

# Formato de instrucción

Arquitectura de Computadoras - Práctico 6

# Set de instrucciones

## RISC:

- conjunto reducido de operaciones y “simples”
- instrucciones de largo fijo

## CISC:

- conjunto más amplio de operaciones y más complejas
- instrucciones de largo variable

# Arquitectura de n bits

¿Qué nos dice?

SI nos dice:

- tamaño de los registros
- ancho (tamaño) del 'data path' (buses que entran a la ALU, tamaño de palabra de ALU)
- tamaño de palabra de memoria estándar
- largo de instrucciones (para RISC)

# Arquitectura de n bits (2)

¿Qué nos dice?

NO nos dice:

- ancho (tamaño) del bus de datos
- largo de las instrucciones (para CISC)
- largo de las direcciones (cantidad de bits para una dirección)

# Ejercicio 1

Considere un microprocesador RISC de 16 bits con 16 registros y las siguientes instrucciones:

Instrucción	Descripción
NOP	No realiza ninguna operación
MOVI Inm, Reg1	Carga el inmediato Inm en los bits menos significativos de Reg1, dejando los bits más significativos en 0
MOVR Reg1, Reg2	Carga en Reg2 el contenido de Reg1
LOAD Reg1, Reg2	Carga el contenido de memoria apuntado por Reg1 en Reg2
SAVE Reg1, Reg2	Guarda el registro Reg1 en la dirección de memoria apuntada por Reg2
ADD Reg1, Reg2, Reg3	Suma Reg1 y Reg2 y guarda el resultado en Reg3
SUB Reg1, Reg2, Reg3	Realiza la operación Reg1-Reg2 y guarda el resultado en Reg3
SHL Reg1, Reg2, Reg3	Desplaza Reg1 a la izquierda Reg2 lugares y guarda el resultado en Reg3
SHR Reg1, Reg2, Reg3	Desplaza Reg1 a la derecha Reg2 lugares y guarda el resultado en Reg3
AND Reg1, Reg2, Reg3	Realiza un AND bit a bit entre Reg1 y Reg2 y guarda el resultado en Reg3
OR Reg1, Reg2, Reg3	Realiza un OR bit a bit entre Reg1 y Reg2 y guarda el resultado en Reg3
XOR Reg1, Reg2, Reg3	Realiza un XOR bit a bit entre Reg1 y Reg2 y guarda el resultado en Reg3
NOT Reg1, Reg2	Realiza un NOT bit a bit de Reg1 y guarda el resultado en Reg2
JMP Reg1	Hace un salto incondicional a la dirección apuntada por Reg1
JC Reg1	Hace un salto a la dirección apuntada por Reg1 si la bandera de C vale 1
JZ Reg1	Hace un salto a la dirección apuntada por Reg1 si la bandera de Z vale 1

Aclaraciones:

- Las flags Z (cero) y C (acarreo) son afectadas por las operaciones aritméticas y lógicas.
- Para el caso de los desplazamientos, la flag C corresponde al último bit que fue desplazado hacia fuera del registro.
- Los 16 registros llevan el nombre R00 hasta R15.

- Defina un formato de instrucción para la CPU e indique el largo de instrucción.
- Implemente en alto nivel un programa que sume el contenido de memoria de las direcciones comprendidas en el rango 0xABCD–0xEF00 y almacene en la dirección 0x00FF el triple de dicha suma.
- Implemente en lenguaje ensamblador el programa de la parte B.
- Implemente en lenguaje máquina las primeras 7 instrucciones del programa de la parte C.

Notas:

- El programa se debe implementar con los nombres de operación dados. Las direcciones y constantes deben representarse en hexadecimal.
- El programa estará cargado en memoria a partir de la dirección 0x0000.

# Parte A

- tenemos 16 instrucciones: ¿cuántos bits necesitamos para identificarlas?
- tenemos 16 registros: ¿cuántos bits necesitamos?
- ¿cuál es el largo de la instrucción?

