Instruction Level Parallelism
Super Escalaridade

Arquitetura de Computadores

Licenciatura em Engenharia Informática

Luís Paulo Santos

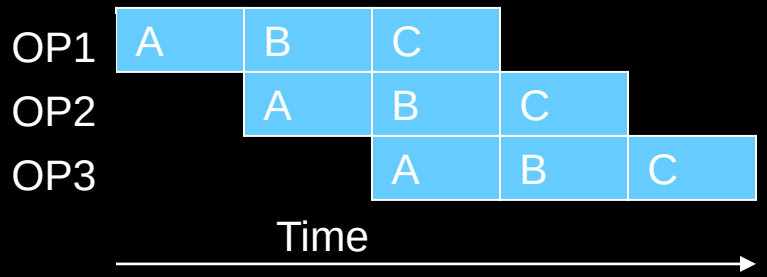
Material de Apoio

- "*Computer Organization and Design: The Hardware / Software Interface*"
  David A. Patterson, John L. Hennessy; 5th Edition, 2013
  - Secção 4.10 (pags. 332 .. 344) – Parallelism via instructions
  - Secção 4.11 (pags. 344 .. 351) – Real Stuff

- "*Computer Systems: a Programmer's Perspective*"; Randal E. Bryant, David R. O'Hallaron--Pearson (2nd ed., 2011)
  - Secção 5.7 (pags. 496 .. 500) – Understanding modern processors
  - ADVANCED MATERIAL:
    Secção 5.7.2 .. 5.9 (pags. 523 .. 446) – Understanding modern processors

Pipelining

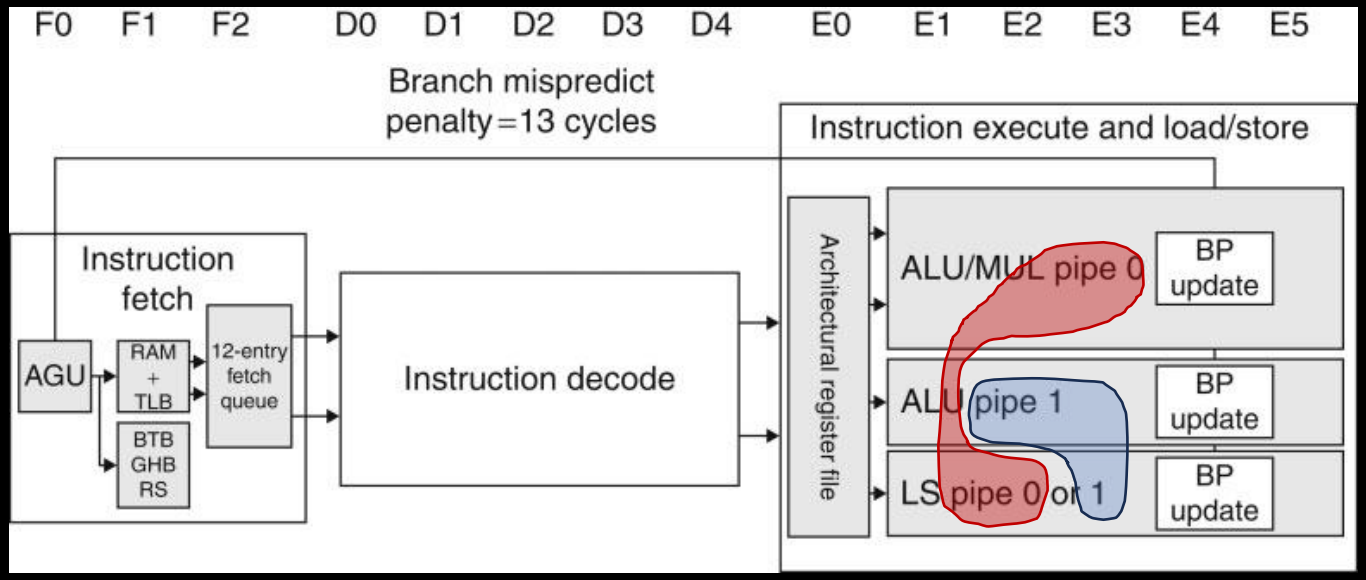
- Explora ILP (*Instruction Level Parallelism*) por permitir a execução simultânea de múltiplas instruções em diferentes estágios



- Permite aumentar a frequência, relativamente a organizações de ciclo único
- O CPI ideal é 1, mas difícil de manter devido a
  - dependências de dados;
  - dependências de controlo;
  - atrasos nos acessos à memória

Multiple Issue

- Uma abordagem complementar consiste em ter múltiplas unidades funcionais, permitindo a execução simultânea de múltiplas instruções -> *multiple issue*



2 unidades funcionais (pipes)

[FIGURE 4.75 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013 --- The A8 pipeline]

Multiple Issue

- As múltiplas unidades funcionais permitem o lançamento (*issue*) de mais do que uma instrução por ciclo, logo, potencialmente podemos ter CPI < 1

- Os processadores actuais mais avançados tentam lançar (*issue*) 3 a 6 instruções por ciclo

Multiple Issue: Limitações

- Existem no entanto limitações ao número de instruções que podem executar em simultâneo:
  - dependências de dados e/ou controlo
  - disponibilidade de recursos: número e tipo de unidades funcionais, latência de diferentes instruções, atrasos no acesso à memória

- Quem determina quais as instruções que podem ser lançadas simultaneamente?

static issue	***dynamic issue***
compilador	**processador**

Static in-order Multiple Issue

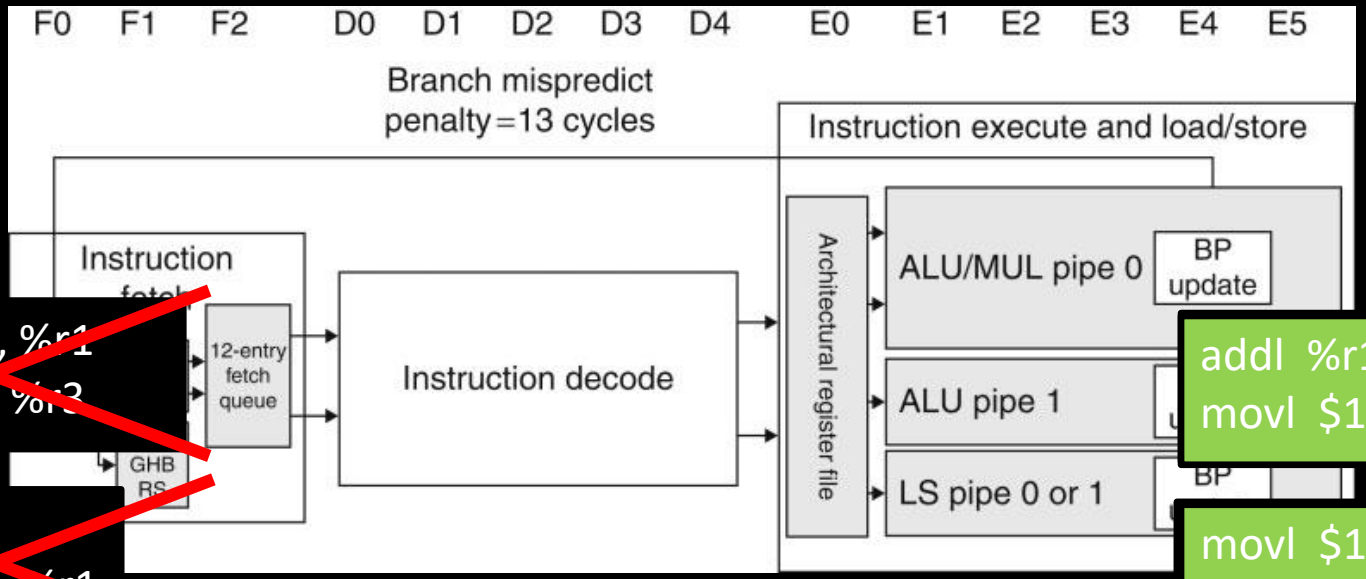
- O **compilador** agrupa em pacotes as instruções que serão lançadas (*issued*) em cada ciclo do relógio

- Estes **pacotes de instruções** concatenadas podem ser vistos como uma única instrução com múltiplas operações:
  VLIW *– Very Long Instruction Word*

- Uma VLIW comporta tantas instruções quantos os *pipelines* (unidades funcionais) disponíveis

- O compilador é responsável por:
  - garantir que não há dependências dentro de uma VLIW
  - que as instruções integradas numa VLIW podem ser executadas pelas unidades funcionais
  - resolver dependências de controlo ou dependências de dados entre diferentes VLIW

