Arquitectura de Computadores II

LESI - 3º Ano

Pipelining Avançado e Paralelismo ao Nível da Instrução (ILP –*Instruction Level Parallelism*)

João Luís Ferreira Sobral
Departamento do Informática
Universidade do Minho

Abril 2002

Pipelining Avançado e ILP

Variantes do DP MIPS já analisadas

- *Single-cycle* – Todas as instruções são executadas num só ciclo de relógio – Pouco eficiente, nunca utilizada em processadores comerciais.

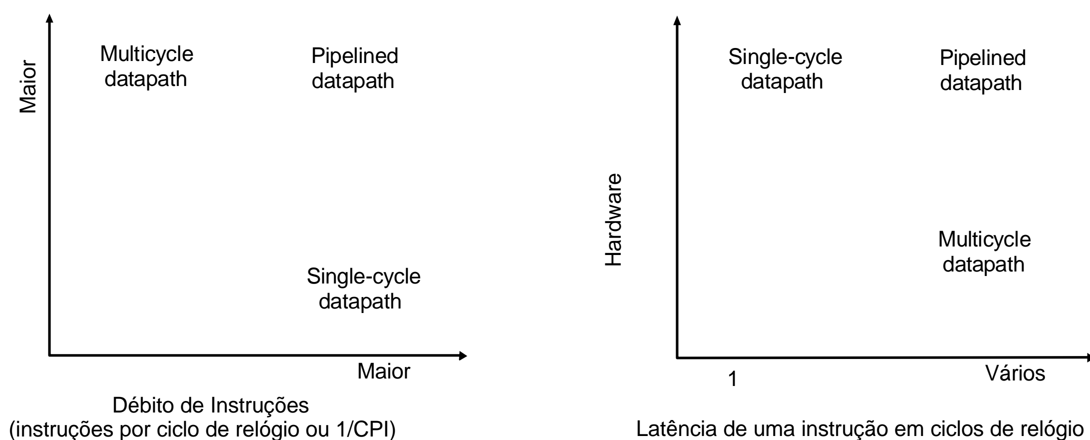
  CPI = 1
  Tcc = duração da instrução mais longa
  Latência de uma instrução = Tcc

- Multi-ciclo – Cada instrução demora vários ciclos a executar.

  CPI = 1 ... número máximo de ciclos numa instrução (depende do Mix)
  Tcc = duração de cada ciclo
  Latência = Tcc x 1 ... CPI*max*

- *Pipeline* – cada fase (estágio) da *pipeline* executa de forma concorrente, como numa linha de montagem, melhorando o CPI relativamente à implementação multi-ciclo

  CPI =1 (em condições ideais)
  Tcc = duração do estágio mais longo da *pipeline*
  Latência = Tcc x número de estágios da *pipeline*



- O desempenho de uma dada arquitectura é dado pela combinação dos vários factores:

  Texec = #I x CPI x Tcc   => Texec decresce com 1/CPI e 1/Tcc

- A latência da execução das instruções é um factor importante, nomeadamente no caso dos saltos, porque influencia o número de ciclos que é necessário empatar a *pipeline*.

Pipelining Avançado e ILP

Aspectos do desenho

- *Pipelining* – Execução sobreposta das instruções

IF	ID	EXE	MEM	WB			
	IF	ID	EXE	MEM	WB		
		IF	ID	EXE	MEM	WB	
			IF	ID	EXE	MEM	WB

- *Super-pipelining* – Aumentar o número de estágios para possibilitar um aumento da frequência interna de relógio

I1	I2	D1	D2	E2	E2	M1	M2	W1	W2			
	I1	I2	D1	D2	E1	E2	M1	M2	W1	W2		
		I1	I2	D1	D2	E1	E2	M1	M2	W1	W2	
			I1	I2	D1	D2	E1	E2	M1	M2	W1	W2

- Super-escalar – Iniciar várias instruções por ciclo de relógio

IF	ID	EXE	MEM	WB	
IF	ID	EXE	MEM	WB	
	IF	ID	EXE	MEM	WB
	IF	ID	EXE	MEM	WB

- VLIW (“EPIC”) – Especificar várias operações paralelas por instrução

IF	ID	EXE	MEM	WB
		EXE	MEM	WB
		EXE	MEM	WB
		EXE	MEM	WB

- Instruções vectoriais - especificar uma série de operações a realizar em vários dados, numa só instrução

IF	ID	EXE	MEM	WB			
			EXE	MEM	WB		
				EXE	MEM	WB	
					EXE	MEM	WB

 © João Luís Sobral 2002

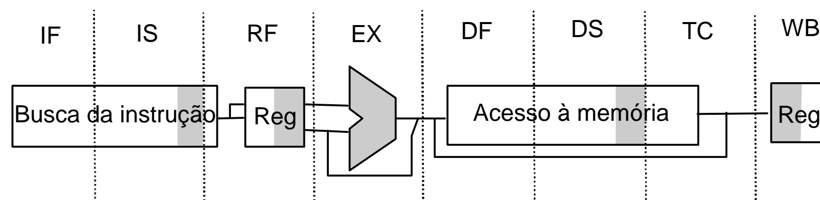
Pipelining Avançado e ILP

Super-*pipelining*

- consiste em aumentar o número de estágios da *pipeline*, conseguindo diminuir Tcc e aumentar a frequência de relógio.