# Parte A (2)

- 4 bits para identificar la operación (lo llamamos opcode)
- 4 bits para identificar los registros
- largo de instrucción de 16 bits (RISC)

Instrucción	Formato		
NOP			
JMP, JZ, JC	opcode (4 bits)	????	????
MOVR, LOAD, SAVE, NOT			
SUB, SHL, SHR, ADD, AND, OR, XOR			
MOVI			

# Parte A (3)

Instrucción	Formato		
NOP	opcode (4 bits)	Reg1	0000
JMP, JZ, JC		0000	
MOVR, LOAD, SAVE, NOT			
SUB, SHL, SHR, ADD, AND, OR, XOR			
MOVI			



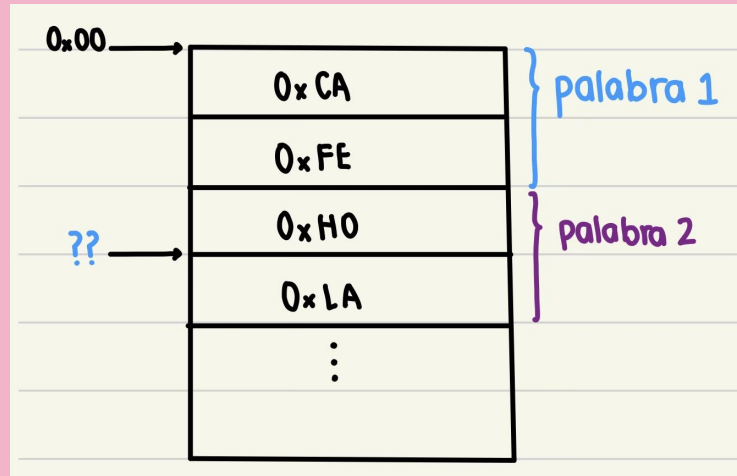
# Parte A (4)

Instrucción	Formato			
NOP		0000	0000	0000
JMP, JZ, JC	opcode (4 bits)	Reg1	0000	0000
MOVR, LOAD, SAVE, NOT		Reg1	Reg2	0000
SUB, SHL, SHR, ADD, AND, OR, XOR		Reg1	Reg2	Reg3
MOVI		Reg1	Inmediato 8 bits	

# Parte B

palabra de memoria: 16 bits

arquitectura direccionable de a byte: cada byte se corresponde con una dirección diferente de memoria



# Parte B (2)

```
Programa ( ) {
```

```
  int dir = 0x ABCD;
```

```
  int hasta = 0x EF00;
```

```
  int suma = 0;
```

```
  int resultado = 0;
```

```
  int paso = ?? ;
```

```
  do {
```

```
    suma = suma + MEM[dir] ;
```

```
    dir = dir + paso ;
```

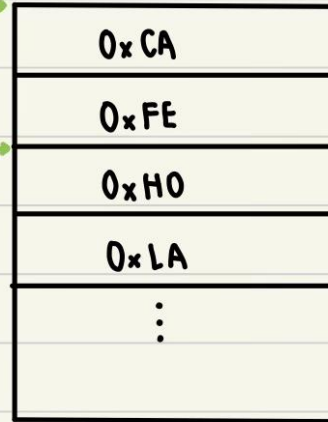
```
  } while (dir < hasta) ;
```

```
  :
```

dir →

dir + paso →

qué hace esto?



} palabra 1

} palabra 2

# Parte B (3)

Programa (1) {

int dir = 0x ABCD;

int hasta = 0x EF00;

int suma = 0;

int resultado = 0;

int paso = 2 ; // 2 bytes por instrucción

do {

suma = suma + MEM[dir];

dir = dir + paso;

} while (dir < hasta);

resultado = suma << 1; // multiplicación x2 0001  $\xrightarrow{\ll 1}$  0010

resultado = resultado + suma;

dir = 0xFF;

MEM[dir] = resultado;

