

Arquitecturas de Computadores

Arquitectura de Computadores – IIC2343

Nuestro computador básico tiene todas las funcionalidades necesarias para ejecutar programas, p.ej., los escritos en lenguajes de alto nivel:

- registros, unidad de ejecución, unidad de control + memoria
- puede ejecutar un programa almacenado en memoria
- puede hacer cálculos aritméticos/lógicos, puede controlar el flujo de ejecución
- permite programar con funciones

La arquitectura de nuestro computador básico es una de muchas arquitecturas posibles

Técnicamente, la arquitectura de un computador está definida por lo que se llama su *microarquitectura* y por lo que se llama la *arquitectura del set de instrucciones*

La **microarquitectura** es el conjunto de componentes de hardware del computador y la forma en la que éstos están interconectados

La **arquitectura del set de instrucciones (ISA)** es la forma, propiedades, etc. de las instrucciones que el computador es capaz de ejecutar

lenguajes orientados o problemas

lenguaje *assembly*

máquina del sistema operativo

arquitectura del set de
instrucciones (ISA)

microarquitectura

lógica digital

5) lenguajes de alto nivel, traducidos por compiladores, o interpretados

4) forma simbólica de los lenguajes de más abajo, traducida por el assembler

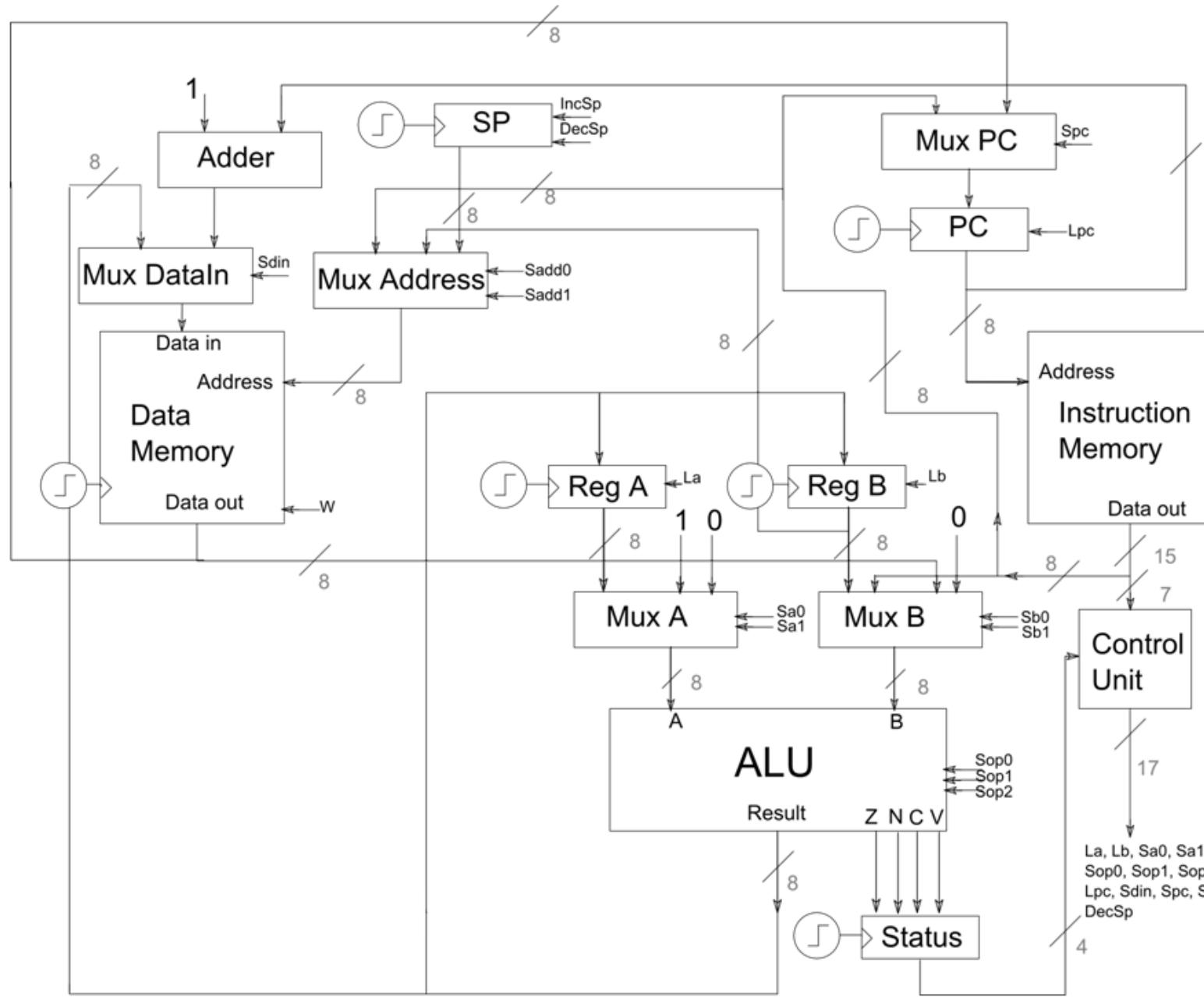
3) instrucciones adicionales, otra organización de memoria, capacidad de ejecución concurrente → interpretado por un programa llamado el *sistema operativo*

2) instrucciones ejecutadas por el microprograma o circuitos del hardware

1) registros + ALU → *datapath*, controlada por hardware o microprogramada

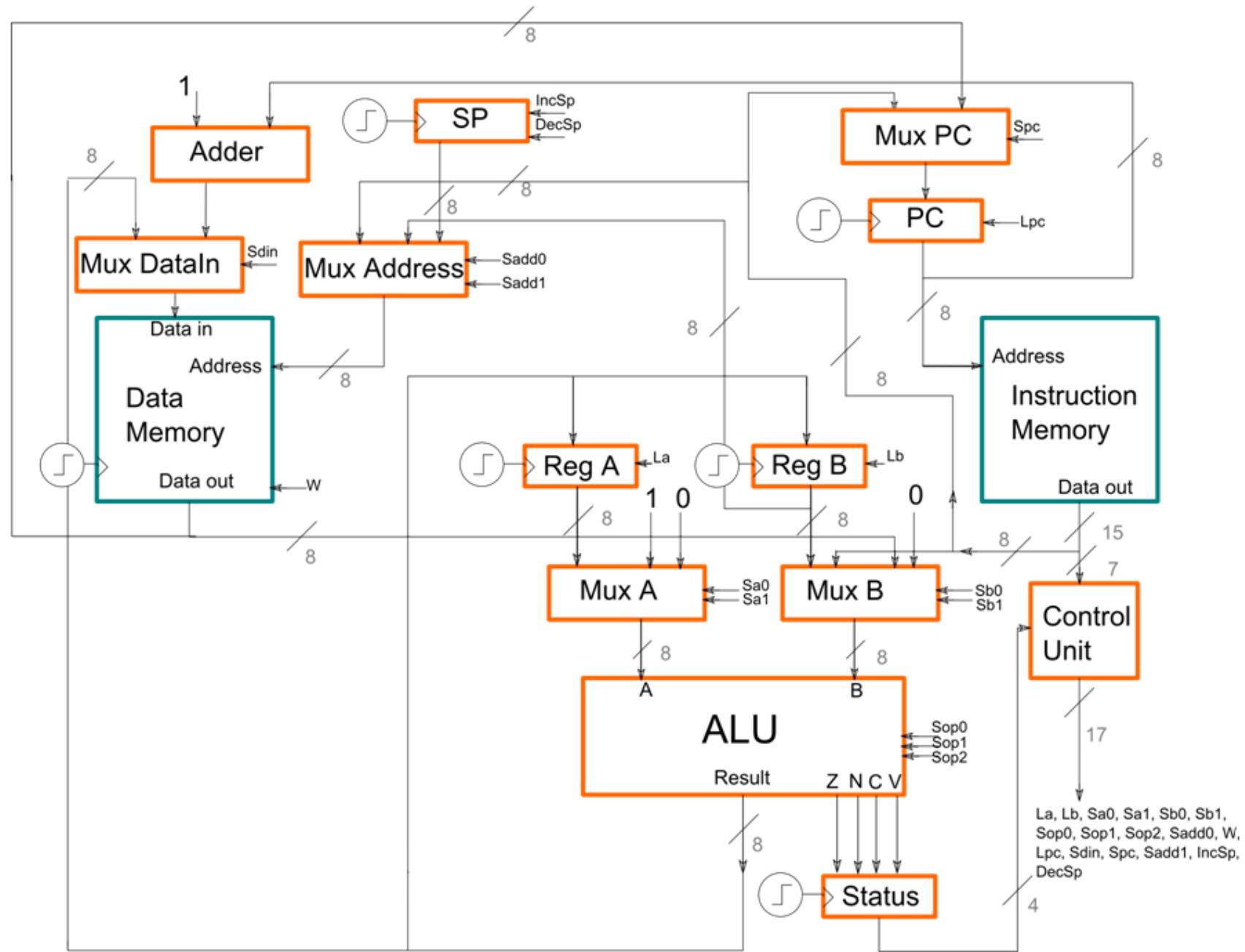
0) compuertas (*gates*), álgebra de Boole, registros, el motor de computación

La microarquitectura del computador básico (en diagrama):

