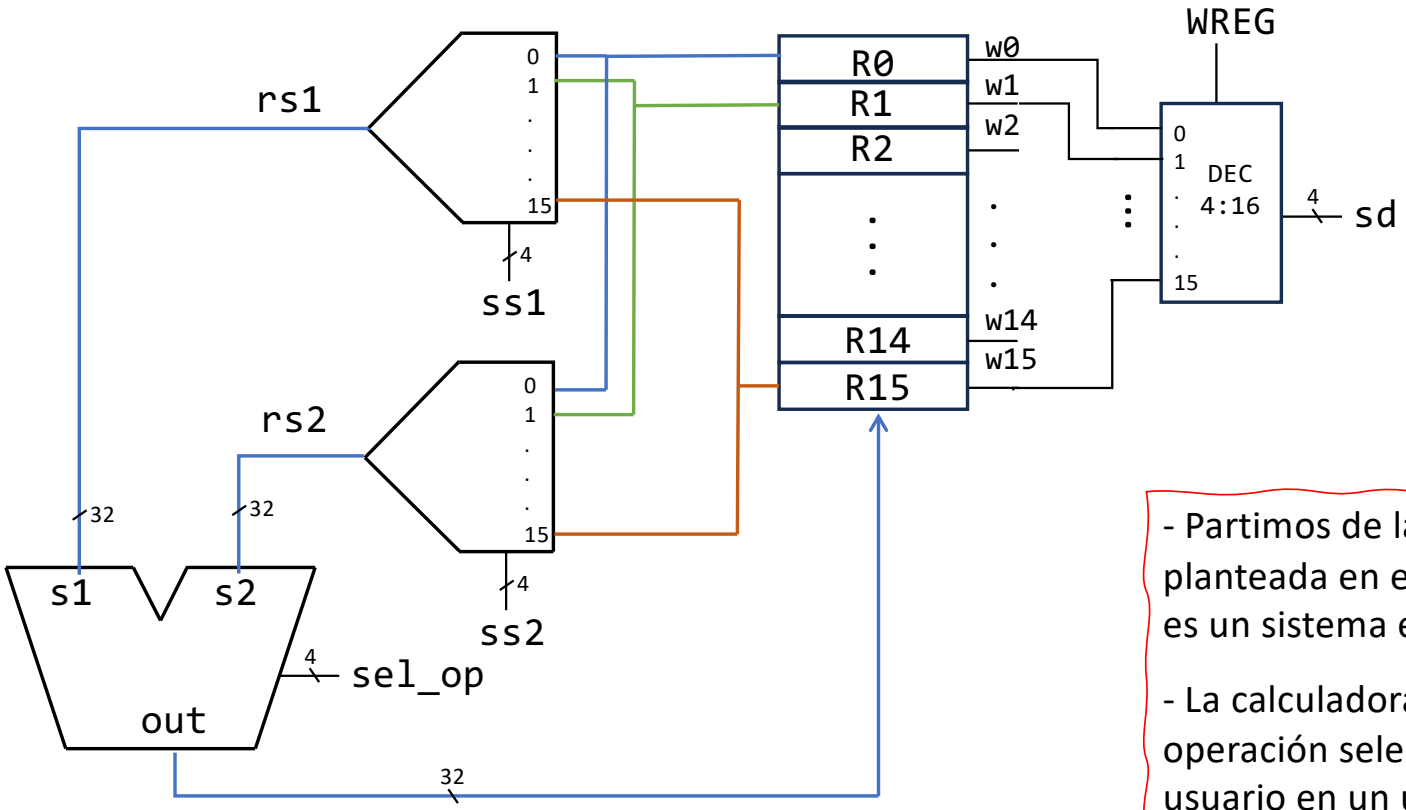
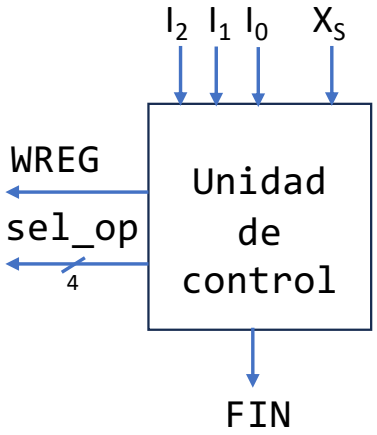
Tema 3

Diseño de un Computador Simple a Nivel RT

Índice

La calculadora como punto de partida

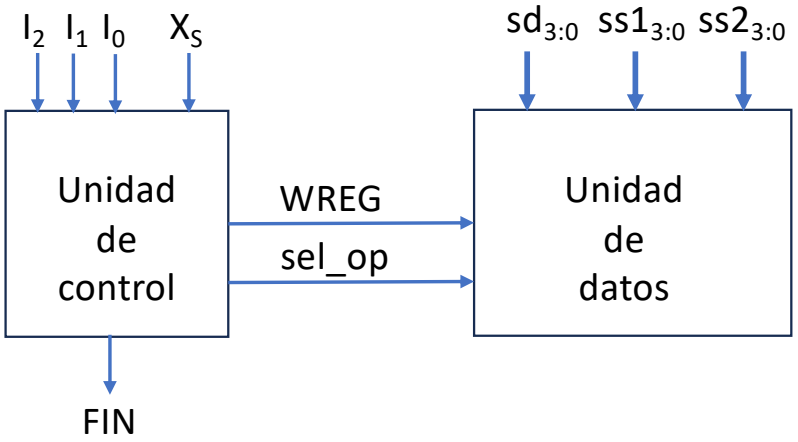


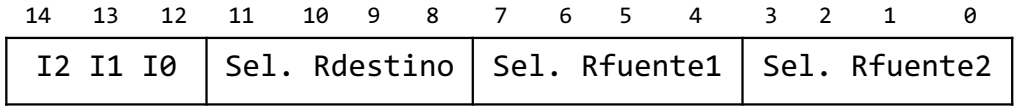


- Partimos de la calculadora planteada en el tema anterior que es un sistema en un único paso.
- La calculadora ejecuta cada operación seleccionada por el usuario en un único ciclo de reloj.

La calculadora como punto de partida

- A partir de sus operaciones es posible realizar tareas más complejas sin conocer la electrónica del sistema pero:
  - LA EJECUCIÓN NO ES AUTOMÁTICA:
    el usuario debe de activar Xs y esperar la señal de fin para cada instrucción
  - NO HAY PROGRAMA ALMACENADO:
    cada vez que se ejecuta una instrucción el usuario debe suministrar la siguiente.



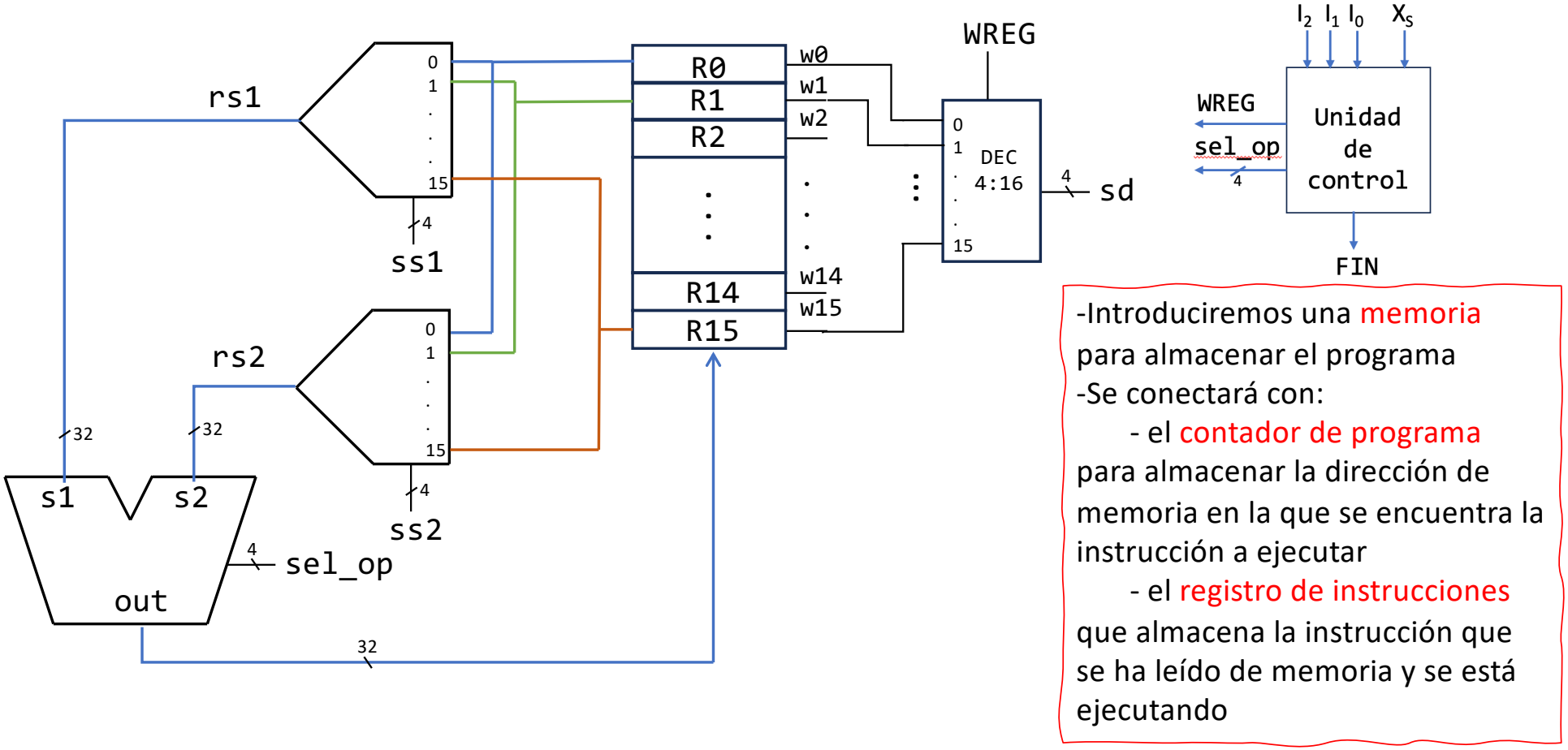


Registro de instrucciones:
podemos usar un registro adicional, que no posee la calculadora, donde se guarda la información que pone el usuario desde el exterior y se conecta a las entradas correspondientes de U. de datos y U. de control.

Índice

1. Introducción.
2. La calculadora como punto de partida
3. Computador Simple 1
   (Automatización en la ejecución y almacenamiento de programa)
4. Computador simple 2
   (Almacenamiento de los datos y ampliación de modos de direccionamiento)
5. Computador Simple 3 / RISCY
   (Ejecución no secuencial: Instrucciones de salto)

