Universidade do Minho

Módulo 8

Y86 PIPE: Resolução de anomalias e escalonamento dinâmico

6/Dez/2012

Y86 PIPE-: Limitações

- **Dependências de dados**
  - Uma leitura de um registo precedida de uma escrita no mesmo registo constitui uma dependência de dados (**RAW**).
  - Se a leitura ocorre antes da conclusão da escrita ocorre uma anomalia.
  - Na versão **PIPE-** estas anomalias são corrigidas empatando o *pipeline* através da injeção de "bolhas" (nops).
- **Dependências de controlo**
  - O desfecho dos saltos condicionais só é conhecido depois da fase de execução. O Y86 prevê que o salto é tomado, executando as instruções no alvo de forma especulativa. Previsões erradas são corrigidas inserindo "bolhas".
  - O destino de um *ret* só é conhecido depois da fase de leitura de memória. O Y86 resolve esta anomalia inserindo "bolhas" até que o endereço da próxima instrução seja conhecido.

Y86 PIPE: Motivação

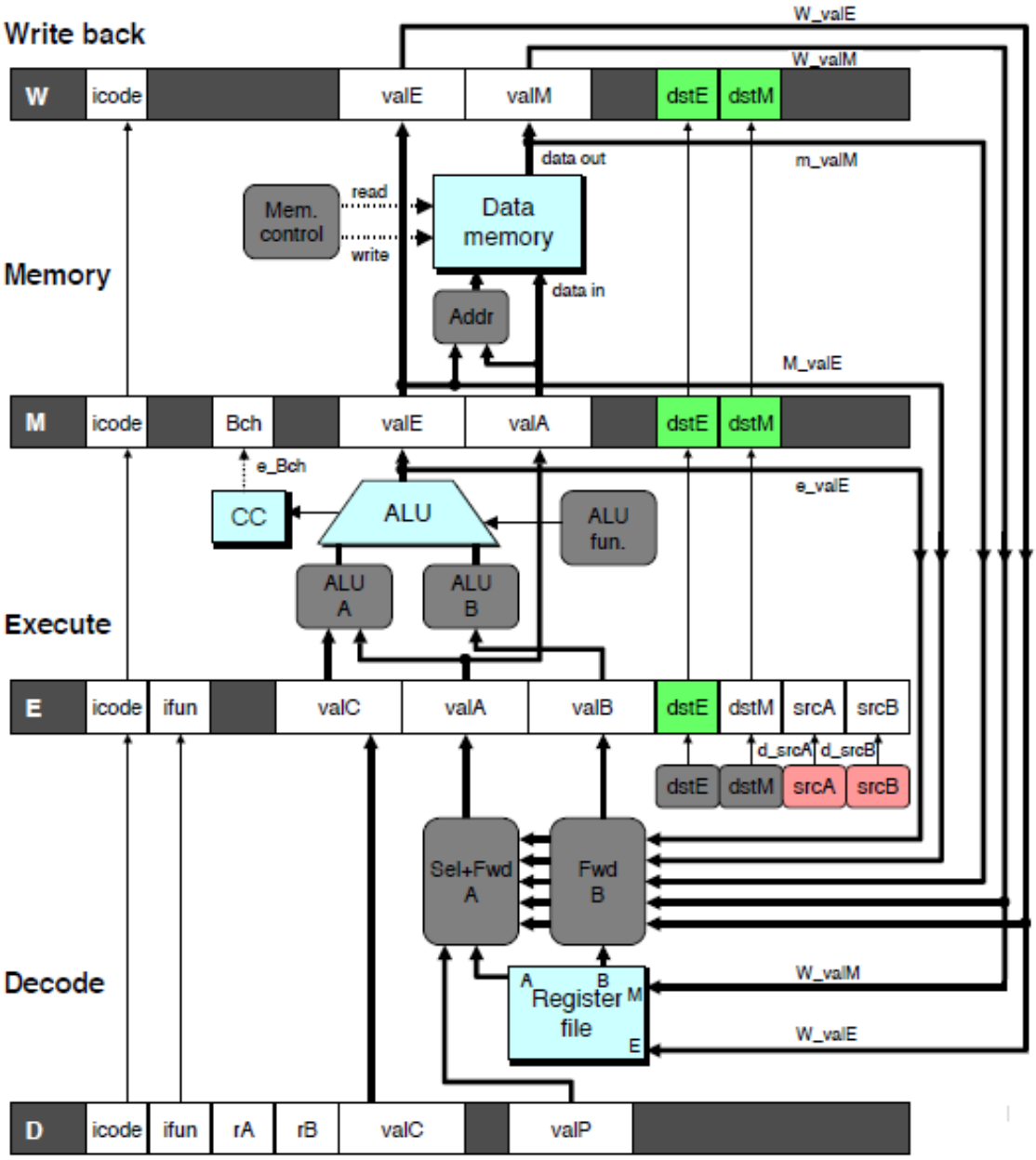
- As dependências de dados são demasiado comuns.
- Resolve-las recorrendo à injeção de "bolhas" resulta no desperdício de um elevado número de ciclos, comprometendo o desempenho do *pipeline.*
- A versão Y86 **PIPE** propõe-se resolver estas dependências de dados, diminuindo o número de bolhas injetadas → logo diminui o número de ciclos desperdiçados.
- As dependências de controlo não sofrem qualquer alteração relativamente ao PIPE-.