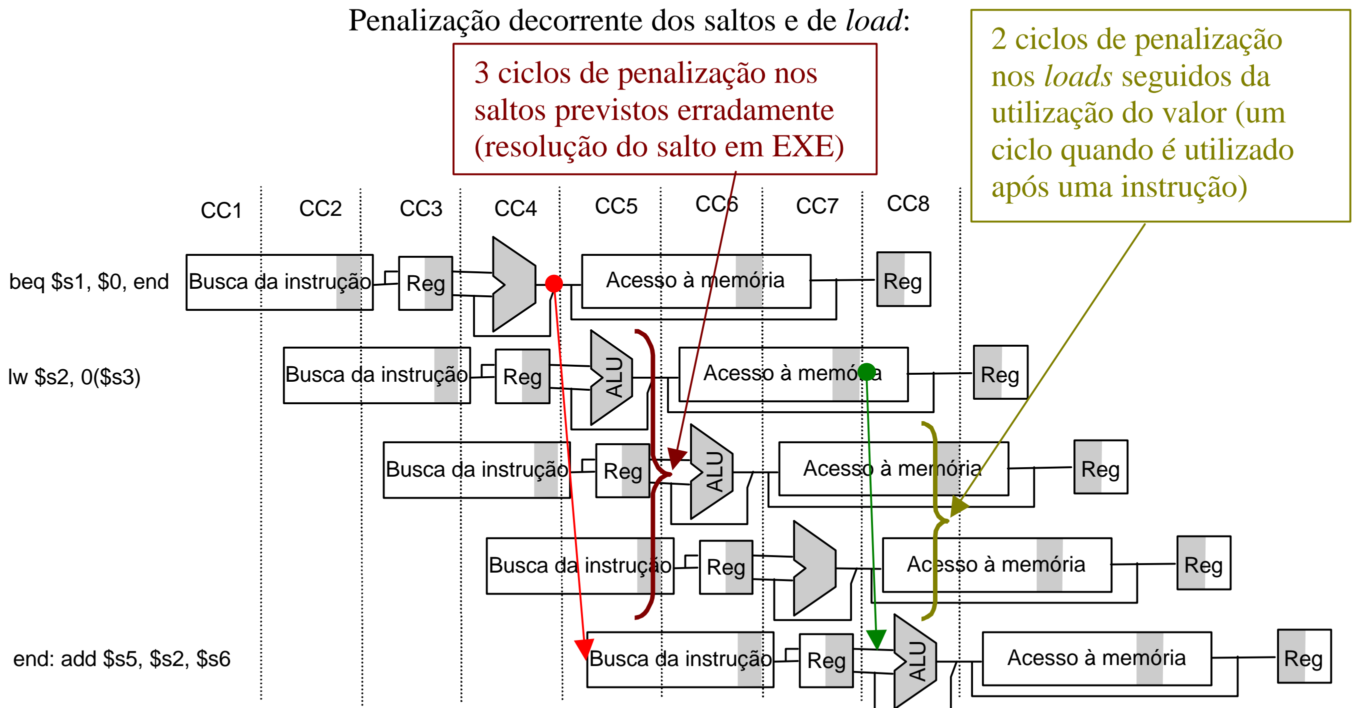
- pode não diminuir a latência das instruções (pode mesmo aumentar) o que implica um aumento das penalizações devido a anomalias de dados e de controlo, se não forem introduzidos outros melhoramentos no DP

- MIPS R4000 – 8 estágios de *pipeline*, sendo os estágios adicionais utilizados para acessos à memória



IF – Primeira metade da busca da instrução (selecção do PC)
IS – Segunda metade da busca da instrução, completando o acesso
RF – descodificação e leitura do valor dos registos, detecção de *hit*
EX – Execução, incluindo a resolução dos saltos
DF – Busca dos dados, primeira metade do acesso a *cache*
DS – segunda metade do acesso a *cache*
TC – *Tag check*, determinar se o acesso foi um *hit*
WB- Escrita dos resultados em registo para *load* e registo-registo

Penalização decorrente dos saltos e de *load*:



© João Luís Sobral 2002

Pipelining Avançado e ILP

- Super-*pipelining* (continuação)

  - Comparar o desempenho entre o MIPS com 5 estágios (MIPS*5*) e o MIPS R4000 com 8 estágios de *pipeline* (MIPS*8*), assumindo que não existem *caches misses*, 50% de *stalls* nos *load* e 75% de saltos previstos correctamente.

    Mistura de instruções (gcc):

Tipo de Instrução	Frequência
Tipo R	49%
Load	22%
Store	11%
Branch	18%

MIPS*5* (resolução dos saltos em EXE, s/ optimizar escritas em PC):

CPI*load* (50% de situações de *stall*) = 0,5 x1 + 0,5 x2= 1,5
CPI*branch* (75% previsões correctas) = ,75 x1 + ,25 x3 = 1,5

CPI*5* = 0,49 x 1 + 0,22 x 1,5 + 0,11 x 1 + 0,18 x 1,5 = 1,2

MIPS*8*:

CPI*load* (50% de situações de *stall*) = 0,5 x1 + 0,5 x3 = 2,0
CPI*branch* (75% previsões correctas) = ,75 x1 + ,25 x4 =1,75

CPI*8* = 0,49 x 1 + 0,22 x 2 + 0,11 x 1 + 0,18 x 1,75 = 1,36

Comparação mantendo a frequência de relógio:

CPI*5* / CPI*8* = 1,36 / 1,2 = 0,88 (degradação de 12%)

Comparação aumentado a frequência de relógio em 50%:

Ganho = #I x CPI*5* xTcc*5* / #I x CPI*8* x Tcc*8*
= 1,5 x CPI*5* / CPI*8* = 1,5 x 1,2 / 1,36 = 1,32x

Comparação melhorando a previsão de saltos para 90% e reduzindo um ciclo ao impacto dos *stall* nos *load*:

CPI*load* (50% de situações de *stall*) = 0,5 x1 + 0,5 x2 = 1,5
CPI*branch* (90% previsões correctas) = 0,9 x1 + 0,1 x4 = 1,4

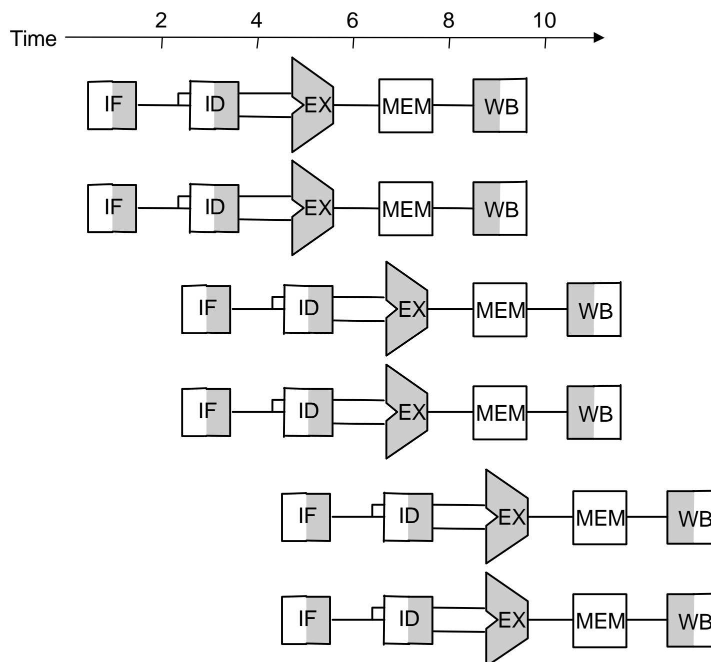
CPI*8a* = 0,49 x 1 + 0,22 x 1,5 + 0,11 x 1 + 0,18 x 1,4 = 1,18
Ganho = 1,5 x 1,2 / 1,13 = 1,53x

© João Luís Sobral 2002

Pipelining Avançado e ILP

Super-escalaridade

- Baseia-se na duplicação de unidades funcionais por forma a ser possível executar mais do que uma instrução em cada ciclo de relógio
- CPI < 1 (em situações ideais)
- Exemplo: MIPS com grau de super-escalaridade 2



