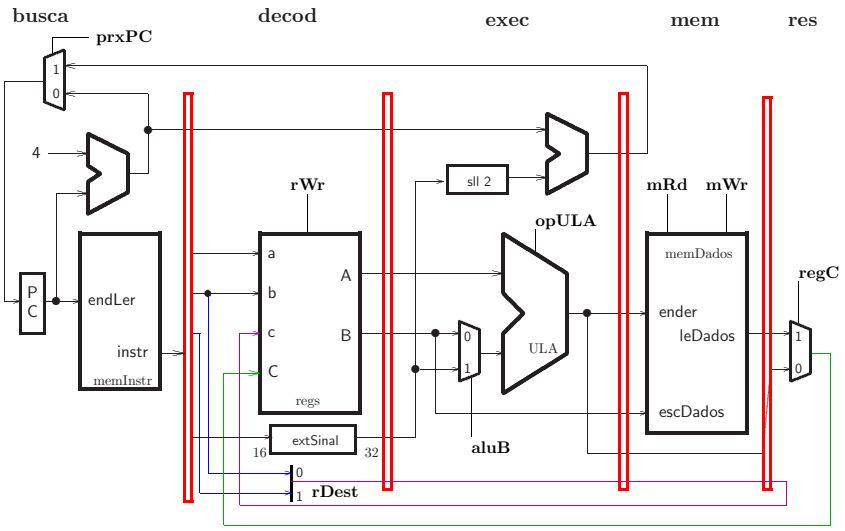
Revisão – segmentação

- Todos os processadores modernos usam segmentação
- segmentação não reduz a latência de uma instrução
  mas ajuda na vazão/produção do programa inteiro
  → várias tarefas em execução simultânea usando recursos distintos
- ganho potencial: ≈ número de estágios CPI: 3-5 → 1
- vazão do pipeline limitada pelo estágio mais lento
  estágios desbalanceados reduzem ganho
  tempo para encher e para drenar segmentos reduz ganho
- controle deve detectar e resolver riscos
  bloqueios afetam vazão negativamente
- esta aula: controle dos segmentos (e de riscos)

Controle em Processador Segmentado



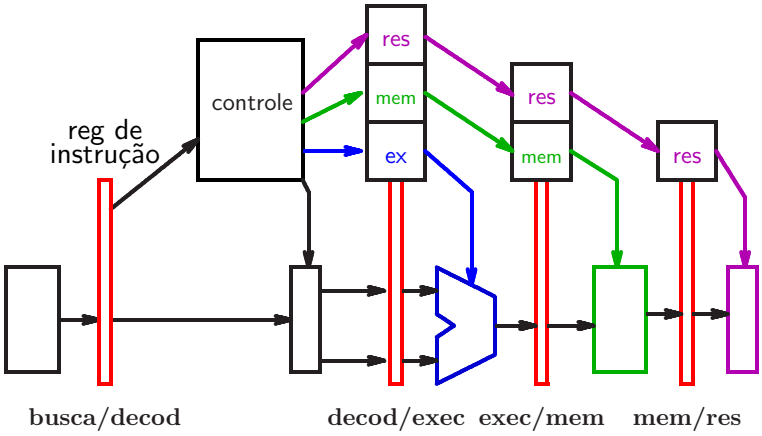
Sinais de controle do processador segmentado

	exec			mem			res	
	rDest	opULA	aluB	prxPC	mRd	mWr	rWr	regC
ALU	1	fun	0	0	0	0	1	0
IMM	0	oper	1	0	0	0	1	0
lw	0	+	1	0	1	0	1	1
sw	x	+	1	0	0	1	0	x
beq	x	−	0	1	0	0	0	x

