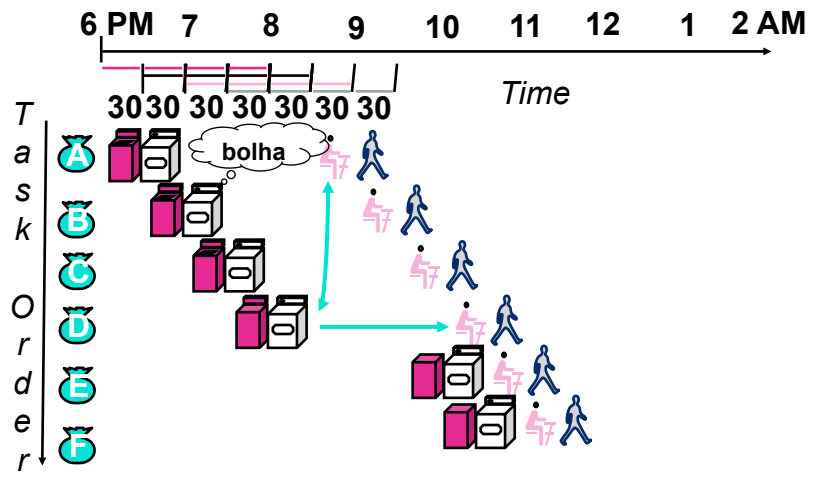
Representação gráfica do "pipeline"



Exemplo

- Vamos considerar: 2 ns para aceder à memória, 2 ns para a operação da ALU e 1 ns para ler ou escrever registos.
- Qual a taxa de instruções?
- Sem "pipeline":
  - lw : IF + Ler Reg + ALU + Memória + Esc. Reg = 2 + 1 + 2 + 2 + 1 = 8 ns
  - add: IF + Ler Reg + ALU + Esc. Reg = 2 + 1 + 2 + 1 = 6 ns
- Com "pipeline":
  - Max(IF,Ler Reg,ALU,Memória,Esc. Reg) = 2 ns

Acidente: Meias em lavagens separadas (A, D)

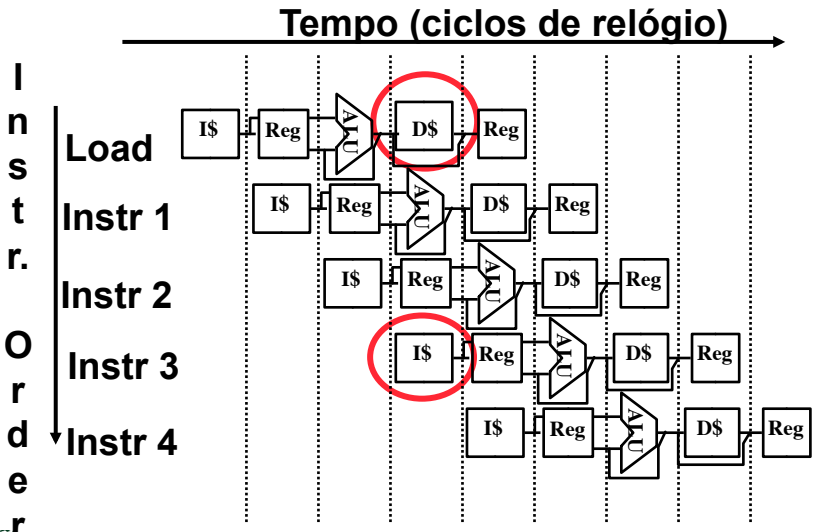


A depende de D; Logo a dobragem tem que PARAR

Problemas para os Computadores

- Limites do "pipelining": Acidentes podem impedir a próxima instrução de ser executada no seu ciclo de relógio:
  - Acidentes estruturais: O hardware não suporta a combinação de instruções (no exemplo, uma única pessoa para dobrar e guardar a roupa)
  - Acidentes de controlo: A execução em pipeline de saltos & outras instruções param o pipeline até ser possível continuar; são inseridas "bolhas" no pipeline
  - Acidentes de dados: Uma instrução que depende do resultado de outra ainda no pipeline (no exemplo, a meia que faltava…)

Acidente estrutural #1: Memória única (1/2)