Data Forwarding

- **Problema**:
  - Um registo é lido na fase de **D**escodificação.
  - A escrita só ocorre na fase de ***W****riteback*.
- **Observação**:
  - O valor a escrever no registo é gerado no estágio de **E**xecução ou **M**emória.
- **Resolução do problema**:
  - Passar o valor necessário diretamente do estágio onde está disponível (**E**, **M** ou **W**) para o estágio de **D**escodificação.

Y86 PIPE: Resumo

- **Dependências de Dados**
  - Tratadas maioritariamente com ***forwarding***
    Não há penalização no desempenho.
  - ***Load/use*** exige que se empate o *pipeline* durante 1 ciclo.
- **Dependências de Controlo** *(não há alterações relativamente a PIPE-)*
  - **salto condicional mal previsto**: cancelar instruções no estágio F e D
    - 2 ciclos do relógio desperdiçados.
  - ***ret***: Empatar o estágio F (injetando bolhas em E) até o endereço de retorno ser lido
    - 3 ciclos do relógio desperdiçados.



Exercício 1

```
I1:    irmovl %10, %edx          # wr EDX
I2:    irmovl %20, %ebx          # wr EBX
I3:    irmovl %30, %ecx          # wr ECX
I4:    mrmovl $0(%esi), %eax     # rd ESI ; wr EAX
I5:    xorl %eax, %eax           # rd EAX ; wr EAX (coloca Z=1)
I6:    jne I9                    # salta se Z=0 ➔ não salta
I7:    rmmovl %eax, $0(%esi)     # rd EAX, ESI
I8:    halt
I9:    rmmovl %eax, $0(%edi)     # rd EAX, EDI
I10:   halt
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	**F**	**D**	**E**	**M**	**W** edx										
I2		**F**	**D**	**E**	**M**	**W** ebx									
I3			**F**	**D**	**E**	**M**	**W** ecx								
I4				**F**	**D** esi	**E**	**M**	**W** eax							
~~I5~~ / bolha					F	D eax	E	M	W						
I5					F	D	**D** eax	**E**	**M**	**W** eax					
I6						**F**	**F**	**D**	**E** (B)	**M** (C)	**W**				
~~I9~~ / bolha								**F**	**D** eax,edi	E	M	W			
~~I10~~/bolha									**F**	D	E	M	W		
I7										**F**	**D** eax,esi	**E**	**M**	**W**	
I8											**F**	**D**	**E**	**M**	**W**

Justificação
(2) Dependência WAW em %EAX entre I4 e I5 → não causa problema.
(3) Ciclo 6 - Dependência RAW em %EAX entre I4 e I5 → I5 precisa %EAX no ciclo 6 mas I4 só tem esse valor no ciclo 7 → 1 bolha.
(4) Ciclo 7 - Encaminha o valor de %EAX de **M** para **D** (I4 para I5) → novos **E_valA = E_valB = m_valM**
(5) Ciclo 9 - Dependência RAW em %EAX entre I5 e I9 → encaminha-se o valor de %EAX de **M** (I5) para **D** (I9) → novo **E_valA = M_valE**
(B) Ciclo 9 - Coloca sinal **e_Bnc=1**.
(C) Ciclo 10 - Corrige previsão de salto → anula I9 e I10 e inicia I7 ➔ insere 2 bolhas.
(6) Dependência RAW em %EAX entre I5 e I7 → resolvida pelas bolhas anteriores.

Exercício 2.1

Reordene o programa anterior por forma a minimizar o número de bolhas injetadas no processador.