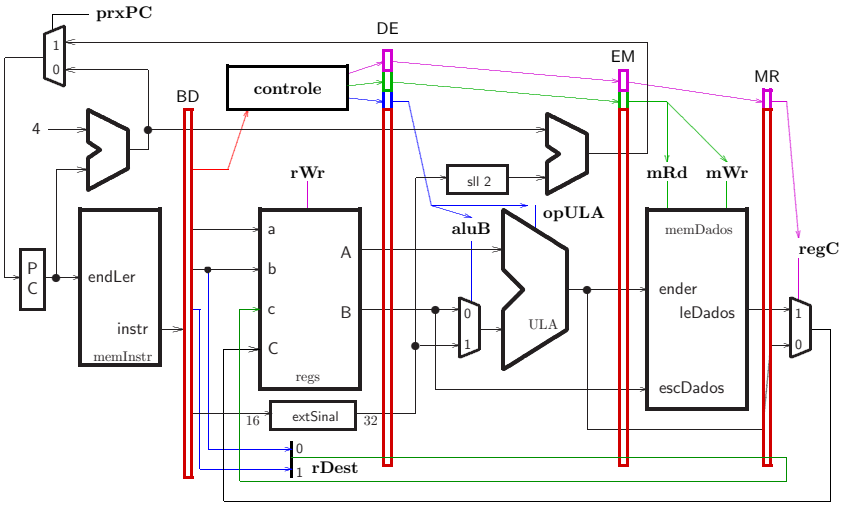
registradores dos segmentos são atualizados a cada ciclo
busca e **decod**: sempre busca instrução e incrementa PC

Controle em Processador Segmentado

Todos os sinais de controle são determinados na decodificação e mantidos nos registradores entre os estágios



Controle do Processador Segmentado



Modelo seqüencial de execução

Conjunto de instruções de processadores "comuns" define um **modelo seqüencial de execução:**

cada instrução é completamente executada
e altera o estado do processador
antes do início da próxima instrução

Este modelo facilita muito a vida do programador!

Como programador em C pensa que cada comando executa?

```
if(p == q) { x = y + z; t = w * z; }; a = b - c;
```

Riscos em processadores segmentados

Riscos são condições que levam a comportamento incorreto
se as medidas apropriadas não forem tomadas

- **riscos estruturais** — structural hazards
  - ▷ quando duas instruções **diferentes** necessitam do **mesmo** recurso no **mesmo** ciclo
- **riscos com dados** — data hazards
  - ▷ quando 2 instr **diferentes** usam **mesmo** local de armazenamento
  - ▷ resolução do risco deve garantir aparência de que instruções executaram na ordem seqüencial correta
- **riscos de controle** — control hazards
  - ▷ quando uma instrução determina **quais** instruções serão executadas a seguir (desvios, saltos, funções)

Riscos com dados

Quando duas instruções **diferentes** usam **mesmo** local de armazenamento — data hazards

Deve parecer que instruções executam na ordem seqüencial correta

```
i:  r1 ⇐ r4 + r5
j:  r2 ⇐ r1 - r9     r1 foi produzido por i
k:  r1 ⇐ r6 ⊕ r3     valor de r1 em i é sobre-escrito
    resultado de i;j;k é o mesmo que i;k;j ?
```

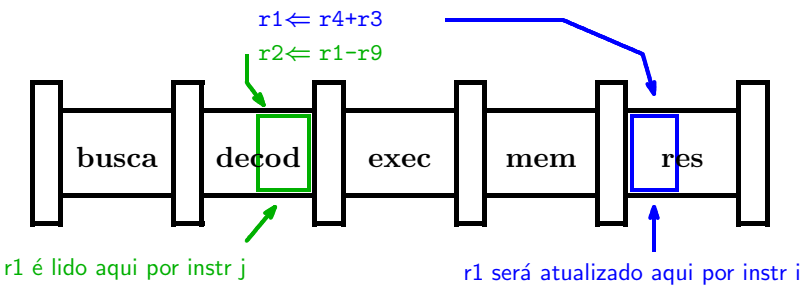
Convenção: **nome do risco** é a ordem do programa que
deve ser preservada pela implementação (segmentada, superescalar)

Riscos com dados - RAW

- Read-After-Write (RAW)
  instr **j** tenta ler operando **r1** ANTES que instr **i** escreva resultado