Static in-order Multiple Issue –ARM Cortex A8



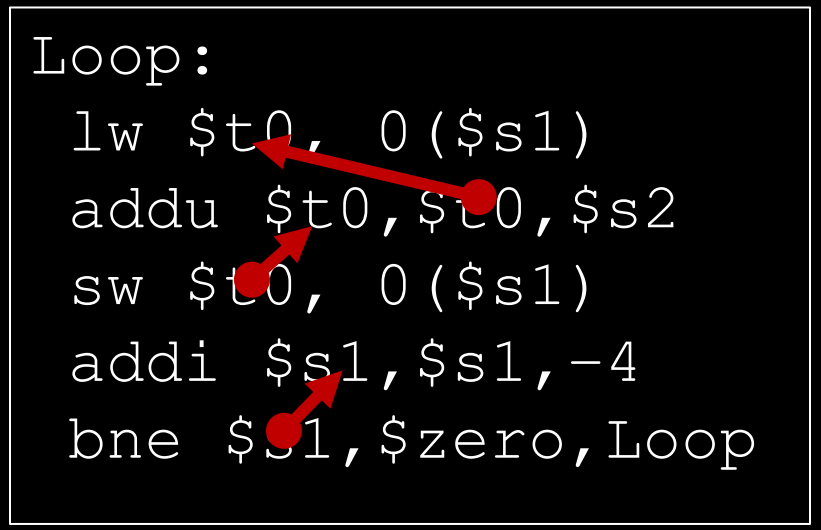
Pipe 0	Pipe 1	Válido ?
movl $10, %r1	movl (%r2), %r3	
movl $10, %r1	addl %r1, %r3	
movl (%r2), %r3	movl (%r4), %r5	
mov $10,%r1	jne label	

Dependências: dados e controlo

- Podem inviabilizar a execução simultânea (ou fora de ordem) de múltiplas instruções

- RAW – Read After Write

A instrução I_j com dependência RAW de I_i tem que ser executada num ciclo posterior ao de I_i

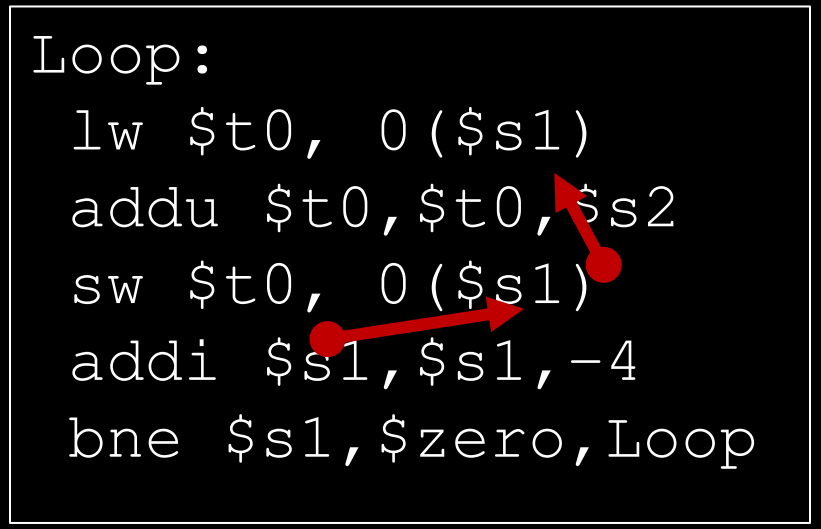


Dependências: dados e controlo

- Podem inviabilizar a execução simultânea (ou fora de ordem) de múltiplas instruções

- WAR –Write After Read

A instrução I_j com dependência WAR de I_i pode ser executada no mesmo ciclo de I_i , mas não antes de I_i

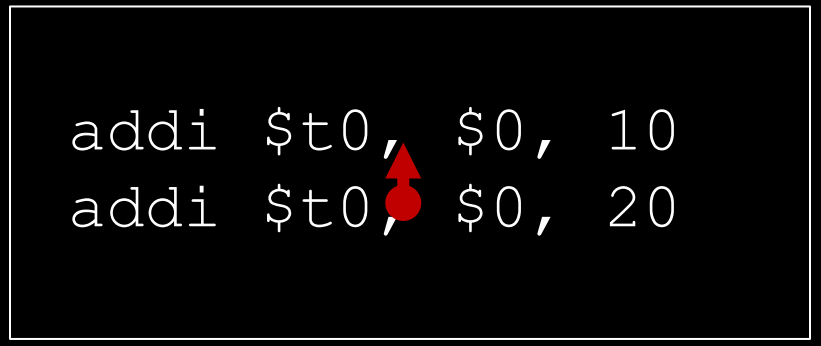


Dependências: dados e controlo

- Podem inviabilizar a execução simultânea (ou fora de ordem) de múltiplas instruções

- WAW –Write After Write

A instrução I_j com dependência WAW de I_i tem que ser executada num ciclo posterior ao de I_i

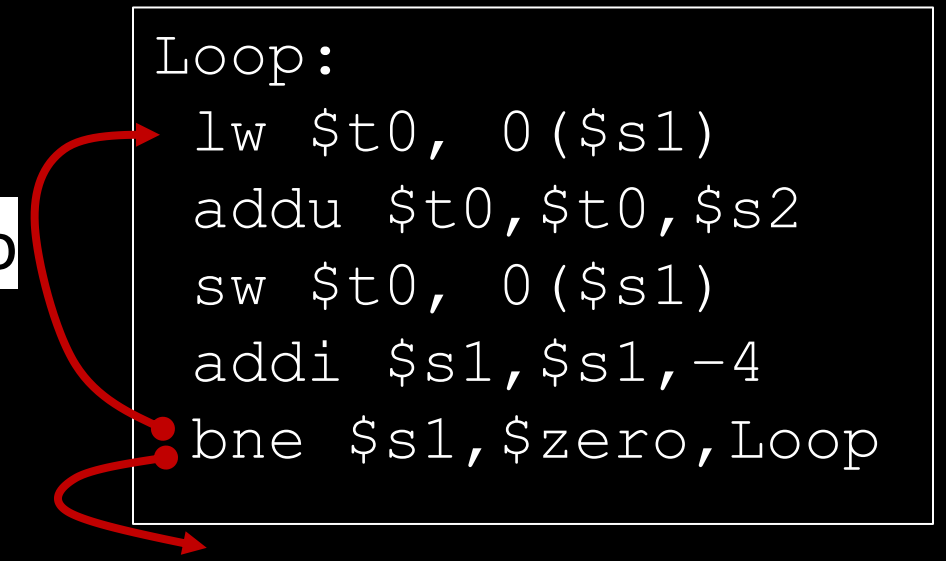


Dependências: dados e controlo

- Podem inviabilizar a execução simultânea (ou fora de ordem) de múltiplas instruções

- dependência de controlo

Instruções após um salto condicional só podem ser executadas no mesmo ciclo (ou antes) do salto se o processador suportar execução especulativa



Grafo de dependências

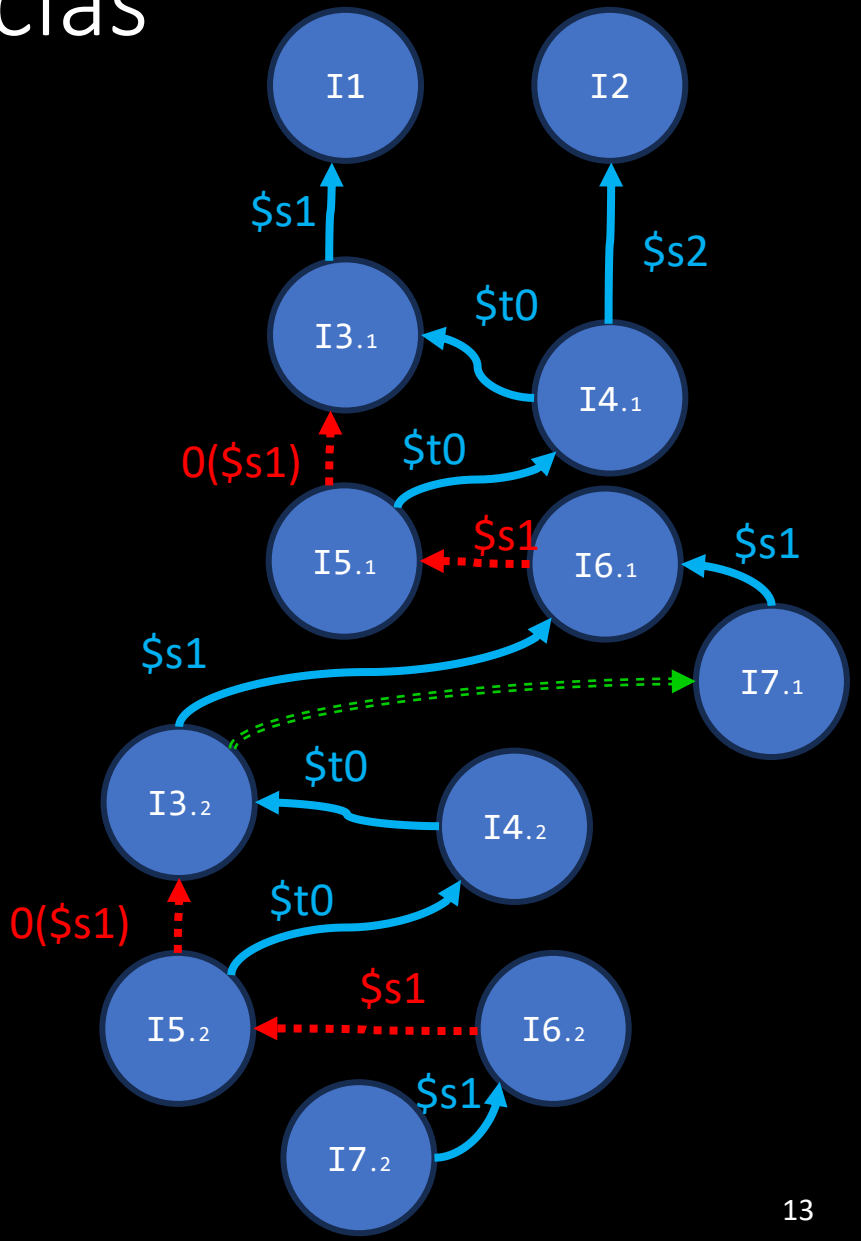
```
I1: addi $s1, $zero, 8
I2: addi $s2, $zero, 100
Loop:
I3: lw $t0, 0($s1)
I4: addu $t0,$t0,$s2
I5: sw $t0, 0($s1)
I6: addi $s1,$s1,-4
I7: bne $s1,$zero,Loop
```