```
I1:   irmovl %10, %edx          # wr EDX
I2:   irmovl %20, %ebx          # wr EBX
I4:   mrmovl $0(%esi), %eax     # rd ESI    ;  wr EAX
I3:   irmovl %30, %ecx          # wr ECX
I5:   xorl %eax, %eax           # rd EAX    ;  wr EAX
I6:   jne I9                    # não salta
I7:   rmmovl %eax, $0(%esi)     # rd EAX/ESI
I8:   halt
I9:   rmmovl %eax, $0(%edi)     # rd EAX/EDI
I10:  halt
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	F	D	E	M	W **edx**									
I2		F	D	E	M	W **ebx**								
I4			F	D **esi**	E	M	W **eax**							
I3				F	D	E	M	W **ecx**						
I5					F	D **eax**	E	M	W **eax**					
I6						F	D	E **(B)**	M **(C)**	W				
~~I9~~ / bolha							F	D **eax,edi**	E	M	W			
~~I10~~/bolha								F	D	E	M	W		
I7									F	D **eax,esi**	E	M	W	
I8										F	D	E	M	W

Explicações
(A) Troca-se I3 com I4 → resolve-se o problema criado pela dependência RAW entre I4 e I5 → não é preciso "bolha".
(1) Ciclo 6 - Dependência RAW em %EAX entre I4 e I5 → encaminha-se o valor **m_valM** do estágio **M** (I4) para o estágio **D** (I5) → novos **E_valA = E_valB = m_valM**
(2) Ciclo 8 - Dependência RAW em %EAX entre I5 e I9 → encaminha-se o valor do registo **M** (I5) para o estágio **D** (I9) → novo **E_valA = M_valM**
(B) Ciclo 8 - Coloca sinal **e_Bnc=1**.
(C) Ciclo 9 - Corrige previsão de salto → anula I9 e I10 e inicia I7 → insere 2 bolhas.
(3) Ciclo 10 - Dependência RAW em %EAX entre I5 e I7 →resolvida pelas bolhas anteriores.

Exercício 2.2

- Qual o tamanho em *bytes* deste programa?
- Qual o valor do PC no ciclo 7, se o programa anterior estiver armazenado em memória a partir do endereço 0x0050?

Resposta
Tamanho = 6 + 6 + 6 + 6 + 2 + **5** + 6 + 1 + **6** + 1 = **45 bytes** (I1 + I2 + I3 + I4 + I5 + **I6** + I7 + I8 + **I9** + I10) No **ciclo 7** o PC aponta para a instrução **I9**. O endereço da instrução é **0x50** + 6 + 6 + 6 + 6 + 2 + **5** + 6 + 1 = 80 + 38 = 118 = **0x76**

Exercício 3

Usando o código apresentado abaixo, identifique para cada ciclo do relógio a ocupação de cada estágio do processador, para a versão PIPE do Y86 – versão com atalhos. Justifique devidamente a sua resposta, indicando quais os valores encaminhados.