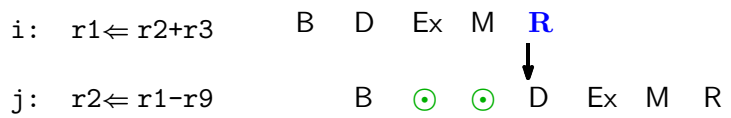
```
i:  r1 ⇐ r4 + r3
j:  r2 ⇐ r1 - r9
```

- Risco decorre de uma **dependência de dados**, causada pela comunicação entre as duas instruções add e sub através de **r1**

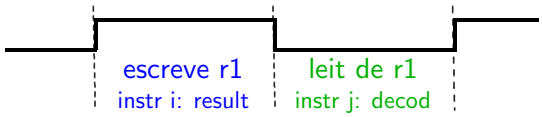


Solução (parcial) simples para RAW

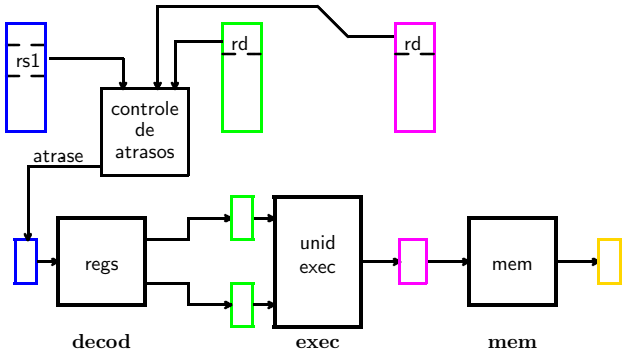
- Circuito de controle detecta risco, e então insere bolha:
  atrasa instrução **j** até ocorrer escrita do resultado de **i**
  - ∗ solução simples – segura instrução dependente na **busca** stall
  - ∗ desempenho ruim por causa dos ciclos desperdiçados



- solução simples pressupõe que, em cada ciclo, registradores são atualizados num semiciclo e são lidos no outro



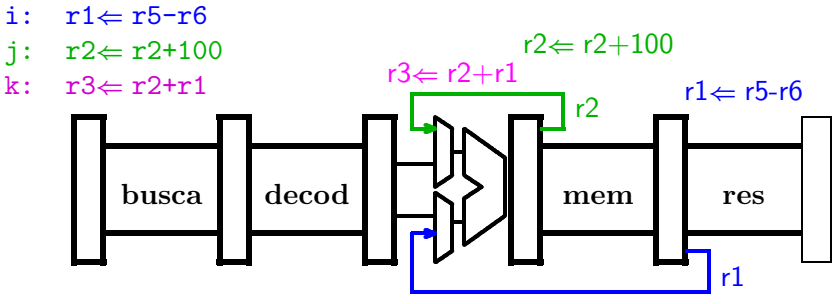
Risco de dados: bloqueio



- Compara com estágios posteriores
  - ▷ `if (rs1(decod)==rd(exec) || rs1(decod)==rd(mem))`
    `{ insere bolha }` mesmo para `rs2`
  - ▷ todos conflitos são riscos? st não escreve reg, addi não lê reg

Risco de dados: adiantamento (i)

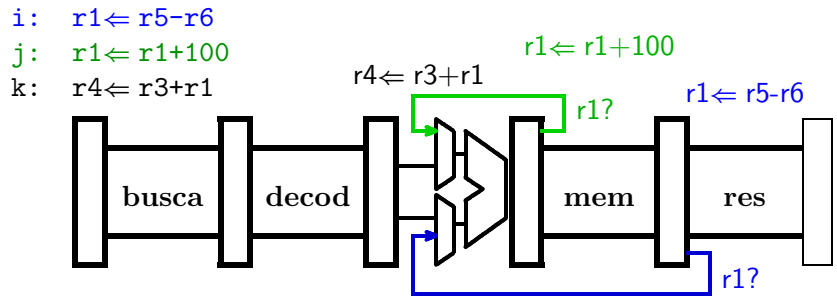
Ao invés de bloquear, adianta resultado para entradas da ULA:
usa controle de atrasos para decidir se deve adiantar, e
usa multiplexadores para escolher fonte do resultado.