RAW

WAW

WAR

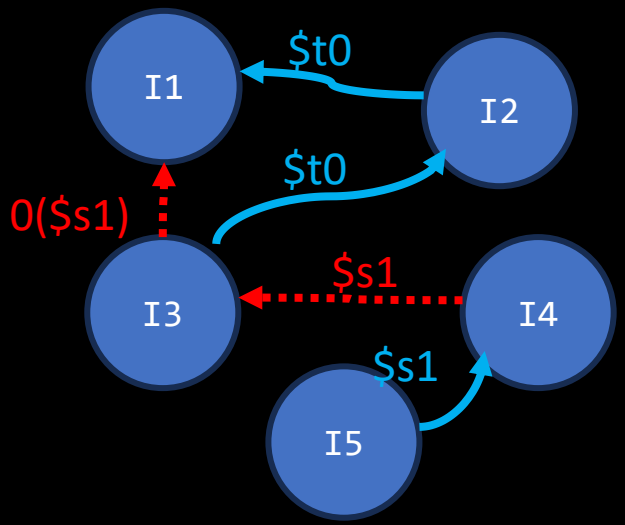
Branch



Static in-order Multiple Issue – MIPS

- Considere uma versão com *static multiple issue* do MIPS, com 2 *pipelines*:
  - Pipe 0 : ALU ou *branch*
  - Pipe 1 : *load / store*

```
Loop:
I1: lw $t0, 0($s1)
I2: addu $t0,$t0,$s2
I3: sw $t0, 0($s1)
I4: addi $s1,$s1,-4
I5: bne $s1,$zero,Loop
```



CC	Pipe 0 (ALU/Branch)	Pipe 1 (LS)
1	nop	lw $t0,0($s1)
2	addu $t0,$t0,$s2	nop
3	addi $s1,$s1,-4	sw $t0,4($s1)
4	bne $s1,$zero,Loop	nop
CPI	4 / 5 = 0.8	

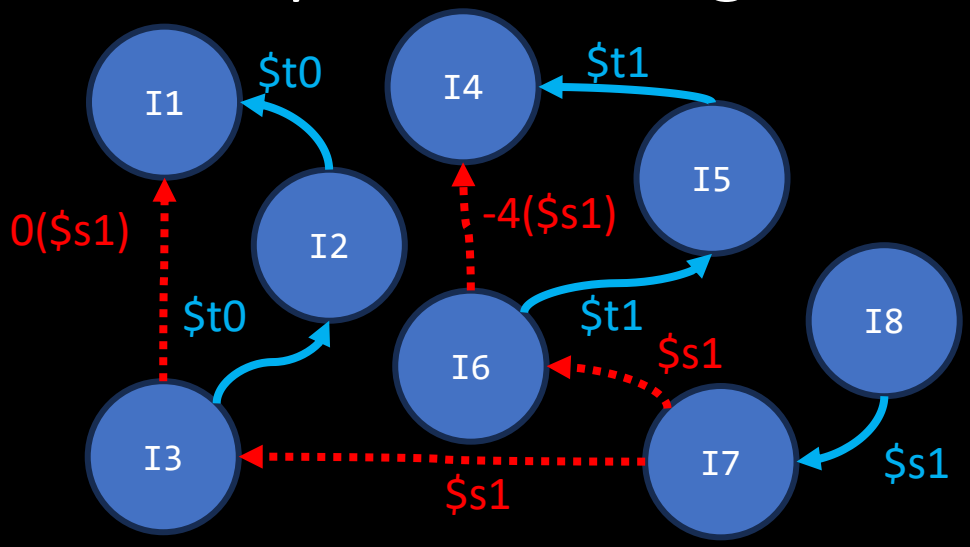
Loop Unrolling– Exemplo: MIPS

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

anti-dependência -> *register renaming*

```
Loop:
 lw $t0, 0($s1)
 addu $t0,$t0,$s2
 sw $t0, 0($s1)
 lw $t1, -4($s1)
 addu $t1,$t1,$s2
 sw $t1, -4($s1)
 addi $s1,$s1,-8
 bne $s1,$zero,Loop
```

Loop Unrolling– Exemplo: MIPS




```
Loop:
I1: lw $t0, 0($s1)
I2: addu $t0,$t0,$s2
I3: sw $t0, 0($s1)
I4: lw $t1, -4($s1)
I5: addu $t1,$t1,$s2
I6: sw $t1, -4($s1)
I7: addi $s1,$s1,-8
I8: bne $s1,$zero,Loop
```

OUT OF ORDER

CC	Pipe 0 (ALU/Branch)	Pipe 1 (LS)
1	nop	lw $t0, 0($s1)
2	addu $t0,$t0,$s2	lw $t1, -4($s1)
3	addu $t1,$t1,$s2	sw $t0, 0($s1)
4	addi $s1,$s1,-8	sw $t1, -4($s1)
5	bne $s1,$zero,Loop	
CPI	5 / 8 = 0.625	

Loop Unrolling

- Consiste em fazer múltiplas cópias do corpo de um ciclo

(normalmente feito pelo compilador)

```
for (i=0 ; i<N ; i++) {
  a[i] = a[i] + var;
}
```

```
for (i=0 ; i<N ; i+=3) {
  a[i] = a[i] + var;
  a[i+1] = a[i+1] + var;
  a[i+2] = a[i+2] + var;
}
```

- O seu efeito no desempenho nem sempre é positivo
- Contribui para:
  - disponibilizar mais instruções independentes para aumentar o ILP, e consequentemente reduzir o CPI
  - reduzir o número de instruções executadas

Static in-order Multiple Issue: Limitação

- O código gerado depende da organização do processador (número, tipo e desempenho das unidades funcionais)

- A mudança para diferentes implementações da mesma arquitectura:
  - frequentemente exige recompilação para que o código execute;
  - noutros casos, exige recompilação para manter um desempenho aceitável

Dynamic Multiple Issue

- O **processador** selecciona dinamicamente, **em tempo de execução**, quais as instruções a executar em cada *pipeline* (unidade funcional)

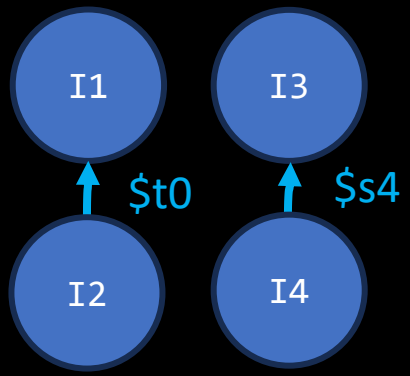
- Diminui a dependência da compilação, relativamente à abordagem estática, pois é o *hardware* que selecciona as instruções de acordo com a sua própria organização

SuperEscalaridade: dynamic *in-order scheduling*

- ***in-order scheduling***: instruções seleccionadas na ordem com que aparecem no programa, baseado na disponibilidade dos operandos e de unidades funcionais apropriadas

```
I1: lw $t0, 20($s2)
I2: addu $t1,$t0,$t2
I3: sub $s4,$s4,$t3
I4: addi $t5, $s4, 20
```

blocos independentes entre si



CC	Pipe 0 (ALU/Branch)	Pipe 1 (LS)
1	`nop`	`lw $t0, 20($s2)`
2	`addu $t1,$t0,$s2`	`nop`
3	`sub $s4,$s4,$t3`	`nop`
4	`addi $t5, $s4, 20`	`nop`
CPI	**4 / 4 = 1**	