		Reg. Lidos	Reg. Escritos
I1:	pushl %eax	# EAX, (ESP)	(ESP)
I2:	popl %ebx	# (ESP)	EBX, (ESP)
I3:	irmovl %20, %eax	#	EAX
I4:	addl %ebx, %eax	# EBX,EAX	EAX
I5:	call I7	# (ESP)	(ESP)
I6:	halt		
I7:	subl %esi, %eax	# ESI,EAX	EAX
I8:	ret	# (ESP)	(ESP)
I9:	…		

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I1	F	D **eax,esp**	E	M	W **esp**										
I2		F	D **esp**	E	M	W **ebx,esp**									
I3			F	D	E	M	W **eax**								
I4				F	DD **ebx,eax**	E	M	W **eax**							
I5					F	D **esp**	E	M	W **esp**						
I7 **(5)**						F	D **esi,eax**	E	M	W **eax**					
I8							F	D **esp**	E	**M**	W **esp**				
Bolha 1 **(8)**								F							
Bolha 2 **(8)**									F						
Bolha 3 **(8)**										F					
I6								stall	stall	stall	**F**	D	E	M	W

Justificação

(1) Ciclo 3 – Dependência (**RAW**) em %**esp** entre I1 e I2 → encaminhar o valor **e_valE** do estágio **E** (I1) para o estágio **D** (I2) → novos **E_valA = E_valB = e_valE**

(2) Ciclo 5 - Dependência em %**ebx** entre I2 e I4 → encaminhar o valor do estágio **M**(I2) para **D**(I4) → novo **E_valA = m_valM**

(3) Ciclo 5 - Dependência em %**eax** entre I3 e I4 → encaminhar o valor do estágio **E** (I3) para **D** (I4) → novo **E_valB = e_valE**

(4) Ciclo 6 - Dependência em %**esp** entre I2 e I5 → encaminhar o valor do registo **W** (I2) para **D** (I5) → novo **E_valB = W_valM**

(5) Ciclo 6 - No Y86 PIPE- e PIPE, nos JMP e **CALL** o próximo **PC** é o valor **valC** incluído na instrução → depois de I5 executa-se I7.

(6) Ciclo 7 - Dependência em %**eax** entre I4 e I7 → encaminhar o valor do registo **M** (I4) para **D** (I7) → novo **E_valB = M_valE**

(7) Ciclo 8 - Dependência em %**esp** entre I5 e I8 → encaminhar o valor do registo **M** (I5) para **D** (I8) → novos **E_valA = E_valB = M_valE**

(8) ***ret*** obriga a empatar o *pipeline* por 3 ciclos, injetando bolhas até o endereço de retorno ser conhecido (até à fase **M**⇔ciclo **10**).

Exercício 4

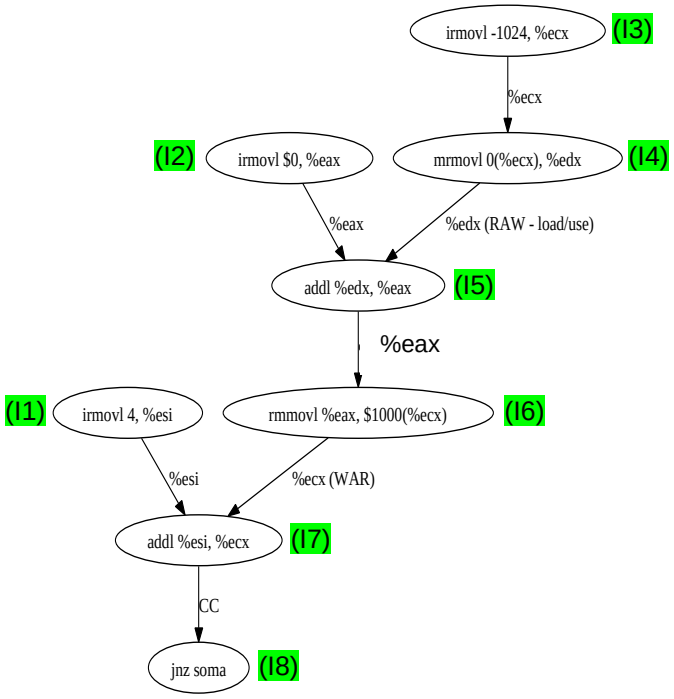
Exercício 4.1

Considerando o seguinte programa escrito em Y86, identifique as dependências de dados existentes neste programa. Será possível reordenar as operações por forma a diminuir o número de bolhas injetadas pelo processador?