La calculadora como punto de partida: cambios

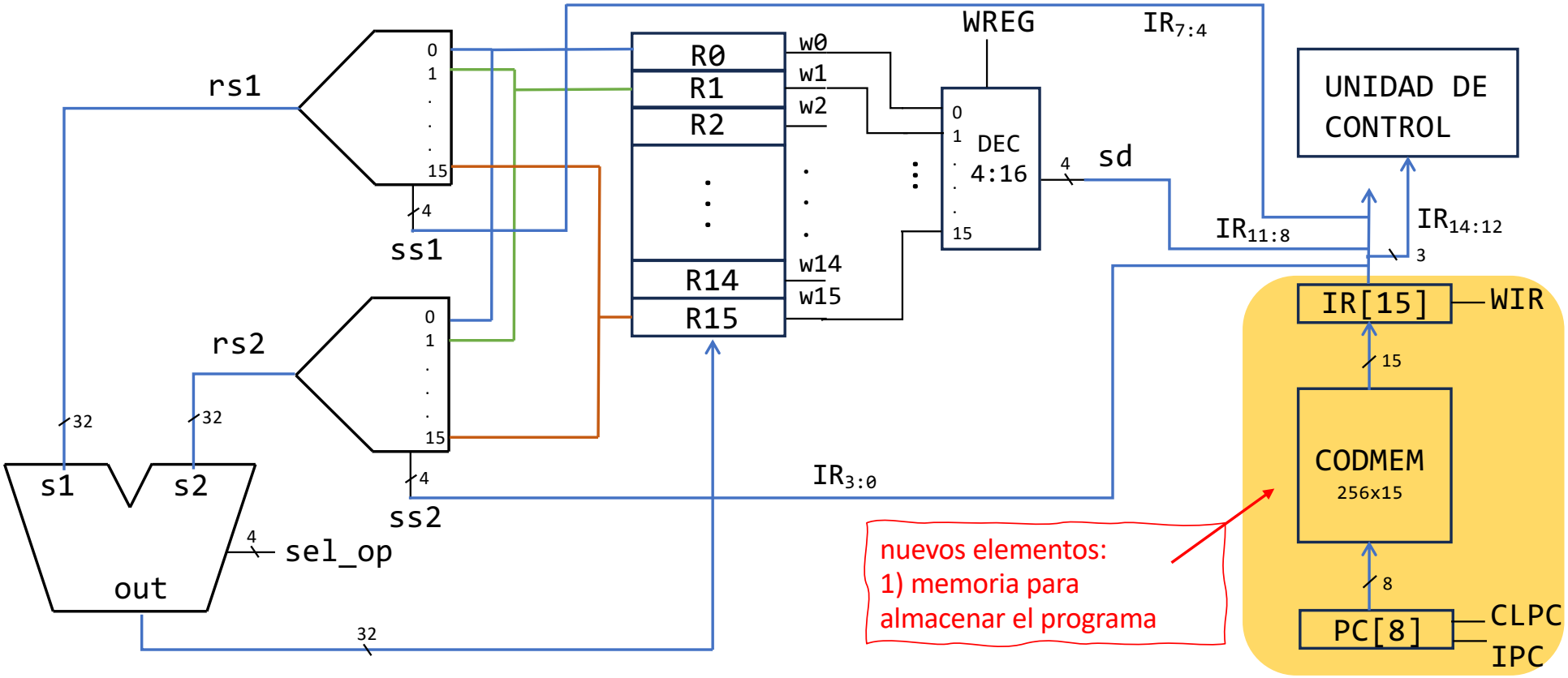


-Introduciremos una memoria para almacenar el programa

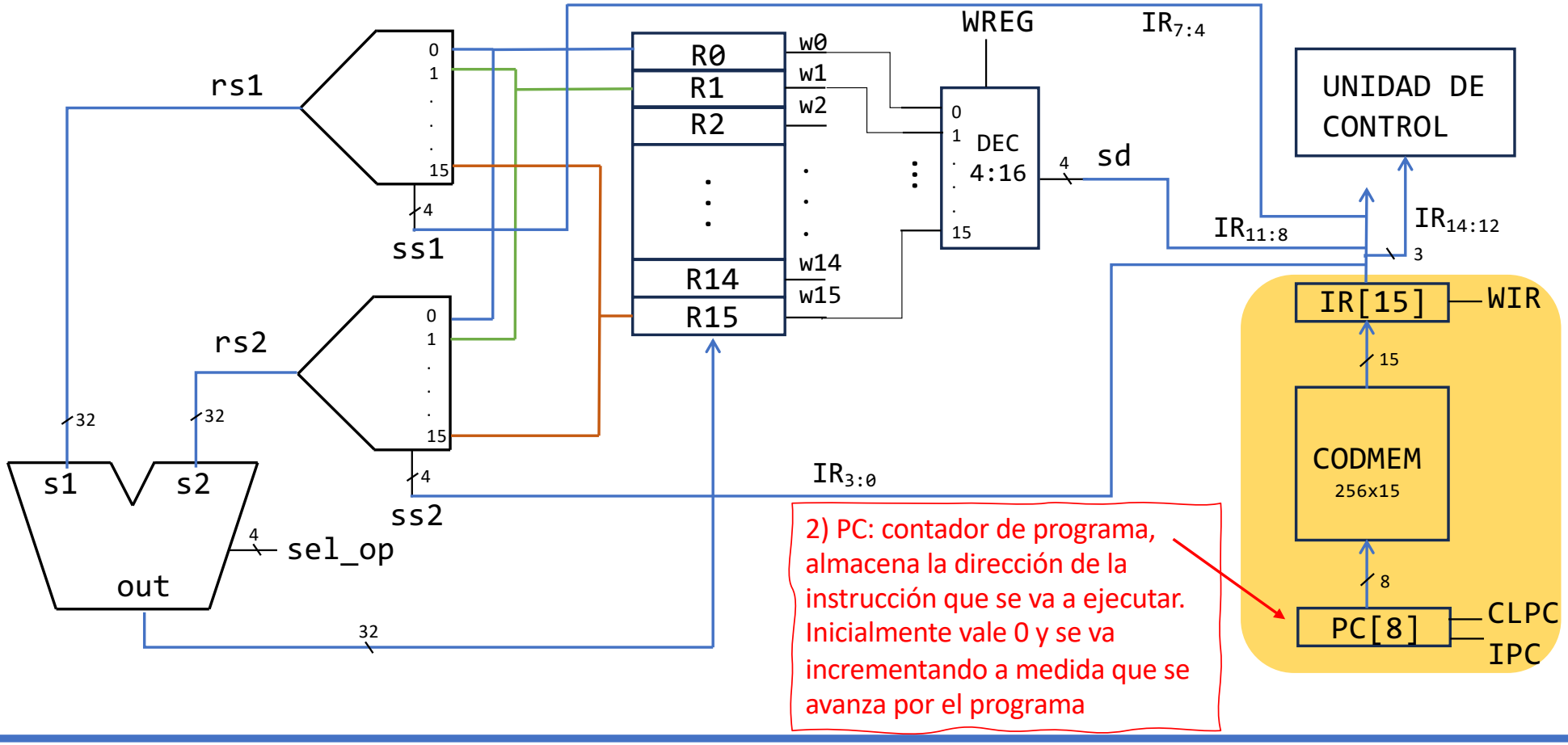
-Se conectará con:

- el contador de programa para almacenar la dirección de memoria en la que se encuentra la instrucción a ejecutar
- el registro de instrucciones que almacena la instrucción que se ha leído de memoria y se está ejecutando

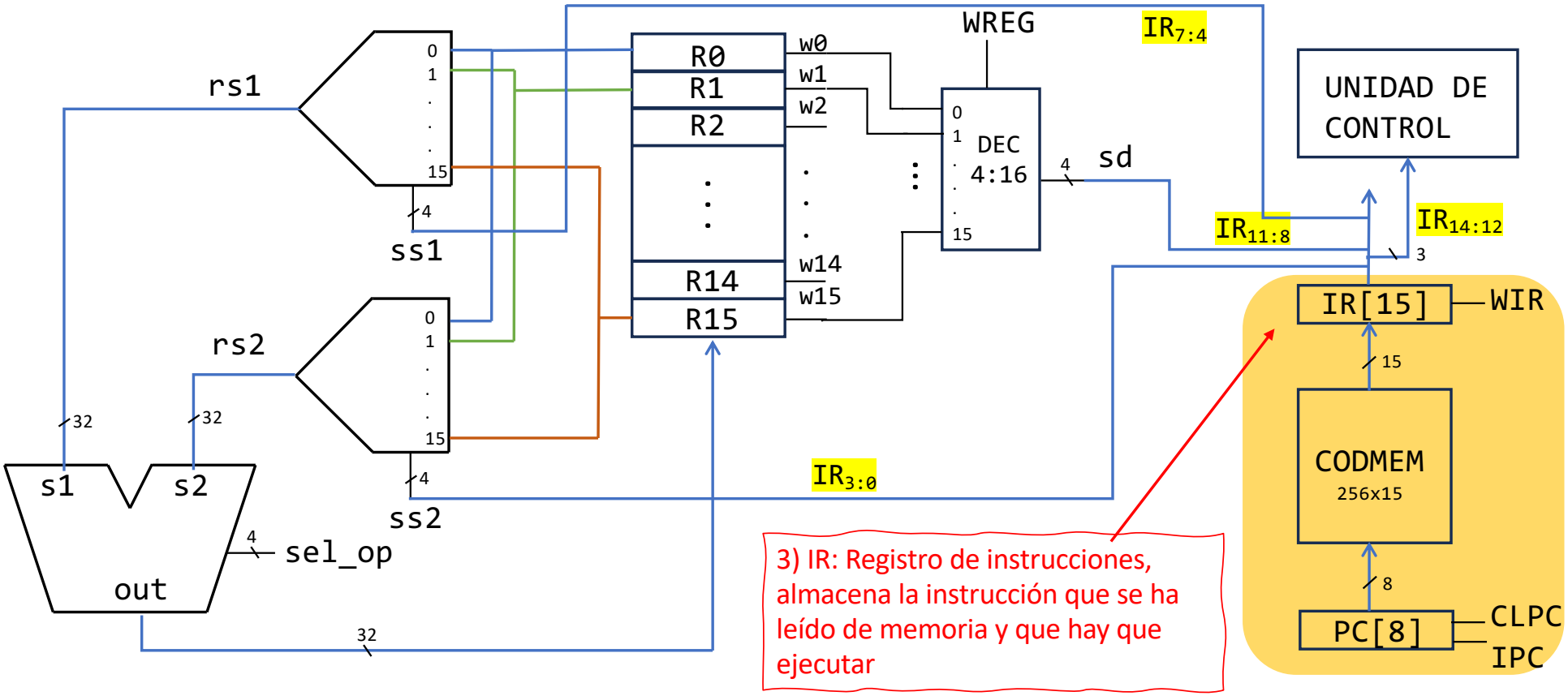
Computador Simple 1



Computador Simple 1



Computador Simple 1



Computador Simple 1

Todas las instrucciones son de una palabra y están codificadas en binario (código máquina).

Formato de instrucción:
indica como debe ser interpretada una instrucción en código máquina (código de operación y operandos).

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CODOP			RD				RS1				RS2			

Código de operación (CODOP): es lo que diferencia a las instrucciones entre sí. En el CS1 se dispone de 3 bits de código de operación lo que permite definir 8 instrucciones.

En el CS1 los datos siempre están almacenados en sus 16 registros que están codificados por 12 bits. Los 4 primeros identifican al registro destino (RD) y los 8 siguientes a los dos registros fuente (RS1 y RS2)

Computador Simple 1: instrucciones

CODOP ($IR_{14:12}$)	Sintaxis	Función
001	add rd,rs1,rs2	rd ← rs1 + rs2
010	sub rd,rs1,rs2	rd ← rs1 - rs2
011	xor rd,rs1,rs2	rd ← rs1 ⊕ rs2
100	or rd,rs1,rs2	rd ← rs1 or rs2
101	and rd,rs1,rs2	rd ← rs1 and rs2
110	sll rd,rs1,rs2	rd ← rs1 << $rs2_{4:0}$
111	srl rd,rs1,rs2	rd ← rs1 >> $rs2_{4:0}$
000	stop	nop

add, sub, xor, or, and, srll y srl son mnemónicos: ayudan a identificar las instrucciones de un modo más cómodo que con el codop

Los bits 14, 13 y 12 del IR son entradas de la unidad de control

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CODOP			RD				RS1				RS2			

Computador Simple 1

Programa almacenado:

La memoria CODMEM contiene el programa a ejecutar. El usuario no tendrá que suministrar las instrucciones una a una como sucedía en la calculadora.

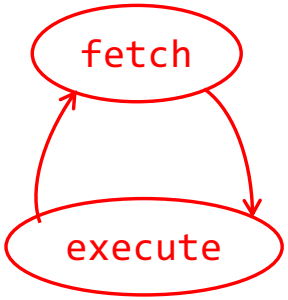
Los programas siempre están almacenados a partir de la dirección 0 de la memoria y la ejecución es lineal.

Automatización en la ejecución:

La unidad de control debe ocuparse de que el programa almacenado en memoria se ejecute de forma automática.