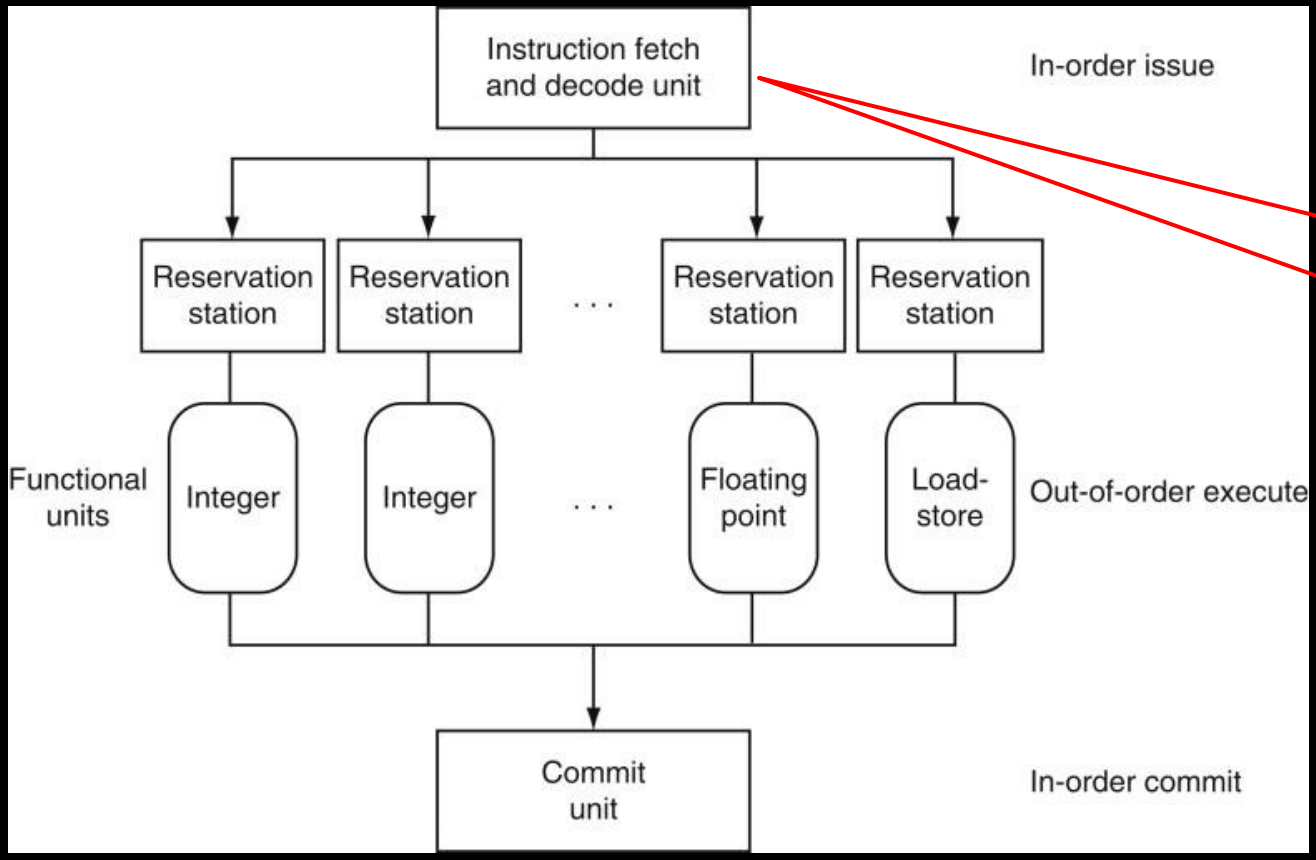
SuperEscalaridade: *dynamic out-of-order scheduling*

- ***out-of-order scheduling***: as instruções são reordenadas pelo processador para minimizar o CPI, garantindo a correcta execução do programa

- ***speculative execution***: as instruções são executadas pelo processador antes de determinar se devem ser executadas. Exemplo: após um salto condicional ainda não avaliado

- ***in-order commit***: os resultados das instruções executadas fora de ordem e/ou especulativamente só são escritos quando é determinado que estão correctos
  Escrita (*commit*) pela ordem do programa sequencial (*in-order*)

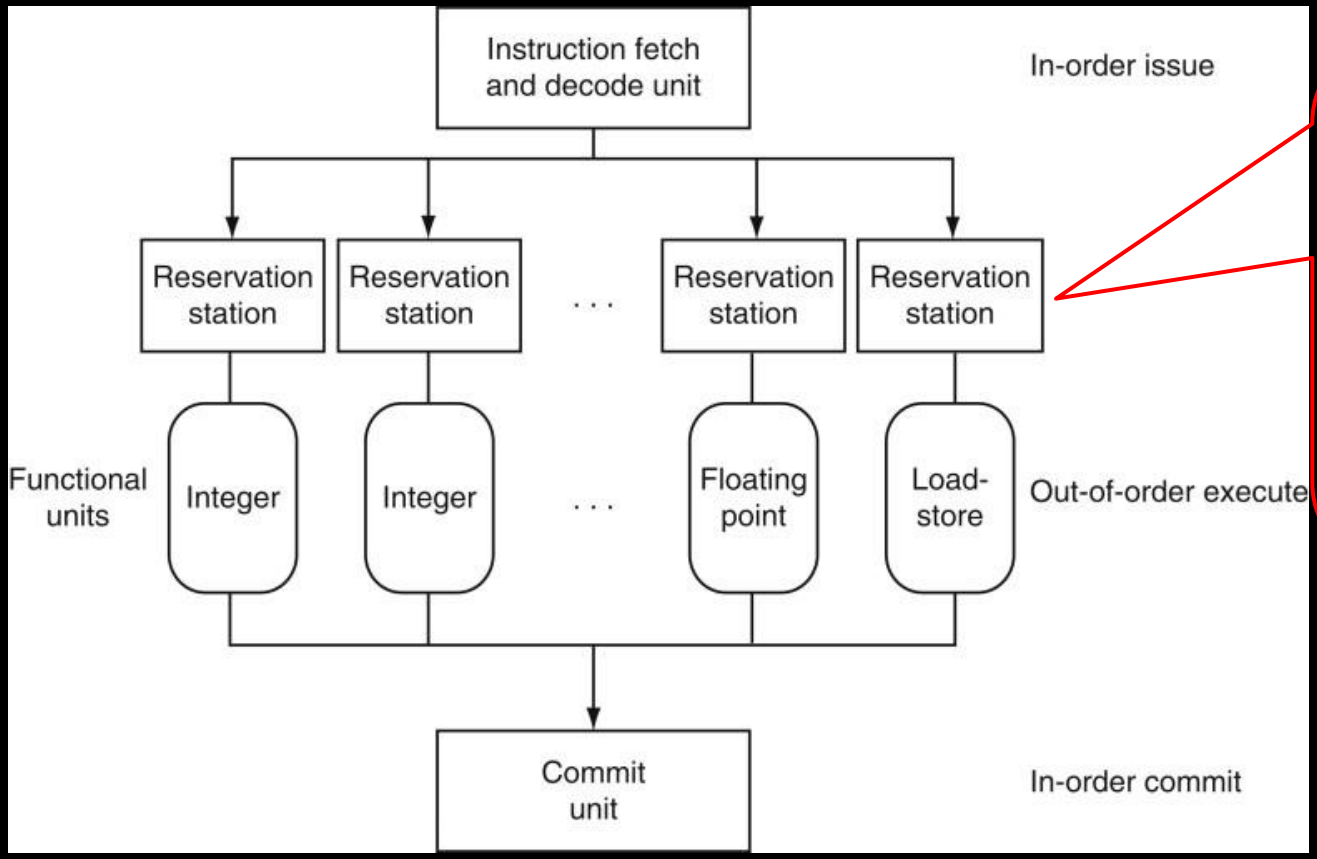
SuperEscalaridade: organização



. *fetch* das instruções
. descodificação
. Envia para a unidade funcional correspondente

[FIGURE 4.72 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013 --- As 3 unidades primárias para *dynamic pipeline scheduling*]

