

## UT 1. Introducción a la Arquitectura de Computadores Tema 1.3 Diseño de los juegos de instrucción

J. Flich, P. López, V. Lorente,  
A. Pérez, S. Petit, J.C. Ruiz, S. Sáez, J. Sahuquillo

Departamento de Informática de Sistemas y Computadores  
Universitat Politècnica de València



**DOCENCIA VIRTUAL**

**Finalidad:**  
Prestación del servicio Público de educación superior (art. 1 LOU)

**Responsable:**  
Universitat Politècnica de València.

**Derechos de acceso, rectificación, supresión, portabilidad, limitación u oposición al tratamiento conforme a políticas de privacidad:**  
<http://www.upv.es/contenidos/DPD/>

**Propiedad intelectual:**  
Uso exclusivo en el entorno de aula virtual.  
Queda prohibida la difusión, distribución o divulgación de la grabación de las clases y particularmente su compartición en redes sociales o servicios dedicados a compartir apuntes.  
La infracción de esta prohibición puede generar responsabilidad disciplinaria, administrativa o civil

 UNIVERSITAT POLITÈCNICA DE VALÈNCIA





## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64
- 3 Tipos de juegos de instrucciones
- 4 Registros y tipos de operando
- 5 Codificación de las instrucciones
- 6 Direccionamiento de la memoria
- 7 Control de flujo
- 8 Instrucciones SIMD

### Bibliografía



**Bostjan Cigan.**

SIMD SSE instructions in C, part one, April 2012.



**John L. Hennessy and David A. Patterson.**

*Computer Architecture, Fourth Edition: A Quantitative Approach.*

Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4 edition, 2006.



**John L. Hennessy and David A. Patterson.**

*Computer Architecture, Fifth Edition: A Quantitative Approach.*

Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5 edition, 2012.



**Microsoft.**

MMX, SSE, and SSE2 intrinsics, April 2013.

## Bibliografía (cont.)



**Microsoft.**

Streaming SIMD extensions 4 instructions, April 2013.



**Microsoft.**

Supplemental streaming SIMD extensions 3 instructions, April 2013.

## Índice

**1** Generalidades sobre los juegos de instrucciones

2 El juego de instrucciones del MIPS64

3 Tipos de juegos de instrucciones

4 Registros y tipos de operando

5 Codificación de las instrucciones

6 Direccionamiento de la memoria

7 Control de flujo

8 Instrucciones SIMD

# 1. Generalidades sobre los juegos de instrucciones

## Factores de diseño del juego de instrucciones

- El juego de instrucciones es la interfaz entre los programas y la ruta de datos
- Las instrucciones son el producto de la compilación
  - Si una instrucción del juego no la usa el compilador, es inútil
- La experiencia con la compilación hace pensar que
  - Aunque algunos programas pueden ser muy complejos, la mayoría son muy simples
- Las instrucciones *simples* pueden ejecutarse rápidamente sobre una ruta de datos simple →  $CPI, T \downarrow$

# 1. Generalidades sobre los juegos de instrucciones

## Propiedades de los juegos de instrucciones

### Principio básico

“Caso frecuente: eficiente; caso raro: correcto”

### Regularidad/Ortogonalidad

*Siempre que tenga sentido*, las operaciones, modos de direccionamiento y tipos de datos deben ser independientes.

→ Simplifica la generación de código, sobre todo si la decisión se toma en dos fases de la compilación distintas.

### Ofrecer primitivas y no soluciones

Evitar incluir soluciones que den soporte directo a construcciones de alto nivel

→ Sólo funcionarán con un lenguaje, al ser muy específicas.

→ Tienen más o menos funcionalidad de la necesaria.

En su lugar, el juego de instrucciones ha de proporcionar primitivas para que el compilador genere código optimizado

# 1. Generalidades sobre los juegos de instrucciones

## Propiedades de los juegos de instrucciones (cont.)

### Principio “uno o todo”

O hay una sola forma de hacer una determinada cosa, o todas las formas son posibles.

→ Simplificar el coste del cálculo de cada alternativa.

### Ejemplo: condiciones de salto.

- $>, =$  Una forma de generar todas las condiciones
- $>, >=, <, <=, =, <>$  Todas las formas de generar las condiciones

Incorporar instrucciones que operen con constantes para las cantidades conocidas en tiempo de compilación.

→ En esos casos, realizar la interpretación en tiempo de ejecución resulta ineficiente.



## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64**
- 3 Tipos de juegos de instrucciones
- 4 Registros y tipos de operando
- 5 Codificación de las instrucciones
- 6 Direccionamiento de la memoria
- 7 Control de flujo
- 8 Instrucciones SIMD

## 2. El juego de instrucciones del MIPS64

### Características generales

- La arquitectura MIPS64 se ha utilizado en estaciones de trabajo, consolas, set-top boxes y soluciones empotradas.
- Ofrece instrucciones de 32 bits con formato fijo.
- Incluye 32 registros de propósito general y 32 registros para cálculos en coma flotante.
- El modelo de programación soportado es el de carga y almacenamiento, lo que significa que sólo las instrucciones de carga y almacenamiento intercambian datos con memoria.
- El control de flujo se efectúa mediante instrucciones de comparación y salto.

## 2. El juego de instrucciones del MIPS64

### Instrucciones lógicas y de cálculo

<i>Arithmetic/logical</i>	<i>Operations on integer or logical data in GPRs; signed arithmetic trap on overflow</i>
DADD, DADDI, DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB, DSUBU	Subtract, signed and unsigned
DMUL, DMULU, DDIV, DDIVU, MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND, ANDI	And, and immediate
OR, ORI, XOR, XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL, DSRL, DSRA, DSLLV, DSRLV, DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT, SLTI, SLTU, SLTIU	Set less than, set less than immediate, signed and unsigned
<i>Floating point</i>	<i>FP operations on DP and SP formats</i>
ADD.D, ADD.S, ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D, SUB.S, SUB.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D, MUL.S, MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D, MADD.S, MADD.PS	Multiply-add DP, SP numbers, and pairs of SP numbers
DIV.D, DIV.S, DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C._.D, C._.S	DP and SP compares: “_” = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