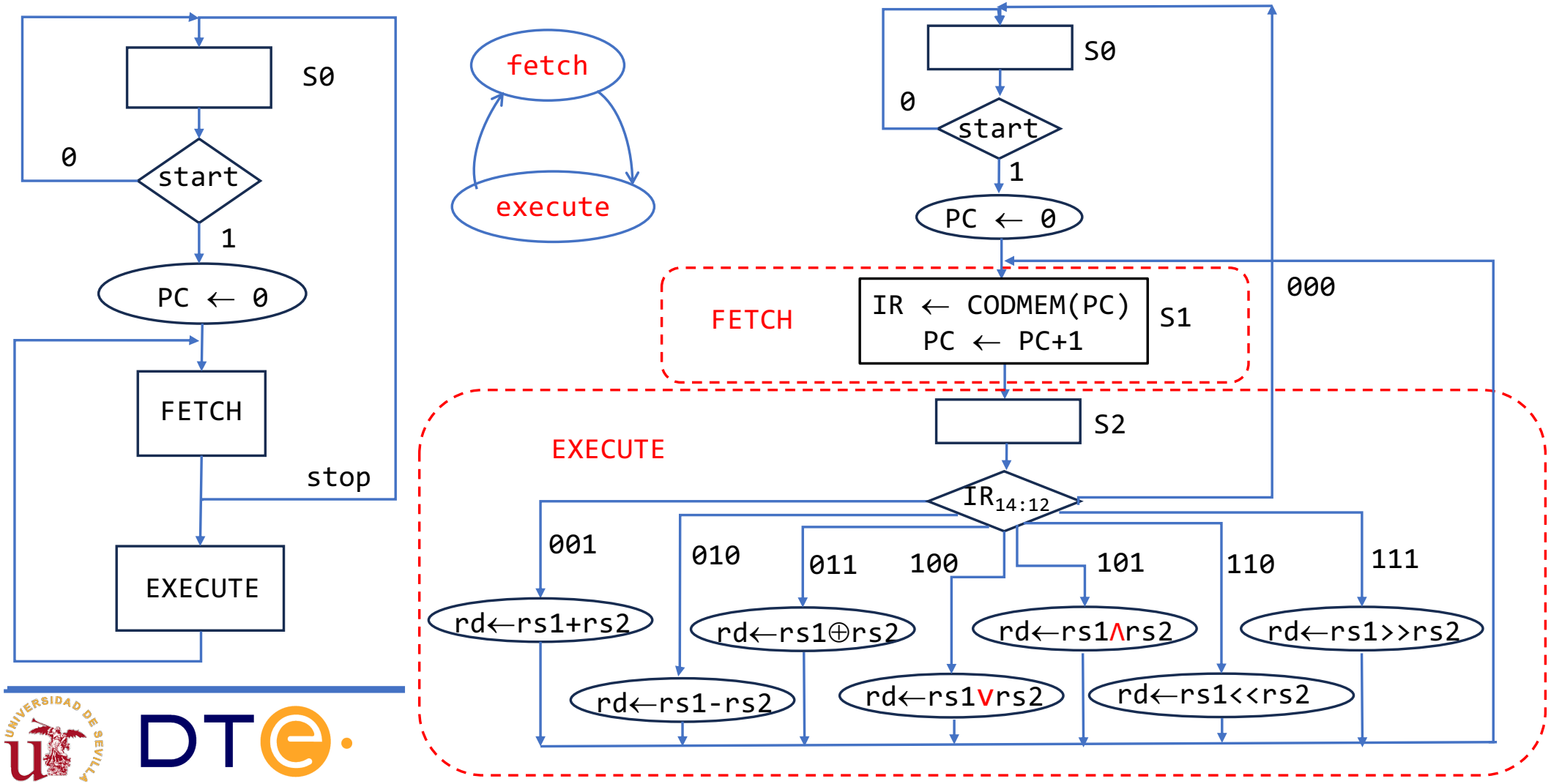
Para ello se establece el ciclo de instrucción en la carta ASM, consistente en dos fases

1) se carga en IR la instrucción apuntada por PC: ciclo de búsqueda o de fetch
2) se ejecuta la instrucción cargada en IR: ciclo de ejecución o execute



Estos ciclos se ejecutarán continuamente hasta alcanzar la instrucción stop

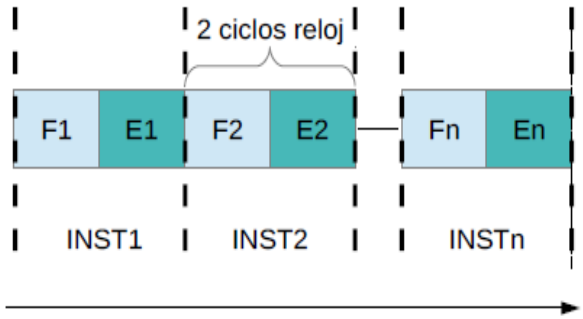
CARTA ASM

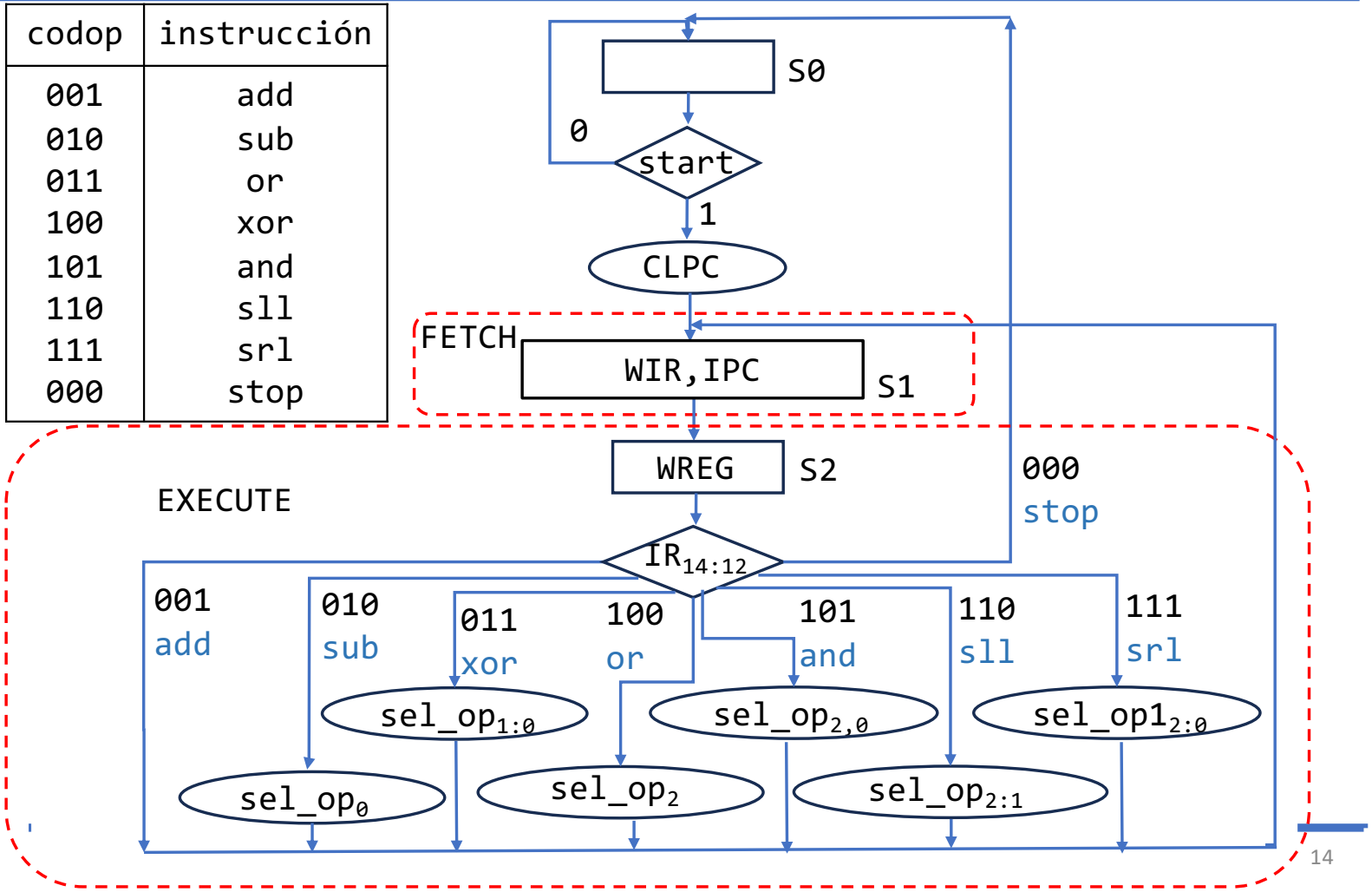

S0
0
start
1
PC ← 0
FETCH
stop
EXECUTE
fetch
execute
S0
0
start
1
PC ← 0
000
FETCH
IR ← CODMEM(PC)
PC ← PC+1
S1
S2
EXECUTE
IR14:12
001
010
011
100
101
110
111
rd←rs1+rs2
rd←rs1-rs2
rd←rs1⊕rs2
rd←rs1vrs2
rd←rs1∧rs2
rd←rs1<<rs2
rd←rs1>>rs2

CARTA ASM

sel_op	operación alu
0000	out=s1+s2
0001	out=s1-s2
0010	out=s2
0011	out=s1⊕s2
0100	out=s1 or s2
0101	out=s1 and s2
0110	out=s1<<$s2_{4:0}$
0111	out=s1>>$s2_{4:0}$
1000	out=s1>>>$s2_{4:0}$

codop	instrucción
001	add
010	sub
011	or
100	xor
101	and
110	sll
111	srl
000	stop





Computador Simple 1

Ejemplo de uso: mediante un programa se realiza la operación R6 ← 3R4-2R1

programa

```
SUB R6,R6,R6
ADD R6,R4,R4
ADD R6,R6,R4
SUB R6,R6,R1
SUB R6,R6,R1
STOP
```

dirección	contenido
0x00	010 0110 0110 0110
0x01	001 0110 0100 0100
0x02	001 0110 0110 0100
0x03	010 0110 0110 0001
0x04	010 0110 0110 0001
0x05	000 0000 0000 0000