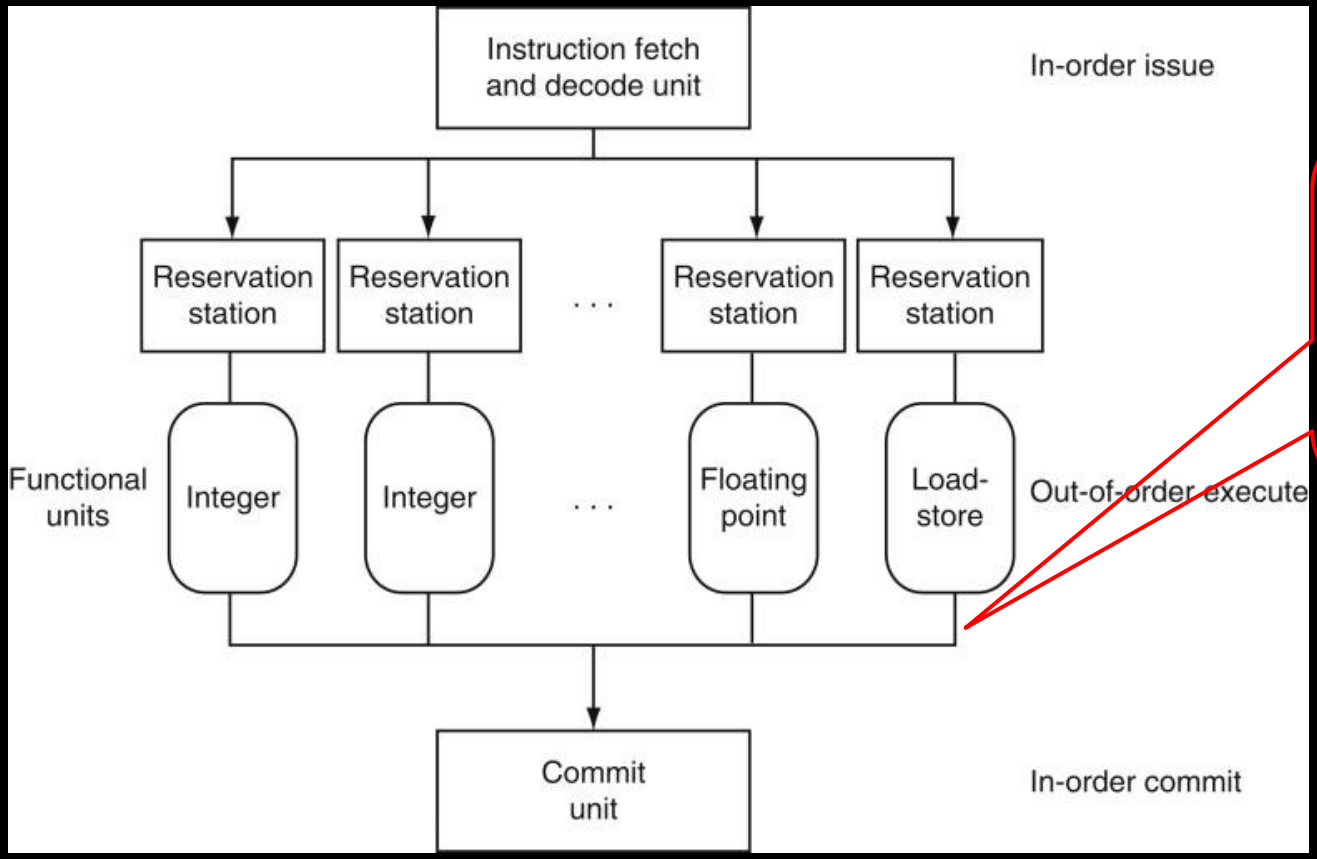
SuperEscalaridade: organização



Buffers que mantêm as operações a realizar e os respectivos operandos. Quando os **operandos de uma operação estão disponíveis** e a **unidade funcional livre**, a operação é escalonada e o resultado calculado.

[FIGURE 4.72 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013 --- As 3 unidades primárias para *dynamic pipeline scheduling*]

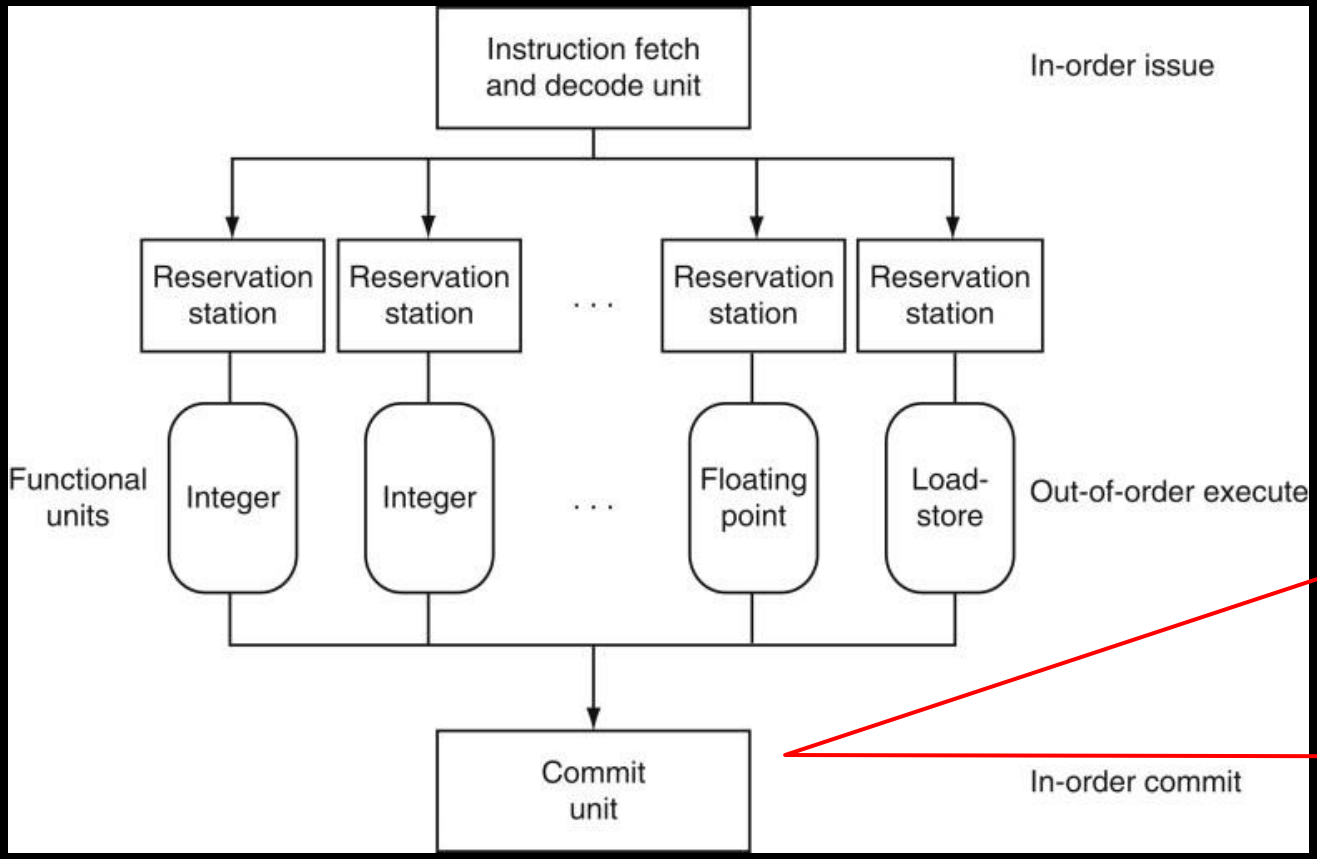
SuperEscalaridade: organização



Assim que um resultado é calculado é enviado para:
. Outras *reservation stations* que possam estar à espera deste valor;
. Para a *commit unit*

[FIGURE 4.72 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013 --- As 3 unidades primárias para *dynamic pipeline scheduling*]

SuperEscalaridade: organização



Armazena resultados no *reorder buffer* até que possam ser escritos no seu destino final (registos/memória), devido à execução especulativa.
O *reorder buffer* é usado para alimentar as *reservation units* com resultados que ainda não tenham sido escritos.

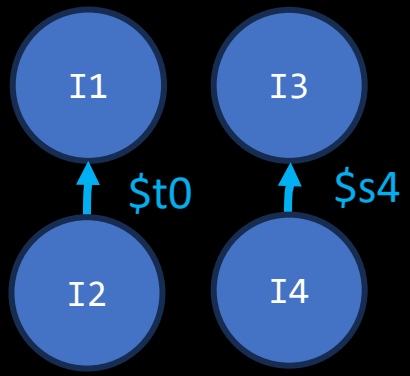
[FIGURE 4.72 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013 --- As 3 unidades primárias para *dynamic pipeline scheduling*]

SuperEscalaridade: *dynamic out-of-order scheduling*

- ***out-of-order scheduling***: as instruções são reordenadas pelo processador para minimizar o CPI, garantindo a correcta execução do programa

```
I1: lw $t0, 20($s2)
I2: addu $t1,$t0,$t2
I3: sub $s4,$s4,$t3
I4: addi $t5, $s4, 20
```

blocos independentes entre si



CC	Pipe 0 (ALU/Branch)	Pipe 1 (LS)
1	`sub $s4,$s4,$t3`	`lw $t0, 20($s2)`
2	`addu $t1,$t0,$s2`	`nop`
3	`addi $t5, $s4, 20`	`nop`
CPI	**3 / 4 = 0.75**	