Programa

```
I1:          irmovl 4, %esi
I2:          irmovl $0, %eax
I3:          irmovl -1024, %ecx
I4:    soma: mrmovl 0(%ecx), %edx
I5:          addl %edx, %eax
I6:          rmmovl %eax, $1000(%ecx)
I7:          addl %esi, %ecx
I8:          jnz soma
```

Grafo de dependências



As dependências de dados limitam a ordem de execução das instruções. Quando uma instrução depende de uma anterior essa ordem deve ser mantida. Convém notar que todas as dependências são do tipo **RAW** (salientando o caso **load/use**), exceto a dependência entre **rmmovl** (**I6**) e **addl** (**I7**) que é **WAR** (logo uma falsa dependência).

Observando o grafo de dependências nota-se que não existem grandes opções para alterar o escalonamento, devido às dependências.

Exercício 4.2

Apresente o escalonamento do programa anterior numa arquitetura superescalar de duas vias, com a capacidade de executar uma operação **aritmética/salto** simultaneamente com um **acesso à memória**. Qual o CPI obtido na execução do programa.

Nota: faça o escalonamento apenas para as instruções executadas no ciclo ⇔ I4 a I8.

Ciclo	Aritmética/salto	Acesso à memória
1		mrmovl 0(%ecx), %edx (I4)
2	addl %edx, %eax (I5)	
3		rmmovl %eax, $1000(%ecx) (I6)
4	addl %esi, %ecx (I7)	
5	jnz soma (I8)	

O corpo do ciclo requer 5 ciclos devido às dependências de dados, logo **CPI** ≅ **1**.

Nota: é mais correto considerar que o corpo do ciclo requer 6 ciclos, inserindo uma bolha entre **mrmovl** (**I4**) e **addl** (**I5**), originada pelo ***load/use***, mas isso complica mais o exercício.

Neste exercício é importante notar que o **CPI** está longe do $\mathbf{CPI_{ideal}}$ que seria **0.5**. Na verdade esta arquitetura super-escalar não apresenta melhor desempenho que a Y86 PIPE.

Exercício 4.3

Desdobre o corpo do ciclo 2 vezes e efetue novamente esse escalonamento, reordenando as instruções. Considere que pode utilizar um número adicional de registos designados por %r8 ... %r15. Qual o CPI obtido agora na execução do programa.