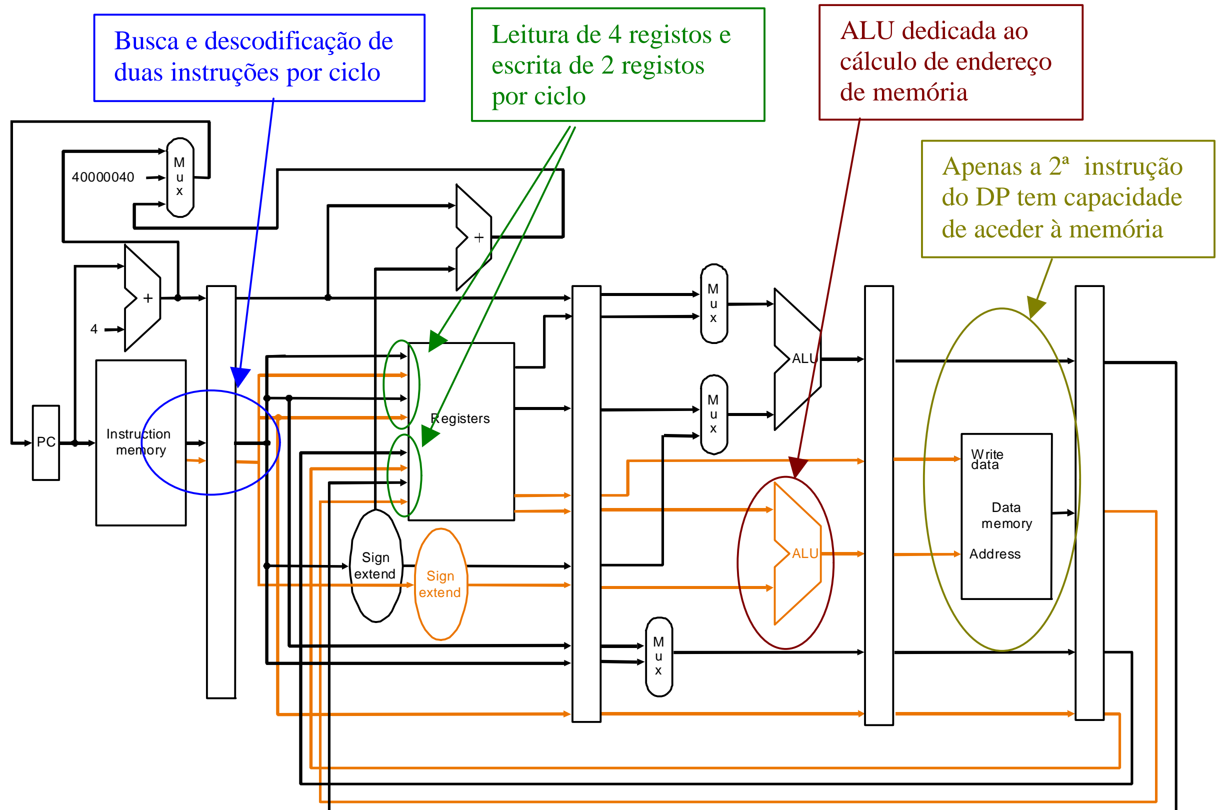
- Este tipo de abordagem requer a capacidade de efectuar a busca de duas instruções por ciclo (64 bits no MIPS), duas descodificações, etc.
- Uma abordagem mais simples consiste em apenas duplicar algumas unidades funcionais, não suportando todas as combinações de instruções.
- Em MIPS a unidade de MEM apenas é utilizada pelos LW e SW, sendo mais simples conceber um MIPS super-escalar que suporta uma operação tipo R ou *branch* em simultâneo com um acesso à memória.

Tipo de instrução	Estágio						
ALU ou branch	IF	ID	EX	MEM	WB		
load ou *store*	IF	ID	EX	MEM	WB		
ALU ou branch		IF	ID	EX	MEM	WB	
load ou *store*		IF	ID	EX	MEM	WB	
ALU ou branch			IF	ID	EX	MEM	WB
load ou *store*			IF	ID	EX	MEM	WB

Pipelining Avançado e ILP

MIPS Super-escalar

- DP para suporte a uma operação tipo R ou *branch* em simultâneo com um acesso à memória.



- Desempenho de MIPS super-escalar (ignorando penalização dos saltos).

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

	ALU || *branch*	*Load* || *store*	Ciclo
cic:		lw $t0, 0($s1)	1
	addi $s1, $s1, -4		2
	addu $t0, $t0, $s2		3
	bne $s1, $0, cic	sw $t0, 4($s1)	4

CPI ≅ número de ciclos / número de instruções = 4 / 5 = 0,8

- Desempenho de MIPS super-escalar com *loop unrolling* (grau 4)

	ALU || *branch*	*Load* || *store*	Ciclo
cic:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
		lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 0($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1, $0, cic	sw $t3, 4($s1)	8

CPI ≅ 8/14 = 0,57

Utiliza mais registos:

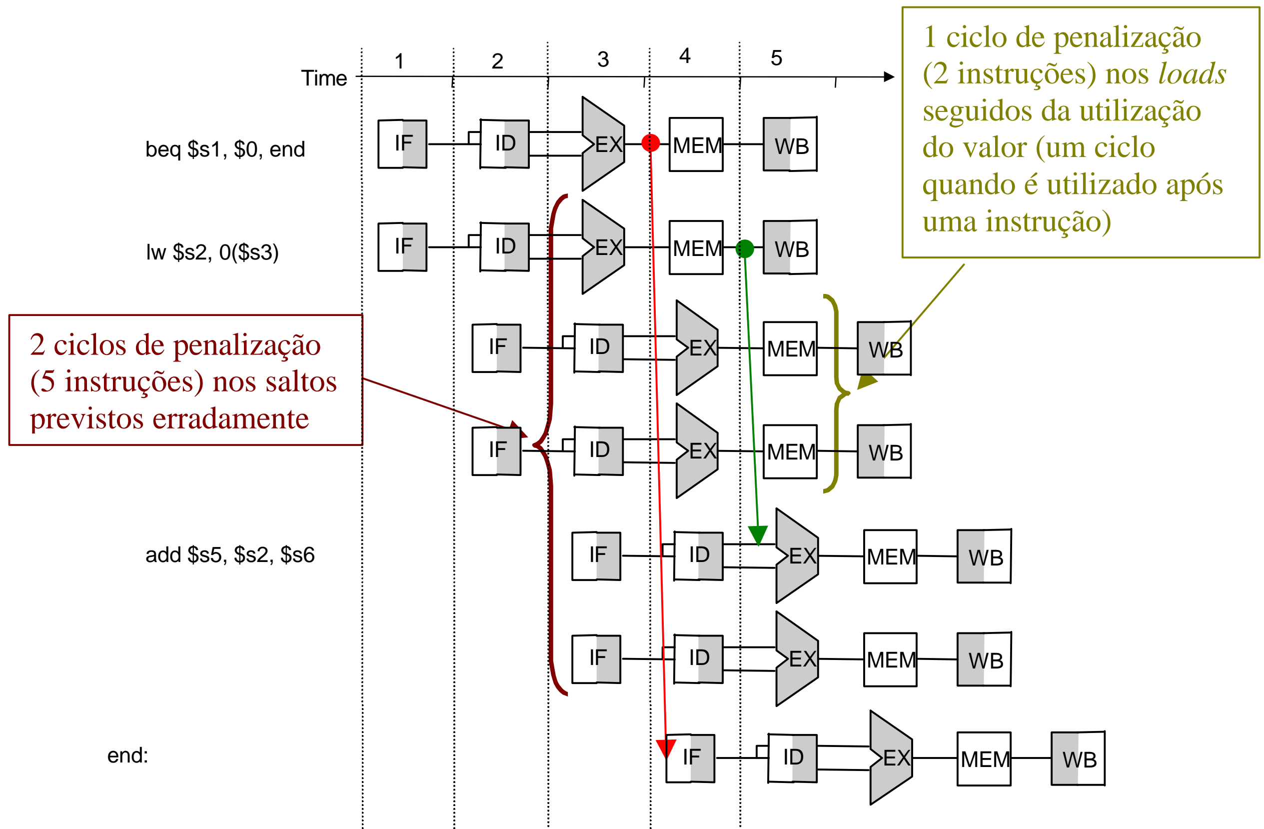
$t1,$t2 e $t3

© João Luís Sobral 2002

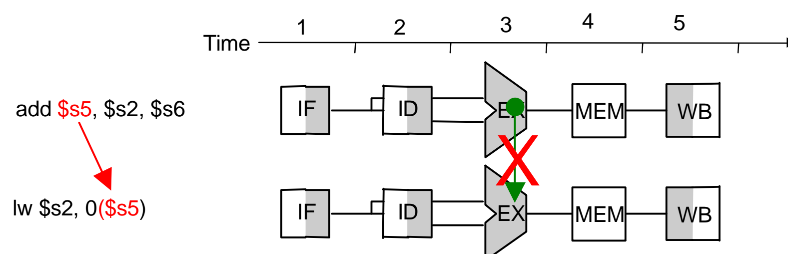
Pipelining Avançado e ILP

- MIPS Super-escalar

  - Penalização decorrente dos saltos e de *load*:



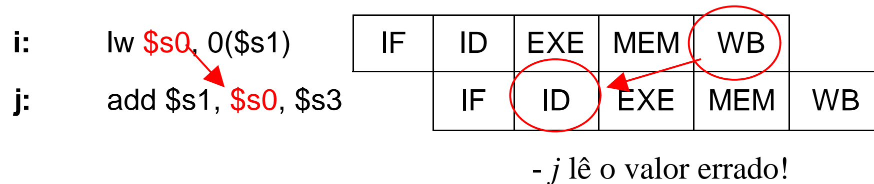
  - A detecção de anomalias e encaminhamento de dados em arquitecturas super-escalares é extremamente complexa. Adicionalmente as instruções executadas em cada ciclo devem ser complemente independentes.



Pipelining Avançado e ILP

Nova visão das dependências de dados

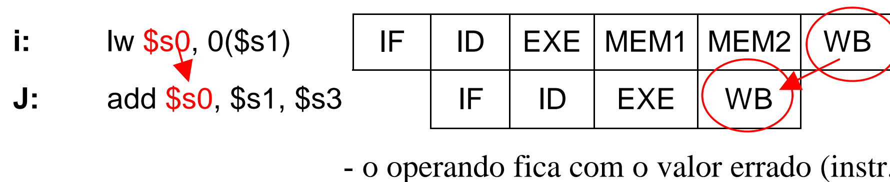
- **RAW** (*Read After Write*) – uma instrução *j* tenta ler um operando antes de uma instrução anterior *i* o escrever. Normalmente resolvido com *stall* ou encaminhamento de dados



- *j* lê o valor errado!

- **WAW** (*Write After Write*) – uma instrução *j* tenta escrever o operando antes de uma instrução anterior *i* o escrever.

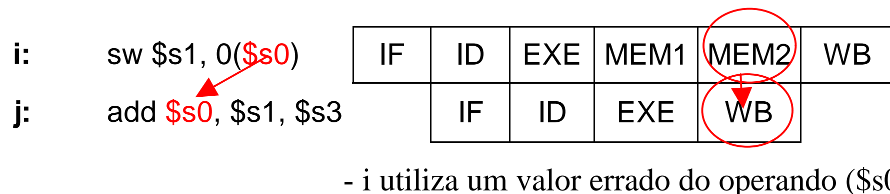
  Está apenas presente em *pipeline* que escrevem valores em mais do que um estágio da *pipeline*, ou quando as instruções não são completadas na ordem do programa



- o operando fica com o valor errado (instr. i)

- **WAR** (*Write After Read*) – uma instrução *j* tenta escrever um novo valor num operando antes de uma instrução anterior *i* o ler.

  Originado quando a leitura de registos pode ocorrer após o WB da instrução seguinte, nomeadamente, quando as instruções não são completadas pela ordem do programa



- i utiliza um valor errado do operando ($s0)

- As dependências WAW e WAR resultam da utilização de um número limitado de registos (dependências de nome), podendo ser eliminadas através de RENOMEAÇÃO de REGISTOS.

- As dependências de dados em acessos à memória são mais complexas:

  - 100 ($s0) = 20 ($s2) ?

  - 20 ($s2) = 20 ($s2) em iterações diferentes de um ciclo?

 © João Luís Sobral 2002

Pipelining Avançado e ILP

Pipelining com operações multi-ciclo

- Na execução em *pipelining* não é viável que todas as operações demorem o mesmo número de ciclos, especialmente quando são suportadas instruções com vírgula flutuante (FP).

- A generalidade das arquitecturas possuem várias unidades funcionais, cada unidade implementando uma funcionalidade específica (ex. operações em FP).

- Nem todas as unidades funcionais podem implementar a execução em *pipelining*. Quando é suportada *pipeline* o estágio EXE divide-se em vários estágios, podendo ser iniciada uma instrução em cada ciclo, caso contrário, o estágio EXE é repetido várias vezes.