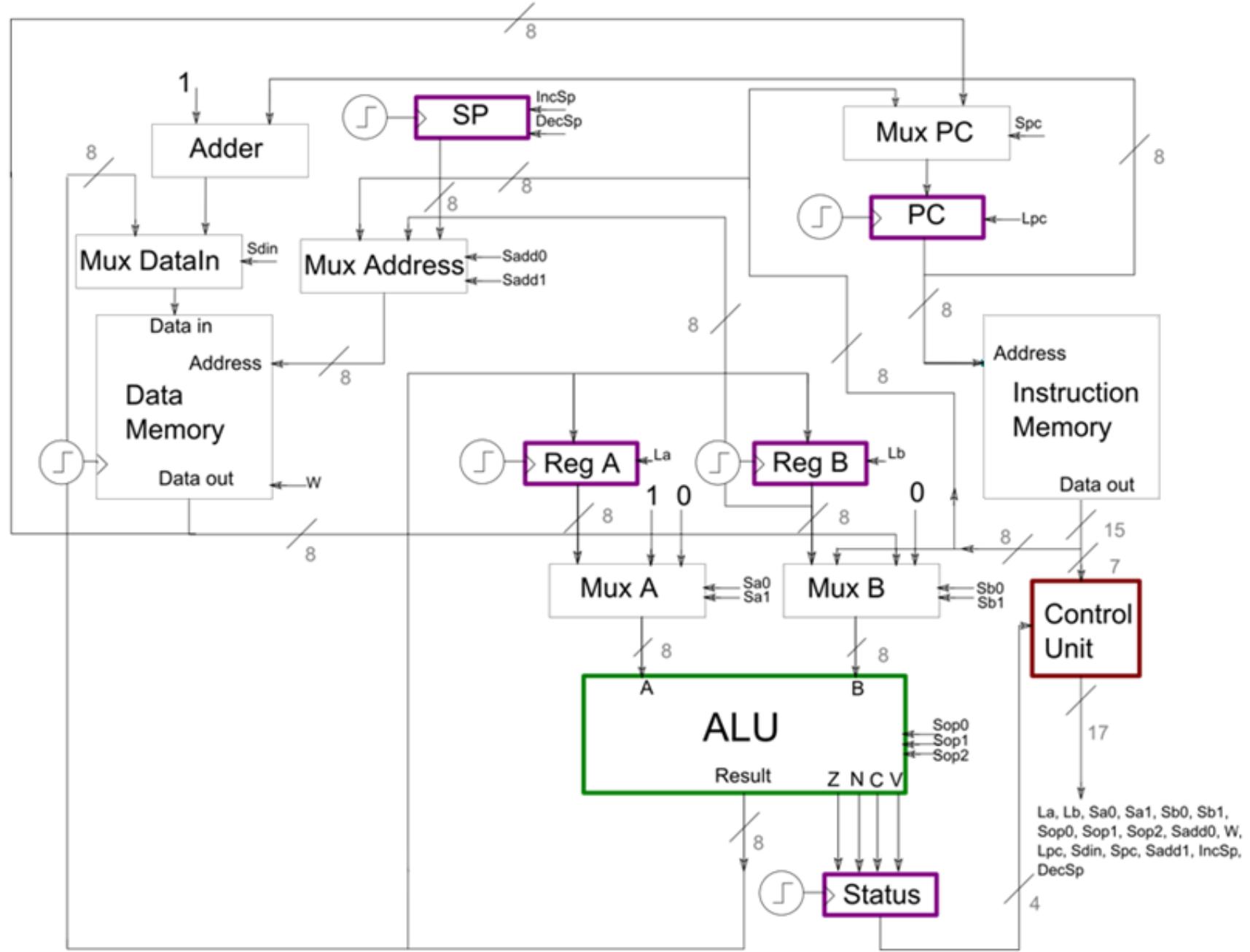


Procesador (CPU)

Memorias



Registros, Unidad de ejecución, Unidad de control



La microarquitectura del computador básico (en palabras):

registros: *A, B, SP, PC, Status*

unidad de control: simple (circuito digital *hardwired*)

unidad de ejecución: ALU (operaciones aritméticas/lógicas sobre enteros)

códigos de condición: *Z, N, C, V* (en el registro *Status*)

memoria: *Instruction Memory + Data Memory* (arquitectura *Harvard*)

stack: en la *Data Memory*

tamaños: registros, direcciones de memoria, etc. de 8 bits

Nota sobre arquitecturas *Harvard* y *von Neumann*:

- difieren sólo en cómo se almacenan los programas y los datos

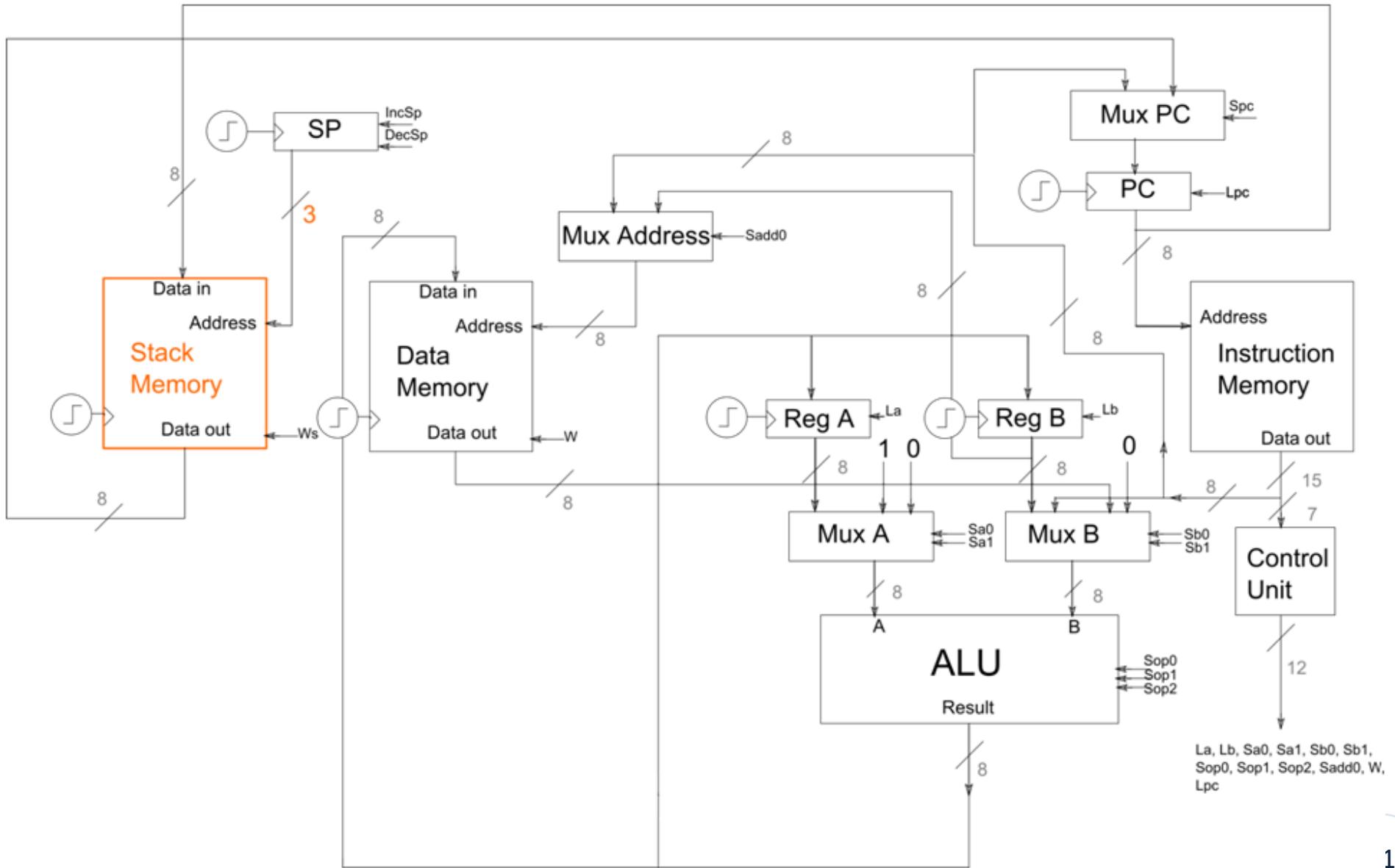
En **Harvard** hay una *Instruction Memory* y una *Data Memory* —como en el computador básico:

- permite que cada memoria sea optimizada para su propósito
... pero es inflexible y obliga a especificar por adelantado cuánta memoria de instrucciones necesitas
- se usa en microcontroladores y algunos computadores embebidos (p.ej., ver apuntes de Alejandro y Hans sobre el microcontrolador PIC16F877A)