## 2. El juego de instrucciones del MIPS64

### Instrucciones de transferencia de datos y salto

<i>Data transfers</i>	<i>Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR</i>
LB, LBU, SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH, LHU, SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW, LWU, SW	Load word, load word unsigned, store word (to/from integer registers)
LD, SD	Load double word, store double word (to/from integer registers)
L.S, L.D, S.S, S.D	Load SP float, load DP float, store SP float, store DP float
MFC0, MTC0	Copy from/to GPR to/from a special register
MOV.S, MOV.D	Copy one SP or DP FP register to another FP register
MFC1, MTC1	Copy 32 bits to/from FP registers from/to integer registers
<i>Control</i>	<i>Conditional branches and jumps; PC-relative or through register</i>
BEQZ, BNEZ	Branch GPRs equal/not equal to zero; 16-bit offset from PC + 4
BEQ, BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T, BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN, MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J, JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL, JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode

## 2. El juego de instrucciones del MIPS64

### Condicionantes

- El juego de instrucciones condiciona el proceso de compilación de los programas
  - ¿Cómo se gestionan los operandos de las instrucciones y sus resultados?
  - ¿Cuántos registros hay disponibles para la ejecución de los programas?
  - ¿De qué manera acceden las instrucciones a los datos?
  - ¿Qué tipos de datos es posible manipular y a través de qué operaciones?
  - ¿Cómo se controla el flujo de instrucciones ejecutadas?
- No existe una única respuesta a cada una de estas preguntas y por eso existen distintos tipos de juegos de instrucciones.

## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64
- 3 Tipos de juegos de instrucciones**
- 4 Registros y tipos de operando
- 5 Codificación de las instrucciones
- 6 Direccionamiento de la memoria
- 7 Control de flujo
- 8 Instrucciones SIMD

### 3. Tipos de juegos de instrucciones

#### Paradigma actual

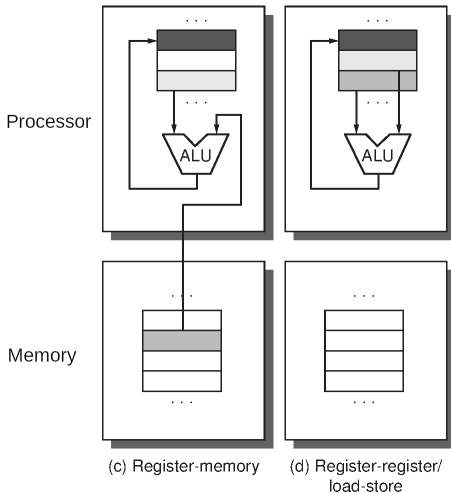
Criterio clásico: almacenamiento de los operandos en la CPU.  
Las instrucciones operan sobre unos datos y generan un resultado.  
Los datos y resultados se almacenan en la memoria y *en la CPU*.

**Registros de propósito general** (Ej.: MIPS64, x86-64).

Los datos y resultados están en el *banco de registros* o en memoria. Todos los operandos deben nombrarse **explícitamente**.

### 3. Tipos de juegos de instrucciones

#### Paradigma actual (cont.)





### 3. Tipos de juegos de instrucciones

#### Paradigma actual

Código para  $C = A + B$

Registros de propósito general.

LOAD R1,A	LOAD R1,A	MOVE C,A
<b>ADD R1,B</b>	LOAD R2,B	<b>ADD C,B</b>
STORE C,R1	<b>ADD R3,R1,R2</b>	
	STORE C,R3	

⇒ El paradigma de procesador con memoria direccionable y banco de registros de propósito general permite la compilación eficiente

### 3. Tipos de juegos de instrucciones

#### Clasificación de las máquinas con registros de propósito general

Dos parámetros significativos:

- 1 Número  $n$  de operandos en una instrucción UAL típica (2 o 3).
  - Si  $n = 2$ , uno de los operandos es a la vez fuente y destino.
  - Si  $n = 3$ , hay dos operandos y un destino.
- 2 Número  $m \leq n$  de direcciones a memoria en una instrucción UAL.

Casos:

- Si  $m = 0$ , máquinas *reg-reg* o *load/store* (típico  $m = 0, n = 3$ ).
  - Todas las instrucciones UAL operan sobre datos en registros, y depositan su resultado en un registro.
  - Las únicas instrucciones que permiten acceder a la memoria son las de carga (*load*) y almacenamiento (*store*).
- Si  $m < n$ , máquinas *reg-mem* (típico  $m = 1, n = 2$ ).
- Si  $m = n$ , máquinas *mem-mem* (típico  $m = 3, n = 3$ ).

### 3. Tipos de juegos de instrucciones

#### Clasificación de las máquinas con registros de propósito general (cont.)

Dos tipos mayoritarios de juego de instrucciones:

**IA** (*Intel Architecture*) Propia de los procesadores de Intel y compatibles (AMD)

**RISC** Propia de los procesadores MIPS, ARM, SPARC, POWER (todavía vigentes) y HP-PA, Alpha, (desaparecidos)

Características esenciales:

	<b>Intel Architecture</b>	<b>RISC</b>
Modelo prog.	<i>R-M</i> Registro-memoria	<i>L/S</i> Load/Store
Registros	Pocos, long. variable	Muchos, long. fija
Formato instr.	Variable	Fijo (32 bits)

### 3. Tipos de juegos de instrucciones

#### Modelos L/S y R-M

##### RISC Modelo L/S

- 1 Las instrucciones de cálculo operan sólo con registros
- 2 Las instrucciones de cálculo tienen tres operandos
- 3 Hay instrucciones especializadas *load* y *store* que transportan los datos entre los registros y la memoria
- 4 La cantidad de trabajo que hacen las instrucciones del juego es parecido (o un cálculo o un acceso a la memoria)

##### IA Modelo R-M

- 1 Las instrucciones de cálculo pueden operar entre registros o con un operando en la memoria
- 2 Las instrucciones de cálculo tienen dos operandos
- 3 Instrucción MOV para transportar datos  $R \Leftrightarrow M$  y  $R \Leftrightarrow R$
- 4 La cantidad de trabajo que hacen las instrucciones es muy diverso

### 3. Tipos de juegos de instrucciones

#### Modelos L/S y R-M (cont.)

La elección afecta al tiempo de ejecución:  $T_{ej} = I \times CPI \times T$

#### RISC Modelo L/S

- 1 Un programa tiene más instrucciones para hacer el mismo trabajo  $\rightarrow I \uparrow$
- 2 Formato más sencillo, más fácil decodificar  $\rightarrow T \downarrow$
- 3 cantidad de trabajo por instrucción homogéneo  $\rightarrow CPI \downarrow$