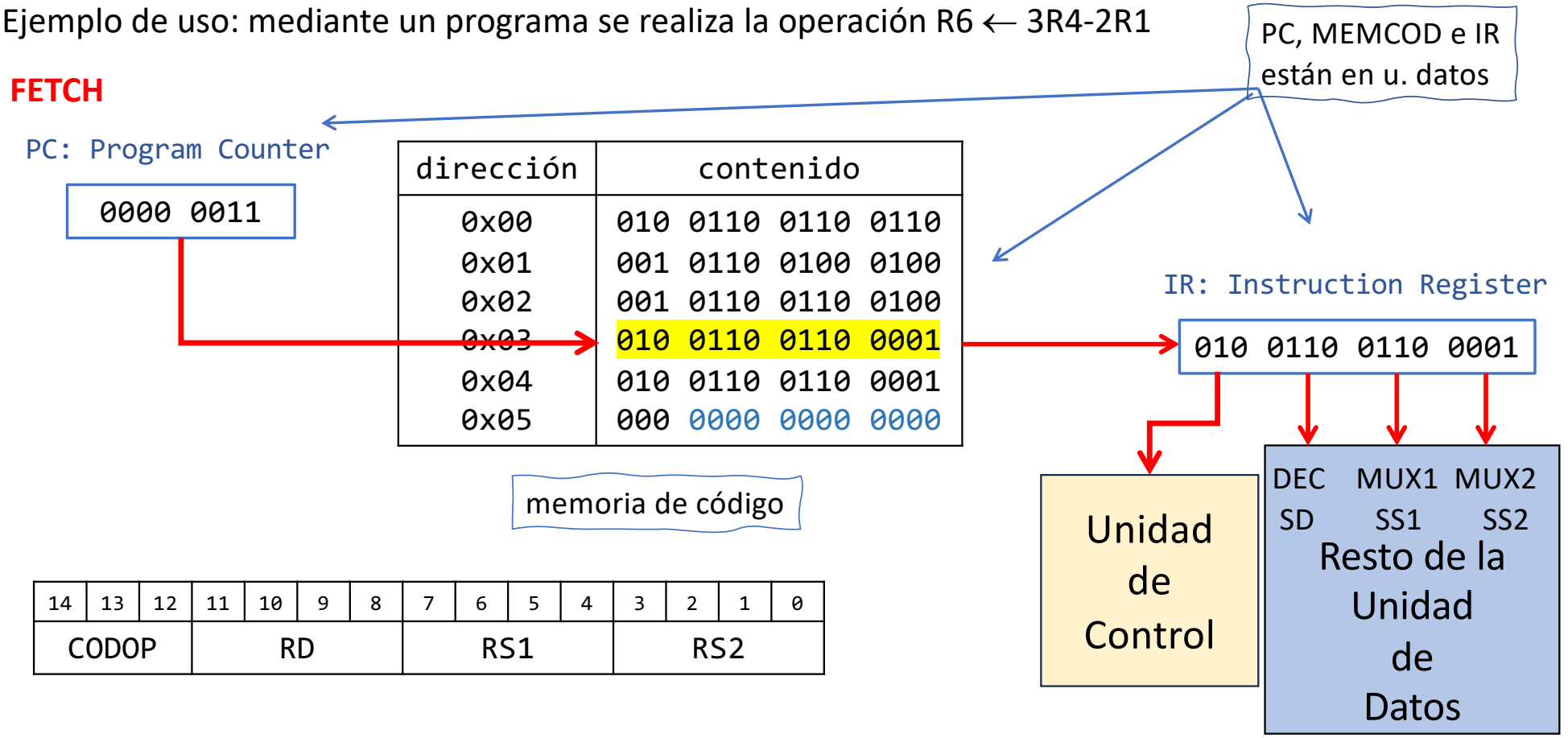
memoria de código

codop	operación
001	add
010	sub
011	or
100	xor
101	and
110	sll
111	srl
000	stop

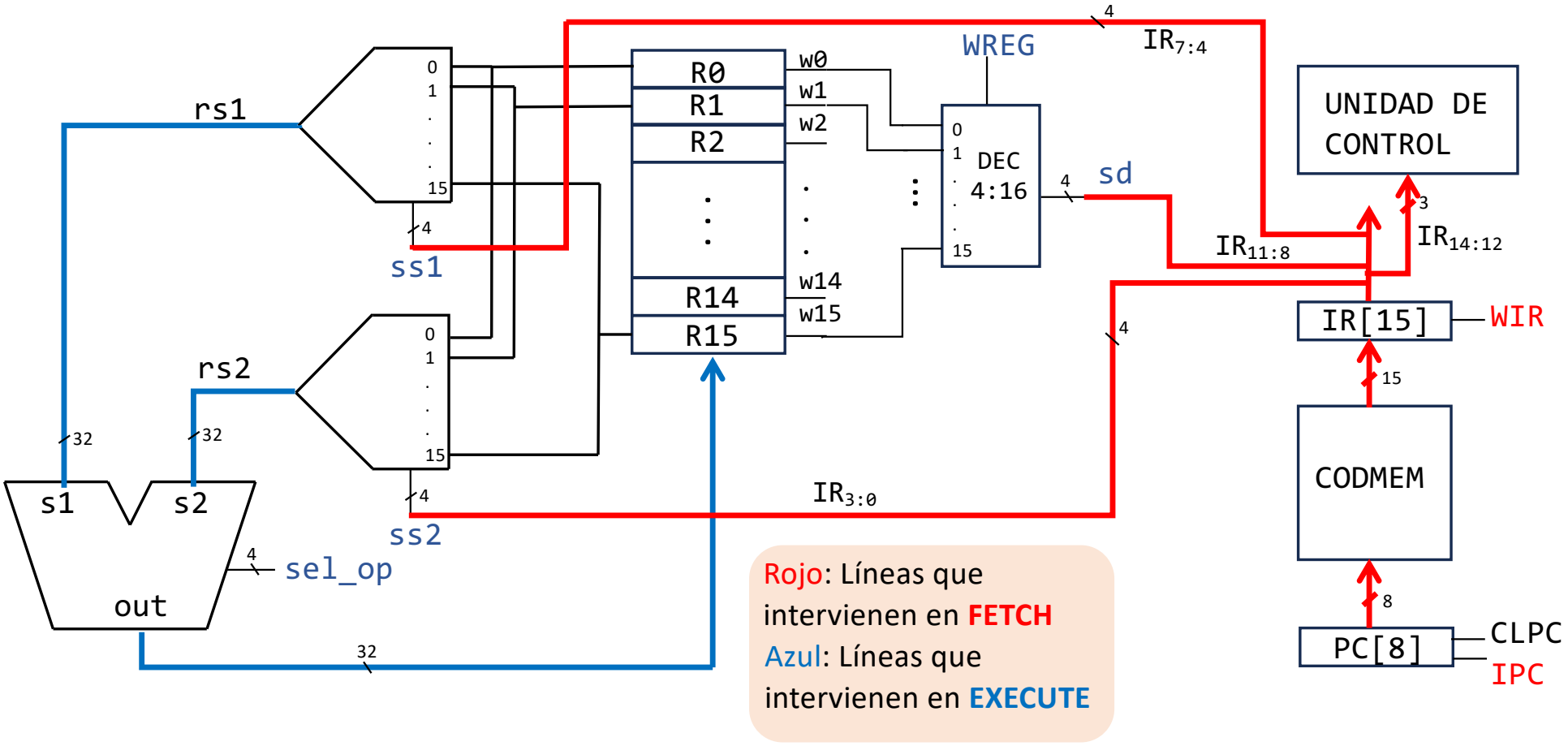
14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CODOP			RD				RS1				RS2			

Computador Simple 1

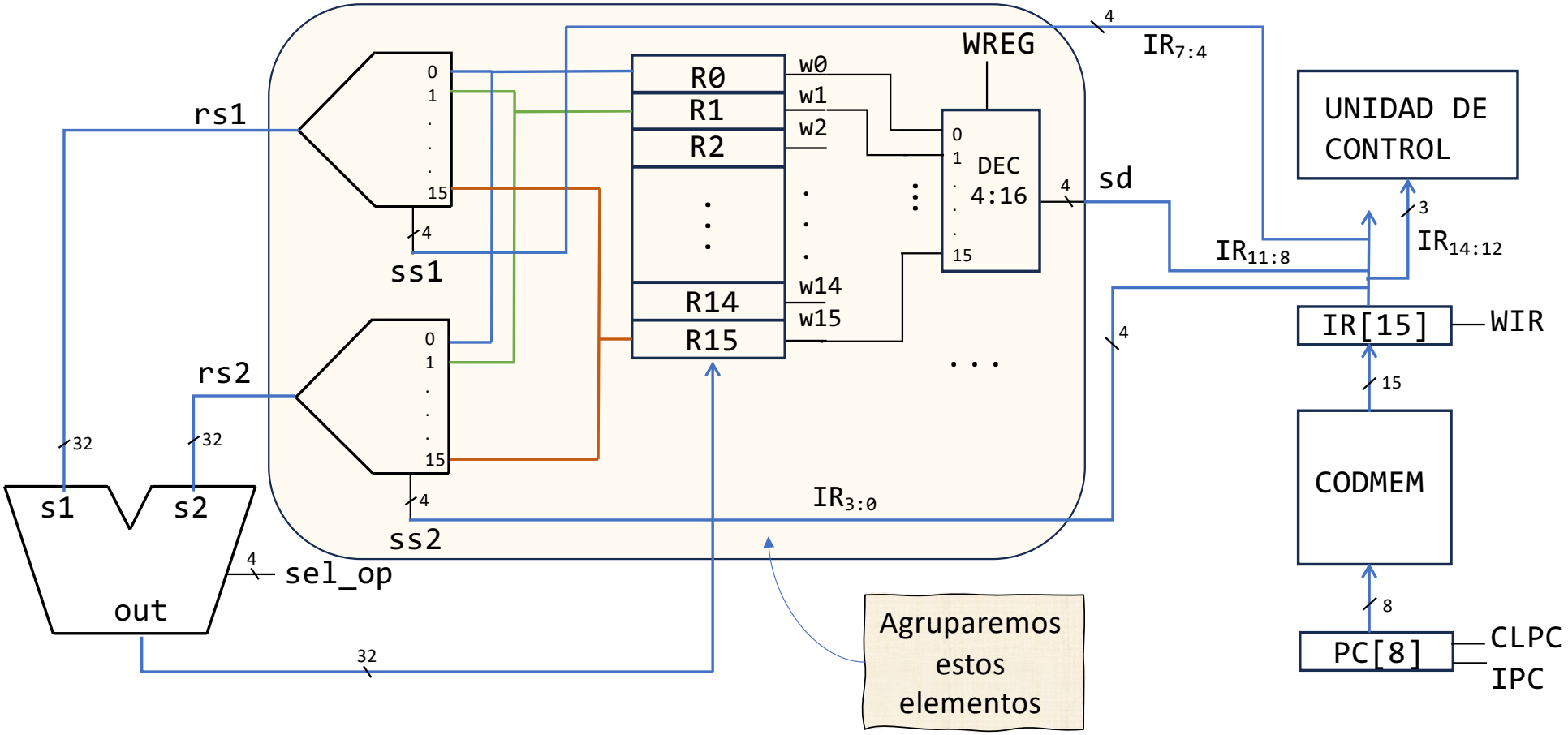
Ejemplo de uso: mediante un programa se realiza la operación R6 ← 3R4-2R1



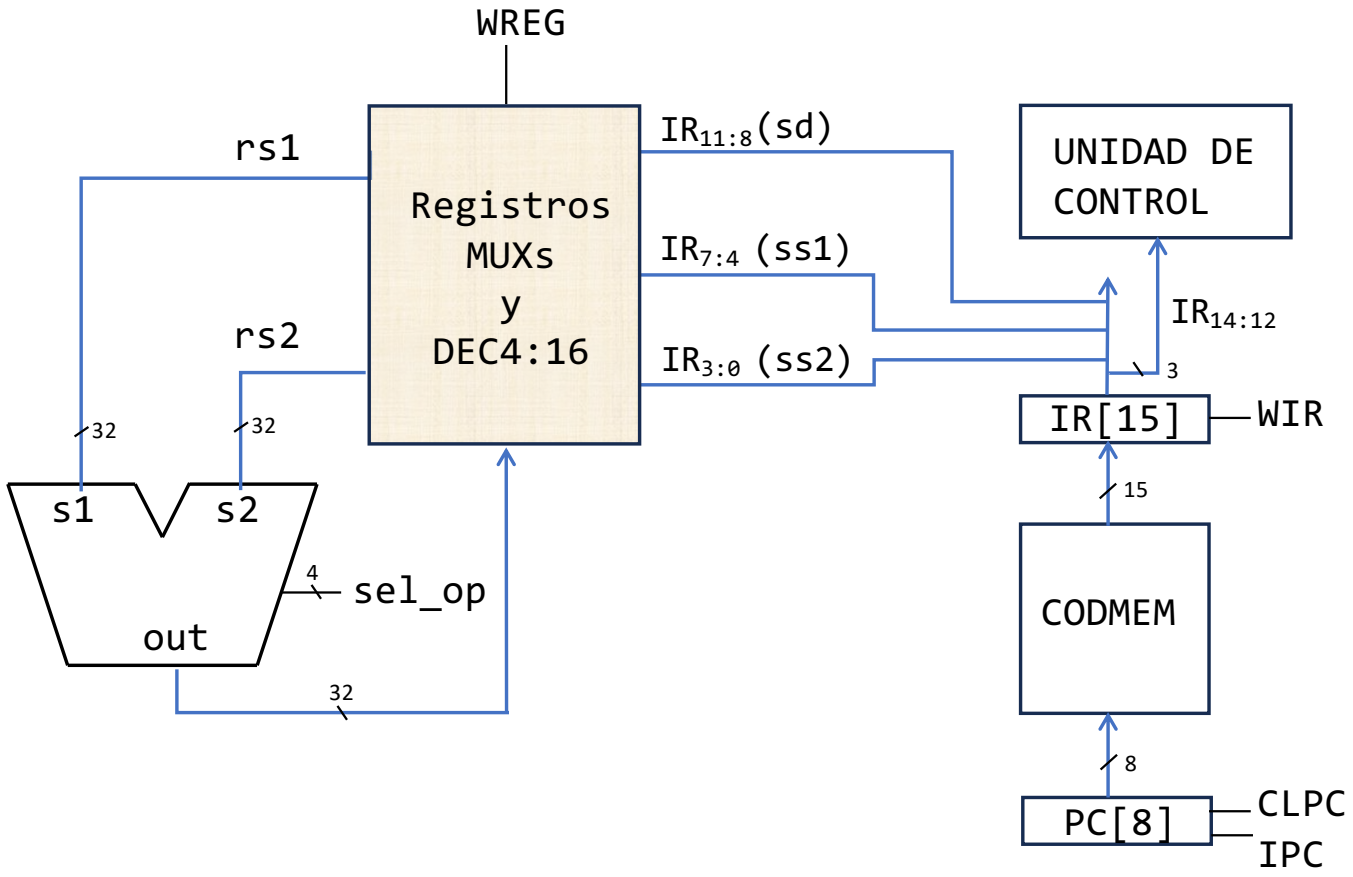
Computador Simple 1



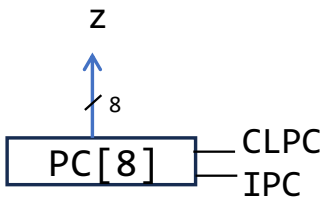
Computador Simple 1



Computador Simple 1

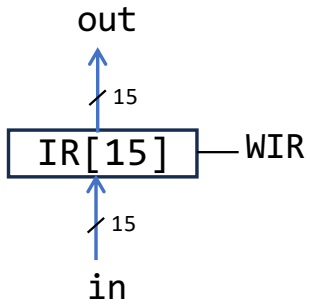


Descripción RT de los nuevos elementos



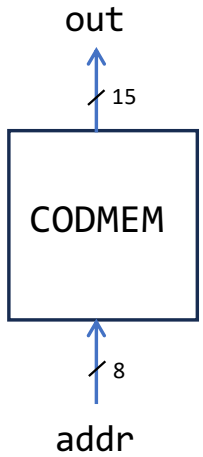
CLPC IPC	operación
1 x	PC ← 0
0 1	PC ← PC + 1
0 0	PC ← PC

z = [PC]