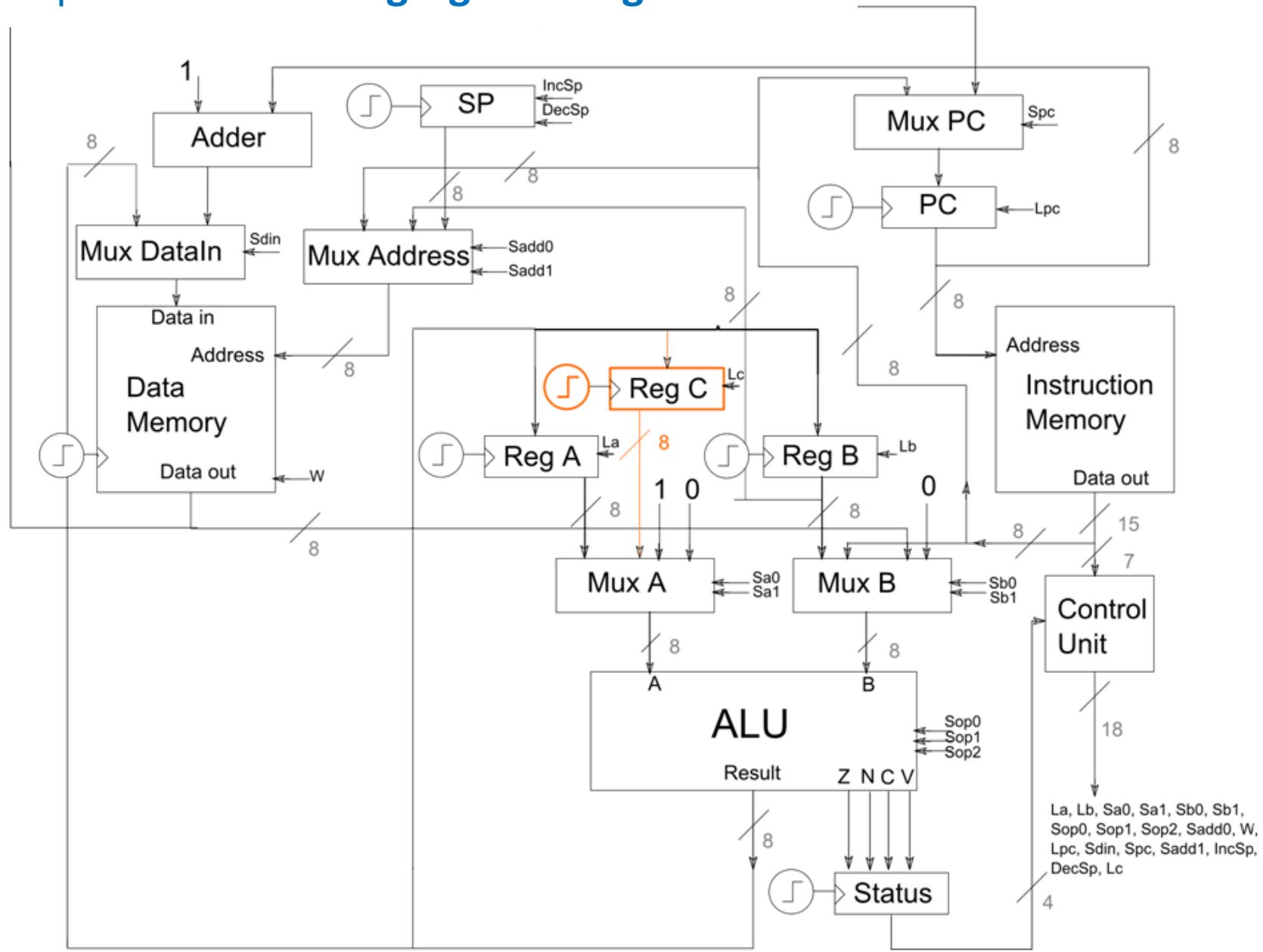
En **von Neumann** hay una única *Memory*:

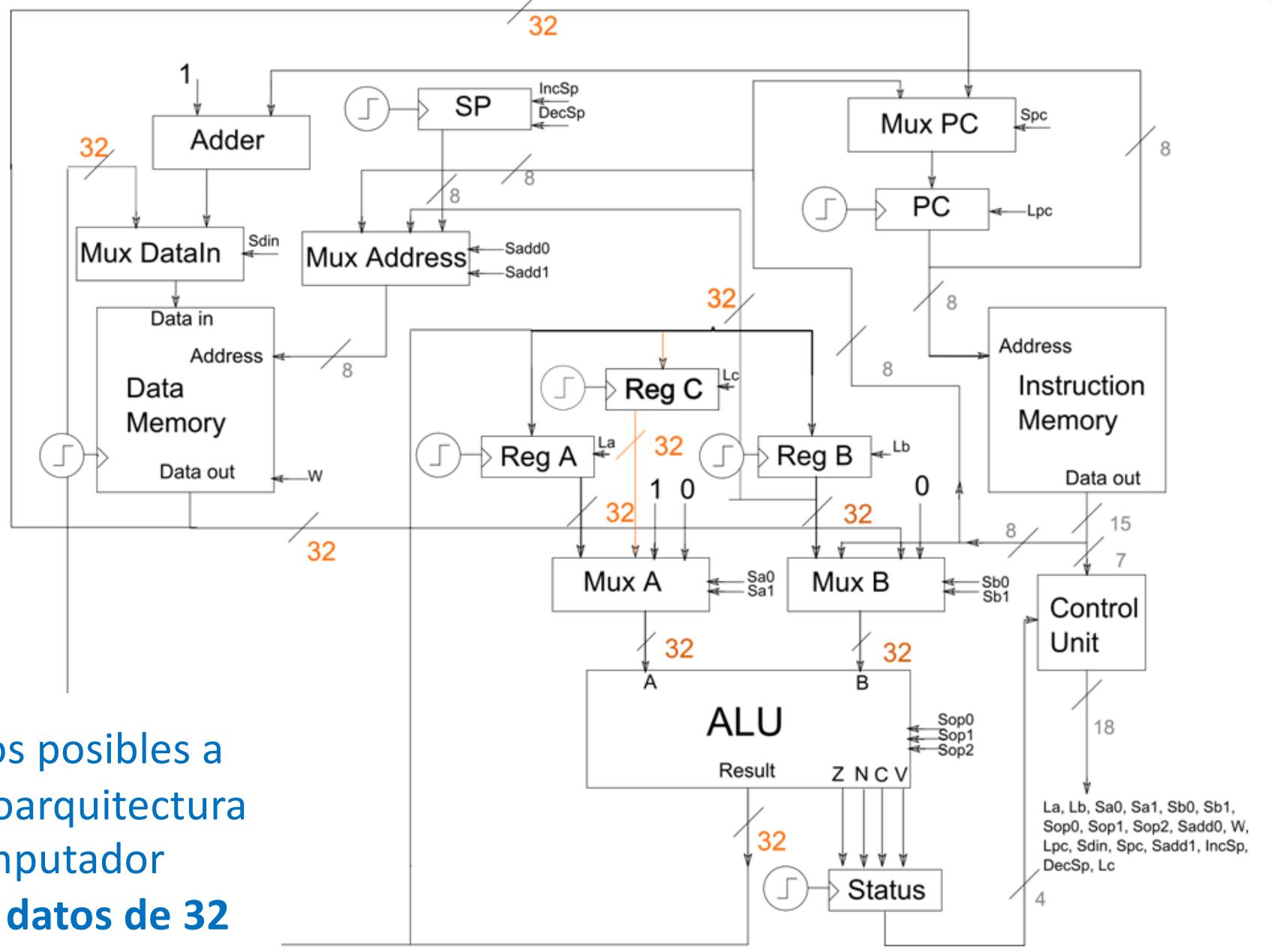
- su ventaja es flexibilidad y además permite que los programas sean cargados en memoria al igual que cualquier dato
- ha sido ampliamente adoptada en casi todos los computadores modernos

Cambios posibles a la microarquitectura del computador básico: agregar una memoria especializada para el manejo del stack

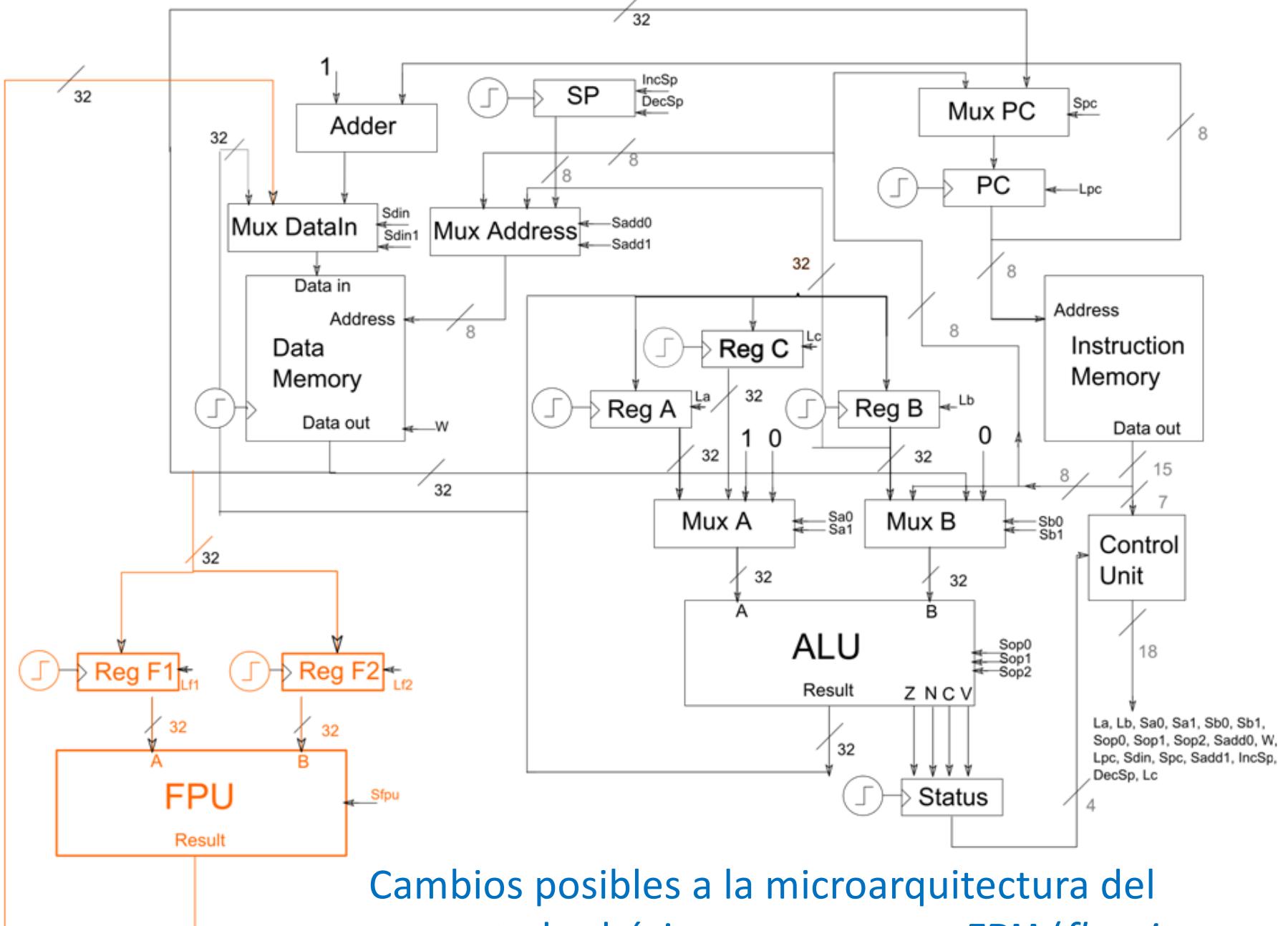


Cambios posibles a la microarquitectura del computador básico: agregar un registro C