#### IA Modelo R-M

- 1 Un programa tiene menos instrucciones para hacer el mismo trabajo  $\rightarrow I \downarrow$
- 2 Formato variable, más difícil decodificar  $\rightarrow T \uparrow$
- 3 cantidad de trabajo por instrucción muy diferente  $\rightarrow CPI \uparrow$

### 3. Tipos de juegos de instrucciones

#### Modelos L/S y R-M (cont.)

##### Ejemplo

Intel 32 bits	Equivalente en MIPS-32
add EAX,EBX	add \$t0,\$t0,\$t1
add EAX,a	lw \$t1,a add \$t0,\$t0,\$t1
add a,EAX	lw \$t1,a add \$t1,\$t1,\$t0 sw \$t1,a

## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64
- 3 Tipos de juegos de instrucciones
- 4 Registros y tipos de operando**
- 5 Codificación de las instrucciones
- 6 Direccionamiento de la memoria
- 7 Control de flujo
- 8 Instrucciones SIMD

## 4. Registros y tipos de operando

### Registros y tipos de operando

#### RISC

- Los registros son numerosos y tienen todos igual tamaño.  
Ejemplo: MIPS64: 32 registros de enteros (64 bits) y 32 de coma flotante (64 bits)
- Cada instrucción de cálculo opera con el contenido completo de los registros → conversión entre tipos si las `load` cargan datos de tamaño inferior al tamaño del registro (extensión de zeros o de signo)

#### IA

- Pocos registros, y adaptados, a los tipos de datos disponibles
- Cada instrucción aritmética tiene un código de operación para cada tipo → En x86-64 hay cuatro versiones de `add` para operandos de 8, 16, 32 y 64 bits

→ Es más fácil cambiar la longitud de palabra en RISC



## 4. Registros y tipos de operando

### Registros y tipos de operando (cont.)

Ejemplo: los registros IA-32

- 8 registros de 32 bits EAX ... EDI
- la parte baja de cuatro de ellos puede ser tratada como 4 registros de 16 bits o 8 registros de 8 bits

Ampliación a 64 bits (x86-64):

- Los 8 registros de 32 bits EAX ... EDI son la parte baja de 8 registros de 64 bits RAX ... RDI
- Hay 8 registros de 64 bits adicionales

31		15	8	7	0
EAX	AX	AH	AL		
ECX	CX	CH	CL		
EDX	DX	DH	DL		
EBX	BX	BH	BL		
ESP	SP				
EBP	BP				
ESI	SI				
EDI	DI				

## 4. Registros y tipos de operando

### Registros y tipos de operando (cont.)

#### Ejemplo: los registros MIPS

- 32 registros de 32 bits (64 bits en el MIPS64): R0 ... R31
- Convenio de uso por parte de los compiladores

Name	Number	Use	Preserved across a call?
\$zero	0	The constant value 0	N.A.
\$at	1	Assembler temporary	No
\$v0-\$v1	2-3	Values for function results and expression evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS kernel	No
\$gp	28	Global pointer	Yes
\$sp	29	Stack pointer	Yes
\$fp	30	Frame pointer	Yes
\$ra	31	Return address	Yes

## 4. Registros y tipos de operando

### Registros y tipos de operando (cont.)

#### Ejemplo

##### Código fuente

```
byte a,b,c;  
...  
c = a + b;
```

##### x86-64

```
a  db ...  
...  
MOV AL,a  
add AL,b  
MOV c,AL
```

##### MIPS64

```
a:  byte ...  
...  
lb $t0,a  
lb $t1,b  
add $t2,$t0,$t1  
sb $t2,c
```

## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64
- 3 Tipos de juegos de instrucciones
- 4 Registros y tipos de operando
- 5 Codificación de las instrucciones**
- 6 Direccionamiento de la memoria
- 7 Control de flujo
- 8 Instrucciones SIMD

## 5. Codificación de las instrucciones

### Estrategias de codificación

Las instrucciones se almacenan en la memoria según un formato, que indica la operación a realizar (código de operación) y los operandos.

#### Formato fijo vs. variable:

**Fijo** Todas las instrucciones se codifican utilizando el mismo número de bits.

- Facilita la búsqueda de instrucciones y su decodificación.
- A veces, derrocha bits en el formato, ya que no todas las instrucciones requieren el mismo espacio para codificarse.

**Variable** El número de bits requerido para codificar la instrucción varía según el tipo de instrucción.

- Optimiza espacio ocupado por las instrucciones, y, por lo tanto, por los programas.
- Complica la búsqueda de instrucciones y su decodificación.

## 5. Codificación de las instrucciones

### Estrategias de codificación (cont.)

#### Nº de bits del formato:

El número de bits destinado al formato impone un límite al espacio destinado a cada uno de los campos, el cual limita el nº de variantes del mismo:

- nº de instrucciones (códigos de operación),
- nº de registros,
- espacio de memoria direccionable,
- etc.

## 5. Codificación de las instrucciones

### Estrategias de codificación (cont.)

#### Nº de formatos de instrucción.

¿Cómo se asignan los bits del formato a los campos requeridos por las instrucciones?

**Formato único** La correspondencia entre los bits del formato y los campos es siempre la misma.

- Facilita la decodificación de la instrucción.
- A veces, derrocha bits en el formato, ya que no todas las instrucciones requieren todos los campos previstos.

**Múltiples formatos** Cada formato puede tener campos distintos y define una correspondencia entre éstos y los bits del formato.

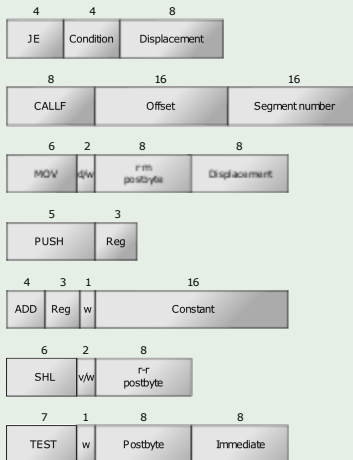
- Permite ajustar mejor los bits ocupados por la instrucción y los campos requeridos.