WIR	operación
0	IR ← IR
1	IR ← in

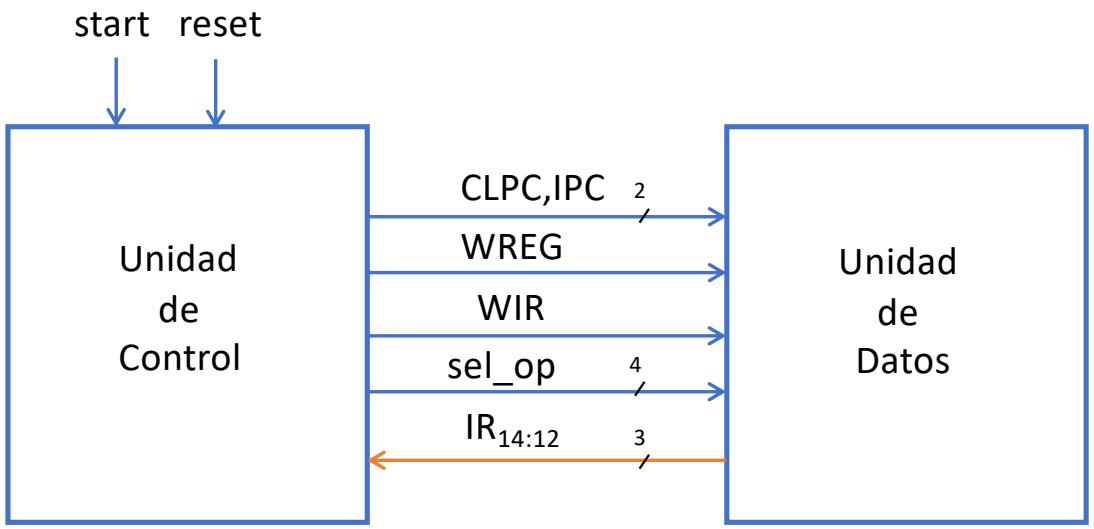
out = [IR]



out = CODMEM[addr]

No hay señal de escritura, damos por hecho que el programa está escrito en CODMEM

Computador Simple 1



Índice

1. Introducción.
2. La calculadora
3. Computador Simple 1
   (Automatización en la ejecución y almacenamiento de programa)
4. Computador simple 2
   (Almacenamiento de los datos y ampliación de modos de direccionamiento)
5. Computador Simple 3 / RISCY
   (Ejecución no secuencial: Instrucciones de salto)

Computador Simple 2

- Es necesario aumentar la capacidad de almacenamiento de datos del sistema y que no quede limitada a sus registros.

- Existen dos opciones para dotar al sistema de almacenamiento de datos y programas:

  - Utilizar un único sistema de memoria para datos e instrucciones (Arquitectura von Neumann)

  - Utilizar sistemas de memoria distintos para datos e instrucciones (Arquitectura Harvard)

Computador Simple 2

- La arquitectura Harvard, al tener sistemas de memoria separados con buses separados, permite acceder simultáneamente a las instrucciones y a los datos. Por ejemplo, permite buscar la siguiente instrucción mientras se está guardando un dato calculado en la ejecución de la instrucción actual.

- La arquitectura von Neumann tiene un único sistema de memoria para instrucciones y programa, residiendo estas en zonas diferentes del mapa de memoria. Al no tener duplicados los buses, el número de pines requeridos en el chip es mucho menor.

- Los procesadores modernos incorporan aspectos de ambas arquitecturas. Por ejemplo, normalmente, la memoria caché interna a la CPU se separa en dos (datos e instrucciones), mientras que la memoria principal es única (von Neumann)

Computador Simple 2

- El computador simple 2 tendrá arquitectura Harvard
- El conjunto de instrucciones también se amplía, entre otras, habrá instrucciones para el intercambio de datos entre los registros y la memoria de datos.
- Para aumentar el número de instrucciones es necesario aumentar los bits del código de operación. La nueva palabra de instrucción será de 32 bits.
- También se añaden nuevas formas de acceso a los operandos (modos de direccionamiento)
- Las direcciones de la memoria son de bytes, como las palabras son de 32 bits, cada palabra tendrá una dirección múltiplo de 4. (son palabras alineadas)

Computador Simple 2: instrucciones (i)

- Se mantienen las instrucciones del Computador Simple 1 con nuevos códigos de operación.

- Todas usan tres registros como operandos y se clasifican como tipo R.

CODOP ($IR_{31:24}$)	Sintaxis	Tipo	Función
0000 0001	add rd,rs1,rs2	R	rd ← rs1+rs2
0000 0010	sub rd,rs1,rs2	R	rd ← rs1-rs2
0000 1010	xor rd,rs1,rs2	R	rd ← rs1⊕rs2
0000 1011	or rd,rs1,rs2	R	rd ← rs1 or rs2
0000 1100	and rd,rs1,rs2	R	rd ← rs1 and rs2
0001 0000	sll rd,rs1,rs2	R	rd ← rs1 << $rs2_{4:0}$
0001 0001	srl rd,rs1,rs2	R	rd ← rs1 >> $rs2_{4:0}$
0000 0101	stop	-	nop

Computador Simple 2 : instrucciones (ii)

- Las nuevas instrucciones pueden tener diversos tipos (I, S o R), según su modo de expresar los operandos.

CODOP ($IR_{31:24}$)	Sintaxis	Tipo	Función
0000 0011	addi rd,rs1,inm12	I	rd ← rs1 + inm
0000 0110	lw rd,inm12(rs1)	I	rd ← mem(rs1 + inm)
0000 0111	sw rs2,inm12(rs1)	S	mem(rs1 + inm) ← rs2
0001 0010	sra rd,rs1,rs2	R	rd ← rs1 >>> $rs2_{4:0}$
0000 1101	xori rd,rs1,inm12	I	rd ← rs1 ⊕ inm
0000 1110	ori rd,rs1,inm12	I	rd ← rs1 or inm
0000 1111	andi rd,rs1,inm12	I	rd ← rs1 and inm
0001 0011	slli rd,rs1,inm12	I	rd ← rs1 << $inm_{4:0}$
0001 0100	srli rd,rs1,inm12	I	rd ← rs1 >> $inm_{4:0}$
0001 0101	srai rd,rs1,inm12	I	rd ← rs1 >>> $inm_{4:0}$

inm es inm12 extendido en signo a 32 bits

Instrucciones para intercambio de datos con memoria

Hay instrucciones cuyo operando aparece en la propia palabra de instrucción

Computador Simple 2

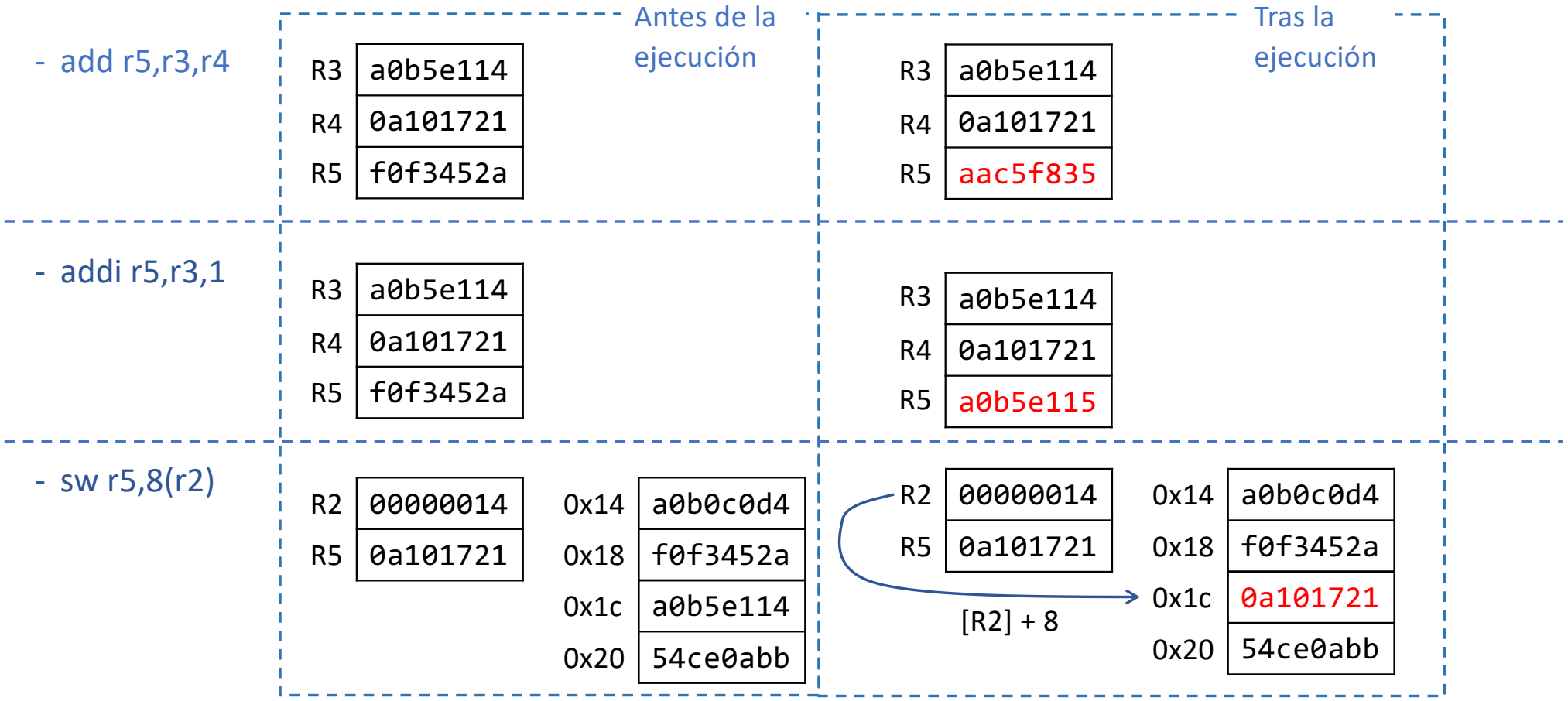
- La forma de acceder a los operandos (modo de direccionamiento) es diferente según el tipo de instrucción.

- En el CS2 hay tres modos de direccionamiento:
  - Direccionamiento de registro: los operandos están en registros (por ej. add r5,r3,r4)

  - Direccionamiento indirecto: uno de los operandos está en la memoria de datos y se localiza mediante su dirección en dicha memoria. Esta dirección se expresa como la suma de un registro y un dato de 12 bits (inm12) contenido en la propia palabra de instrucción. (por ej. lw r5,20(r2))

  - Direccionamiento inmediato: uno de los operandos es un dato de 12 bits (inm12) contenido en la propia palabra de instrucción. (por ej. addi r5,r3,25)

Computador Simple 2



Computador Simple 2

- sra r5,r3,r4

(r5 ← r3 >>> $r4_{4:0}$)

Antes de la ejecución

R3	a0b5e114
R4	0a101724
R5	f0f3452a

1010 0000 1011 0101 1110 0001 0001 0100

Tras la ejecución

R3	a0b5e114
R4	0a101724
R5	fa0b5e11

1111 1010 0000 1011 0101 1110 0001 0001 0100

- slli r5,r3,0xa74

(r5 ← r3 << 0b10100)
(r5 ← r3 << $inm_{4:0}$)

R3	a0b5e114
R4	0a101721
R5	f0f3452a

1010 0000 1011 0101 1110 0001 0001 0100

R3	a0b5e114
R4	0a101721
R5	11400000

0001 0001 0100 0000 0000 0000 0000 0000

1010 0000 1011 0101 1110

Computador Simple 2

Todas las instrucciones tienen 32 bits.