Cambios posibles a
la microarquitectura
del computador
básico: **datos de 32
bits** (en lugar de 8)



Cambios posibles a la microarquitectura del computador básico: **agregar una FPU (floating point unit) y los registros correspondientes**

La arquitectura del set de instrucciones (ISA) del computador básico:

formato de instrucciones: *opcode* + cero, uno o dos operandos, 15 bits

tipos de instrucciones: aritméticas, lógicas, acceso a memoria

direccionamiento: inmediato, directo, por registro, indirecto por registro

tipos de datos: números enteros con y sin signo

flujo de control: secuencial, condicional (*jumps*), *call* y *return* (funciones)

ciclos por instrucción: uno (sólo RET, POP A y POP B requieren dos ciclos)

Instrucción	Operandos	Opcode	Condition	Lpc	La	Lb	Sa0,1	Sb0,1	Sop0,1,2	Sadd0,1	Sdin0	Spc0	W	IncSp	DecSp
MOV	A,B	0000000		0	1	0	ZERO	B	ADD	-	-	-	0	0	0
	B,A	0000001		0	0	1	A	ZERO	ADD	-	-	-	0	0	0
	A,Lit	0000010		0	1	0	ZERO	LIT	ADD	-	-	-	0	0	0
	B,Lit	0000011		0	0	1	ZERO	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0000100		0	1	0	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	B,(Dir)	0000101		0	0	1	ZERO	DOUT	ADD	LIT	-	-	0	0	0
	(Dir),A	0000110		0	0	0	A	ZERO	ADD	LIT	ALU	-	1	0	0
	(Dir),B	0000111		0	0	0	ZERO	B	ADD	LIT	ALU	-	1	0	0
	A,(B)	0001000		0	1	0	ZERO	DOUT	ADD	B	-	-	0	0	0
	B,(B)	0001001		0	0	1	ZERO	DOUT	ADD	B	-	-	0	0	0
	(B),A	0001010		0	1	0	A	ZERO	ADD	B	ALU	-	1	0	0
ADD	A,B	0001011		0	1	0	A	B	ADD	-	-	-	0	0	0
	B,A	0001100		0	0	1	A	B	ADD	-	-	-	0	0	0
	A,Lit	0001101		0	1	0	A	LIT	ADD	-	-	-	0	0	0
	A,(Dir)	0001110		0	1	0	A	DOUT	ADD	LIT	-	-	0	0	0
	A,(B)	0001111		0	0	1	A	DOUT	ADD	B	-	-	0	0	0
	(Dir)	0010000		0	0	0	A	B	ADD	LIT	ALU	-	1	0	0
SUB	A,B	0010001		0	1	0	A	B	SUB	-	-	-	0	0	0
	B,A	0010010		0	0	1	A	B	SUB	-	-	-	0	0	0
	A,Lit	0010010		0	1	0	A	LIT	SUB	-	-	-	0	0	0
	A,(Dir)	0010011		0	1	0	A	DOUT	SUB	LIT	-	-	0	0	0
	A,(B)	0010100		0	1	0	A	DOUT	SUB	B	-	-	0	0	0
	(Dir)	0010101		0	0	0	A	B	SUB	LIT	ALU	-	1	0	0
AND	A,B	0010110		0	1	0	A	B	AND	-	-	-	0	0	0
	B,A	0010111		0	0	1	A	B	AND	-	-	-	0	0	0
	A,Lit	0011000		0	1	0	A	LIT	AND	-	-	-	0	0	0
	A,(Dir)	0011001		0	1	0	A	DOUT	AND	LIT	-	-	0	0	0
	A,(B)	0011010		0	1	0	A	DOUT	AND	B	-	-	0	0	0
	(Dir)	0011011		0	0	0	A	B	AND	LIT	ALU	-	1	0	0
OR	A,B	0011100		0	1	0	A	B	OR	-	-	-	0	0	0
	B,A	0011101		0	0	1	A	B	OR	-	-	-	0	0	0
	A,Lit	0011110		0	1	0	A	LIT	OR	-	-	-	0	0	0
	A,(Dir)	0011111		0	1	0	A	DOUT	OR	LIT	-	-	0	0	0
	A,(B)	0100000		0	1	0	A	DOUT	OR	B	-	-	0	0	0
	(Dir)	0100001		0	0	0	A	B	IR	LIT	ALU	-	1	0	0
NOT	A,A	0100010		0	1	0	A	-	NOT	-	-	-	0	0	0
	B,A	0100011		0	0	1	A	-	NOT	-	-	-	0	0	0
	(Dir)	0100111		0	0	0	A	B	NOT	LIT	ALU	-	1	0	0

Instrucción	Operandos	Opcode	Condition	Lpc	La	Lb	Sa0,1	Sb0,1	Sop0,1,2	Sadd0,1	Sdin0	Spc0	W	IncSp	DecSp
XOR	A,B	0101000		0	1	0	A	B	XOR	-	-	-	0	0	0
	B,A	0101001		0	0	1	A	B	XOR	-	-	-	0	0	0
	A,Lit	0101010		0	1	0	A	LIT	XOR	-	-	-	0	0	0
	A,(Dir)	0101011		0	1	0	A	DOUT	XOR	LIT	-	-	0	0	0
	A,(B)	0101100		0	1	0	A	DOUT	XOR	B	-	-	0	0	0
	(Dir)	0101101		0	0	0	A	B	XOR	LIT	ALU	-	1	0	0
SHL	A,A	0101110		0	1	0	A	-	SHL	-	-	-	0	0	0
	B,A	0101111		0	0	1	A	-	SHL	-	-	-	0	0	0
	(Dir)	0110011		0	0	0	A	B	SHL	LIT	ALU	-	1	0	0
SHR	A,A	0110100		0	1	0	A	-	SHR	-	-	-	0	0	0
	B,A	0110101		0	0	1	A	-	SHR	-	-	-	0	0	0
	(Dir)	0111001		0	0	0	A	B	SHR	LIT	ALU	-	1	0	0
INC	B	0111010		0	0	1	ONE	B	ADD	-	-	-	0	0	0
CMP	A,B	0111011		0	0	0	A	B	SUB	-	-	-	0	0	0
	A,Lit	0111100		0	0	0	A	LIT	SUB	-	-	-	0	0	0
JMP	Dir	0111101		1	0	0	-	-	-	-	-	LIT	0	0	0
JEQ	Dir	0111110	Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JNE	Dir	0111111	Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JGT	Dir	1000000	N=0 y Z=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLT	Dir	1000001	N=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JGE	Dir	1000010	N=0	1	0	0	-	-	-	-	-	LIT	0	0	0
JLE	Dir	1000011	N=1 o Z=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JCR	Dir	1000100	C=1	1	0	0	-	-	-	-	-	LIT	0	0	0
JOV	Dir	1000101	V=1	1	0	0	-	-	-	-	-	LIT	0	0	0
CALL	Dir	1000101		1	0	0	-	-	-	SP	PC	LIT	1	0	1
RET		1000110		0	0	0	-	-	-	-	-	-	0	1	0
		1000111		1	0	0	-	-	-	SP	-	DOUT	0	0	0
PUSH	A	1001000		0	0	0	A	ZERO	ADD	SP	ALU	-	1	0	1
PUSH	B	1001001		0	0	0	ZERO	B	ADD	SP	ALU	-	1	0	1
POP	A	1001010		0	1	0	-	-	-	-	-	-	0	1	0
		1001011		0	1	0	ZERO	DOUT	ADD	SP	ALU	-	0	0	0
POP	B	1001100		0	0	1	-	-	-	-	-	-	0	1	0
		1001101		0	0	1	ZERO	DOUT	ADD	SP	ALU	-	0	0	0

La arquitectura del set de instrucciones (ISA) (en general)

El nivel ISA se podría definir según cómo ve a la máquina un programador de lenguaje de máquina:

- pero hoy no se programa mucho en lenguaje de máquina

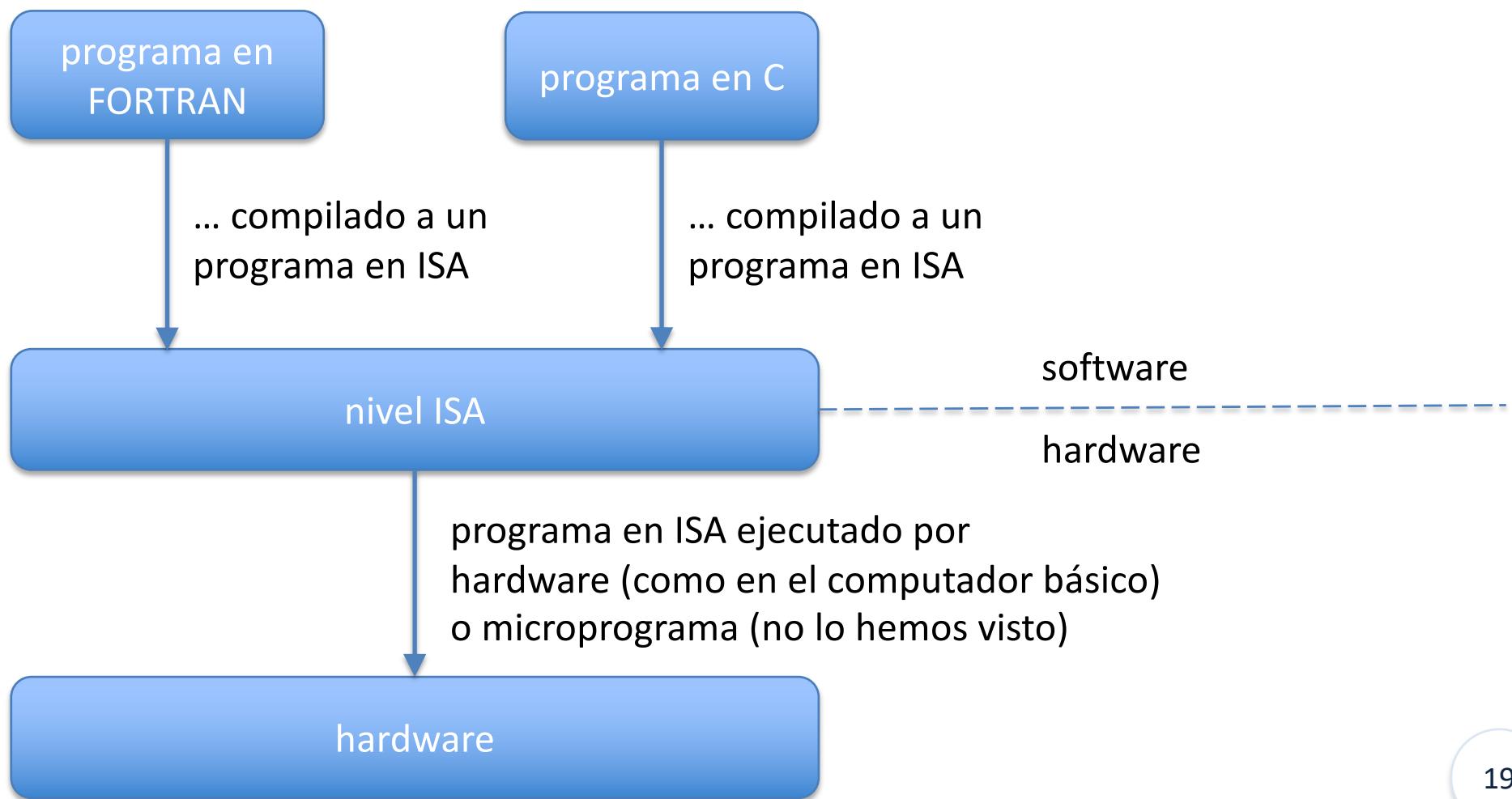
Hoy definimos el nivel ISA así:

el código del nivel ISA es el código que produce un compilador:

- el programador del compilador tiene que saber qué instrucciones están disponibles
 - ... qué registros hay
 - ... cuál es el modelo de memoria
 - ... qué tipos de datos están disponibles, etc.

Así, el nivel ISA es la interfaz entre el software (los compiladores) y el hardware —es el lenguaje que ambos tienen que entender:

- permite que los computadores puedan ejecutar programas escritos en múltiples lenguajes de alto nivel



Las instrucciones propiamente tales son el componente principal del nivel ISA :

- qué instrucciones hay y qué formato tienen

... pero también es necesario tener claros otros aspectos:

- el modelo de memoria
- la cantidad y tipos de los registros
- los modos de direccionamiento
- los tipos de datos

El formato de las instrucciones es la representación binaria de las instrucciones

Una instrucción consiste en un *opcode*:

- qué hace la instrucción —se especifica mediante un número único para cada instrucción, es decir, un *código*

... más información adicional:

- típicamente, dónde se encuentran los *operандos* —es decir, dónde se encuentran los valores necesarios para ejecutar la operación
- ... y a dónde va a parar el *resultado* —un operando adicional

opcode | *operando1* | *operando2* | ...

Una instrucción siempre tiene un *opcode*, y puede tener cero, uno, dos o más operandos

Todas las instrucciones pueden ser del mismo largo —mismo número de bits	... o bien el largo de las instrucciones puede variar
<ul style="list-style-type: none"> • desperdicia espacio, ya que hay instrucciones que evidentemente podrían ser más cortas, por lo que se ve como ineficiente • ... pero hace que el hardware sea más simple, facilitando la lectura (<i>fetch</i>) y decodificación de las instrucciones • los bits que no se usan de una instrucción simplemente son ignorados por el hardware (p.ej., en el computador básico, los 8 bits del literal) 	<ul style="list-style-type: none"> • hay procesadores en que algunas instrucciones ocupan una palabra, otras ocupan media y hasta un cuarto de palabra, y aún otras ocupan dos palabras • el hardware necesario para leer y decodificar una instrucción es más complejo

En general, instrucciones más cortas (menos bits) son mejores que instrucciones más largas:

- n instrucciones de 16 bits ocupan la mitad de espacio en memoria que n instrucciones de 32 bits
- el ancho de banda de la memoria no ha progresado lo mismo que la velocidad de los procesadores —las memorias (y las *caches*) no son capaces de entregar instrucciones y operandos tan rápidamente como el procesador los puede usar

Por otra parte, la instrucción debe tener suficiente espacio para poder expresar todas las operaciones necesarias:

- n bits en el *opcode* significa un máximo de 2^n operaciones
- ... y siempre hay que considerar la necesidad de no usar todos los *opcodes* inicialmente para poder agregar instrucciones en el futuro

Finalmente, supongamos una memoria de 2^{32} bytes

... la unidad de direccionamiento podría ser el byte (de 8 bits):

- memoria de 2^{32} bytes, numerados 0 al 4,295 millones
- **cada dirección necesita 32 bits**
- p.ej., para comparar dos caracteres, simplemente los lee

... o, en el otro extremo, la palabra (de 4 bytes = 32 bits):

- memoria de 2^{30} palabras, numeradas 0 al 1,074 millones
- **cada dirección necesita solo 30 bits** → instrucciones más cortas y más rápidas de leer

... o bien, manteniendo direcciones de 32 bits, puede direccionar 16 GB de memoria (y no sólo 4 GB)

- p.ej., para comparar dos caracteres tiene que leer dos palabras y extraer los caracteres (de 8 bits) de cada una → toma instrucciones adicionales

Los **tipos de instrucciones** tienen que ver con lo que las instrucciones hacen:

- movimiento de datos
- operaciones de dos operandos
- operaciones de un operando
- comparaciones, saltos condicionales, y control de loops
- llamadas a funciones
- input/output ... *pendiente*

En el caso del computador básico, hemos visto ejs. de prácticamente todos estos tipos de instrucciones —sólo nos queda pendiente ver el tema de las instrucciones de input y output

A continuación, unos puntos adicionales con respecto a las instrucciones de movimiento de datos y las operaciones de uno y dos operandos

Movimiento de datos (mejor dicho, duplicación de datos):

copiar un dato de un lugar a otro (en realidad, crear un nuevo objeto con un patrón de bits idéntico al original) es la operación más fundamental

- debido a asignaciones en el programa del usuario
- debido a la necesidad de llevar datos de la memoria a los registros (*load*) o vice versa (*store*), o copiar datos entre registros (*move*), ya que muchas instrucciones de máquina pueden tener acceso a variables sólo cuando éstas están en los registros

Operaciones de dos operandos:

combinan dos operandos para producir un resultado

- suma, resta, multiplicación y división de números enteros
- funciones booleanas de dos variables —*and, or, not, exclusive or, nor, nand*:

p.ej., para extraer el segundo carácter de una palabra *P* de 4 bytes y dejarlo como el único carácter en otra palabra —el de más a la derecha, con puros ceros en los otros bytes:

10110111	10111100	11011011	10001011	P
<u>00000000</u>	<u>11111111</u>	<u>00000000</u>	<u>00000000</u>	Q (<i>mask</i>)
00000000	10111100	00000000	00000000	P AND Q

... y este resultado lo desplazamos (*shift*) 16 bits a la derecha

- operaciones de punto flotante

Operaciones de un operando:

tienen un solo operando y producen un resultado

- la función booleana *not*
- *shift* a la derecha o a la izquierda —*shift* a la derecha puede mantener el signo o no

shift a la izquierda (*left shift*) de k bits → multiplicación por 2^k

shift a la derecha (*right shift*) de k bits → división por 2^k (excepto en el caso de números negativos)

- *rotate* (*shift* en que los bits que salen por un extremo reaparecen en el otro)
p.ej., para chequear cada bit de una palabra, poniéndolos uno por uno en la posición del bit de signo, y restaurando el valor de la palabra al final
- operaciones de dos operandos en que uno de ellos es muy común
p.ej., asignar 0 (**CLR address**) o sumar 1 (**INC address**) a una variable

El **modelo de memoria** es cómo “ve” la memoria una instrucción

Todos los computadores dividen la memoria en celdas, normalmente de 8 bits —un **byte**— que tienen direcciones consecutivas:

Los bytes son agrupados en **palabras** de 4 u 8 bytes

... y hay instrucciones para manejar palabras enteras

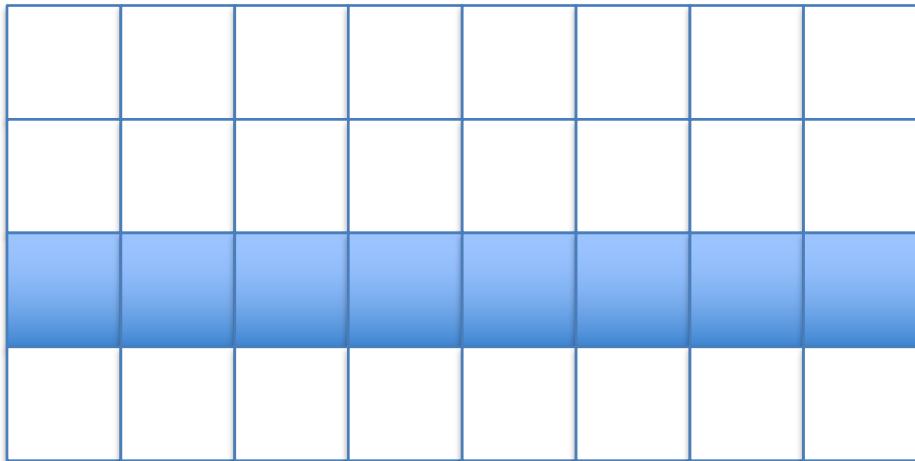
Muchas arquitecturas requieren que las palabras estén alineadas según su límite natural —**alineación**:

- p.ej., las de 4 bytes comienzan en las direcciones 0, 4, 8, etc.
- las memorias operan más eficientemente de esta manera
 - ... p.ej., el Core i7 lee 8 bytes a la vez,
 - ... y la interfaz de memoria requiere direcciones que sean múltiplos de 8

... aunque a veces esto causa problemas:

- en el mismo Core i7, los programas pueden hacer referencia a palabras que empiezan en cualquier dirección, no sólo múltiplos de 8 (ya que en el procesador 8088 el bus de datos era de un byte)

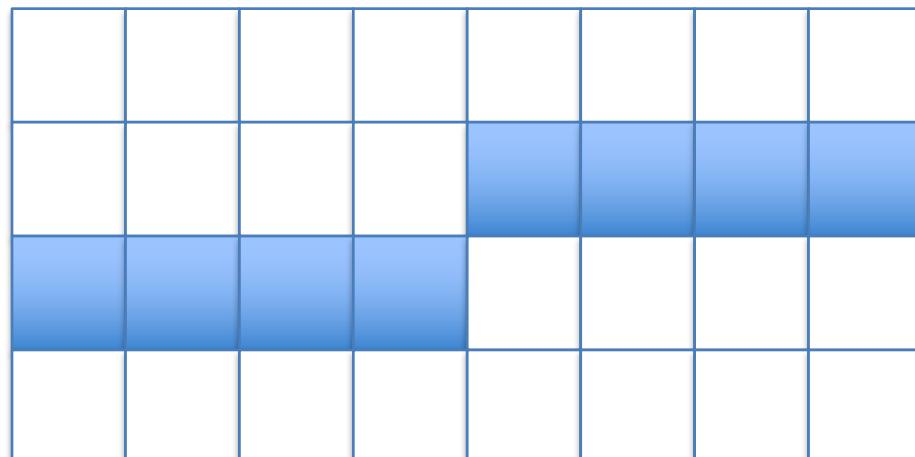
El significado de la alineación y el problema que puede presentar, se ilustran en la próxima diap.



24
16
8
0

palabra de 8 bytes **alineada**
que empieza en la dirección 8:
requiere un acceso a memoria

palabra de 8 bytes **no alineada**
que empieza en la dirección 12:
requiere dos accesos a memoria
y luego armar la palabra



24
16
8
0

Como acabamos de ver, la mayoría de las instrucciones tienen operandos

Direccionamiento es cómo especificar en la instrucción —en bits que son parte de la instrucción— dónde están los operandos y a dónde va a parar el resultado:

- hay varias formas de interpretar los bits que corresponden a los operandos
—los *modos de direccionamiento*

Modos de direccionamiento (algunos):

- inmediato
- directo
- por registro
- indirecto por registro

Direccionamiento **inmediato**

La parte de la instrucción en que (normalmente) se especifica una dirección en este caso *contiene al operando propiamente tal* (en lugar de su dirección):

- el operando es automáticamente traído desde la memoria (la *Instruction Memory*) al mismo tiempo que (el resto de) la instrucción
 - ... por lo que está inmediatamente disponible para ser usado
 - ... y no requiere una referencia adicional a la memoria (la *Data Memory*)
- p.ej., en el computador básico **MOV A, Lit** carga el registro A con **el valor especificado en el campo *literal*** de la instrucción (los 8 bits adicionales al *opcode*)
- ¿desventajas?

Direccionamiento **directo**

Se especifica explícitamente la dirección de memoria del operando:

- la instrucción siempre va a tener acceso a la misma dirección de memoria en la *Data Memory* —puede cambiar el valor almacenado en la dirección, pero no la dirección
- sólo para tener acceso a variables cuya dirección es conocida al momento de compilar —típicamente, variables globales
- p.ej., en el computador básico **MOV A, (var)** carga el registro A con **Mem[var]**: **el contenido de la dirección de memoria var, dirección que está especificada en el campo literal de la instrucción**
- ¿es útil?

Direccionamiento por registro (*modo registro*)

Conceptualmente igual a direccionamiento directo,

... pero especifica un registro en lugar de una dirección de memoria:

- modo de direccionamiento más común
- las variables que van a ser usadas más a menudo (p.ej., el índice de un *loop*) van en los registros
- p.ej., en el computador básico **MOV A, B** carga el registro *A* con **el contenido del registro *B*** (en el computador básico, simplemente el *opcode* especifica la instrucción completa; los bits correspondientes al *literal* son ignorados)

P.ej., en la arquitectura ARM, (casi) todas las instrucciones usan solo este modo:

- el “casi” es debido a las instrucciones que llevan un dato de la memoria a un registro (**LDR**), o viceversa (**STR**)

Direccionamiento **indirecto por registro**

El operando viene de o va a la memoria (*la Data Memory*)

... la dirección de memoria correspondiente está contenida en un registro —este registro es un **puntero**

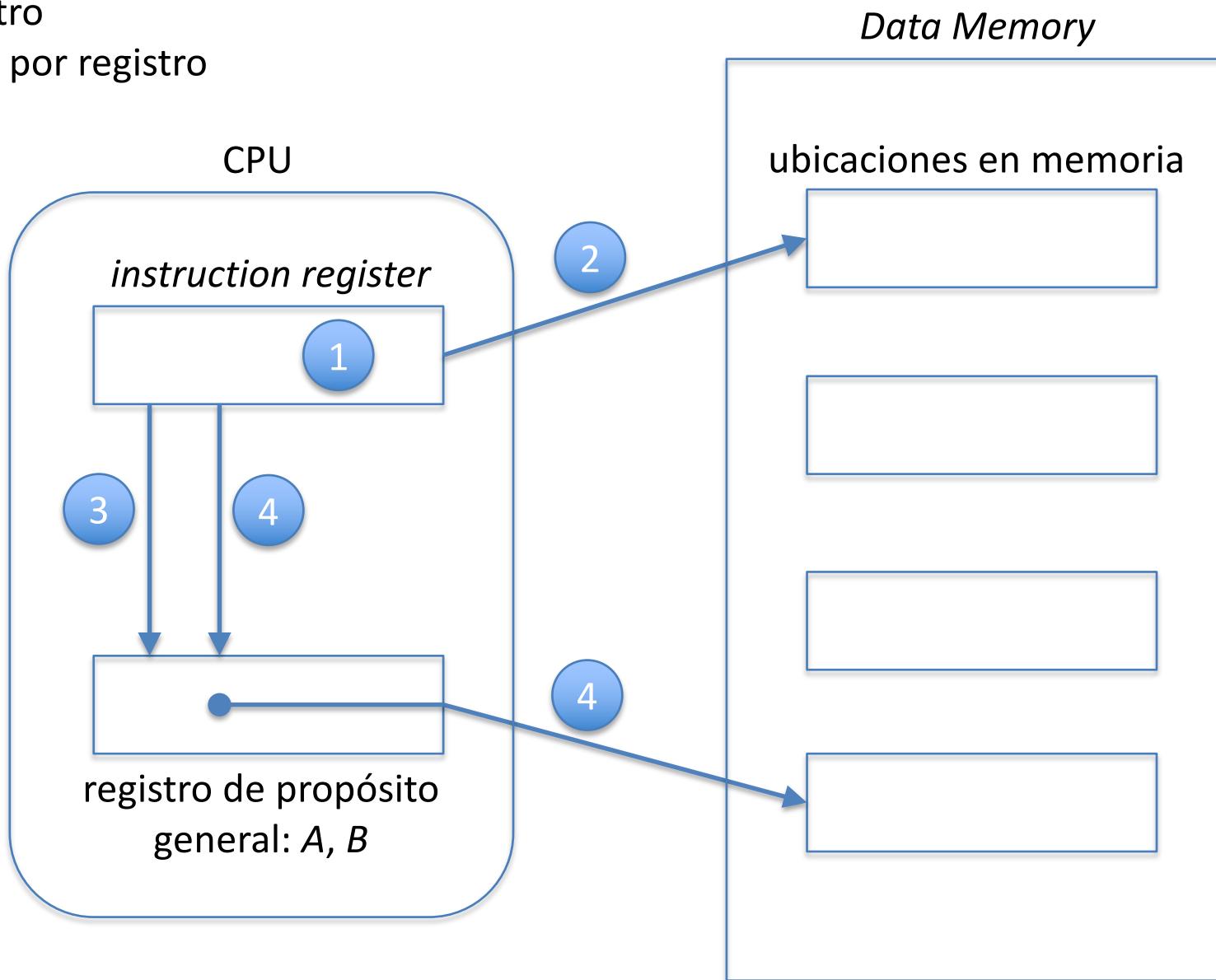
... y es este registro el que se especifica en la instrucción:

- p.ej., en el computador básico **MOV A, (B)** carga el registro *A* con **Mem[B]**: el contenido de la dirección de memoria almacenada en el registro *B*

Ventajas:

- puede hacer referencia a la memoria sin tener que especificar explícitamente una dirección de memoria —p.ej., de 32 bits— en la instrucción
- puede hacer referencia a diferentes direcciones de memoria en diferentes ejecuciones de la instrucción

- 1) Inmediato
- 2) Directo
- 3) Por registro
- 4) Indirecto por registro



Finalmente, un tema clave en el diseño de una ISA es si hay soporte de hardware para un **tipo de datos** en particular:

- si hay instrucciones que esperan datos en un formato particular
 - ... entonces el usuario no es libre de elegir otro formato
- p.ej., los números enteros con signo exigen que el signo sea el bit más significativo

Los tipos de datos pueden ser divididos en dos categorías:

- numéricos
- no numéricos

El principal tipo de datos numérico son los **números enteros**:

- 8, 16, 32 y 64 bits
- sin signo, con signo
- p.ej., todas las combinaciones están en el Core i7

... también se usan **números de punto flotante**:

- 32, 64 y 128 bits
- muchos computadores tienen registros separados para operandos enteros y para operandos de punto flotante
- p.ej., el Core i7 implementa el estándar IEEE 754, en 32 y 64 bits

[Algunos lenguajes de programación —p.ej., COBOL— manejan números decimales (en base 10):

- dos dígitos decimales por byte, de 4 bits cada uno (¿es eficiente?)]