Adiantamento = forwarding, bypassing, short-circuiting

Risco de dados: adiantamento (ii)

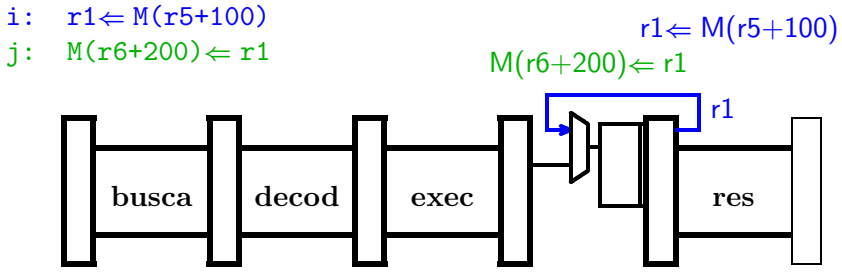
Quem fornece r1 para instrução **k**?



Implementação deve satisfazer modelo seqüencial de execução
senão, os clientes ficarão muito irritados...

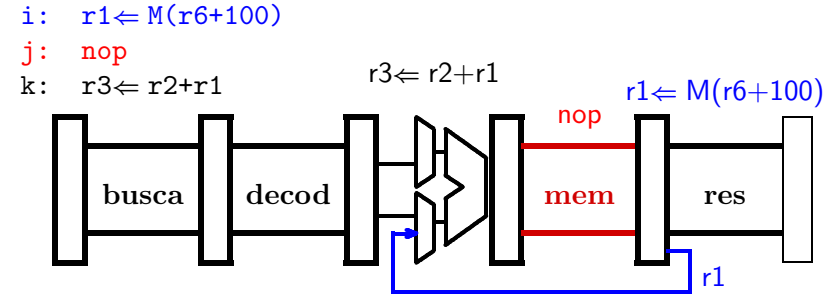
Risco de dados: adiantamento (iii)

Adiantamento para o estágio de memória
`load` seguido de `store`



Risco de dados: adiantamento (iv)

Adiantamento para o estágio de memória: `load` seguido por `add`



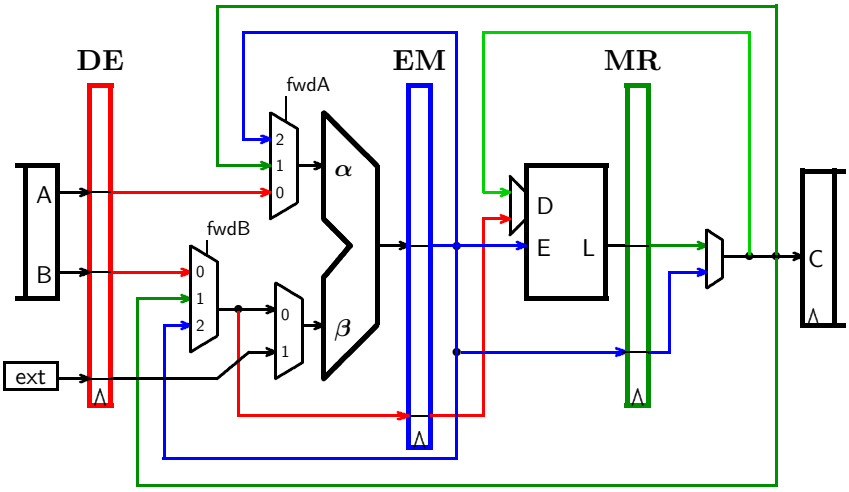
★ Risco deve ser detectado por hardware e bolha inserida *stall*
→ desempenho cai por causa da bolha

★ Compilador deve tentar preencher bolha com instrução "boa"
load delay slot introduzido no conj de instr MIPS-I
e eliminado do CdI MIPS-II **por que foi eliminado?**