En el CS2 los operandos de las instrucciones pueden encontrarse en registros, memoria o en la propia palabra de instrucción. Esto da lugar a 3 posibles formatos de instrucción (tipos R, I y S):

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
CODOP								RD				RS1				RS2				-												R
CODOP								RD				RS1				-				inm12												I
CODOP								-				RS1				RS2				inm12												S

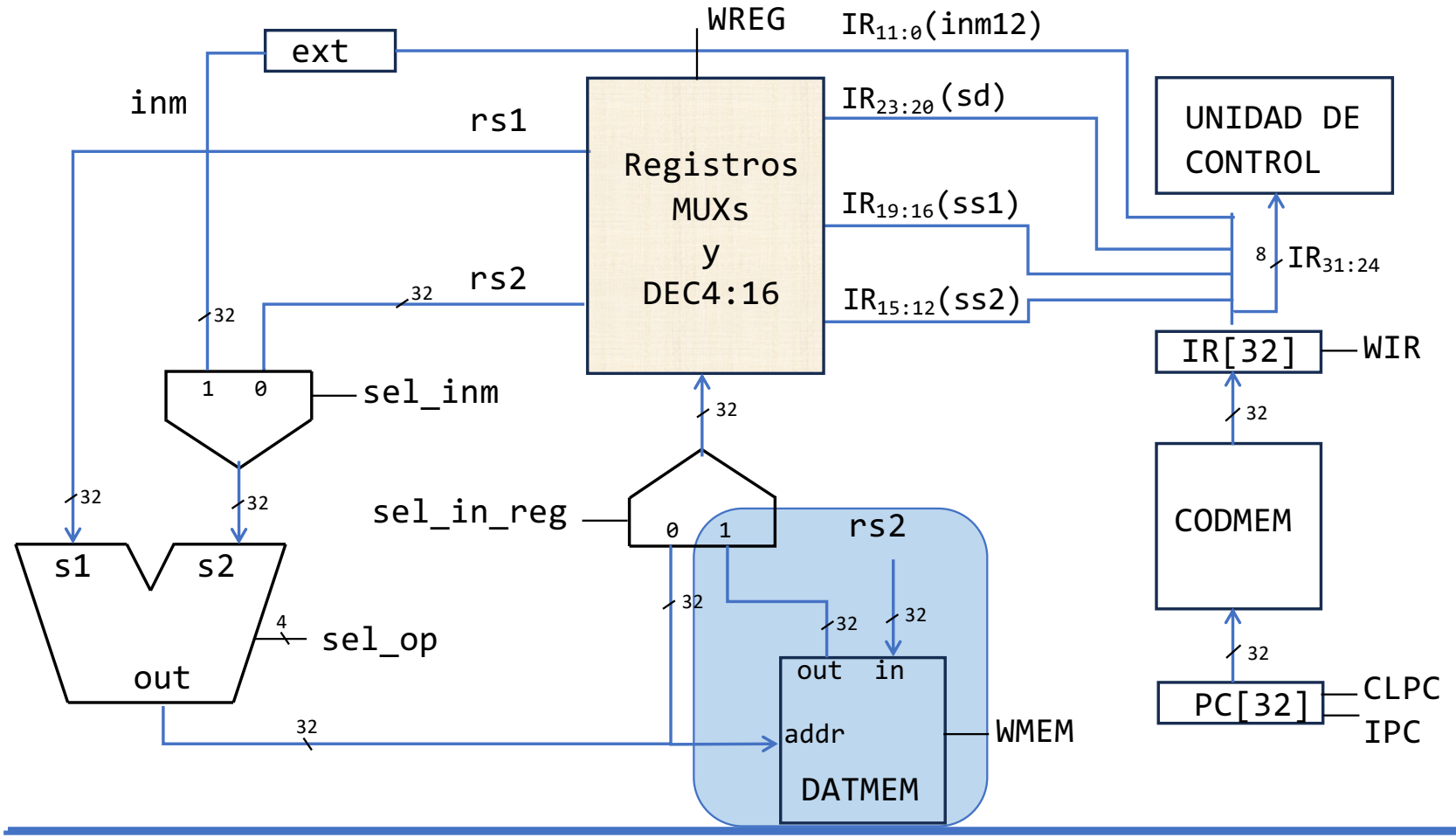
```
add r5,r3,r4:
addi r5,r3,1:
sw r5,3(r2):
```

0000 0001	0101	0011	0100	xxxx xxxx xxxx
0000 0011	0101	0011	xxxx	0000 0000 0001
0000 0111	xxxx	0010	0101	0000 0000 0011

código máquina

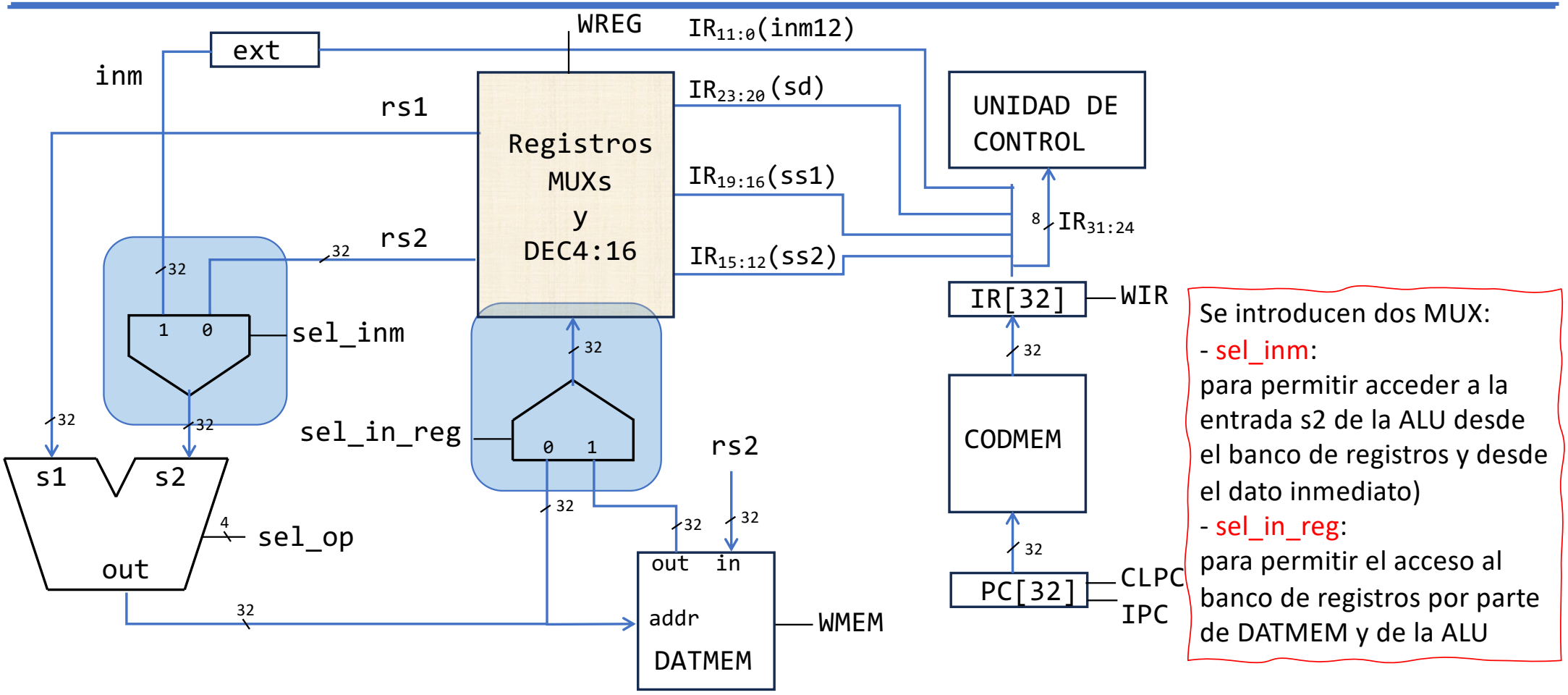
Las posiciones marcadas con “x” han de tomar el valor: 0 o 1, no importa cuál. Pero ha de especificarse un valor.

Computador Simple 2



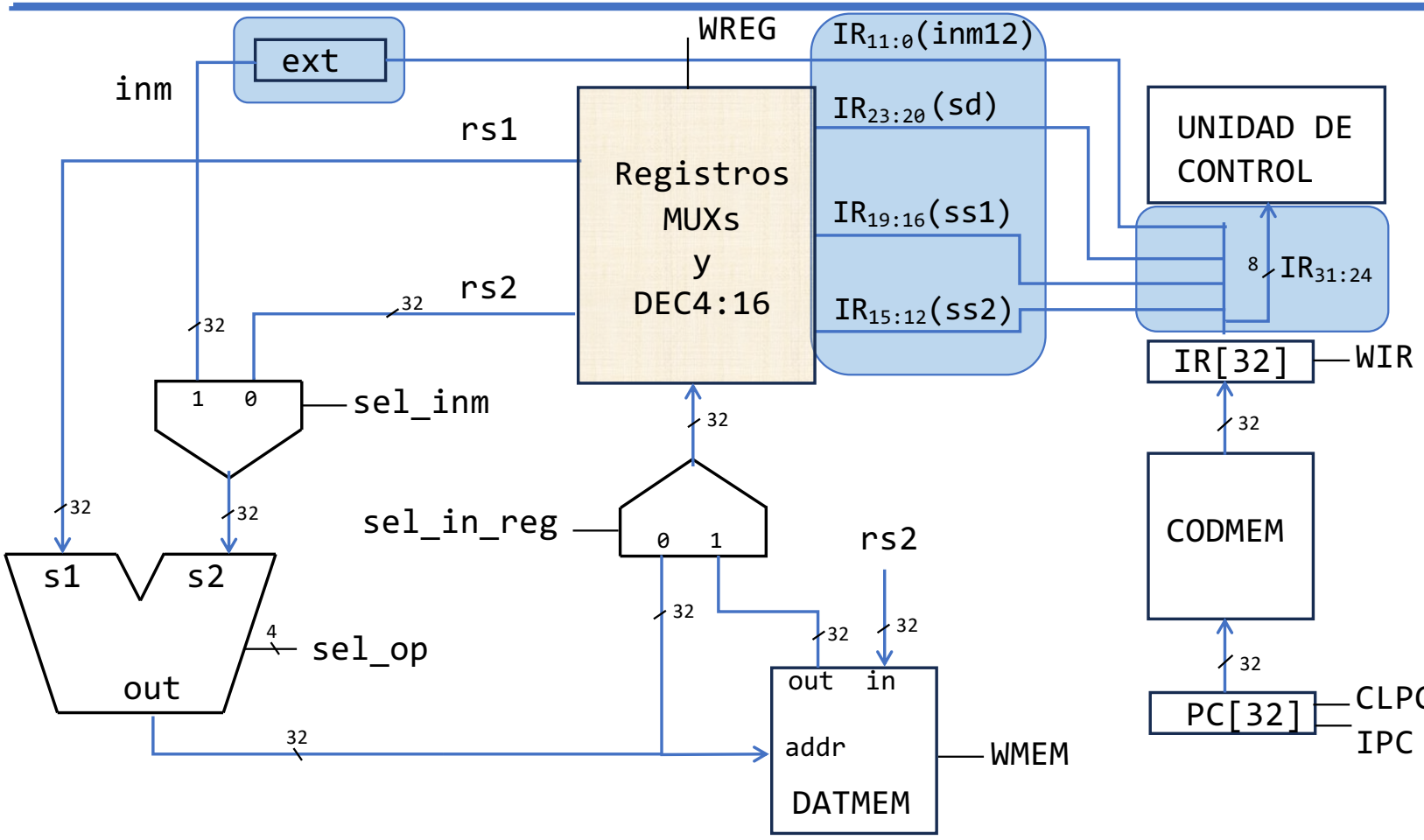
Se introduce la memoria de datos DATMEM
Se conecta:
- con el banco de registros mediante sus 2 buses de datos (in y out), para trasladar datos en ambos sentidos
- con la ALU mediante su bus de direcciones, ya que la ALU calcula rs1 + inm12 (instrucciones lw y sw)

Computador Simple 2



Se introducen dos MUX:
- sel_inm:
para permitir acceder a la entrada s2 de la ALU desde el banco de registros y desde el dato inmediato)
- sel_in_reg:
para permitir el acceso al banco de registros por parte de DATMEM y de la ALU