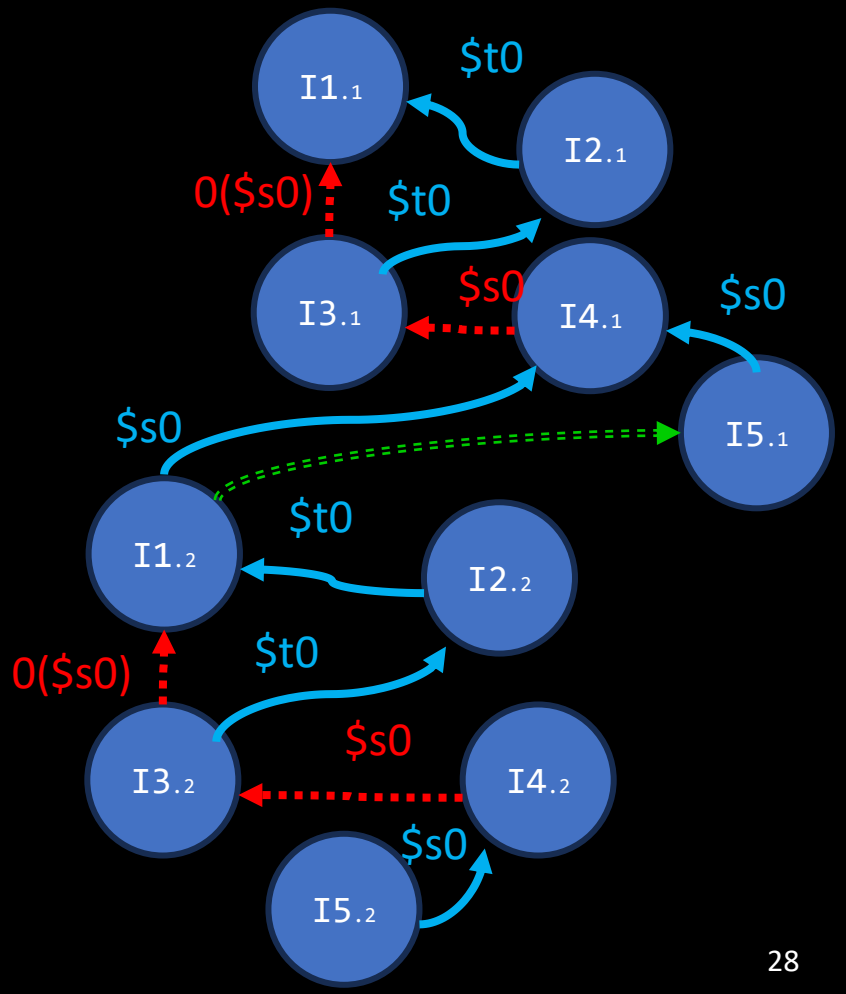
Scheduling - Exercícios

- Implementação do MIPS com 2 unidade funcionais:
  - 0 : operações sobre inteiros e saltos (1 ciclo)
  - 1 : *loads* (2 ciclos) e *stores* (1 ciclo)
    Apesar de 1 *load* tomar 2 ciclos, pode-se iniciar uma instrução a cada ciclo (*pipeline* profundidade 2)
- Previsão de salto tomado
- Para o programa abaixo e para 2 iterações, calcule #I, #cc e CPI, para cada política de escalonamento

```
I1:    lw     %t0, 0(%s0)
I2:    addi   %t0, %t0, $20
I3:    sw     %t0, 0(%s0)
I4:    addi   %s0, %s0, -4
I5:    bnz    I1
```

Grafo de dependências

```
I1: lw $t0, 0($s0)
I2: addi $t0,$t0, 20
I3: sw $t0, 0($s0)
I4: addi $s0,$s0,-4
I5: bne $s0,$zero,I1
```



RAW

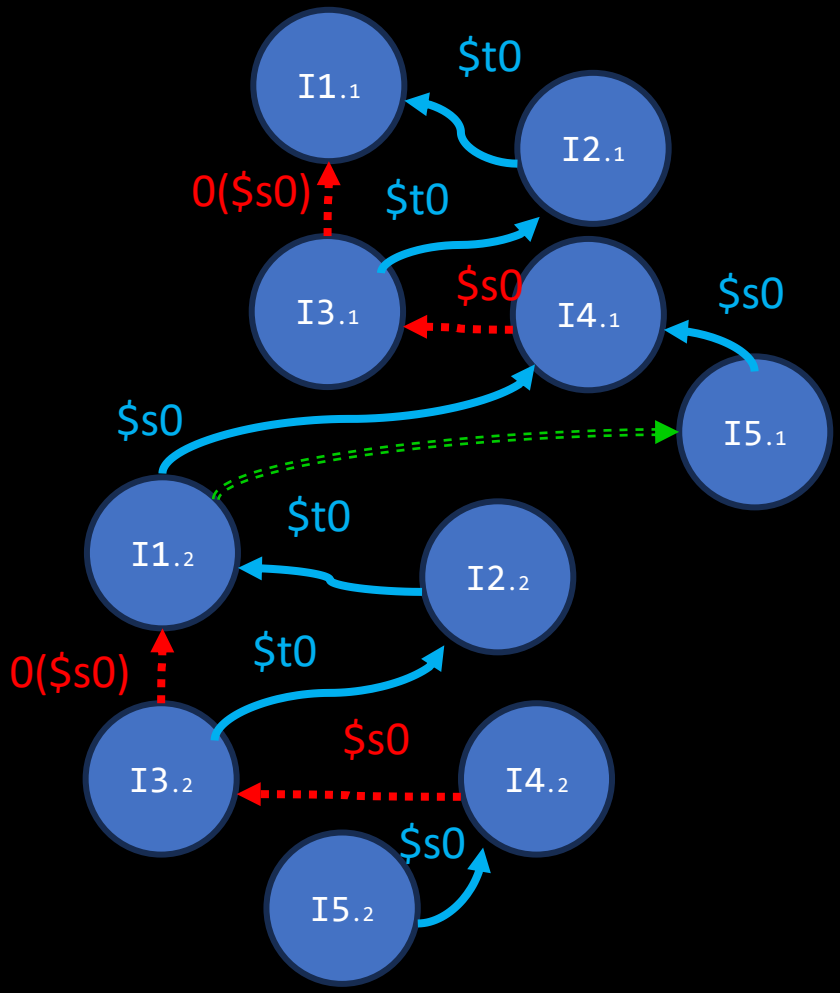
WAW

WAR

Branch

```
I1: lw  %t0, 0(%s0)
I2: addi   %t0, %t0, $20
I3: sw  %t0, 0(%s0)
I4: addi   %s0, %s0, -4
I5: bne $s0,$zero,I1
```

Scheduling: *in-order multiple issue*

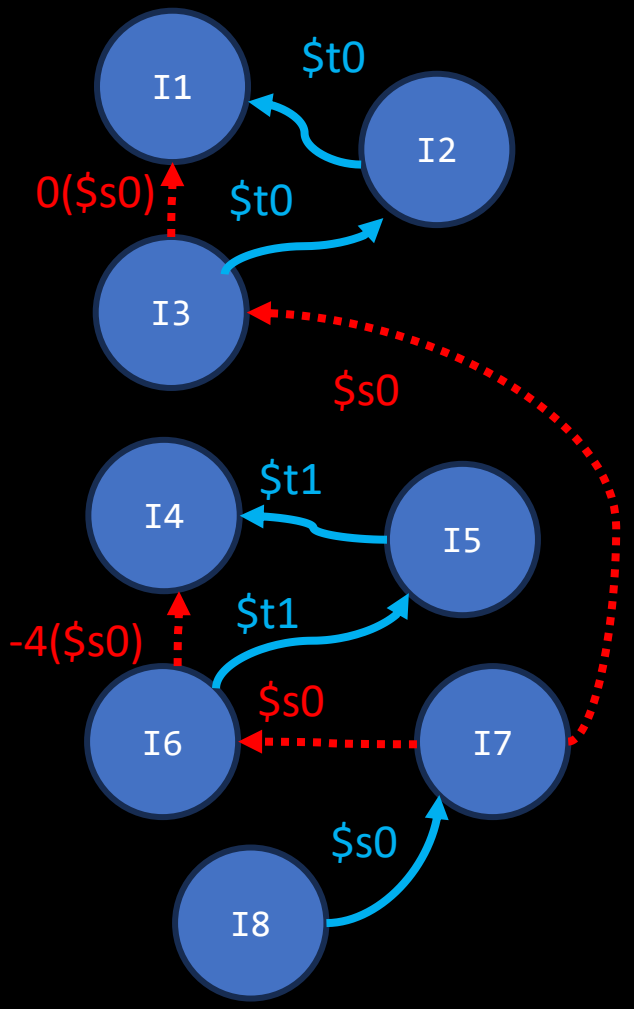


cc	0 (int + b)	1 (load / store)
1	nop	lw %t0, 0(%s0)
2	nop	
3	addi %t0,$t0,20	nop
4	addi %s0,$s0,-4	sw %t0, 0(%s0)
5	bnz I1	lw %t0, 0(%s0)
6	nop	
7	addi %t0,$t0,20	nop
8	addi %s0,$s0,-4	sw %t0, 0(%s0)
9	bne $s0,$zero,I1	
10		

#I = 10
#cc = 9
CPI = 0.9

Scheduling: *in-order multiple issue; unroll = 2*

```
I1: lw  %t0, 0(%s0)
I2: addi   %t0, %t0, $20
I3: sw  %t0, 0(%s0)
I4: lw  %t1, -4(%s0)
I5: addi   %t1, %t1, $20
I6: sw  %t1, -4(%s0)
I7: addi   %s0, %s0, -8
I8: bne $s0,$zero,I1
```



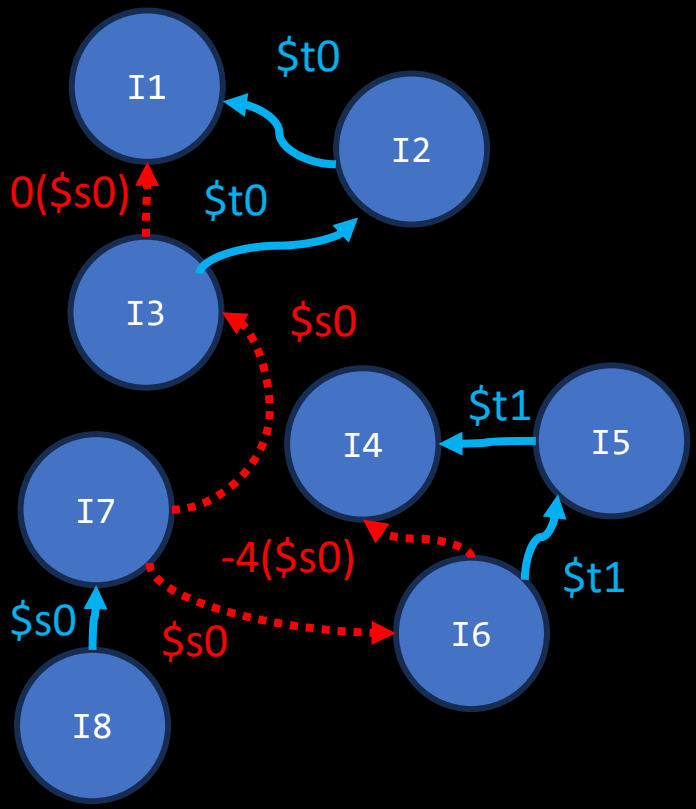
RAW

WAW

WAR

Branch

```
I1: lw  %t0, 0(%s0)
I2: addi   %t0, %t0, $20
I3: sw  %t0, 0(%s0)
I4: lw  %t1, -4(%s0)
I5: addi   %t1, %t1, $20
I6: sw  %t1, -4(%s0)
I7: addi   %s0, %s0, -8
I8: bne $s0,$zero,I1
```