1º Reescrever o programa com o ciclo desdobrado duas vezes

I1: irmovl 4, %esi

I2: irmovl $0, %eax

I3: irmovl -1024, %ecx

I4: soma: **mrmovl 0(%ecx), %edx**

I5: **addl %edx, %eax**

I6: **rmmovl %eax, $1000(%ecx)**

I7: **addl %esi, %ecx**

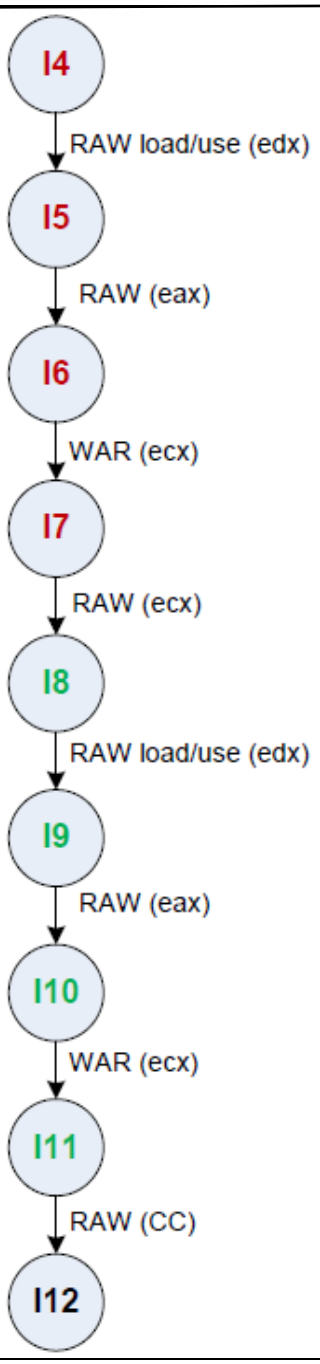
I8: **mrmovl 0(%ecx), %edx**

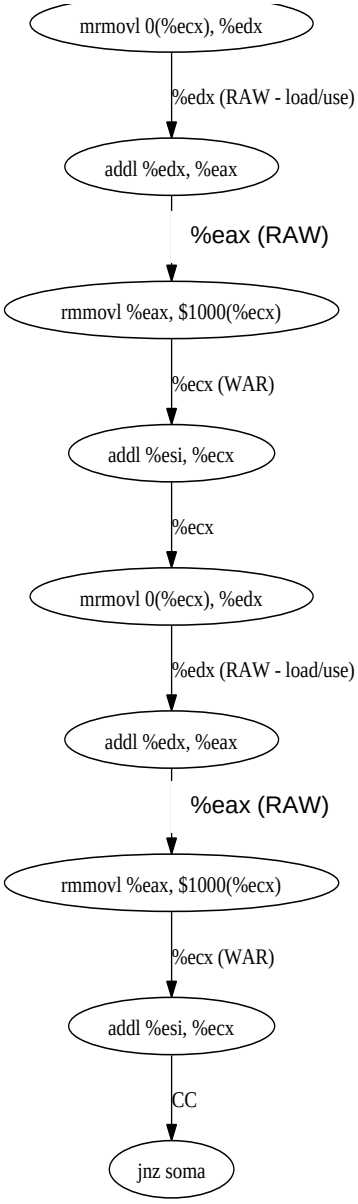
I9: **addl %edx, %eax**

I10: **rmmovl %eax, $1000(%ecx)**

I11: **addl %esi, %ecx**

I12: **jnz soma**





➔ se eliminarmos os 2 WAR's podemos escalonar 3 blocos de instruções em separado (I4-I5-I6 , I7-I8-I9-I10 e I11-I12).

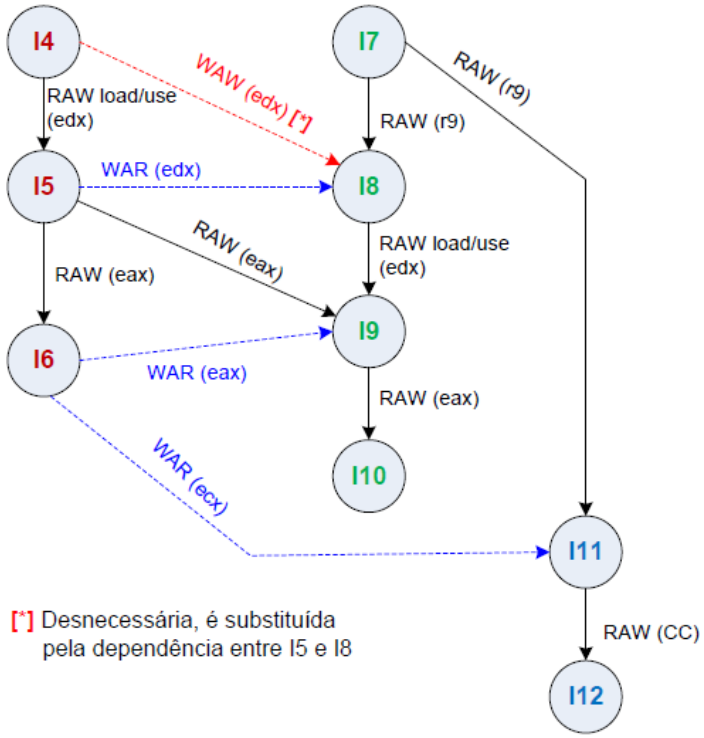
2º Remover as duas dependências WAR (I6→I7 e I10→I11) usando a instrução **addl rs1, rs2, rd** e o registo %r9.