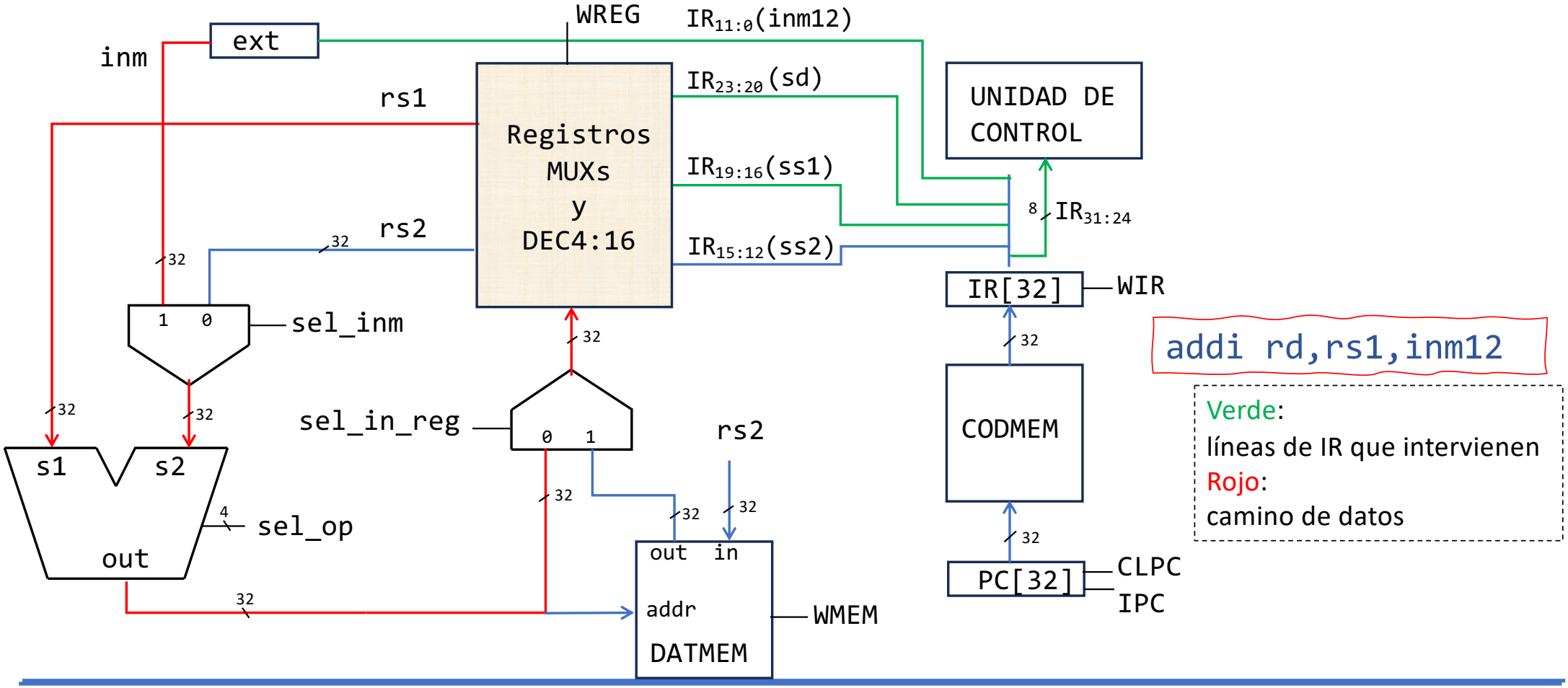
Computador Simple 2



Se introduce una unidad de extensión de signo para los inmediatos de 12 bits ya que han de sumarse con datos de 32 bits.

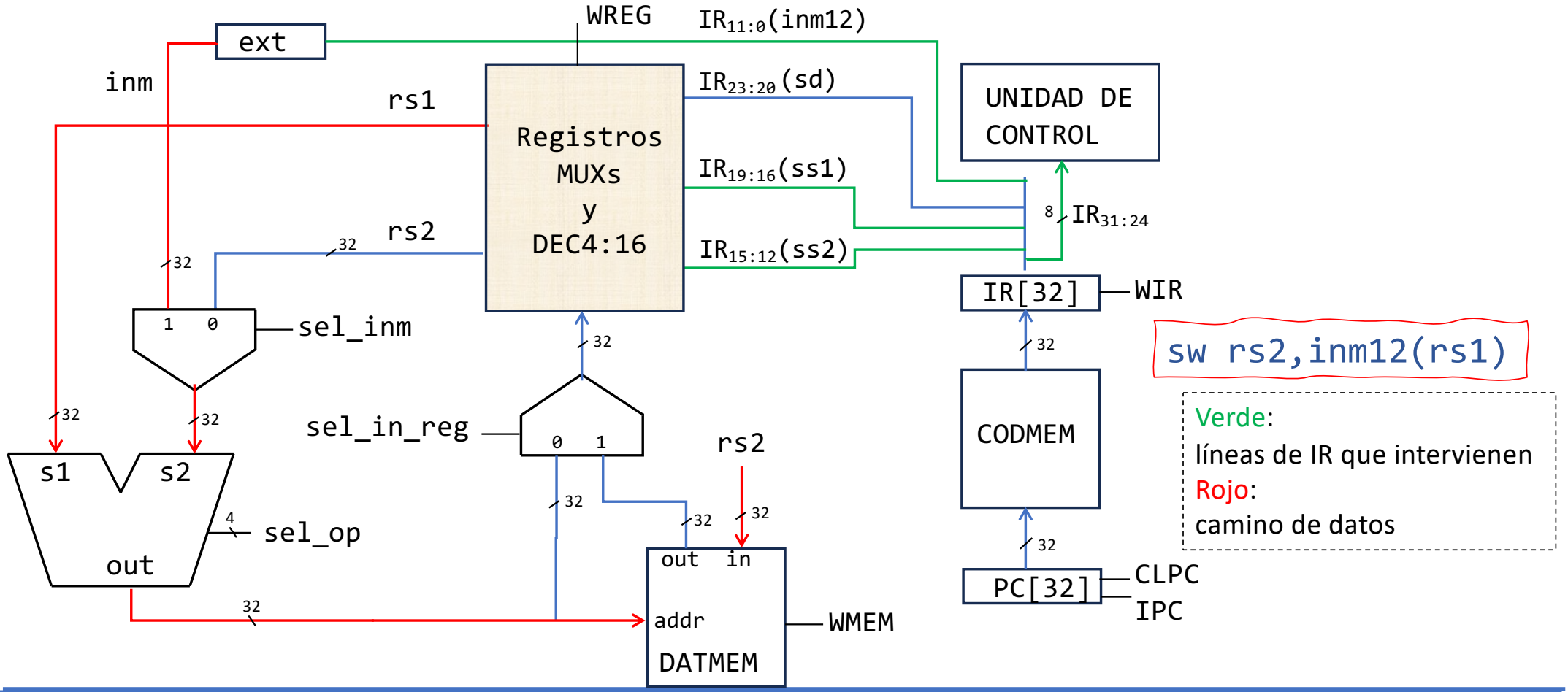
Se han adecuado los tamaños de los buses

Computador Simple 2: instrucción addi



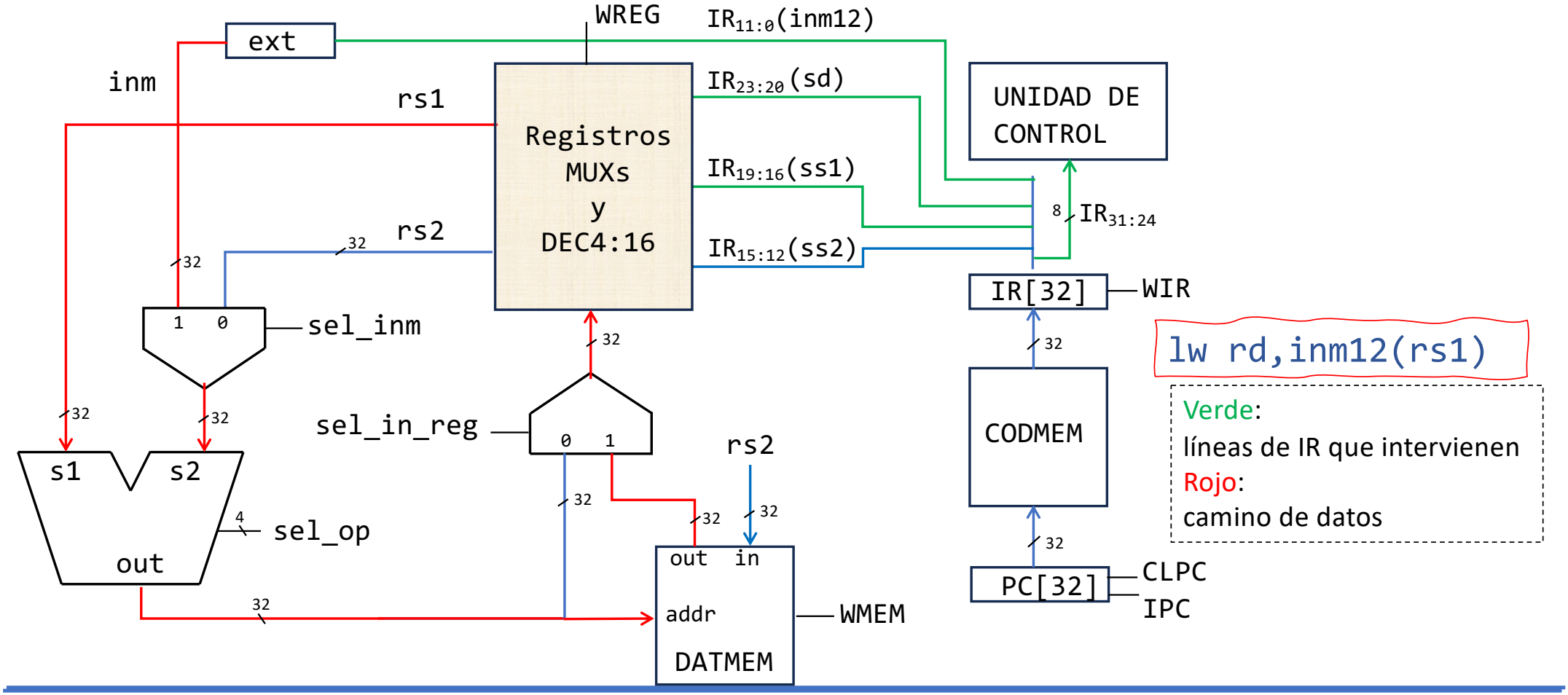
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
CODOP								RD				RS1				-				inm12												I

Computador Simple 2: instrucción sw



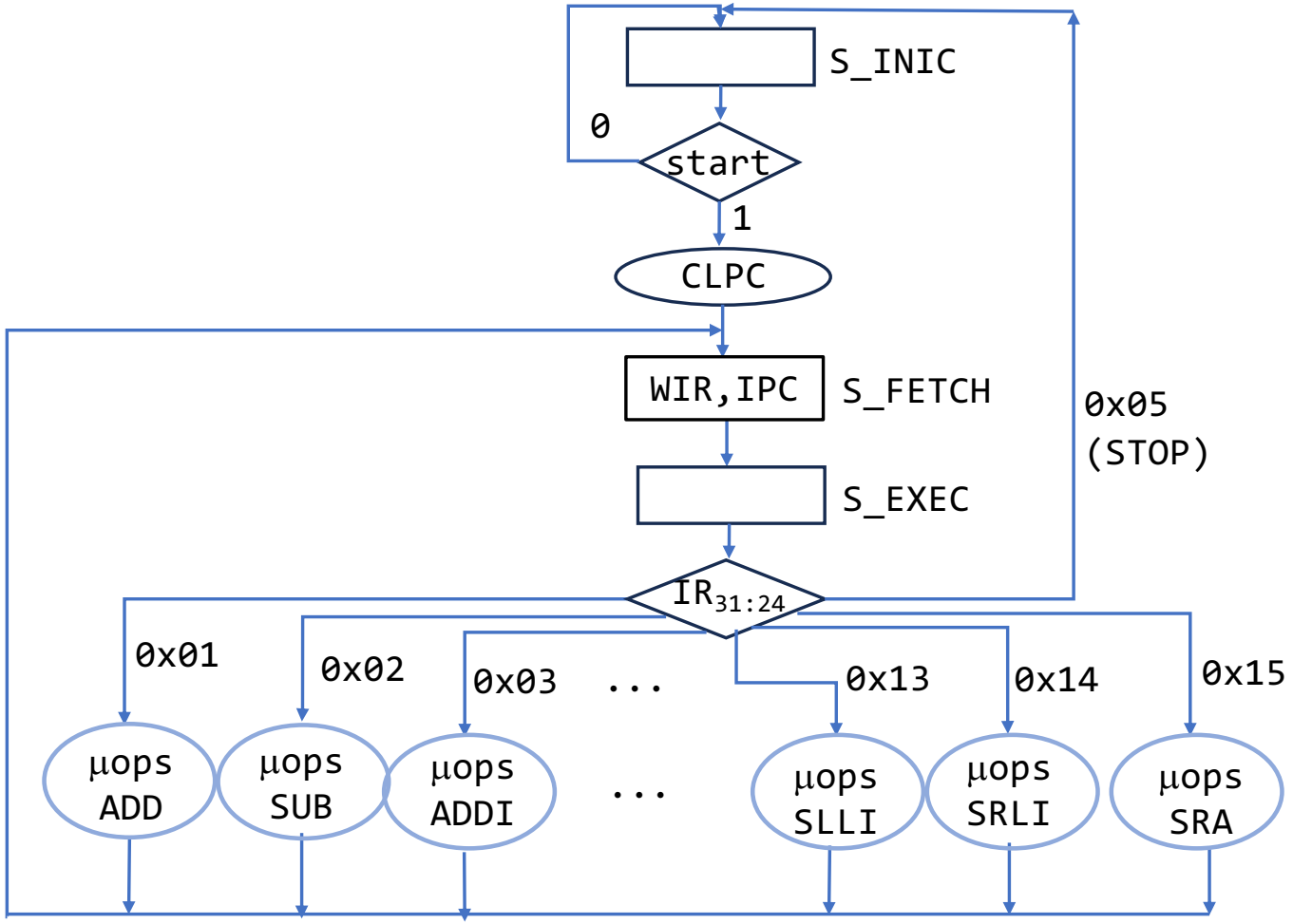
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
CODOP								-				RS1				RS2				inm12												S