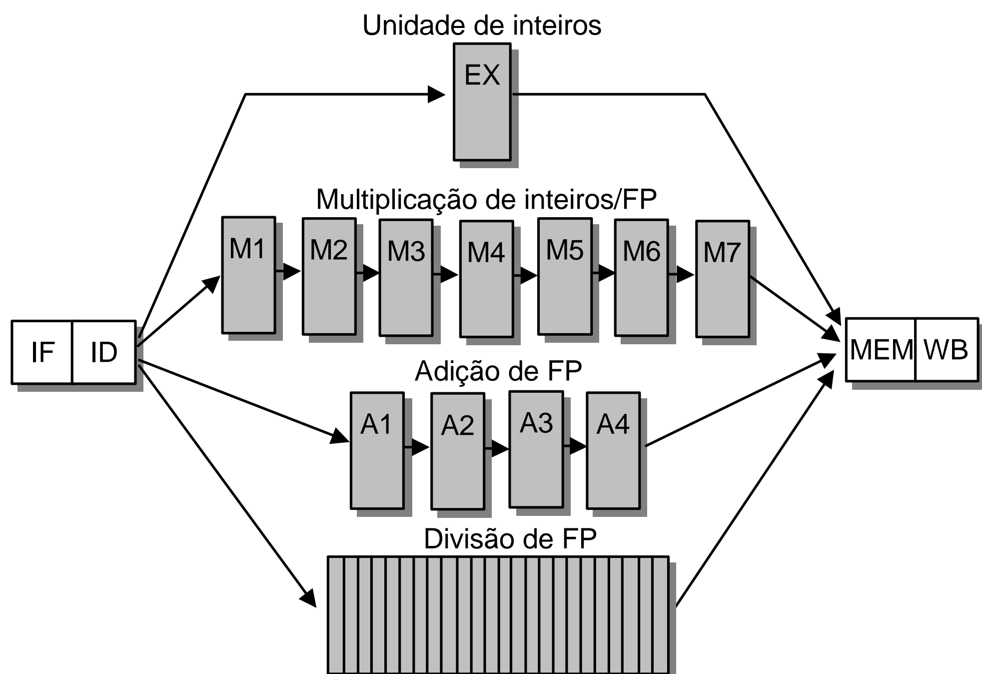
- Exemplo com quatro unidades funcionais:

    1. Processamento de inteiros, que processa operações ALU em inteiros, *load* e *store* e saltos, 1 ciclo

    2. multiplicação de FP e de inteiros, 7 ciclos

    3. Adição e subtracção de FP, 4 ciclos

    4. divisão de FP e de inteiros, 24 ciclos, sem *pipeline*



Pipelining Avançado e ILP

Pipelining com operações multi-ciclo

- **latência** - número de ciclos necessários entre uma instrução que produz um valor e uma instrução que utiliza esse valor (dependências RAW, com encaminhamento de dados)

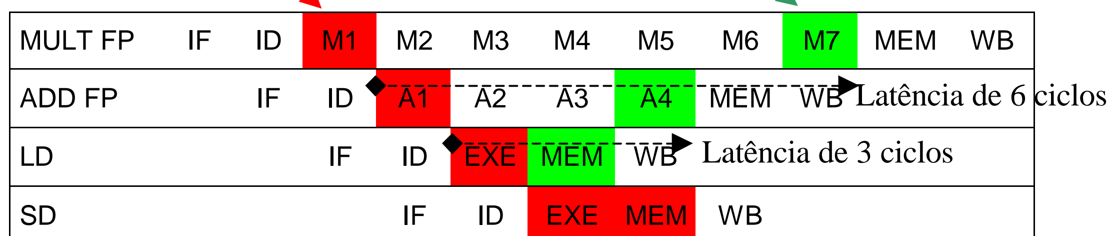
- **intervalo de iniciação** - número de ciclos que deve separar duas instruções que utilizam a mesma unidade (para evitar anomalias estruturais)

- Exemplo para o MIPS com 4 unidades funcionais

Unidade funcional	Latência	Intervalo de iniciação
ALU de inteiros	0	1
Memória de dados (lw)	1	1
Adição FP	3	1
Multiplicação FP	6	1
Divisão FP	24	24

Estágio em que são necessários os dados a processar

Estágio em que são disponibilizados os dados processados



- A unidade de divisão pode originar anomalias estruturais, uma vez que não implementa *pipeline*, o que origina *stalls*

- O número de escritas em registos num ciclo pode ser superior a 1 (anomalia estrutural) porque a conclusão das instruções (WB) não é efectuada pela ordem que são iniciadas

	Ciclo										
Instrução	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EXE	MEM	WB					
…			IF	ID	EXE	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
					IF	ID	EXE	MEM	WB		
						IF	ID	EXE	MEM	WB	
LD F8, 0 (R2)							IF	ID	EXE	MEM	WB

Pipelining Avançado e ILP

Pipelining com operações multi-ciclo (exemplo)

- As dependências de dados (RAW) impõem limitações às instruções que podem iniciar a execução em cada ciclo (devem ser respeitadas a latência e intervalos de iniciação) => *stalls* mais frequentes:

Instrução	Ciclo															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
LD F4, 0(R2)	IF	ID	EX	M	WB											
MULTD F0, F4, F6		IF	ID	*st*	M1	M2	M3	M4	M5	M6	M7	M	WB			
ADDD F2, F0, F8			IF	*st*	ID	*st*	*st*	*st*	*st*	*st*	*st*	A1	A2	A3	A4	M
SD F2, 0(R2)				*st*	IF	*st*	*st*	*st*	*st*	*st*	*st*	ID	*st*	*st*	st	EX

- Podem surgir problemas com WAW porque a conclusão das instruções (WB) não é efectuada pela ordem que são iniciadas

- As dependências WAR não causam problemas porque todas as instruções lêem os operandos (2ºestágio) antes da seguinte os escrever (5º estágio ou mais tarde)

- Existem problemas com o processamento de excepções, uma vez que as instruções não são completadas pela ordem do programa.

Escalonamento estático de *pipeline* com operações multi-ciclo

- Na fase ID verifica-se se a instrução pode iniciar a fase de execução, caso contrário é originado um *stall* das instruções em IF e ID:

  1. Não existem anomalias estruturais (apenas uma instrução pode fazer WB em cada ciclo e/ou a unidade funcional deve estar livre)

  2. As dependências RAW podem ser resolvidas com encaminhamento (não existem instruções pendentes que escrevam num registo utilizado)

  3. A instrução não origina anomalias WAW (nenhuma das instruções pendentes escreve no mesmo registo que a actual)

- Origina entre 0,65 e 1,21 *stalls* por instrução (SPEC FP)

- **O escalonamento estático com a iniciação das instruções pela ordem do programa, apenas quando não existem dependências é demasiado limitativo!**

Pipelining Avançado e ILP

Escalonamento dinâmico da *pipeline*

- O processador rearranja as instruções para diminuir os *stalls* da *pipeline* => **instruções podem ser iniciadas e/ou completadas fora de ordem.**

- Diminui a complexidade dos compiladores e a dependência do código gerado de uma arquitectura específica

- Aumenta fortemente a complexidade do hardware.

- Elimina *stalls* devidos a dependências de dados através da execução fora de ordem de instruções:

```
DIVD  F0, F2, F4
ADDD  F10, F0, F8
SUBD  F12, F8, F14
```

Com execução fora de ordem SUBD pode iniciar a execução antes de ADDD

- A execução fora de ordem pode originar anomalias WAR e WAW (em MIPS)

```
DIVD  F0, F2, F4
ADDD  F10, F0, F8
SUBD  F8, F8, F14
SUBD  F10, F0, F4
```

WAR: SUBD pode escrever em F8 antes de ADDD ler o valor

WAW: SUBD pode escrever em F10 antes de ADDD escrever o valor

- Exemplo de um algoritmo de escalonamento dinâmico (c/*scoreboard*)

Scoreboard - mantém um registo dos registos em utilização pelas instruções em execução, permite determinar quando uma instrução pode iniciar a execução e quando pode escrever o resultado nos registos

ID (passos 1 e 2)

1. ***issue*** – se a unidade funcional se encontra livre e nenhuma instrução activa escreve no mesmo registo, inicia a instrução e actualiza o *scoreboard*. Caso contrário efectua o *stall* da *pipeline*. Evita anomalias WAW e estruturais.