Adiantamento – implementação

- Adiantamento nas linhas de dependência para trás no tempo
  - ▷ estágio EXEC produz resultado de instr de ULA ou ender efetivo
  - ▷ estágio MEM produz resultado de `lw`
- Adianta para entradas da ULA valor na saída de qualquer registrador de segmento e não somente do reg. DE:
  - ∗ adiciona multiplexadores nas entradas da ULA
    para passar **rd** para as entradas **rs** e **rt** da ULA
    - 0: entrada normal (registrador DE)
    - 2: adianta da instrução anterior (registrador EM)
    - 1: adianta de duas instruções atrás (registrador MR)
  - ∗ circuito adicional de controle
- permite execução sem bolhas, mesmo com dependências de dados
  ⇝ exceto no uso do valor do `load`...

Adiantamento - circuito completo



Controle de Adiantamento (1/4)

- Risco EX/MEM:
  - ⋆ Lembre que o núm do reg de destino (Rd) viaja junto com a instrução
  - ⋆ `rd` é registrador destino **rd** ou **rt**
  - ⋆ `rs` é o número do registrador **rs**
  - ⋆ `rt` é o número do registrador **rt**
  - ⋆ `fwdA, fwdB` controlam os multiplexadores

```
if ( EM.rd == DE.rs ) fwdA = 2          anterior
if ( EM.rd == DE.rt ) fwdB = 2
```

- risco MEM/RES:

```
if ( MR.rd == DE.rs ) fwdA = 1          2 antes
if ( EM.rd == DE.rt ) fwdB = 1
```

- O que está errado no controle?
  Quando pode adiantar indevidamente?
  Quais seqüências de instruções revelariam o erro?

Controle de Adiantamento (2/4)

- Risco EX/MEM:

```
if ( EM.regWR                          se instr escreve
  and ( EM.rd == DE.rs ) )                   anterior
     fwdA = 2
```

  O MESMO PARA `fwdB`

- risco MEM/RES:

```
if ( MR.regWR                          se instr escreve
  and ( MR.rd == DE.rs ) )                  2 antes
     fwdA = 1
```

  O MESMO PARA `fwdB`

- O que está errado no controle?
  Quando pode adiantar indevidamente?
  Quais seqüências de instruções revelariam o erro?

Controle de Adiantamento (3/4)

- Risco EX/MEM:

```
if ( EM.regWR                          se instr escreve
  and ( EM.rd != 0 )                    dest não é $r0
  and ( EM.rd == DE.rs ) )                   anterior
     fwdA = 2
```

  O MESMO PARA `fwdB`

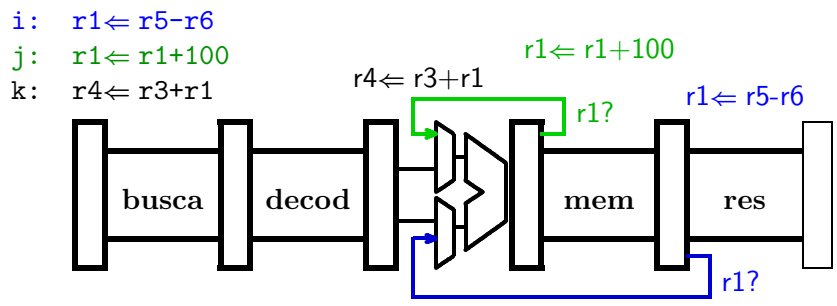
- risco MEM/RES:

```
if ( MR.regWR                          se instr escreve
  and ( MR.rd != 0 )                    dest não é $r0
  and ( MR.rd == DE.rs ) )                  2 antes
     fwdA = 1
```

  O MESMO PARA `fwdB`

- O que está errado no controle?