## 5. Codificación de las instrucciones

### Estrategias de codificación (cont.)

#### Codificación IA-32

- Formato variable: una instrucción puede ocupar entre 1 y 17 bytes
- La decodificación es secuencial: hay que conocer el valor de los bits de un campo para decodificar el siguiente





## 5. Codificación de las instrucciones

### Estrategias de codificación (cont.)

#### Codificación MIPS64

- Formato fijo de 32 bits.
- Pueden descodificarse varios campos en paralelo.
- 3 tipos de formatos: R, I y J
- N° de bits. Hasta  $2^6$  codops +  $2^6$  extensiones (formato R); hasta  $2^5$  registros; desplazamientos de  $2^5$ ; saltos relativos, acceso a memoria con desplazamiento y valores inmediatos de  $2^{16}$ .

I-type instruction



LOAD/STORE: LD rt,Imm(rs) SD rt,Imm(rs)  
ALU con constantes DADD rt,rs,Imm  
Saltos condicionales BEQZ rs,Imm(PC) BEQ rs,rt,Imm(PC)  
Saltos incondicionales JR rs JALR rs

R-type instruction



ALU reg-reg DADD rd,rs,rt  
func es una extension del Cod. op.  
Indica la operación a realizar en la ALU  
Desplazamientos DSLL R1,R2,#shamt  
Transferencia entre regs. MOVN rd,rs,rt MFC0 rt,rd

J-type instruction



Jump and jump and link  
Trap and return from exception

## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64
- 3 Tipos de juegos de instrucciones
- 4 Registros y tipos de operando
- 5 Codificación de las instrucciones
- 6 Direccionamiento de la memoria**
- 7 Control de flujo
- 8 Instrucciones SIMD

## 6. Direccionamiento de la memoria

### Interpretación de las direcciones

- Las unidades físicas de lectura/escritura (las palabras) están formadas por  $W = 2^w$  unidades direccionables (bytes)
- Las palabras se acceden mediante la dirección del menor de sus bytes.
  - La palabra de dirección  $A$  contiene los bytes de dirección  $A, A + 1, \dots, A + W - 1$

- Dos alternativas, sin ninguna consecuencia importante:

#### IA Little endian

El byte de dirección  $A$  ocupa la posición menos significativa de la palabra

Direcciones  
de palabra

0

4

Direcciones  
de byte

3	2	1	0
7	6	5	4

#### RISC Big endian

El byte de dirección  $A$  ocupa la posición más significativa de la palabra

Direcciones  
de palabra

0

4

Direcciones  
de byte

0	1	2	3
4	5	6	7

## 6. Direccionamiento de la memoria

### Alineamiento

- Los juegos de instrucciones pueden dar acceso a unidades de  $1, 2, \dots, 2^w$  bytes.

En el MIPS-32: *byte*, *halfword*, *word*, etc.

- Dos alternativas:

#### RISC Acceso alineado

La dirección del objeto de  $2^i$  bytes es múltiplo de  $2^i$ .

En el MIPS-32: la dirección de un *halfword* es siempre par, y la dirección de un *word* es múltiplo de 4.

#### IA Acceso no alineado

No hay restricción.

Una instrucción puede acceder a bytes contiguos en dos palabras consecutivas, y su ejecución exigirá dos accesos físicos a la memoria.

→ Influencia sobre la complejidad del hardware:  $CPI \uparrow$ ,  $T \uparrow$

## 6. Direccionamiento de la memoria

### Modos de direccionamiento

¿Cómo se especifican los operandos de las instrucciones? → modos de direccionamiento.

¿Interesa tener modos de direccionamiento sofisticados?

- Reducción del número de instrucciones de los programas →  $I \downarrow$
- *Hardware* más complejo →  $CPI \uparrow$  y/o  $T \uparrow$ .

## 6. Direcccionamiento de la memoria

### Modos de direccionamiento (cont.)

#### Ejemplos de modos

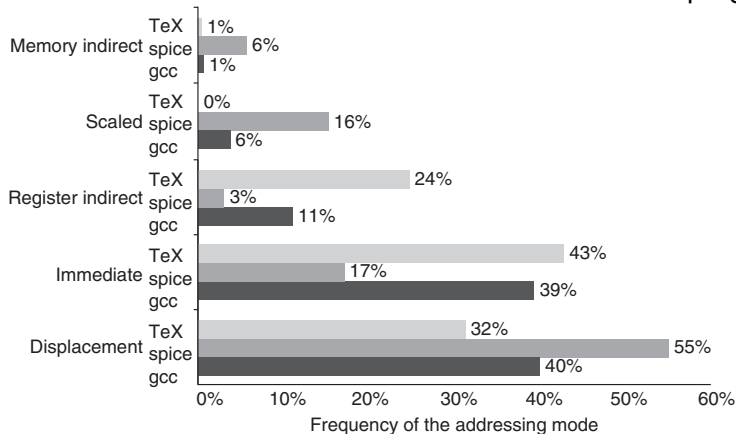
Modo	Ejemplo	Significado
Directo a registro	add r1,r2,r3	$r1 \leftarrow r2 + r3$
Inmediato	add r1,r2,#1	$r1 \leftarrow r2 + 1$
Directo o Absoluto	lw r1,(1000)	$r1 \leftarrow \text{Mem}[1000]$
Registro indirecto	lw r1,(r2)	$r1 \leftarrow \text{Mem}[r2]$
Desplazamiento	lw r1,100(r2)	$r1 \leftarrow \text{Mem}[100 + r2]$
Indexado	lw r1,(r2+r3)	$r1 \leftarrow \text{Mem}[r2 + r3]$
Indirecto a mem.	lw r1,@(r2)	$r1 \leftarrow \text{Mem}[\text{Mem}[r2]]$
Autoincremento	lw r1,(r2)+	$r1 \leftarrow \text{Mem}[r2]$ $r2 \leftarrow r2 + d$
Autodecremento	lw r1,-(r2)	$r2 \leftarrow r2 - d$ $r1 \leftarrow \text{Mem}[r2]$
Escalado	lw r1,100(r2)(r3)	$r1 \leftarrow \text{Mem}[100 + r2 + r3 * d]$

d: tamaño en bytes del operando

## 6. Direcccionamiento de la memoria

### Modos de direccionamiento (cont.)

Estadísticas sobre la utilización de los modos en los programas:



## 6. Direcccionamiento de la memoria

### Modos de direccionamiento (cont.)

**IA** x86-64 incluye hasta el modo escalado

$([Reg] + [Reg] \times d + \text{desplazamiento})$

Combina libremente los modos con los códigos de operación.