2. **leitura dos operandos** – quando os operandos da instrução estão disponíveis e mais nenhuma instrução activa escreve nesses operandos estes são lidos dos registos e a instrução inicia a execução. Evita anomalias RAW.

3. **execução** – a unidade funcional executa a instrução, podendo demorar vários ciclos. Quando termina notifica o *scoreboard*

4. **escrita dos valores** – quando não existem instruções anteriores para execução que lêem o registo destino, este é actualizado. Evita WAR.

© João Luís Sobral 2002

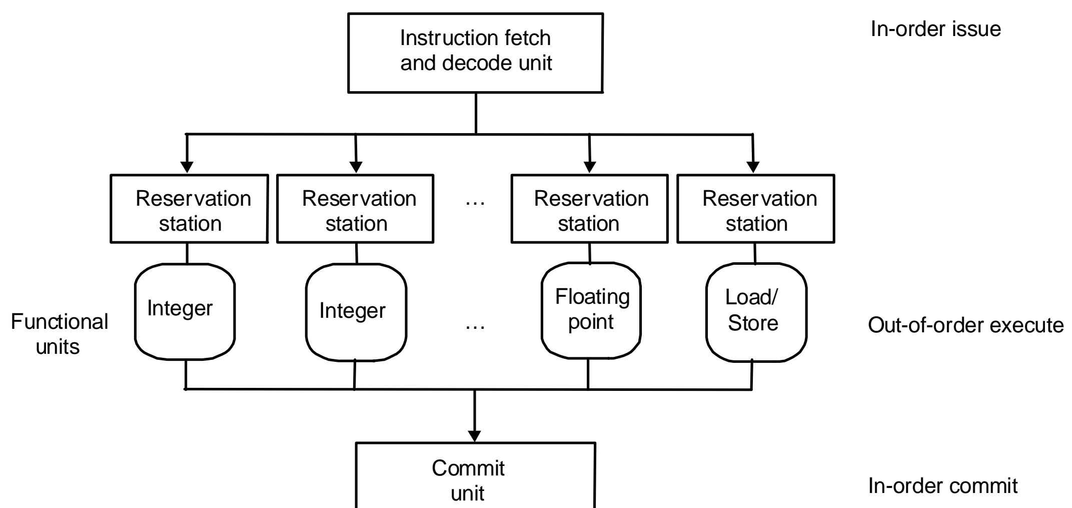
Pipelining Avançado e ILP

Escalonamento dinâmico da *pipeline*

- Algoritmo de escalonamento dinâmico com renomeação (*Tomasulo*)

  Baseia-se num processo de mudança de nome de registos e em estações de reserva para eliminar as dependências WAR e WAW. As estações de reserva contêm instruções pontas para execução e cópias dos operandos que já estão disponíveis. A renomeação de registos atribui registos físicos diferentes sempre que é detectada uma dependência

  1. ***issue*** – se a estação de reserva da unidade funcional se encontra livre envia a instrução para a estação de reserva correspondente com cópias dos operandos já disponíveis e atribui registos físicos os operandos. Caso contrário efectua o *stall* da *pipeline*. Evita anomalias estruturais, elimina WAR e WAW.
  2. **execução** – quando os operandos estão disponíveis inicia a execução da instrução. Evita RAW.
  3. **Escrita do resultado** – o resultado é enviado às unidades à espera do resultado, e, posteriormente escritos no banco de registos.



- A *pipeline* é dividida em três grupos de unidades: busca de instruções e descodificação e *issue*, unidades de execução e unidade de conclusão.

- A unidade de busca e descodificação é responsável por descodificar as instruções e enviá-las para a unidade de execução correspondente.

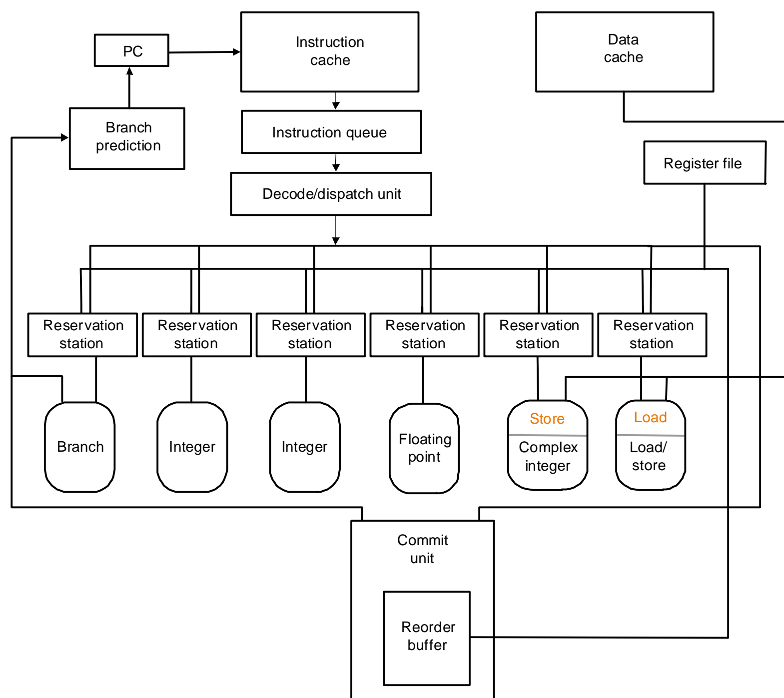
- A unidade de conclusão das instruções é responsável por verificar quando é seguro tornar os resultados da instrução visíveis (saltos)

- O escalonamento dinâmico da *pipeline* é bastante mais complexo que as abordagens anteriores, especialmente, por em geral surge ligado à execução super-escalar, sendo possível em cada ciclo iniciar e/ou concluir entre 4 e 6 instruções.

Pipelining Avançado e ILP

Pipeline do PowerPC 604 e do Pentium Pro

- O PowerPC 604 efectua a busca de 4 instruções por ciclo (128 bits), possui uma história de saltos de 512 entradas e possui 6 unidades funcionais para execução.