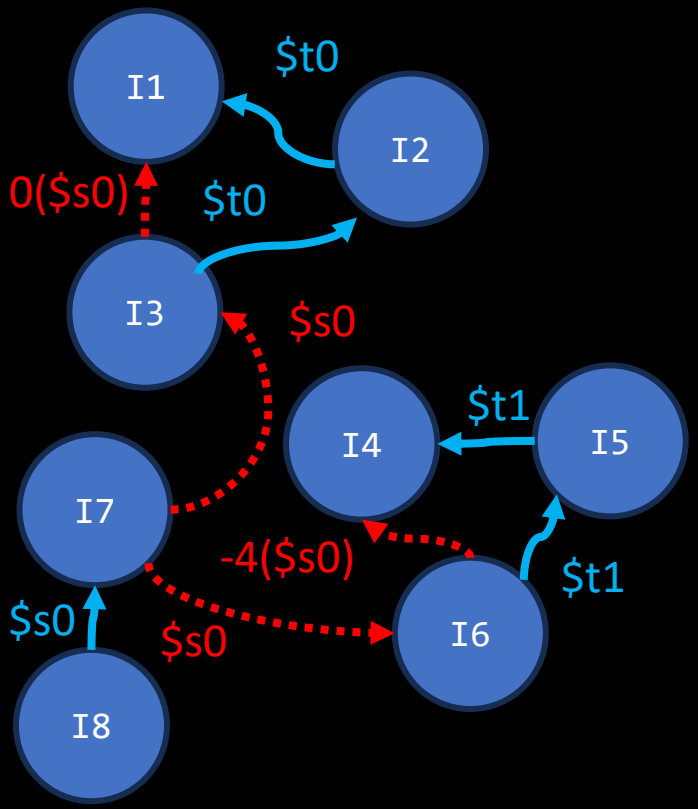


Scheduling: *in-order multiple issue; unroll = 2*

cc	0 (int + b)		1 (load / store)
1	nop		lw %t0, 0(%s0)
2	nop		
3	addi %t0,%t0,20		nop
4	nop		sw %t0, 0(%s0)
5	nop		lw %t1, -4(%s0)
6	nop		
7	addi %t1,%t1,20		nop
8	addi %s0,%s0,-8		sw %t1, -4(%s0)
9	bne $s0,$zero,I1		...
10			

#I = 8
#cc = 9
CPI = 1.125

```
I1: lw  %t0, 0(%s0)
I2: addi   %t0, %t0, $20
I3: sw  %t0, 0(%s0)
I4: lw  %t1, -4(%s0)
I5: addi   %t1, %t1, $20
I6: sw  %t1, -4(%s0)
I7: addi   %s0, %s0, -8
I8: bne $s0,$zero,I1
```



Scheduling: *out of order multiple issue; unroll = 2*

cc	0 (int + b)		1 (load / store)
1	nop		lw %t0, 0(%s0)
2	nop		lw %t1, -4(%s0)
3	addi %t0,%t0,$20		
4	addi %t1,%t1,$20		sw %t0, 0(%s0)
5	addi %s0, %s0, -8		sw %t1, -4(%s0)
6	bne $s0,$zero,I1		...
7			
8			
9			
10			

#I = 8
#cc = 6
CPI = 0.75

VoxVote

- Go to http://live.voxvote.com

- Enter pin number: 127845

ILP: limitações

- *Pipelining* e *multiple issue* são usados para explorar *Instruction Level Parallelism* (ILP)

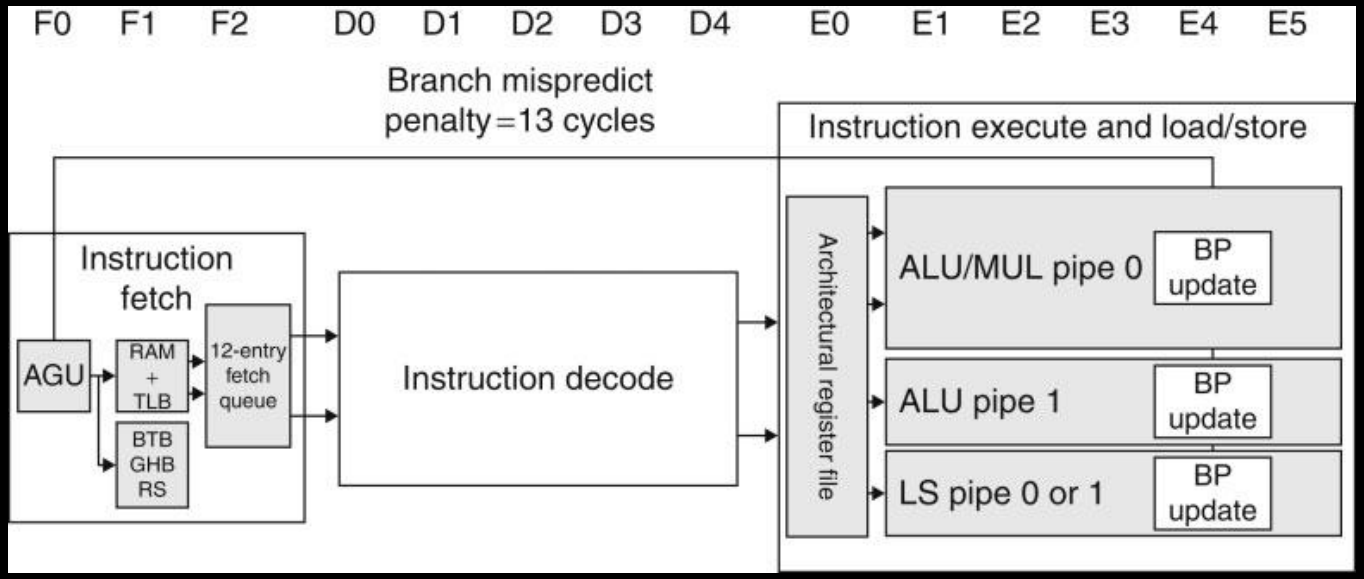
- No entanto, na maior parte das aplicações é difícil lançar mais do que duas instruções por ciclo:
  1. Existem dependências de controlo e dados no código que limitam o paralelismo disponível;
  2. Penalizações devido ao acesso à memória limitam a capacidade de manter os múltiplos *pipelines* constantemente ocupados.

ARM Cortex-A8 vs Intel Core i7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128–1024 KiB	256 KiB
3rd level cache (shared)	–	2–8 MiB

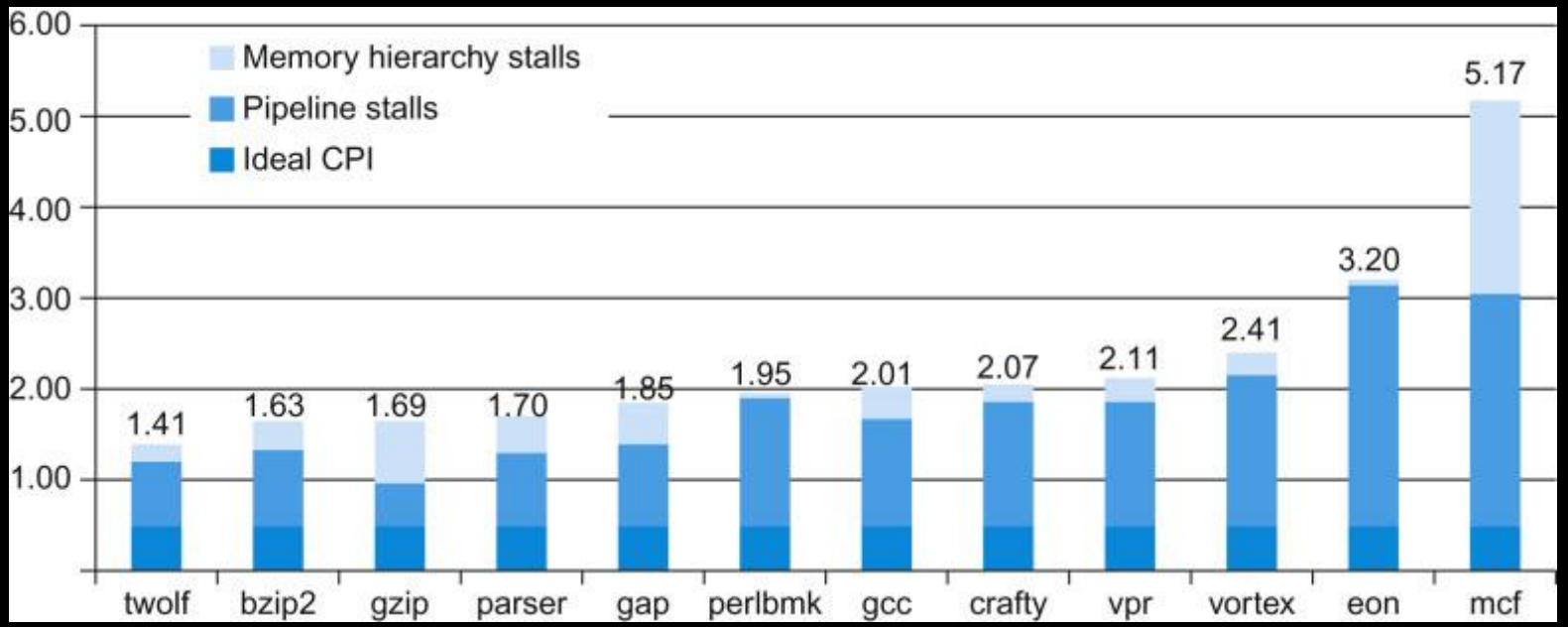
[FIGURE 4.74 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013]