Con los modos más complejos, una instrucción puede tener mucho trabajo:

### Ejemplo:

<b>Intel 32 bits</b>	<b>Equivalente en MIPS-32</b>
<code>add EAX, [BX] [SI] .X</code>	<code>add \$t0, \$t0, \$t1</code> <code>lw \$t2, X(\$t0)</code> <code>add \$t3, \$t3, \$t2</code>
<code>add EAX, a</code>	<code>lw \$t1, a</code> <code>add \$t0, \$t0, \$t1</code>

No tienen modos autoincrementados ni autodecrementados



## 6. Direcccionamiento de la memoria

### Modos de direccionamiento (cont.)

**RISC** Para simplificar:

- Modo inmediato: las instrucciones de cálculo tienen dos versiones:

**R-format** :  $Rd \leftarrow Rs \text{ op } Rt$ .

Ejemplo: `ADD`

**I-format** :  $Rd \leftarrow Rs \text{ op } X$

Ejemplo: `ADDI`

Rango de valores inmediatos: solución de compromiso entre disponer de constantes grandes y el número de bits ocupados en el formato. 16 bits en el MIPS.

Aunque se incluyen instrucciones para facilitar el trabajo con valores inmediatos más grandes. Por ejemplo,

`LUI R1, #Uvalor` :  $\text{Regs}[R1] \leftarrow Uvalor \ll 16$

`ORI R1, R1, #Lvalor` :  $\text{Regs}[R1] \leftarrow \text{Regs}[R1] \text{ or } Lvalor$

## 6. Direccionamiento de la memoria

### Modos de direccionamiento (cont.)

- Para acceder a la memoria principal, sólo hay el modo de direccionamiento desplazamiento:  $X(R_n)$ 
  - Haciendo  $X=0$  se obtiene el modo registro indirecto
  - Haciendo  $R_n = \$zero$  queda el modo absoluto

Rango de desplazamiento: solución de compromiso entre disponer de desplazamientos grandes y el número de bits ocupados en el formato. 16 bits en el MIPS.

Algunos procesadores RISC disponen de modos indexados

## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64
- 3 Tipos de juegos de instrucciones
- 4 Registros y tipos de operando
- 5 Codificación de las instrucciones
- 6 Direccionamiento de la memoria
- 7 Control de flujo**
- 8 Instrucciones SIMD

## 7. Control de flujo

### Clases de instrucción de control

#### Tipos:

- Saltos condicionales (*branch*),
- Saltos incondicionales (*jump*),
- Llamadas/retorno a/de procedimiento (*call/return*)

#### Estadísticas de uso:

- Saltos incondicionales, *call* y *return* representan  $\frac{1}{3}$ , y saltan siempre.
- Saltos condicionales. Otro  $\frac{1}{3}$  corresponde con bucles (efectivos casi en el 100 %). El resto ( $\frac{1}{3}$ ) son efectivos en un 50 %.

→ Es más probable *saltar*.  $\frac{5}{6}$  son efectivos (saltan) y  $\frac{1}{6}$  son no efectivos (no saltan).

## 7. Control de flujo

### Modos de direccionamiento en el salto

#### Relativo al PC

- El destino suele estar cerca de la instrucción actual → las direcciones relativas consumen pocos bits.
- N<sup>º</sup> bits para el desplazamiento. Valores típicos son 16–20 bits en saltos condicionales y 26 bits en incondicionales.

#### Indirecto a registro

Útil si el destino del salto es desconocido durante la compilación

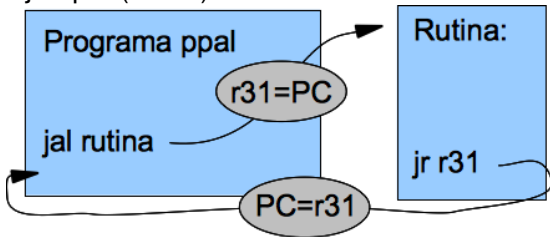
- Sentencias que seleccionan una de entre varias alternativas (v.g., *switch*, *case*).
- Métodos virtuales en lenguajes orientados a objetos.
- Paso de funciones como parámetros a otra función.
- Librerías enlazadas dinámicamente.

## 7. Control de flujo

### Modos de direccionamiento en el salto (cont.)

#### Salto con enlace (*jump and link*)

- Se utiliza para realizar llamadas a subrutinas (*call*).
- Es un salto con modo relativo al PC o indirecto a registro que guarda la dirección de retorno.
- El retorno de la subrutina (*return*) se realiza mediante un salto con modo indirecto a registro, utilizando el registro que se empleó para almacenar el valor de retorno.
- Ejemplo (MIPS):



### Las condiciones de salto

¿Cómo especificar la **condición** de salto?

Varias alternativas:

**1** Códigos de condición (80x86, PowerPC, SPARC).

- El juego de instrucciones define un “estado” que se modifica según el resultado de la última operación aritmética.
- Habitualmente hay unos códigos de condición o *flags*: *C* (acarreo), *Z* (cero), *N* (negativo), etc.
- La instrucción de salto sólo tiene que comprobar la condición correspondiente.
- Ejemplo (Intel): Saltar a *ETI* si  $EAX \leq EBX$

```
CMP EAX,EBX # Resta sin escribir resultado
# Si EAX <= EBX, queda Z=1 o N=1
JLE ETI # Salta si Z=1 o N=1
```

### Las condiciones de salto (cont.)

- Inconvenientes:

- La generación de los códigos de condición no es trivial y además requiere espacio para ello en el chip.
- El hecho de que todas las instrucciones modifiquen los códigos de condición plantea problemas cuando se pretende reordenar el código o se lanzan múltiples instrucciones aritméticas simultáneamente.

**2** Comprobación explícita (MIPS). El resultado de las operaciones se comprueba explícitamente por medio de instrucciones específicas. No hay códigos de condición.