}

0x00

0x CA

0xFE

0xH0

0xLA

⋮

} palabra 1

0x02

} palabra 2

↗ carga 2 bytes

# Parte C

`int dir = 0xABCD; ⇒ 0x0 MOV 0xAB, R2 // 0x00AB`

`0x2 MOV 0xCD, R3`

`0x4 MOV 0x08, R1 // SHL solo usa reg`

`0x6 SHL R2, R1, R2 // en R2 0xAB00`

`0x8 OR R2, R3, R2 // dir = 0xABCD (R2)`

`int hasta = 0xEF00; ⇒ ...`

# Parte C (2)

int hasta = 0xEF00;  $\Rightarrow$  0xA MOV 0xEF, R3

0xC SHL R3, R1, R3 // R3 = hasta

int suma = 0;  $\Rightarrow$  0xE XOR R4, R4, R4 // R4 = suma

int resultado = 0;  $\Rightarrow$  0x10 XOR R5, R5, R5 // R5 = res

int paso = 2;  $\Rightarrow$  0x12 MOV 0x02, R1 // R1 = paso

# Parte C (3)

```
0x0 MOV 0xAB, R2 // 0x00AB
0x2 MOV 0xCD, R3
0x4 MOV 0x08, R1 // SHL solo usa reg
0x6 SHL R2, R1, R2 // en R2 0xAB00
0x8 OR R2, R3, R2 // dir = 0xABCD (R2)
```

```
0xA MOV 0xEF, R3
0xC SHL R3, R1, R3 // R3 = hasta
0xE XOR R4, R4, R4 // R4 = suma
0x10 XOR R5, R5, R5 // R5 = res
0x12 MOV 0x02, R1 // R1 = paso
```

```
suma = suma + MEM[dir]; => ??? // cargar MEM[dir] en R6
                                ADD R4, R6, R4 // R4 = suma
dir = dir + paso;                => ADD R2, R1, R2
while (dir < hasta);             => ???
```

# Parte C (4)

```
0x0 MOV 0xAB, R2 // 0x00AB
0x2 MOV 0xCD, R3
0x4 MOV 0x08, R1 // SHL solo usa reg
0x6 SHL R2, R1, R2 // en R2 0xAB00
0x8 OR R2, R3, R2 // dir = 0xABCD (R2)
```

```
0xA MOV 0xEF, R3
0xC SHL R3, R1, R3 // R3 = hasta
0xE XOR R4, R4, R4 // R4 = suma
0x10 XOR R5, R5, R5 // R5 = res
0x12 MOV 0x02, R1 // R1 = paso
```

```
0x14 MOV 0x16, R7
suma = suma + MEM[dir]; => 0x16 LOAD R2, R6 // cargar MEM[dir] en R6
0x18 ADD R4, R6, R4 // R4 = suma
dir = dir + paso; => 0x1A ADD R2, R1, R2
while (dir < hasta); => 0x1C SUB R2, R3, R2 // resta dir - hasta
0x1E JC R7
```



# jumps relativos vs absolutos

JMP INM

Salto incondicional de INM instrucciones (INM es un número con signo de 12 bits) desde la posición actual.

# Parte C (5)

```
0x0 MOV 0xAB, R2 // 0x00AB
0x2 MOV 0xCD, R3
0x4 MOV 0x08, R1 // SHL solo usa reg
0x6 SHL R2, R1, R2 // en R2 0xAB00
0x8 OR R2, R3, R2 // dir = 0xABCD (R2)
```

```
0xA MOV 0xEF, R3
0xC SHL R3, R1, R3 // R3 = hasta
0xE XOR R4, R4, R4 // R4 = suma
0x10 XOR R5, R5, R5 // R5 = res
0x12 MOV 0x02, R1 // R1 = paso
```

```
0x14 MOV 0x16, R7
suma = suma + MEM[dir]; => 0x16 LOAD R2, R6 // cargar MEM[dir] en R6
0x18 ADD R4, R6, R4 // R4 = suma
dir = dir + paso; => 0x1A ADD R2, R1, R2
while (dir < hasta); => 0x1C SUB R2, R3, R2 // resta dir - hasta
0x1E JC R7
```

```
resultado = suma << 1; => ???
resultado = resultado + suma; => ???
dir = 0xFF; => ???
MEM[dir] = resultado; => ???
```

# Parte C (6)

```
0x0 MOV 0xAB, R2 // 0x00AB
0x2 MOV 0xCD, R3
0x4 MOV 0x08, R1 // SHL solo usa reg
0x6 SHL R2, R1, R2 // en R2 0xAB00
0x8 OR R2, R3, R2 // dir = 0xABCD (R2)
```

```
0xA MOV 0xEF, R3
0xC SHL R3, R1, R3 // R3 = hasta
0xE XOR R4, R4, R4 // R4 = suma
0x10 XOR R5, R5, R5 // R5 = res
0x12 MOV 0x02, R1 // R1 = paso
```

```
0x14 MOV 0x16, R7
suma = suma + MEM[dir]; => 0x16 LOAD R2, R6 // cargar MEM[dir] en R6
0x18 ADD R4, R6, R4 // R4 = suma
dir = dir + paso; => 0x1A ADD R2, R1, R2
while (dir < hasta); => 0x1C SUB R2, R3, R2 // resta dir - hasta
0x1E JC R7
```

```
resultado = suma << 1; => 0x20 MOV 0x01, R1
0x22 JHL R4, R1, R4
resultado = resultado + suma; => 0x24 ADD R5, R4, R5
dir = 0xFF; => 0x26 MOV 0xFF, R2
MEM[dir] = resultado; => 0x28 SAVE R2, R5
```