ARM Cortex A8



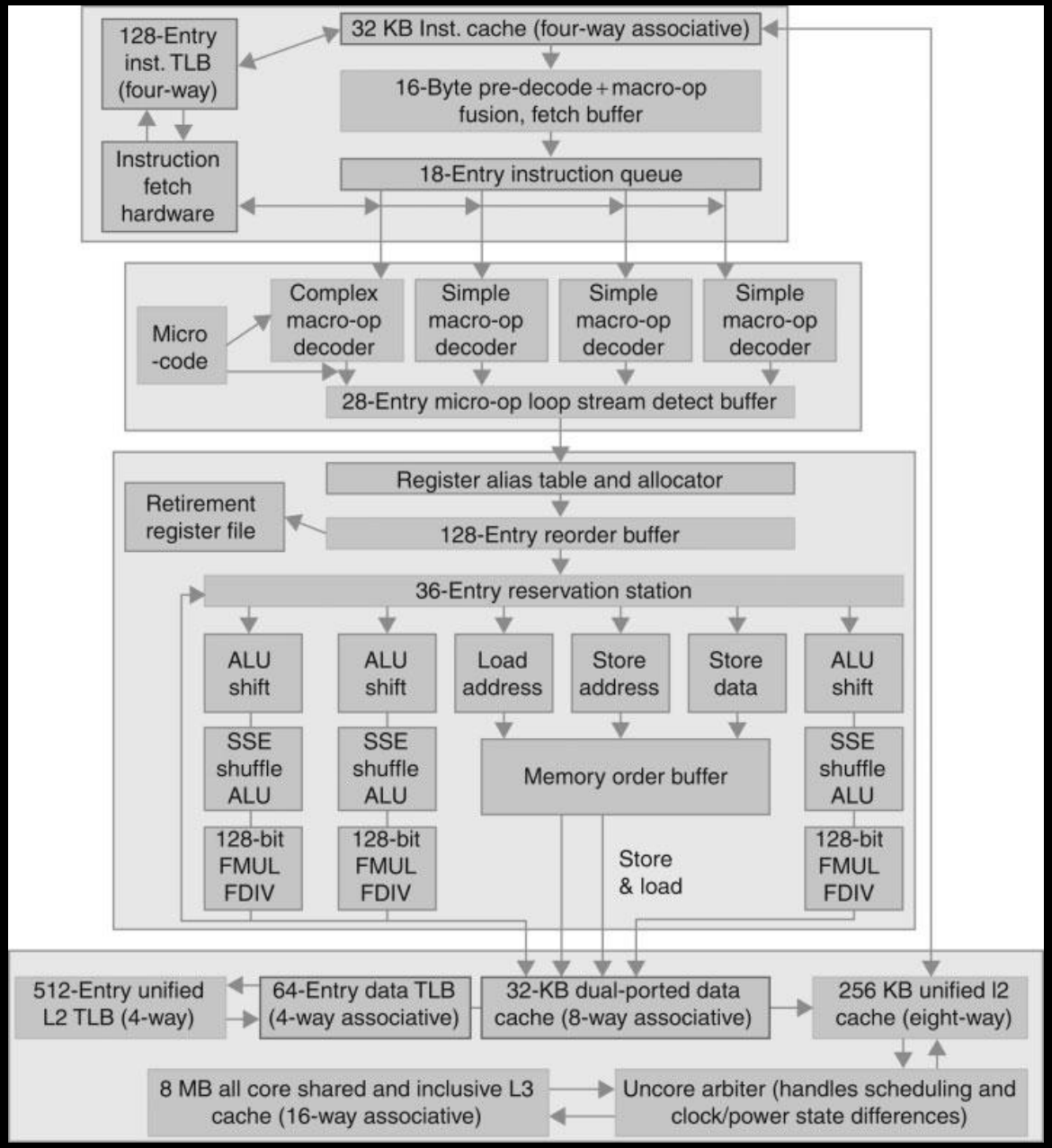
[FIGURE 4.75 – Patterson & Hennessy; Computer Organization & Design – 5th Edition, Elsevier, 2013 --- The A8 pipeline]

ARM Cortex A8 - Desempenho

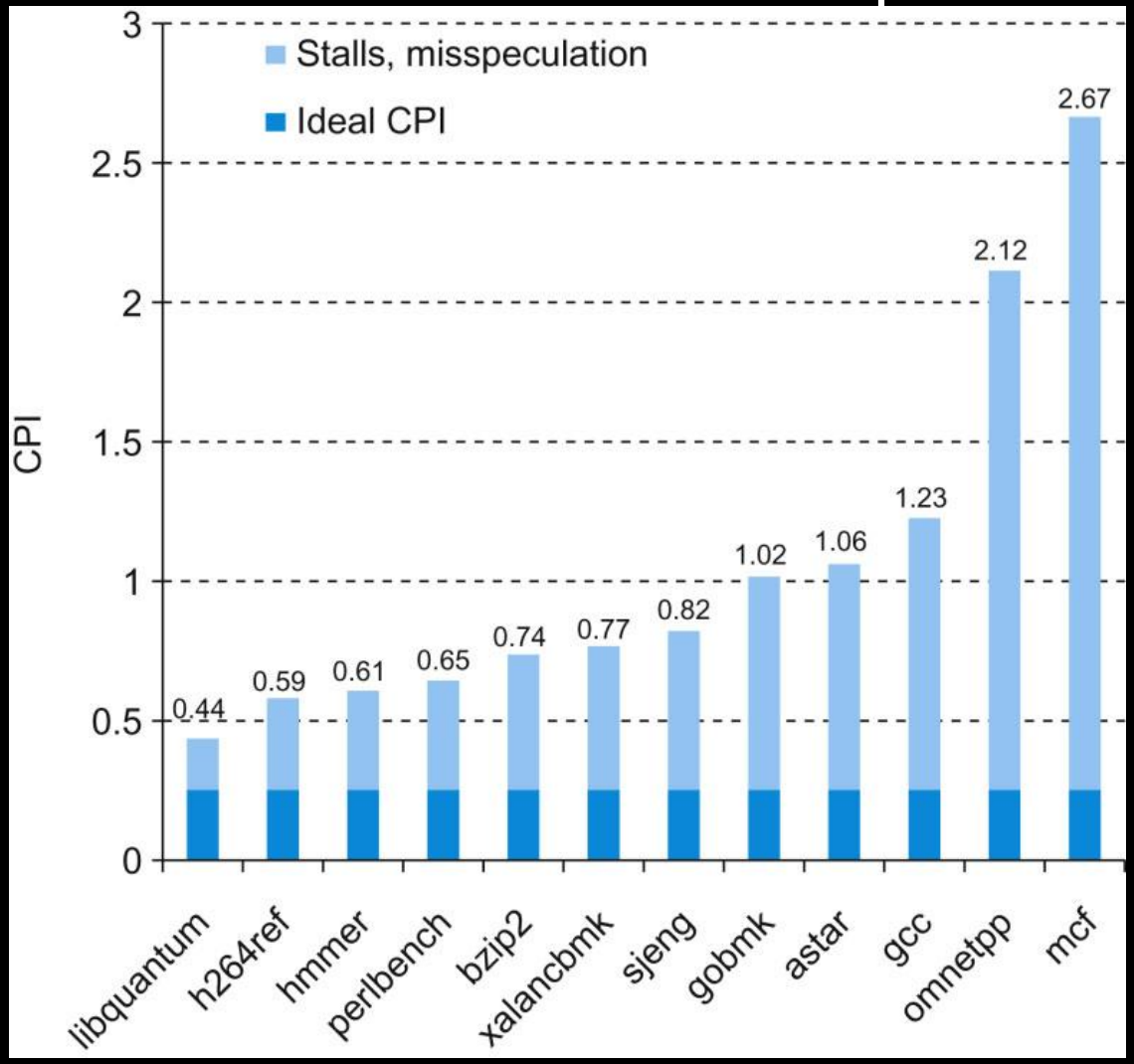


CPI on ARM Cortex A8 for the Minnespec benchmarks, which are small versions of the SPEC2000 benchmarks.

Intel Core i7



Intel Core i7 920 - Desempenho



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.