- Ejemplos (MIPS):

```
; Comparacion, salto  
dadd r1,r2,#1  
slt r10,r1,r3  
bnez r10, salto
```

```
; Comparacion + salto  
dadd r1,r2,#1  
blt r1,r3,salto
```



## Índice

- 1 Generalidades sobre los juegos de instrucciones
- 2 El juego de instrucciones del MIPS64
- 3 Tipos de juegos de instrucciones
- 4 Registros y tipos de operando
- 5 Codificación de las instrucciones
- 6 Direccionamiento de la memoria
- 7 Control de flujo
- 8 Instrucciones SIMD**

### Precedentes

El cálculo intensivo con vectores es una de las aplicaciones importantes en la historia de los computadores.

- Tiene aplicaciones en el control, en la simulación de procesos físicos, en el procesamiento de imágenes, etc

Durante los años 1970 a 2000 (aprox), se diseñaron supercomputadores “vectoriales” con instrucciones que operaban con vectores.

- Una instrucción vectorial podía operar con vectores de dimensión dada (típicamente 64 o 256) formados por elementos de cierto tamaño (32 o 64 bits)
- Los tipos de datos que manejaban eran, casi siempre, CF en simple o doble precisión

Cuando la escala de integración lo permitió, los microprocesadores ampliaron su ISA con instrucciones SIMD y sustituyeron a los computadores vectoriales.

## 8. Instrucciones SIMD

### Datos empaquetados

Las instrucciones SIMD (*Single Instruction - Multiple Data*) operan sobre registros que contienen varios datos empaquetados.

- Los tipos de datos utilizables son muchos: enteros de 8, 16, 32, etc. bits y CF en simple y doble precisión.
- El número de datos contenido en un registro es

$$n = \frac{\text{longitud del registro}}{\text{longitud del tipo de datos}}$$

- Los registros tienen longitud fija, pero el juego de instrucciones permite empaquetar datos de diferente longitud  
Ejemplo: con registros de 64 bits se pueden empaquetar
  - 8 datos de tipo *byte*
  - 4 datos de tipo *halfword*
  - 2 datos de tipo *word* o *float*
  - 1 dato de tipo *double*

## 8. Instrucciones SIMD

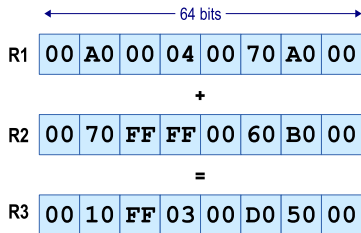
### Datos empaquetados (cont.)

Una instrucción SIMD realiza  $n$  operaciones idénticas.

Ejemplo de dos operaciones de suma con registros de 64 bits

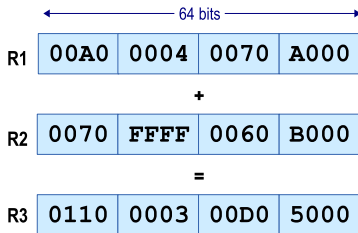
`add_pb R3, R1, R2`

(8 datos de tipo byte)



`add_ph R3, R1, R2`

(4 datos del tipo halfword)



## 8. Instrucciones SIMD

### Algunos ejemplos de instrucciones SIMD en diversas arquitecturas

Instruction category	Alpha MAX	HP PA-RISC MAX2	Intel Pentium MMX	Power PC Altivec	SPARC VIS
Add/subtract		4H	8B,4H,2W	16B, 8H, 4W	4H,2W
Saturating add/sub		4H	8B,4H	16B, 8H, 4W	
Multiply			4H	16B, 8H	
Compare	8B (>=)		8B,4H,2W (=,>)	16B, 8H, 4W (=,>,<,<=)	4H,2W (=,not=,>,<=)
Shift right/left		4H	4H,2W	16B, 8H, 4W	
Shift right arithmetic		4H		16B, 8H, 4W	
Multiply and add				8H	
Shift and add (saturating)		4H			
And/or/xor	8B,4H,2W	8B,4H,2W	8B,4H,2W	16B, 8H, 4W	8B,4H,2W
Absolute difference	8B			16B, 8H, 4W	8B
Maximum/minimum	8B, 4W			16B, 8H, 4W	
Pack (2n bits --> n bits)	2W->2B, 4H->4B	2*4H->8B	4H->4B, 2W->2H	4W->4B, 8H->8B	2W->2H, 2W->2B, 4H->4B
Unpack/merge	2B->2W, 4B->4H		2B->2W, 4B->4H	4B->4W, 8B->8H	4B->4H, 2*4B->8B
Permute/shuffle		4H		16B, 8H, 4W	

## 8. Instrucciones SIMD

### Evolución de las instrucciones SIMD en los procesadores Intel

- 1997 Pentium **MMX** con ocho registros (MM0-MM7) de 64 bits.
- 1999 Pentium-III con juego **SSE**. Nuevas instrucciones y registros de 128 bits (XMM0-XMM7).
- 2001 El Pentium 4 con SSE2 añade nuevas instrucciones.
- 2004 SSE3
- 2006 SSSE3 y SSE4
- 2008 **AVX**, con registros de 256 bits renombrados como YMM0-YMM15
- 2011 AVX2
- 2015 AVX-512 con 32 registros ZMM de 512 bits cada uno.

### Tendencia

Un porcentaje significativo de los transistores adicionales en cada nueva generación de procesadores se dedica a instrucciones vectoriales cada vez más potentes y bancos de registros vectoriales más grandes.

## 8. Instrucciones SIMD

### Banco de registros vectoriales AVX-512

El esquema de los registros AVX-512 ( $ZMM$ ) como extensión de los registros AVX ( $YMM$ ) y los SSE ( $XMM$ ) sería el siguiente:

511	256	255	128	127	0
ZMM0		YMM0		XMM0	
ZMM1		YMM1		XMM1	
...		...		...	
ZMM15		YMM15		XMM15	
ZMM16					
...					
ZMM31					

En procesadores con soporte para los Intel AVX-512, las instrucciones SSE y AVX operan sobre los 128 o 256 bits de menor peso de los primeros 16 registros  $ZMM$ .

## 8. Instrucciones SIMD

### Ejemplo 1: SAXPY

El cálculo de  $\vec{Y} = a\vec{X} + \vec{Y}$  es muy frecuente en el cálculo numérico. Se denomina bucle SAXPY (en simple precisión) o DAXPY (en doble precisión).

```
void SAXPY(int n, float a, float *X, float *Y) {  
    int i;  
    for(i=0; i<n; i++)  
        Y[i] = a*X[i] + Y[i];  
}
```

Con instrucciones no vectoriales

$n$  iteraciones,  $n$  multiplicaciones y  $n$  sumas.