Los computadores manejan mucha información no numérica:

- e-mail, Web, fotos digitales, multimedia

Los **caracteres**:

- codificación *ASCII*, de 7 bits (más 1 bit de paridad)
- codificación *Unicode*, de 16 bits —codificación universal de los alfabetos de la mayoría de los lenguajes humanos; usado por Java
- suele haber instrucciones para strings: copia, búsqueda, edición, etc.

Los valores Boolean:

- 0 es falso; todo lo demás, verdadero
- bastaría con un bit, pero se usa un byte o una palabra (los bits no tienen dirección propia)
- ... excepto cuando hay un arreglo de valores Boolean, o *bit-map* (p.ej., para seguirle la pista a los bloques del disco)

Los punteros, es decir, direcciones de memoria:

- *stack pointer* y *program counter* (o *instruction pointer*) son punteros
- tener acceso a una variable a una distancia fija —un desplazamiento— de un puntero —una dirección base— es muy común en todos los computadores
- útiles, pero también son la fuente de muchos errores de programación con graves consecuencias —hay que usarlos con mucho cuidado

Dos grandes clases de sets de instrucciones

RISC (*Reduced Instruction Set Computer*):

- conjunto mínimo de instrucciones (p.ej., sólo 32 instrucciones) que sea suficiente para realizar todas las operaciones
- c/u ejecuta una operación básica, normalmente en un ciclo de reloj
- todas las instrucciones tienen el mismo tamaño
- p.ej., los sets de instrucciones de los procesadores ARM y MIPS

CISC (*Complex Instruction Set Computer*):

- muchas instrucciones (varios cientos)
- c/u puede ejecutar una operación arbitrariamente compleja, que puede tomar mucho tiempo
- las instrucciones pueden tener tamaños diferentes
- p.ej., el set de instrucciones de los procesadores Intel (IA-32) y AMD

El nivel ISA del procesador MIPS

Ej. de set de instrucciones RISC

Los procesadores MIPS son populares en sistemas embebidos

Memoria de 2^{30} palabras de 4 bytes c/u:

- cada byte tiene su propia dirección
→ las direcciones de las palabras van desde 0 hasta 4,294,967,292, de cuatro en cuatro

... en formato ***big endian***:

- el byte de más a la izquierda de la palabra tiene la dirección numéricamente menor

32 registros, de 32 bits c/u:

\$zero (registro número 0), el valor constante 0

\$at (registro número 1), reservado para el *assembler*

\$v0 - \$v1 (2, 3), para el valor de retorno de una función

\$a0 - \$a3 (4 a 7), para pasar parámetros en llamadas a funciones

\$t0 - \$t7 (8 a 15), registros temporales necesarios al compilar el programa

\$s0 - \$s7 (16 a 23), variables del programa en C u otro lenguaje de alto nivel

\$t8 - \$t9 (24, 25), más registros temporales

\$k0 - \$k1 (26, 27), reservados para el sistema operativo

\$gp (28), *global pointer*

\$sp (29), *stack pointer*

\$fp (30), *frame pointer*

\$ra (31), dirección de retorno

Las 32 instrucciones de un procesador MIPS

(las instrucciones aritméticas y lógicas son de tres operandos; los operandos no se incluyen en esta lista)

ADD

SUBSTRACT

ADD immediate

ADD unsigned

SUBSTRACT unsigned

ADD immediate unsigned

MOVE from coprocessor

MULTIPLY

MULTIPLY unsigned

DIVIDE

DIVIDE unsigned

MOVE from Hi

MOVE from Lo

AND

OR

AND immediate

OR immediate

SHIFT left logical

SHIFT right logical

LOAD word

STORE word

LOAD upper immediate

MOVE from coprocessor register

BRANCH equal

BRANCH not equal

SET on less than

SET less than immediate

SET less than unsigned

SET less than unsigned immediate

JUMP

JUMP register

JUMP and link

Puede parecer que las 32 instrucciones son insuficientes:

- p.ej., no hay instrucciones para copiar el contenido de un registro en otro
... ni para sumar un valor en memoria al contenido de un registro

... pero el diseño está basado en dos principios:

- **velocidad** (p.ej., acceso rápido al valor 0, ...)
- **minimalidad** ... almacenado permanentemente en el registro 0)

(Adicionalmente, hay un set de instrucciones de punto flotante —precisión simple y doble— y otros 32 registros de punto flotante)

En MIPS, todas las instrucciones son del mismo largo —**32 bits**—

... y hay tres formatos —R, I y J:

- la próxima diap. ilustra estos formatos, detallando sus campos y el número de bits de cada campo

... y da ejs. de cómo se usan estos campos en 6 instrucciones típicas

¿Qué significa cada campo?

op : el *opcode*, la operación básica de la instrucción

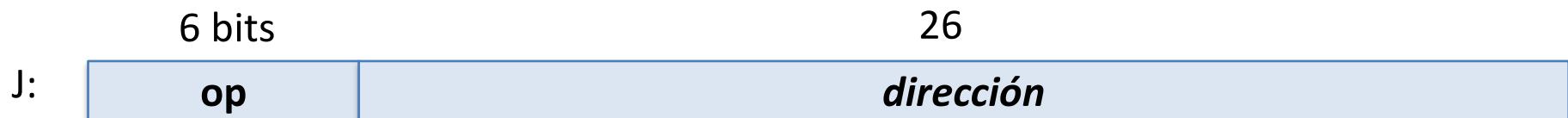
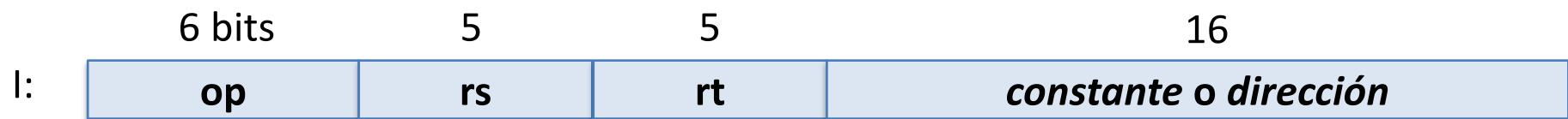
rs : el registro del primer operando fuente

rt : el registro del segundo operando fuente

rd : el registro del operando destino

shamt : cantidad de bits del *shift*

funct : variante particular de la operación especificada en *op*; p.ej., **add** y **sub** tienen el mismo valor en *op*, pero se diferencian por sus valores en *funct*



instrucción	formato	op	rs	rt	rd	shamt	funct	dirección
add	R	0	<i>reg</i>	<i>reg</i>	<i>reg</i>	0	32	n.a.
sub	R	0	<i>reg</i>	<i>reg</i>	<i>reg</i>	0	34	n.a.
add immediate	I	8	<i>reg</i>	<i>reg</i>	n.a.	n.a.	n.a.	<i>constante</i>
lw (<i>load word</i>)	I	35	<i>reg</i>	<i>reg</i>	n.a.	n.a.	n.a.	<i>dirección</i>
sw (<i>store word</i>)	I	43	<i>reg</i>	<i>reg</i>	n.a.	n.a.	n.a.	<i>dirección</i>
j (<i>jump</i>)	J	2	n.a.	n.a.	n.a.	n.a.	n.a.	<i>dirección</i>

reg: número de registro, entre 0 y 31

dirección: una dirección de 16 bits

n.a.: no aplicable

¿Cómo tener operandos inmediatos —constantes— de 32 bits?