Parâmetro	PowerPC 604	Pentium Pro
Número máximo de instruções iniciadas por ciclo de relógio	4	3
#I máximo que completam a execução por ciclo de relógio	6	5
Número máximo de instruções completadas por ciclo de relógio	6	3
Número de bytes lidos da *cache* de instruções	16	16
Número de bytes na fila de instruções	32	32
Número de instruções no *reorder buffer*	16	40
Número de entradas na tabela de história de saltos	512 (2 bits)	512 (4 bits)
Número de *rename buffers*	12 Int. + 8 FP	40
Número total de *reservation stations*	12	20
Número total de unidades funcionais	6 2 - Inteiros + 1 - Inteiros cpl. + 1 - FP + 1 - saltos + 1 - *load* e *store*	6 2 – inteiros + 1 – FP 1 – Saltos 1 – *Load* 1 - *Store*

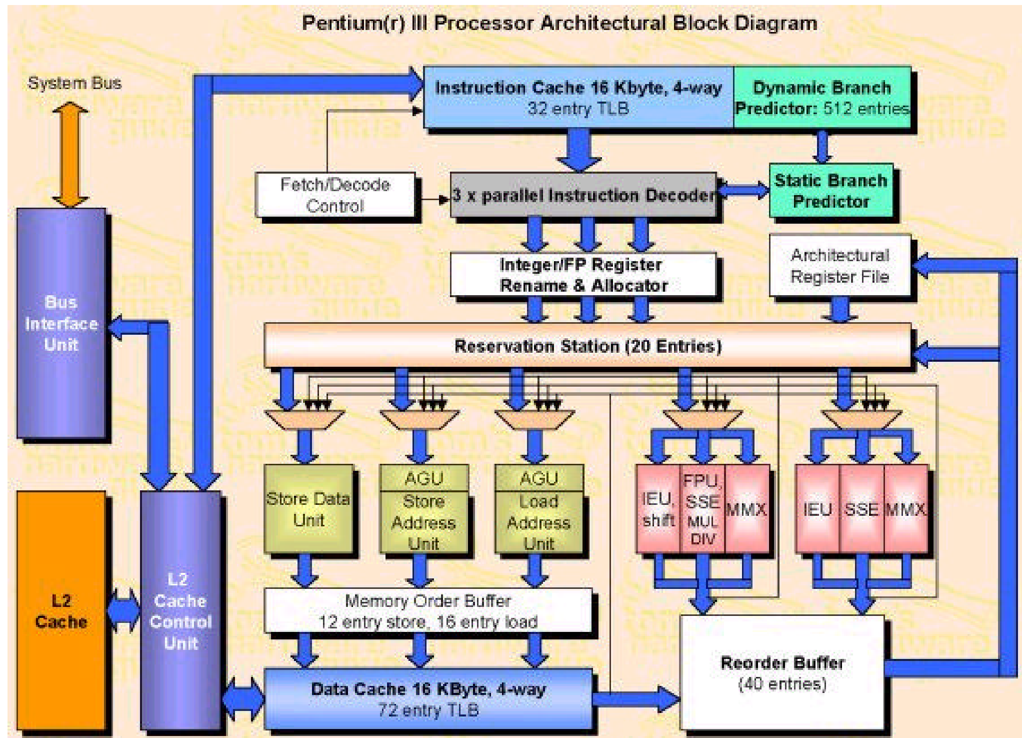
- Os *rename buffers* são registos internos utilizados pelas instruções em execução para armazenar dos resultados até a instrução ser completada. São atribuídos às instruções na fase de descodificação

- Os *reorder buffers* são utilizados pela unidade de *commit* para acompanhar as instruções em execução.

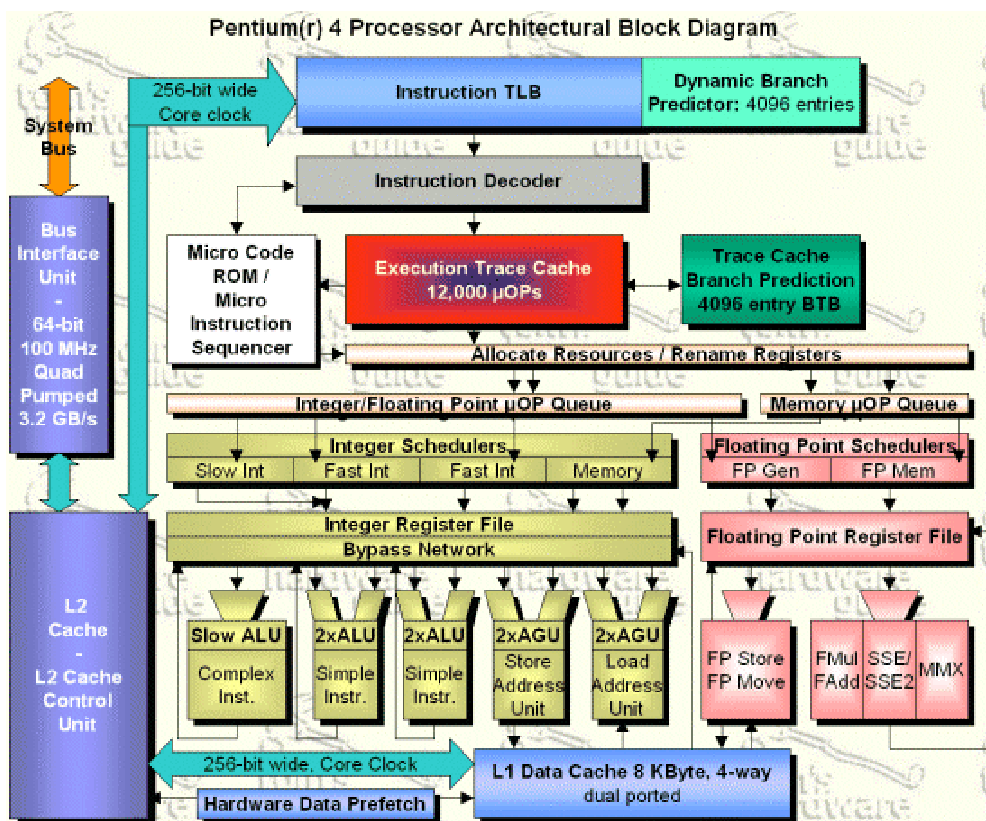
Pipelining Avançado e ILP

- Pentium III

  - características idênticas ao Pentium Pro, excepto a possibilidade de execução de instruções MMX/SSE nas duas unidades de inteiros



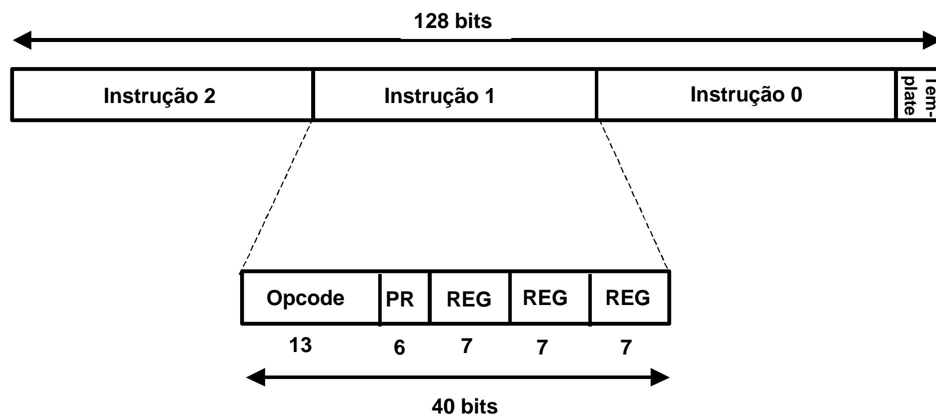
- Pentium 4



Pipelining Avançado e ILP

VLIW (*Very Long Instruction Word*)