## 8. Instrucciones SIMD

### Ejemplo 1: SAXPY con instrucciones SIMD

Esquema de la solución:

- Las componentes del vector ocupan 32 bits (simple precisión)
- Registros de 128 bits: en cada uno cabe un bloque de cuatro componentes del vector.
- Supongamos tres registros SIMD, de nombres `a_reg`, `X_reg` y `Y_reg`
- Hay que inicializar el registro SIMD `a_reg` con cuatro copias del valor `a`
- Supongamos las instrucciones:

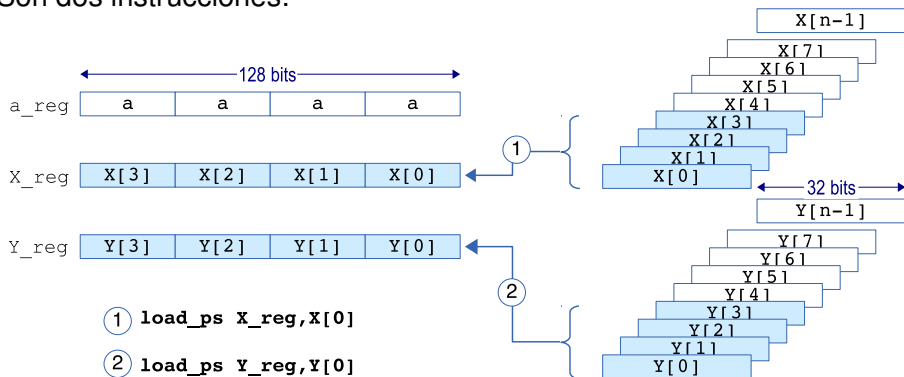
<code>load_ps reg, dir</code>	Lee de memoria un bloque
<code>add_ps rd, rf1, rf2</code>	Suma vectorial de 4 componentes
<code>mul_ps rd, rf1, rf2</code>	Multiplicación vectorial de 4 componentes
<code>store_ps reg, dir</code>	Escribe en memoria un bloque

## 8. Instrucciones SIMD

### Ejemplo 1: SAXPY con instrucciones SIMD (cont.)

Comienza el bucle leyendo 4 componentes de cada vector para depositarlas en un registro del banco SIMD

Son dos instrucciones:



## 8. Instrucciones SIMD

### Ejemplo 1: SAXPY con instrucciones SIMD (cont.)

Una única instrucción permite hacer las cuatro multiplicaciones  $a \cdot X[i]$ .  
Otra instrucción hace las cuatro sumas:

a\_reg

a	a	a	a
---	---	---	---

X\_reg

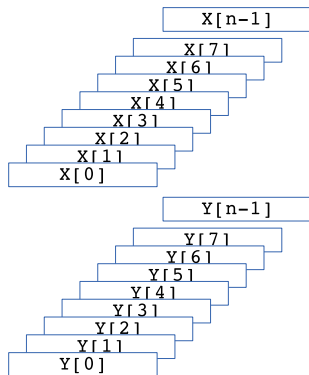
$a \cdot X[3]$	$a \cdot X[2]$	$a \cdot X[1]$	$a \cdot X[0]$
----------------	----------------	----------------	----------------

Y\_reg

$a \cdot X[3]$ +Y[3]	$a \cdot X[2]$ +Y[2]	$a \cdot X[1]$ +Y[1]	$a \cdot X[0]$ +Y[0]
-------------------------	-------------------------	-------------------------	-------------------------

③ `mul_ps X_reg,X_reg,a_reg`

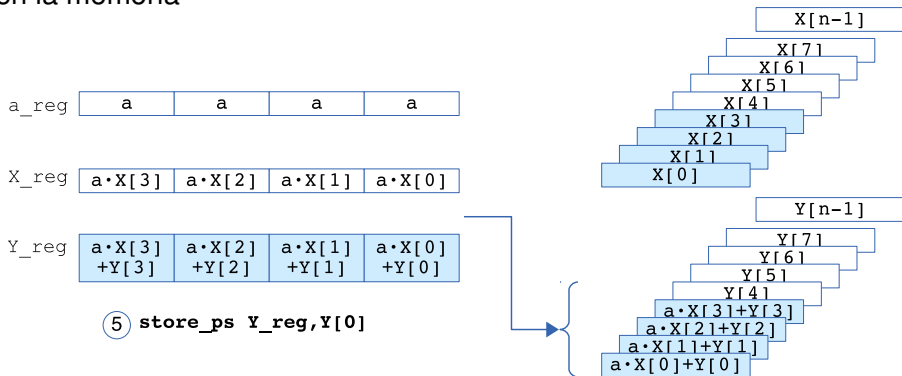
④ `add_ps Y_reg,X_reg,Y_reg`



## 8. Instrucciones SIMD

### Ejemplo 1: SAXPY con instrucciones SIMD (cont.)

La quinta instrucción actualiza las cuatro primeras componentes  $Y[i]$  en la memoria



## 8. Instrucciones SIMD

### Ejemplo 1: SAXPY con instrucciones SIMD (cont.)

Una sola iteración del bucle con instrucciones SIMD equivale a cuatro iteraciones con instrucciones no vectoriales.

#### Con instrucciones SIMD

Hay que iterar  $n/4$  veces

En total:  $n/2$  instrucciones de carga,  $n/4$  instrucciones de multiplicación,  $n/4$  instrucciones de suma y  $n/4$  instrucciones de almacenamiento

## 8. Instrucciones SIMD

### Soporte del compilador

La compilación se basa en los tipos de datos y en las expresiones para generar código.