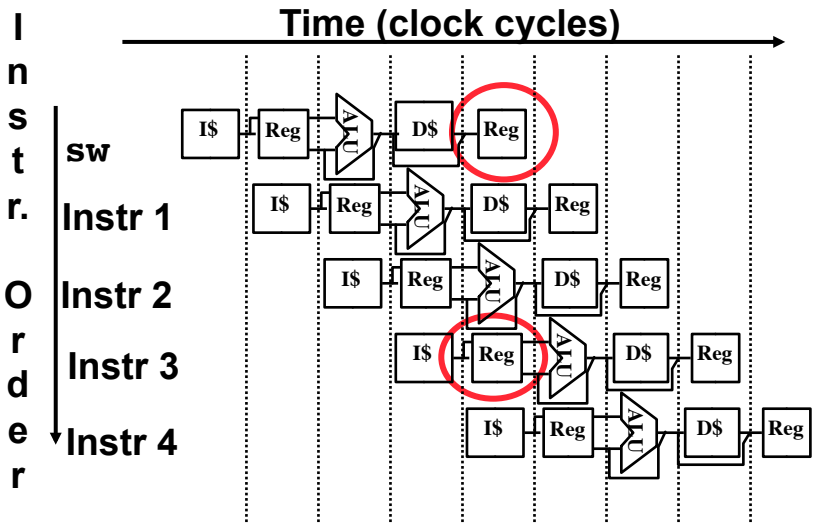


Ler a mesma memória 2 vezes no mesmo ciclo

Acidente estrutural #1: Memória única (2/2)

- Solução:
  - Pouco prático e eficiente criar uma segunda memória…
  - Portanto vamos simular isto com 2 caches de nível 1 (uma cópia de uma pequena parte da memória príncipal…)
  - Temos portanto uma cache de instruções (L1) e uma cache de dados (L1)
  - Claro que é necessário um hardware mais complexo para gerir ambas as caches…

Acidente estrutural #2: Registos (1/2)

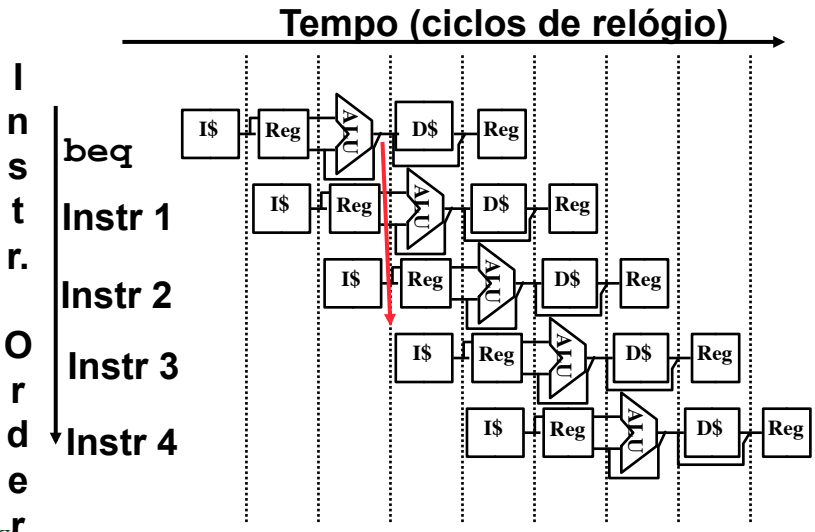


Não podemos ler e escrever os registos simultâneamente

Acidente estrutural #2: Registos (2/2)

- Facto: O acesso aos registos é *MUITO* rápido: leva menos de metade do tempo da ALU
- Solução: introduzir uma convenção
  - **Escrever sempre** para os registos durante a primeira metade do ciclo de relógio
  - **Ler sempre** dos registos durante a segunda metade do ciclo de relógio.
  - **Resultado: Podemos ler e escrever no mesmo ciclo de relógio**

Acidentes de controlo: Saltos (1/7)



Onde efectuamos a comparação para o salto?