Lembre do modelo seqüencial



Adiantamento deve entregar resultado mais recente,
que é o da instrução **j**

Controle de Adiantamento (4/4)

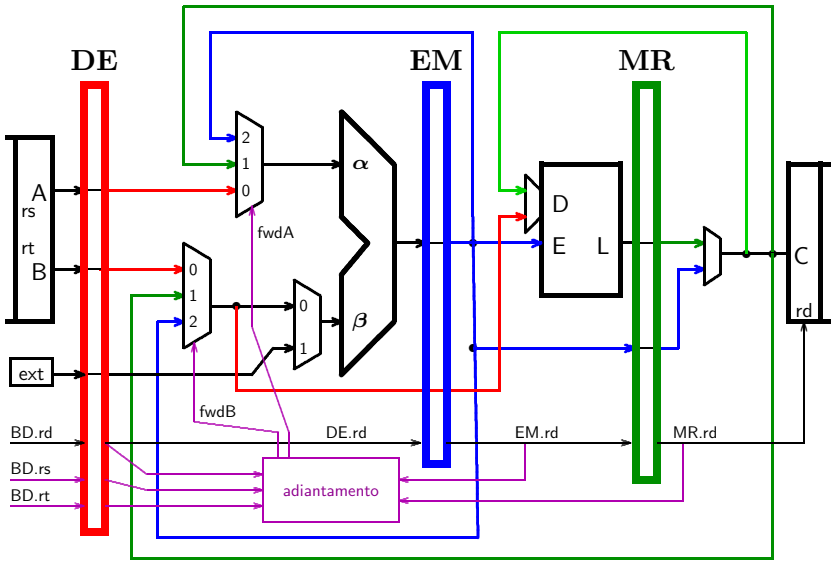
- risco MEM/RES:

```
if ( MR.regWR                                    se instr escreve
  and ( MR.rd != 0 )                             dest não é $r0
  and ( MR.rd == DE.rs )                         anterior
  and ( EM.rd != DE.rs || ~EM.regWR ))           +novo
  fwdA = 1
```

  O MESMO PARA `fwdB`

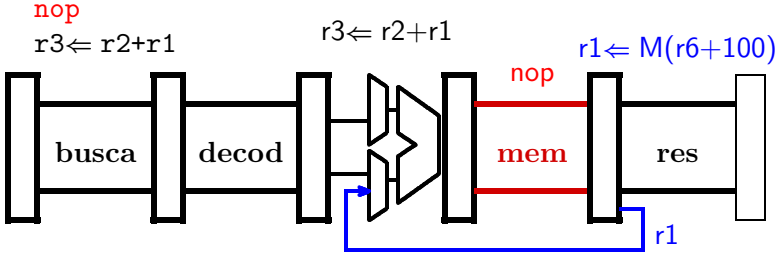
- adiante
  SE esta instrução escreve
  E não escreve em $r0
  E reg destino da anterior é igual ao fonte desta (+velha)
  E (registrador "no meio" não é o destino
  ou instrução não escreve registrador)

Adiantamento - circuito completo + controle



Risco com uso do resultado do LD

```
i:  r1⇐ M(r6+100)
j:  nop
k:  r3⇐ r2+r1
```



Risco com uso do resultado do LD

- DECOD deve detectar risco entre LD e usos do seu resultado

```
if ( DE.memRD
     and ( ( DE.rt == BD.rs )
           or ( DE.rt == BD.rt ) ) )
  bloqueia_DE                                  segura segmentos
```

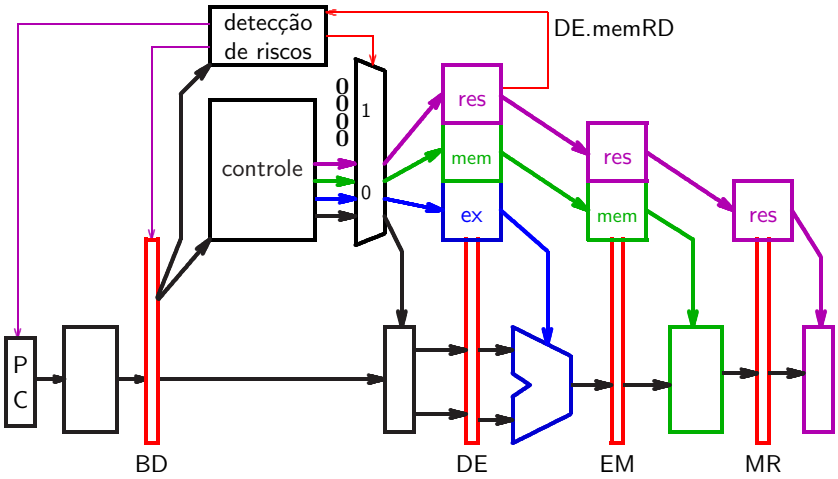
- segura segmentos
  SE instrução em EXEC lê memória
  E LD em EXEC produz operando da instr em DECOD
- depois deste ciclo parado,
  lógica de adiantamento resolve os demais riscos