# Parte C (7)

0x0 MOV 0xAB, R2 // 0x00AB

0x2 MOV 0xCD, R3

0x4 MOV 0x08, R1 // SHL solo usa reg

0x6 SHL R2, R1, R2 // en R2 0xAB00

0x8 OR R2, R3, R2 // dir = 0xABCD (R2)

0xA MOV 0xEF, R3

0xC SHL R3, R1, R3 // R3 = hasta

0xE XOR R4, R4, R4 // R4 = suma

0x10 XOR R5, R5, R5 // R5 = res

0x12 MOV 0x02, R1 // R1 = paso

0x14 MOV 0x16, R7

0x16 LOAD R2, R6 // cargar MEM[dir] en R6

0x18 ADD R4, R6, R4 // R4 = Suma

0x1A ADD R2, R1, R2

0x1C SUB R2, R3, R2 // resta dir - hasta

0x1E JC R7

0x20 MOV 0x01, R1

0x22 JHL R4, R1, R4

0x24 ADD R5, R4, R5

0x26 MOV 0xFF, R2

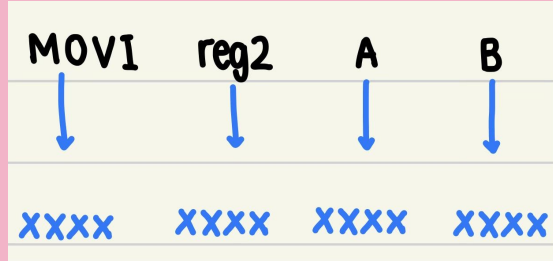
0x28 SAVE R2, R5

0x2A MOV 0x2A, R7

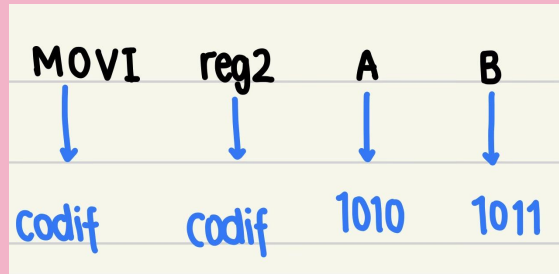
0x2C XOR R5, R5, R5

0x2E JZ R7

# Parte D



# Parte D (2)



NOP	0000
MOVI	0001
MOVR	0010
LOAD	0011
SAVE	0100
ADD	0101
SUB	0110
SHL	0111
SHR	1000
AND	1001
OR	1010
XOR	1011
NOT	1100
JMP	1101
JC	1110
JZ	1111

# Parte D (3)

MOVI	reg2	A	B
↓	↓	↓	↓
codif	codif	1010	1011

NOP	0000
MOVI	0001
MOVR	0010
LOAD	0011
SAVE	0100
ADD	0101
SUB	0110
SHL	0111
SHR	1000
AND	1001
OR	1010
XOR	1011
NOT	1100
JMP	1101
JC	1110
JZ	1111

MOVI	reg2	A	B
↓	↓	↓	↓
0001	0010	1010	1011

# Parte D (4)

<b>MOVI</b>	<b>reg2</b>	<b>A</b>	<b>B</b>
0001	0010	1010	1011
<b>MOVI</b>	<b>r3</b>	<b>C</b>	<b>D</b>
0001	0011	1100	1101
<b>MOVI</b>	<b>r1</b>	<b>0</b>	<b>8</b>
0001	0001	0000	1000
<b>SHL</b>	<b>r2</b>	<b>r1</b>	<b>r2</b>
0111	0010	0001	0010
<b>OR</b>	<b>r2</b>	<b>r3</b>	<b>r2</b>
1010	0010	0011	0010
<b>MOVI</b>	<b>r3</b>	<b>E</b>	<b>F</b>
0001	0011	1110	1111
<b>SHL</b>	<b>r3</b>	<b>r1</b>	<b>r3</b>
0111	0011	0001	0011