Acidentes de controlo: Saltos (2/7)

- O hardware decide o salto na fase da ALU
  - Portanto duas instruções após o salto vão ser sempre carregadas no pipeline independentemente de fazermos ou não o salto.
- O ideal para os saltos:
  - Se não vamos saltar não se deve perder tempo e continuar a execução normalmente…
  - Se vamos saltar, não executar nenhuma instrução após o salto e continuar na "Label" desejada

Acidentes de controlo: Saltos (3/7)

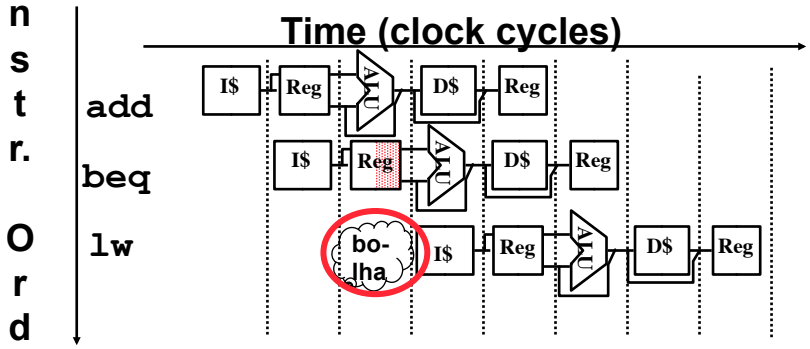
- Solução inicial: Parar o pipeline até conhecer a decisão:
  - inserir instruções `nop` "no-operation" que não fazem nada, apenas perdem tempo…
  - Inconveniente: Saltos levam 3 ciclos de relógio cada (assumindo que o comparador é colocado na fase da ALU)

Acidentes de controlo: Saltos (4/7)

- Optimização #1:
  - mover a comparação para a fase 2 (Dcd)
  - Assim que a instrução é descodificada (Opcode=salto) imediatamente tomar a decisão e alterar o valor do PC
  - Vantagem: Como o salto se completa na fase 2, apenas uma instrução adicional é carregada no pipeline portanto apenas é necessário um "nop"
  - Nota: Isto quer dizer que as instruções de salto não usam as fases 3,4 e 5.

Acidentes de controlo: Saltos (5/7)

- Inserir uma "nop" (bolha)



- Impacto: 2 ciclos de relógio por instrução de salto ⇒ LENTO

Acidentes de controlo: Saltos (6/7)

- Optimização #2: Redefinir os saltos
  - Definição antiga: Se efectuamos o salto, nenhuma das instruções após o salto é executada
  - Nova definição: Quer se efectue o salto ou não, a instrução imediatamente após o salto é executada (Chama-se "branch-delay slot")
- O termo "Delayed Branch" significa que executamos sempre a instrução após o salto

Acidentes de controlo: Saltos (7/7)

- Notas sobre o "Branch-Delay Slot"
  - Pior cenário: podemos sempre colocar um "nop" após o salto.
  - Melhor caso: Conseguimos encontrar uma instrução que se encontra antes do salto que pode ser colocada no "branch-delay slot" sem afectar o resultado do programa
    - Reordenar instruções é um método comum de acelerar a execução de programas
    - O compilador tem que ser muito inteligente de forma a encontrar estas instruções
    - Geralmente encontra-as em pelo menos 50% das situações
    - Os saltos incondicionais também têm "delay slot"…

Exemplo: Saltos com e sem "delay"

Sem "delay"

```
or   $8, $9 ,$10
add $1 ,$2,$3
sub $4, $5,$6
beq $1, $4, Exit
xor $10, $1,$11
Exit:
```

Com "delay"

```
add $1 ,$2,$3
sub $4, $5,$6
beq $1, $4, Exit
or   $8, $9 ,$10
xor $10, $1,$11
Exit:
```

Exercício: "Branch Delay Slot"

- Como usar o "branch delay slot" neste caso?
  - Atenção que há dois saltos! (beq e j)

```
main:  and $t0,$0,$0
       addi $s0,$0,-3
loop:  slti $t1, $t0, 3
       beq $t1,$0, end
       sll $s0,$s0,2
       addi $t0,$t0,1
       j loop
end:
```

```
main:  and $t0,$0,$0
       addi $s0,$0,-3
loop:  slti $t1, $t0, 3
       beq $t1,$0, end
       nop
       addi $t0,$t0,1
       j loop
       sll $s0,$s0,2
end:
```