Circuito de controle de bloqueios

Além de detectar os riscos, implementação de bloqueios/*stalls* deve:

- impedir que instruções em BUSCA e DECOD avançem
  → mantém os conteúdos do PC e do registrador BD
  re-executa a mesma instrução, sem que ela altere estado
  * desativar os sinais de controle (muda para $\mathbf{0}$) nos campos de controle dos estágios EXEC, MEM e RES
  * circuito detector de riscos controla MUX que seleciona entre valores de controle e $\mathbf{0}$s
  * pressupõe que $\mathbf{0}$s são valores inócuos → nada muda
  esta instrução não altera o estado da computação
- instruções nos demais estágios (EX,MEM,RES) completam normalmente

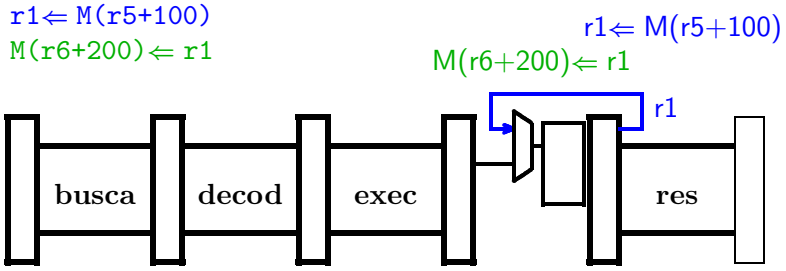
Circuito de controle de bloqueios



Risco com LD seguido de ST

Nas cópias memória-memória (LD;ST) pode adiantar
saída do registrador MR para a entrada da memória
→ necessita circuito de adiantamento para estágio de memória

```
i:  r1⇐ M(r5+100)
j:  M(r6+200)⇐ r1
```



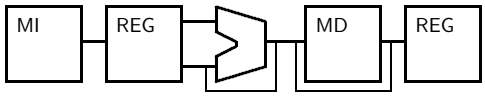
Esta é uma operação comum: copia de um endereço para outro
É necessário adiantar os endereços efetivos?
Cuidados especiais com `p = *p;` `*p = *p;` `*p = p;`

Falta alguma coisa?

```
addu r5, ...
sw r5, ...
```

```
addu r5, ...
subu r6, ...
sw r5, ...
```

Necessita caminho para adiantamento da saída da ULA
para entrada de dados da memória.



Resumo (i)

Modelo seqüencial de execução

Conjunto de instruções de processadores "comuns" define um
modelo seqüencial de execução:

cada instrução é completamente executada
e altera o estado do processador
antes do início da próxima instrução

programador considera que cada comando C executa atomicamente
isso não é verdade com segmentação!

bolhas reduzem o paralelismo ⇝ serializam a computação

Resumo (ii)

- Dependências de dados resolvidas com adiantamento (quase sempre)
- Deve garantir que instruções anteriores escreverão resultado,
  destino é mesmo que fonte,
  e instrução anterior não tem prioridade
- Acrescentar circuito de adiantamento onde pode-se adiantar
  $\rightarrow$ força bloqueio se precisa esperar por resultado
  estágio EXEC, MEM para store, DECOD para desvio
- LOADs necessitam parada porque sobrepõem EXEC com MEM
  desvios podem necessitar de parada também stall=parada
- Próxima aula: riscos de controle e previsão de desvios.
- **Exercício:** Desenhe, numa folha A3 quadriculada,
  o circuito completo do processador segmentado,
  incluindo todos os circuitos mostrados nos slides 5 e 23.