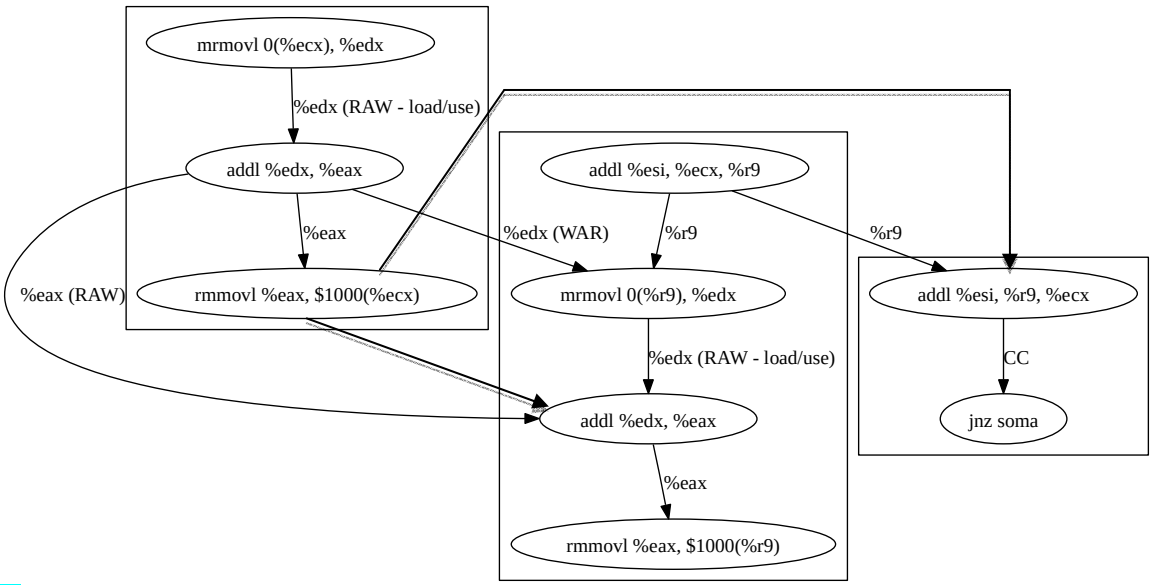
```
I4   soma:  mrmovl 0(%ecx), %edx
I5          addl %edx, %eax
I6          rmmovl %eax, $1000(%ecx)

I7          addl %esi, %ecx, %r9
I8          mrmovl 0(%r9), %edx
I9          addl %edx, %eax
I10         rmmovl %eax, $1000(%r9)

I11         addl %esi, %r9, %ecx
I12         jnz soma
```

3º Desenhar o novo grafo de dependências





4º Novo escalonamento

ciclo	Aritmética/salto	Acesso à memória
1	addl %esi, %ecx, %r9 (I7)	mrmovl 0(%ecx), %edx (I4)
2	addl %edx, %eax (I5)	mrmovl 0(%r9), %edx (I8)
3	addl %edx, %eax (I9/~~I11~~)	rmmovl %eax , $1000(%ecx) (I6)
4	addl %esi, %r9, %ecx (I11)	rmmovl %eax, $1000(%r9) (I10)
5	jnz soma (I12)	

WAR entre I5 e I8 OK

WAR entre I6 e I9 OK

O corpo do ciclo requer **5** ciclos mas executa **9** instruções, logo **CPI** = 5/9 = **0.55** (próx. do $\mathbf{CPI_{ideal}}$ = **0.5**).

5º Escalonamento removendo a dependência WAR entre **I5→I8** em **%edx**, com **%r10**.

ciclo	Aritmética/salto		Acesso à memória	
1	addl %esi, %ecx, %r9	(I7)	mrmovl 0(%ecx), **%edx**	(I4)
2	addl **%edx**, %eax	(I5)	mrmovl 0(%r9), **%r10**	(I8)
3	addl **%r10**, %eax	(I9)	rmmovl %eax , $1000(%ecx)	(I6)
4	addl %esi, %r9, %ecx	(I11)	rmmovl %eax, $1000(%r9)	(I10)
5	jnz soma	(I12)		

➔ contudo esta alteração não altera o escalonamento anterior.