Exemplo

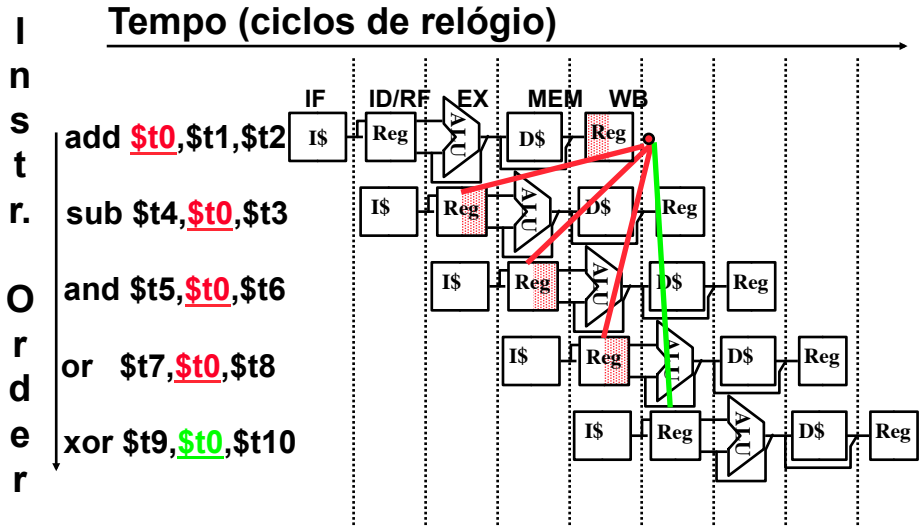
Acidentes de dados (1/2)