- O escalonamento dinâmico incrementa de forma considerável a complexidade do Hardware.
- VLIW efectua um escalonamento estático, sendo o compilador responsável por indicar as instruções que podem ser realizadas em paralelo.
- O formato de instrução indica as operações que são realizadas em paralelo por cada unidade funcional.
- Exemplo IA-64:



- Limitações de VLIW

1. O código gerado tende a ser de maior dimensão, porque é necessário inserir *nop* nos campos da instrução não preenchidos.
2. Compatibilidade de código entre gerações dos mesmo processador uma vez que tende a expor a arquitectura interna do processador
3. É mais penalizado com *stalls* que o escalonamento dinâmico

EPIC – IA-64 / Itanium

- 64 registos de inteiros + 60 registos FP, ambos com 64 bits
- 3 instruções em 128 bits (LIW?)
  - menos bits que VLIW clássico, produzindo código mais compacto
  - possibilidade de ligação entre os vários grupos de instruções
- Verificação de dependências em HW => compatibilidade de código

Pipelining Avançado e ILP

Processadores vectoriais

- Alguns processadores incluem unidades dedicadas à execução de operações sobre vectores de forma mais eficiente.

- Uma operação típica sobre vectores efectua uma adição de dois vectores de 64 elementos, em vírgula flutuante. A operação é equivalente a uma iteração sobre os elementos do vector, efectuando uma multiplicação por iteração.

- Vantagens das operações vectoriais

  1. o cálculo de um resultado é independente do anterior, possibilitando a utilização de *pipeline* com bastante estágios, sem gerar anomalias de dados.

  2. uma só instrução especifica um grande número de operações, equivalente à execução de um ciclo, o que reduz a quantidade de buscas de instruções

  3. os acessos à memória para carregar os elementos do vector em registos podem tirar partido da optimização do débito da memória, amortizando o custo elevado dos acessos à memória

  4. as anomalias de controlo são reduzidas porque os ciclos são transformados numa só instrução.

- Exemplo calcular

  $Y = \boldsymbol{a} X + Y$ Y, X - vectores de 64 elementos
  $\boldsymbol{a}$ – escalar

  Unidade de processamento vectorial e 8 registos de 64 valores FP

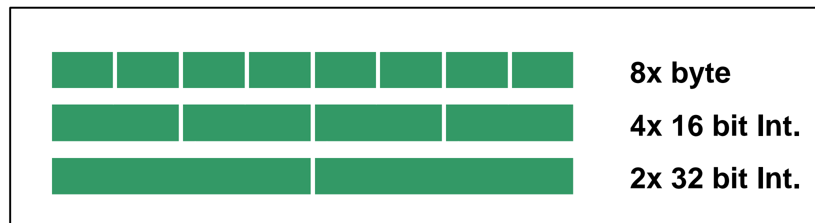
MIPS	MIPS vectorial

<pre>   LD   F0, a
     ADDI   R4, Rx, 512   # último elemento
Cic: LD   F2, 0 (Rx)   # lê X(i)
     MULTD F2, F0, F2   # α X(i)
     LD   F4, 0(Ry)   # lê Y(i)
     ADDD   F4, F2, F4   # α X(i) + Y(i)
     SD   F4, 0(Ry)   # guarda Y(i)
     ADDI   Rx, Rx, 8   # inc. índice X
     ADDI   Ry, Ry, 8   # inc. índice Y
     SUB   R20, R4, Rx   # calcula limite
     BEQZ   R20, Cic</pre>
	<pre>LD F0, a
	
	LV V1, 0(Rx) # lê vector X
	MULTSV V2, F0, V1 # α X
	LV V2, 0(Ry) # lê vector Y
	ADDV V4, V2, V3 # add
	SV V4, 0(Ry) # guarda Y</pre>

Pipelining Avançado e ILP

Vector para multimédia: instruções MMX, SSE e SSE2 de IA-32

- Conceito próximo de instruções vectoriais
- **MMX** – 57 instruções operando sobre registos de 64 bits que podem representar vectores com 2 valores inteiros de 32 bits, 4 valores inteiros de 16 bits ou 8 valores inteiros de 8 bits



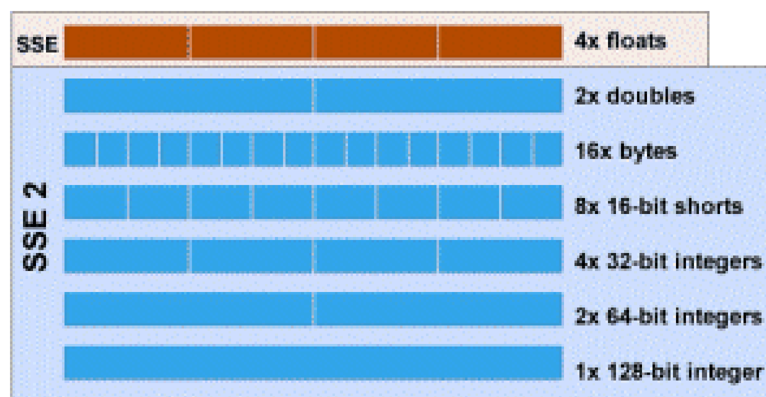
  - Reutiliza os 8 registos FP (Não há mistura entre MMX e FP)

- Contém instruções ler e escrever vectores na memória, aritméticas e lógicas entre vectores e conversão entre tipos de vectores:

  - *Load* e *store* de 32 ou 64 bytes
  - *Add, sub* em paralelo: 8 x 8 bits, 4 x 16 bits ou 2 x 32 bits
  - Deslocamentos (sll, srl), And, And Not, Or, Xor em paralelo
  - Multiplicação e mult-add em paralelo
  - Comparações em paralelo (=,>)
  - *Pack* – conversão entre tipos 32b <->16b, 16b <->8b

- **SSE** - introduz registos de 128 bits com suporte o operações sobre operandos em vírgula flutuante e uma gestão mais eficiente dos registos.

- **SSE2** - substitui MMX com registo de 128 bits e novos tipos de elementos de vector



Bibliografia

- **Secções 6.8, 6.9, 6.10, 6.11** de *Computer Organization and Design: the hardware/software interface*, D. Patterson and J. Hennessy, Morgan Kaufmann, 2ª edição, 1998.

- **Adicional:**

  - Secções 3.4, 3.7, 3.8, 3.9, 4.2, 4.4, B.1, B.2, B.3 de *Computer Architecture: A Quantitative Approach*, J. Hennessy and D. Patterson, Morgan Kaufmann, 2ª edição, 1996.