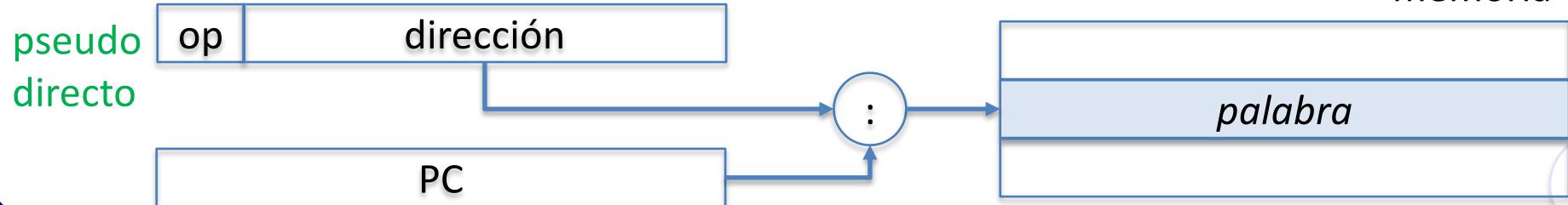
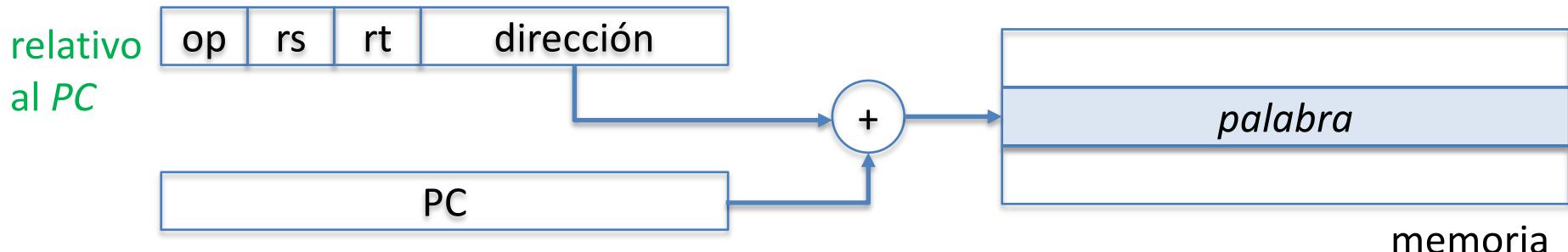
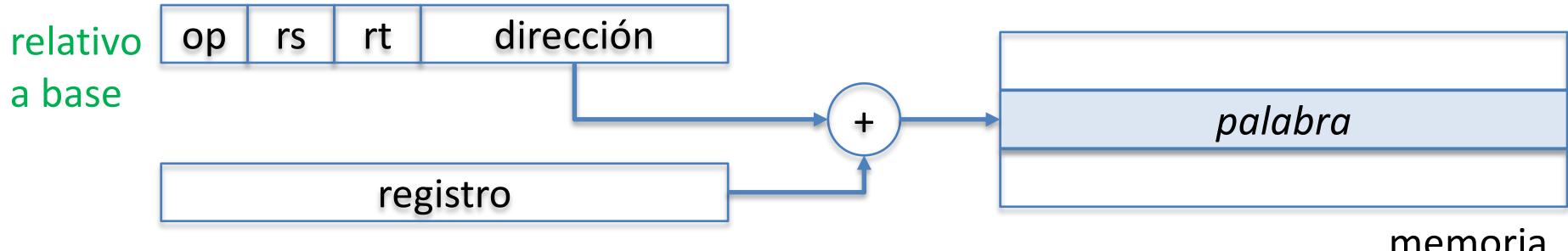
- la instrucción **lui** (*load upper immediate*) carga los 16 bits más significativos del registro; seguida por la instrucción **ori** (*or immediate*) que especifica los 16 bits menos significativos

Direcciones en saltos:

- la instrucción **j** (*jump* incondicional) usa el formato J, con 26 bits para (parte de) la dirección de la instrucción a la cual saltar
- los *jumps* condicionales —llamados *branch*, p.ej., **bne**— usan el formato I; el campo *dirección* se interpreta como una *distancia* que se suma al valor del registro *PC* —modo de direccionamiento **relativo al PC**

La próxima diap. ilustra los 5 modos de direccionamiento de MIPS:

- *inmediato* y *por registro*, ya explicados
- **relativo a una base** y **relativo al PC**, similares, pero el primero es para localizar un dato en la memoria, y el segundo es para saltar a una instrucción
- **pseudodirecto**, para *jumps* —los 26 bits del campo *dirección* se concatenan con los bits más significativos del *PC*



Principios de diseño RISC

Todas las instrucciones deben ser directamente ejecutables por el hardware —no interpretadas por microinstrucciones

Maximizar la tasa a la cual se inicia la ejecución de instrucciones

Las instrucciones deben fáciles de decodificar —de largo fijo, con un número pequeño de campos, y organizadas en unos pocos formatos regulares

Sólo las instrucciones de tipo *load* y *store* deben hacer referencia a la memoria —para el resto, los operandos deben estar en registros

Tener muchos registros —ya que el acceso a la memoria es comparativamente más lento

El nivel ISA del procesador Core i7

Ej. de set de instrucciones CISC

El procesador 80386 (1985) fue el primero de 32 bits de Intel

Los procesadores que siguieron —el 80486 (1989), la familia Pentium (1993–2000), el Core Duo (2006), y hasta el Core i7 (2011)— tienen esencialmente la misma arquitectura de 32 bits, llamada **IA-32**:

- los cambios arquitectónicos posteriores principales han sido la adición de instrucciones especializadas para mejorar el desempeño en aplicaciones multimediales
... y la versión de 64 bits (x86-64), en realidad introducida por AMD

El Core i7 tiene 1,160 millones de transistores, una frecuencia de reloj de 3.5 GHz y el ancho de los cables (que van de un transistor a otro) es de 32 nanómetros (mil veces más fino que un cabello humano)

El espacio de direcciones —para la mayoría de los sistemas operativos (p.ej., UNIX y Windows)— es lineal con 2^{32} bytes:

- en rigor, está organizado en 16,384 segmentos de 2^{32} bytes c/u

Cada byte tiene su propia dirección, desde 0 hasta $2^{32} - 1$

... y las palabras tienen 4 bytes (32 bits) en formato ***little endian***:

- el byte de más a la derecha tiene la dirección numéricamente menor

Los 16 registros del Core i7

EAX, EBX, ECX y EDX: 32 bits; más o menos generales:

EAX es el principal registro aritmético

EBX almacena punteros (direcciones de memoria)

ECX se usa en los *loops*

EDX se usa en multiplicaciones y divisiones, para almacenar productos y dividendos de 64 bits junto a EAX

cada uno contiene un registro de 16 bits (AX, BX, CX y DX) y uno de 8 bits (AL, BL, CL y DL)

ESI, EDI, EBP y ESP: 32 bits; más o menos generales:

ESI y EDI almacenan punteros para manejo de strings

EBP típicamente apunta a la base del registro de activación vigente

ESP es el *stack pointer*

CS, SS, DS, ES, FS y GS: 16 bits; pueden ser ignorados al usar un único espacio de direcciones de 32 bits (recuerdos del 8088)

EIP: 32 bits; *program counter*

EFLAGS: 32 bits; PSW

Las instrucciones aritméticas, lógicas y de transferencia de datos son instrucciones de sólo dos operandos (ver tabla):

- en las instrucciones aritméticas y lógicas uno de los operandos actúa también como (especificación del destino del) resultado
- casi en cualquier instrucción, uno de los operandos puede estar en memoria
- los operandos inmediatos pueden ser de 8, 16 o 32 bits
- el registro puede ser cualquiera de los 14 registros (no *EIP* o *EFLAGS*)

primer operando/resultado	segundo operando
registro	registro
registro	inmediato
registro	memoria
memoria	registro
memoria	inmediato

Algunas instrucciones típicas

Control

jnz, jz: saltos condicionales

jmp: salto incondicional

call: llamado a función

ret: “pop” dirección de retorno
desde el stack y saltar a ella

loop: salto condicional de vuelta al
inicio del loop; involucra ECX y EIP

Transferencia de datos

move: registro-registro, registro-
memoria

push, pop: “push” fuente al stack;
“pop” del tope del stack a registro

les: carga ES y uno de los 14 registros
desde memoria

Aritméticas, lógicas

add, sub: suma/resta fuente a/destino; registro-memoria

cmp: registro-memoria

shl, shr, rcr: “shifts” left y logical right, rotar a la derecha con CCC como relleno

cbw: convierte byte AL de EAX a 16 bits AX de EAX

test: fuente AND destino definen códigos de condición

inc, dec: incrementar/decrementar destino

or, xor: registro-memoria

Strings

moves: de fuente a destino incrementando ESI y EDI

lodsi: de memoria a EAX

La tabla muestra ejs. de instrucciones típicas

La próxima diap. muestra ejs. de formatos típicos de (algunas de estas instrucciones), incluyendo el número de bits de cada campo:

- el largo de una instrucción puede variar desde uno hasta 15 bytes

instrucción	función
JE <i>name</i>	if equal(<i>código de condición</i>) : EIP = <i>name</i> EIP-128 ≤ <i>name</i> < EIP+128
JMP <i>name</i>	EIP = <i>name</i>
CALL <i>name</i>	SP = SP-4 ; M[SP] = EIP+5 ; EIP = <i>name</i>
MOVW EBX, [EDI+45]	EBX = M[EDI+45]
PUSH ESI	SP = SP-4 ; M[SP] = ESI
POP EDI	EDI = M[SP] ; SP = SP+4
ADD EAX, #6765	EAX = EAX+6765
TEST EDX, #42	asignar códigos de condición según EDX y 42
MOVESL	M[EDI] = M[ESI] ; EDI = EDI+4 ; ESI = ESI+4

JE EIP + *displ.*

JE	<i>condition</i>	<i>displ.</i>
4	4	8

CALL

CALL	<i>offset</i>
8	32

MOV EBX, [EDI + 45]

MOV	<i>d</i>	<i>w</i>	R/M postbyte	<i>displ.</i>
6	1	1	8	8

PUSH ESI

PUSH	<i>reg</i>
5	3

ADD EAX, #6765

ADD	<i>reg</i>	<i>w</i>	<i>immediate</i>
4	3	1	32

TEST EDX, #42

TEST	<i>w</i>	<i>postbyte</i>	<i>immediate</i>
7	1	8	32

Modos de direccionamiento:

- inmediato, directo, por registro, indirecto por registro ...
 - ... indexado (contenido de un registro base + desplazamiento)
 - ... SIB → modo especial para direccionar elementos de arreglos, sin y con desplazamiento
- la próxima diap. muestra la relación entre formato y direccionamiento

modo	descripción	restricciones
indirecto por registro	la dirección está en el registro	ni <i>ESP</i> o <i>EBP</i>
base + desplazamiento	la dirección es contenido del registro <i>base</i> + <i>desplazamiento</i>	<i>base</i> : no <i>ESP</i>
SIB base + índice escalado	la dirección es <i>base</i> + (2^{escala} x <i>índice</i>) <i>escala</i> es 0, 1, 2 o 3	<i>base</i> : cualquier <i>Exx</i> <i>índice</i> : no <i>ESP</i>
SIB base + índice escalado + desplazamiento	la dirección es <i>base</i> + (2^{escala} x <i>índice</i>) + <i>desplazamiento</i> <i>escala</i> es 0, 1, 2 o 3	<i>base</i> : cualquier <i>Exx</i> <i>índice</i> : no <i>ESP</i>