- Considere a seguinte sequência de instruções

```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

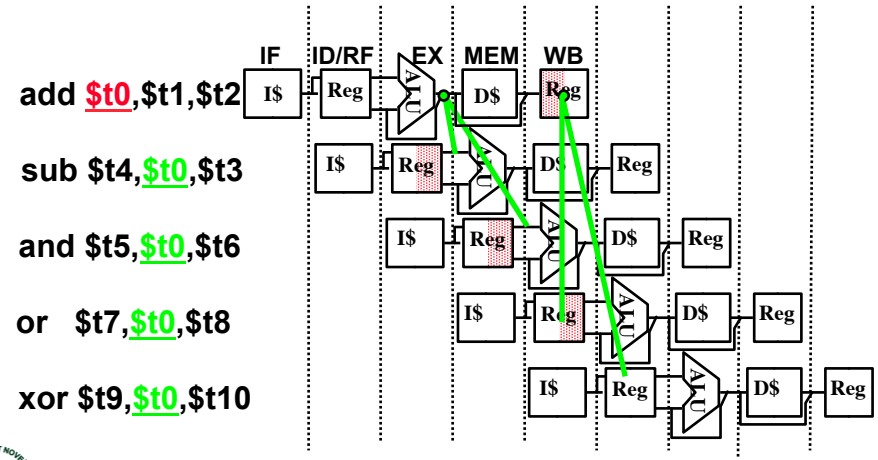
O valor de $t0 calculado na 1ª instrução é necessário nas instruções seguintes. Isto traz problemas ao pipeline. Chama-se um ***acidente de dados***

Acidentes de dados (2/2)



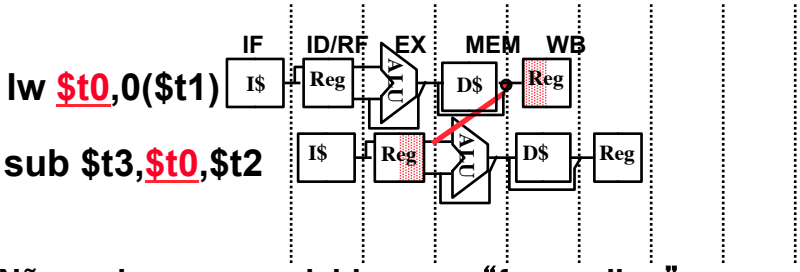
Solução para acidentes de dados: "Forwarding"

- Enviar ("Forward") o resultado de uma fase para outra posterior usando hardware do pipeline ("atalhos")



Acidentes de dados: "Loads" (1/4)

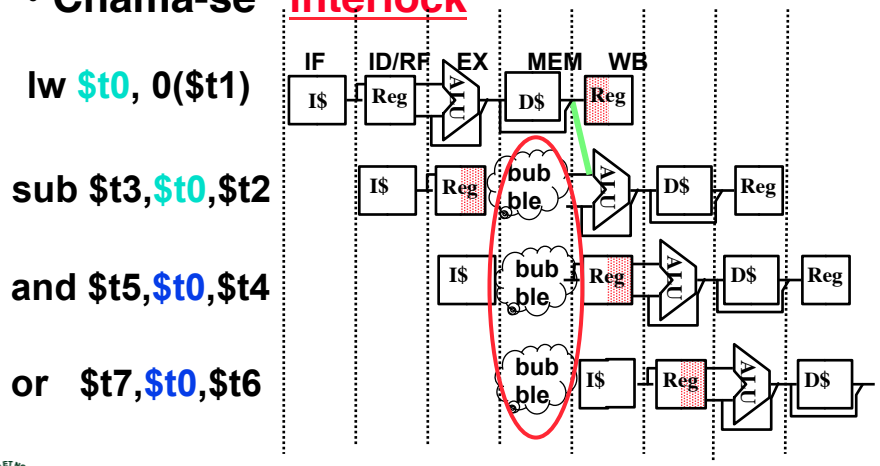
- Dependências para trás no tempo são também acidentes…



- Não podem ser resolvidos com "forwarding"
- É necessário parar a instrução dependente do "load" e depois fazer o "forwarding" (mais hardware…)

Acidentes de dados: "Loads" (2/4)

- O Hardware tem que PARAR o pipeline
- Chama-se "interlock"



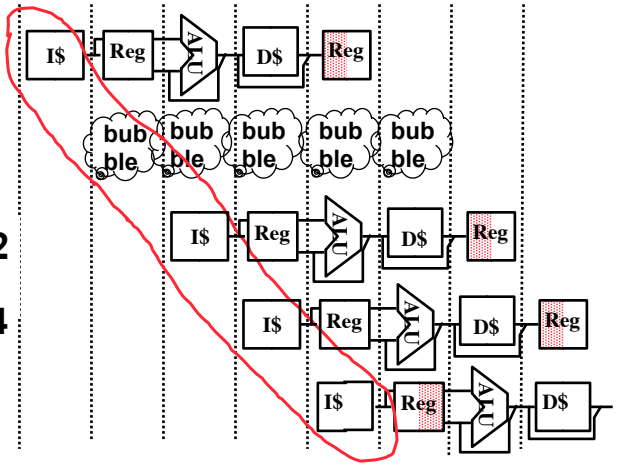
Acidentes de dados: “Loads” (3/4)

- A posição da instrução a seguir ao “load” chama-se “load delay slot”
- Se essa instrução usa o resultado do “load”, então o “interlock” pára o pipeline durante um ciclo de relógio.
- Se o compilador colocar uma instrução não relacionada com o “load” no “load delay slot” então o pipeline não necessita de parar.
- Parar o pipeline é equivalente a colocar uma instrução “nop” no “ load delay slot” (embora ocupe mais espaço no código)

Acidentes de dados: “Loads” (4/4)

- Parar o pipeline é equivalente a um “nop”

```
lw $t0, 0($t1)
nop
sub $t3,$t0,$t2
and $t5,$t0,$t4
or  $t7,$t0,$t6
```



Curiosidade

- O primeiro desenho do MIPS não executava o "interlock" do pipeline no caso de um acidente de dados com o "load"
- A razão do nome MIPS:

  Microprocessor without
  Interlocked
  Pipeline
  Stages

  - E não um termo muito comum….
    Millions of Instructions Per Second,
    também com a sigla MIPS

Questão

Assuma 1 instr/ciclo, "delayed branch", pipeline com 5 fases, "forwarding", "interlock" em acidentes de dados com o "load" (depois de 10^3 ciclos, portanto o pipeline está cheio)

```
Loop:       lw    $t0, 0($s1)
            addu  $t0, $t0, $s2
            sw    $t0, 0($s1)
            addi  $s1, $s1, -4
            beq   $s1, $zero, Loop
```

1
2
3
4
5
6
7
8
9
10

- Quantos ciclos de relógio são necessários para cada iteração do ciclo indicado ?