- Si el programador expresa el código de forma escalar un compilador clásico no utilizará las instrucciones SIMD

Tres maneras de insertar instrucciones SIMD en el código:

#### Vectorización automática

El programador no explicita el paralelismo

Un compilador avanzado extrae el paralelismo del código fuente escalar

#### Usar tipos de datos SIMD

El programador define las variables de interés con un tipo específico y las utiliza en expresiones comunes del lenguaje

#### *Intrinsic functions*

El lenguaje dispone de una biblioteca de funciones SIMD

El programador las utiliza explícitamente

### Compilación del ejemplo 1

#### Vectorización automática

El código fuente no contiene ninguna referencia a la vectorización

```
void SAXPY(int n, float a, float *X, float *Y) {  
    int i;  
    for(i=0; i<n; i++)  
        Y[i] = a*X[i] + Y[i];  
}
```

La opción de compilación -O3

```
gcc -O3 saxpy.c -o saxpy
```

optimizará el código (si puede) insertando instrucciones SIMD

### Compilación del ejemplo 1 (cont.)

#### Con tipos de datos SIMD

El tipo “`__m128`” permite especificar bloques de 4 elementos.

```
void saxpy(int n, float a, float *X, float *Y) {  
    __m128 *x_ptr, *y_ptr;  
    int i;  
    x_ptr = (__m128 *) X;  
    y_ptr = (__m128 *) Y;  
    for (i=0; i<n/4; i++)  
        y_ptr[i] = a*x_ptr[i] + y_ptr[i];  
}
```

El compilador deduce que la expresión “`a*x_ptr[i] + y_ptr[i]`” se compila en instrucciones SIMD.



### Compilación del ejemplo 1 (cont.)

#### *Intrinsic functions*

Hay bibliotecas que permiten insertar código específico.

Las funciones “`_mm_load_ps`”, “`_mm_add_ps`”, “`_mm_mul_ps`” y “`_mm_store_ps`” se traducen en las instrucciones SIMD apropiadas.

```
void saxpy(int n, float a, float *X, float *Y) {  
    __m128 x_vec, y_vec, a_vec;  
    int i;  
    a_vec = _mm_set1_ps(a);  
    for (i=0; i<n; i+=4) {  
        x_vec = _mm_load_ps(&X[i]);  
        y_vec = _mm_load_ps(&Y[i]);  
        x_vec = _mm_mul_ps(a_vec, x_vec);  
        y_vec = _mm_add_ps(x_vec, y_vec);  
        _mm_store_ps(&Y[i], y_vec);  
    }  
}
```

## 8. Instrucciones SIMD

### Ejemplo 2: Producto escalar

#### Código convencional

```
float Scalar(int n, float *X, float *Y) {  
    int i;  
    float prod = 0.0;  
  
    for(i=0; i<n; i++)  
        prod += X[i] * Y[i];  
    return prod;  
}
```

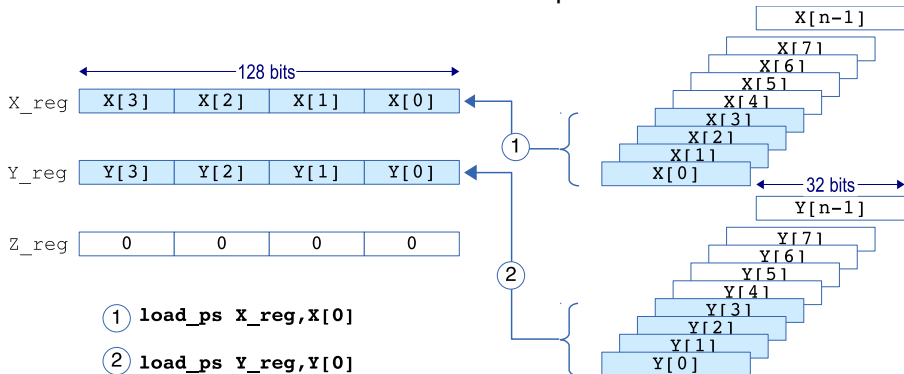
#### Sin instrucciones SIMD

- $n$  iteraciones,
- $n$  multiplicaciones y
- $n$  sumas.

## 8. Instrucciones SIMD

### Ejemplo 2: Producto escalar (cont.)

Vectorización: Cada iteración leerá dos bloques...



## 8. Instrucciones SIMD

### Ejemplo 2: Producto escalar (cont.)

...y hará cuatro productos y cuatro sumas.

X\_reg

X[3]	X[2]	X[1]	X[0]
------	------	------	------

Y\_reg

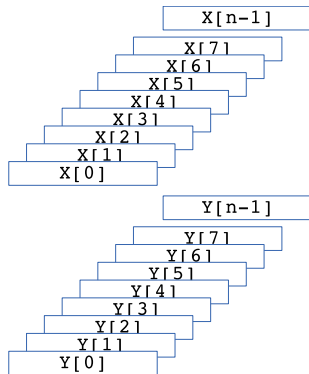
X[3] •Y[3]	X[2] •Y[2]	X[1] •Y[1]	X[0] •Y[0]
---------------	---------------	---------------	---------------

Z\_reg

X[3] •Y[3]	X[2] •Y[2]	X[1] •Y[1]	X[0] •Y[0]
---------------	---------------	---------------	---------------

③ `mul_ps Y_reg,X_reg,Y_reg`

④ `add_ps Z_reg,Z_reg,Y_reg`



## 8. Instrucciones SIMD

### Ejemplo 2: Producto escalar (cont.)

Al final de las  $n/4$  iteraciones, en el registro  $Z\_reg$  quedarán, para  $i = 0 \dots (\frac{n}{4} - 1)$ :

$\sum(X_{4i+3} \cdot Y_{4i+3})$ ,  $\sum(X_{4i+2} \cdot Y_{4i+2})$ ,  $\sum(X_{4i+1} \cdot Y_{4i+1})$  y  $\sum(X_{4i} \cdot Y_{4i})$   
El producto escalar resulta de sumar los cuatro valores.

### Con instrucciones SIMD

- $n/4$  iteraciones,
- $n/4$  multiplicaciones y
- $4 + n/4$  sumas.

### Ejemplo 2: Producto escalar (cont.)

#### Código SIMD

```
float Scalar(int n, float *X, float *Y) {  
    float prod = 0.0;  
    int i;  
    __m128 X_reg, Y_reg, Z_reg;  
    Z_reg = _mm_setzero_ps();  
    for(i=0; i<n; i+=4) {  
        X_reg = _mm_load_ps(&X[i]);  
        Y_reg = _mm_load_ps(&Y[i]);  
        Y_reg = _mm_mul_ps(X_reg, Y_reg);  
        Z_reg = _mm_add_ps(Y_reg, Z_reg);  
    }  
    for(i=0; i<4; i++)  
        prod += Z_reg[i];  
    return prod;  
}
```