Computador Simple 2: instrucción lw



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
CODOP								RD				RS1				-				inm12												I

CARTA ASM


S_INIC
0
start
1
CLPC
WIR,IPC
S_FETCH
S_EXEC
IR31:24
0x05
(STOP)
0x01
0x02
0x03
...
0x13
0x14
0x15
µops ADD
µops SUB
µops ADDI
...
µops SLLI
µops SRLI
µops SRA

sel_op	operación alu
0000	out=s1+s2
0001	out=s1-s2
0010	out=s2
0011	out=s1⊕s2
0100	out=s1 or s2
0101	out=s1 and s2
0110	out=s1<<$s2_{4:0}$
0111	out=s1>>$s2_{4:0}$
1000	out=s1>>>$s2_{4:0}$

CARTA ASM (instrucciones)

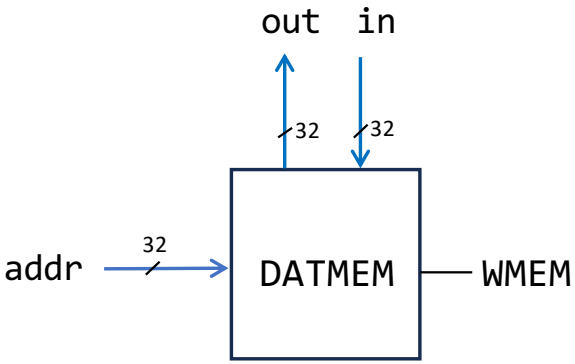
En el CS2, el ciclo de ejecución dura un ciclo de reloj para todas las instrucciones.
La instrucción más lenta determina la duración del ciclo de reloj.

add	sub	addi	lw	sw	xor	or	and	xori
WREG	WREG sel_op_0	WREG sel_inm	WREG sel_inm sel_in_reg	WMEM sel_inm	WREG $sel\_op_{1:0}$	WREG sel_op_2	WREG $sel\_op_{2,0}$	WREG $sel\_op_{1:0}$ sel_inm

ori	andi	sll	srl	sra	slli	srli	srai
WREG sel_op_2 sel_inm	WREG $sel\_op_{2,0}$ sel_inm	WREG $sel\_op_{2:1}$	WREG $sel\_op_{2:0}$	WREG sel_op_3	WREG $sel\_op_{2:1}$ sel_inm	WREG $sel\_op_{2:0}$ sel_inm	WREG sel_op_3 sel_inm

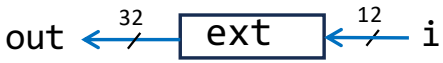
Es importante recordar que cuando no se activa ningún bit de sel_op, la ALU va a sumar.
Eso pasa en add, addi, sw y lw, que necesitan que la ALU sume. Hay que recordar que la dirección de memoria a la que se accede en lw y sw se obtiene sumando un registro con un dato inmediato.

Descripción RT de los elementos nuevos

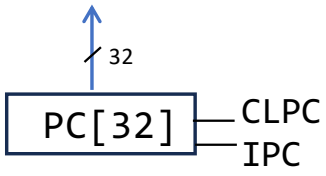


WMEM	operación
0	DATMEM ← DATMEM
1	DATMEM[addr] ← in

out = DATMEM[addr]

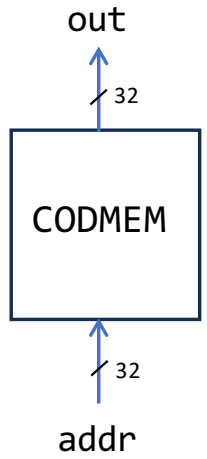


$out = in_{11}\ in_{11} \ldots in_{11}\ in_{10} \ldots in_0$

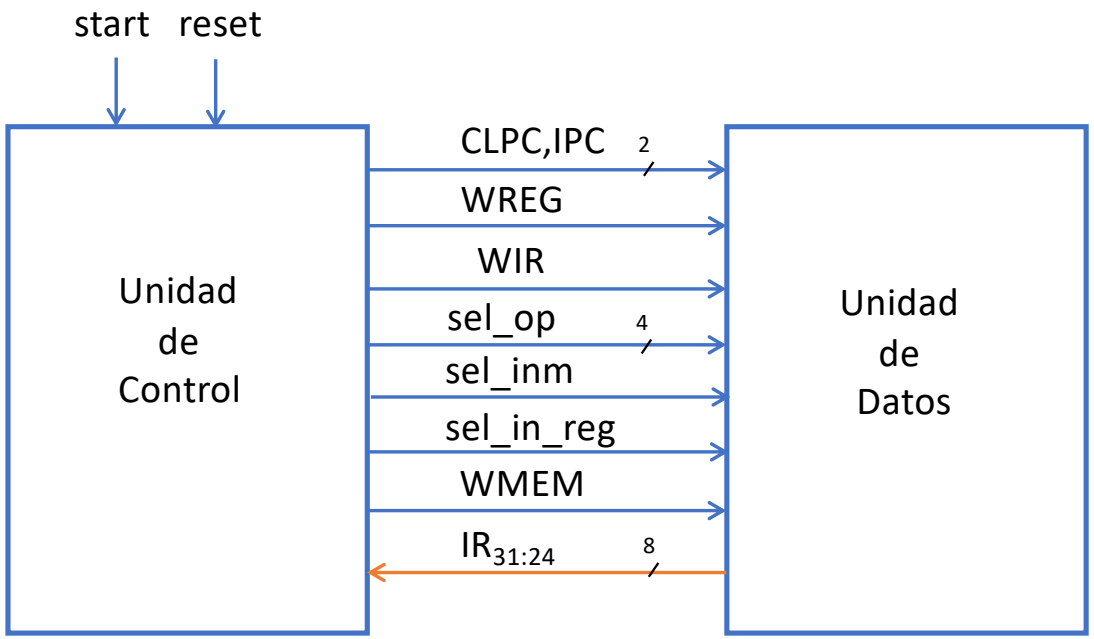


CLPC IPC	operación
1 X	PC ← 0
0 1	PC ← PC + 4
0 0	PC ← PC

z = [PC]



Computador Simple 2



Índice

1. Introducción.
2. La calculadora
3. Computador Simple 1
   (Automatización en la ejecución y almacenamiento de programa)
4. Computador simple 2
   (Almacenamiento de los datos y ampliación de modos de direccionamiento)
5. Computador Simple 3 / RISCY
   (Ejecución no secuencial: Instrucciones de salto)

Computador Simple 3 : RISC Y

- En el computador simple 2 se han introducido mejoras al CS1, pero hay una deficiencia importante ya que no es posible realizar saltos en los programas.

- En el CS3 se van a incorporar instrucciones de salto al juego de instrucciones del CS2

- Para introducir las nuevas instrucciones no son necesarios nuevos formatos ni nuevos modos de direccionamiento. Tampoco habrá grandes cambios en la unidad de datos.

- Este computador, como los anteriores, se ha diseñado de modo que tenga características similares a las del computador RISC V que veremos en el tema siguiente, por este motivo al CS3 (que es el más completo de la serie de los CS) le denominaremos RISC Y

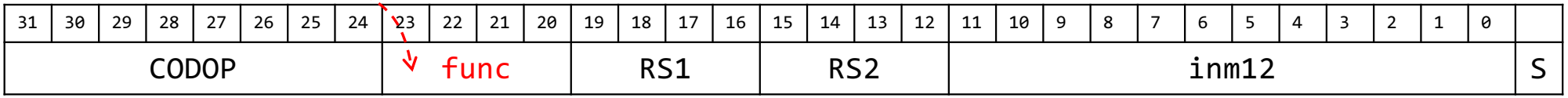
RISC Y

Se incluyen 6 nuevas instrucciones:

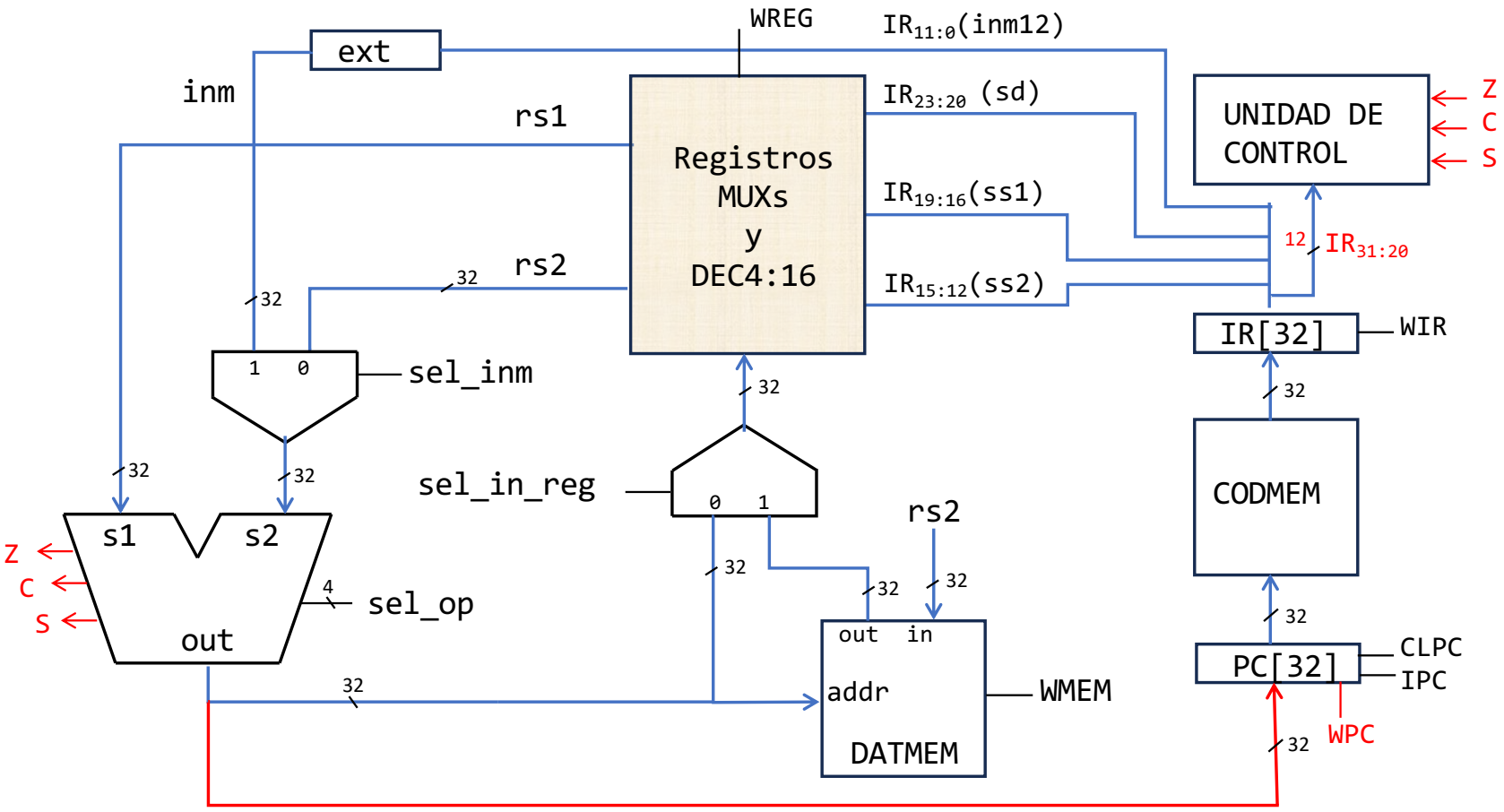
- son saltos condicionales, se comparan los datos contenidos en rs1 y rs2 y según el resultado de la comparación se realiza o no el salto.

Hay 6 tipos de saltos. Todos tienen el mismo código de operación. Se distinguen por el valor de func.

CODOP ($IR_{31:24}$)	func	Sintaxis	Tipo	Función
0000 1001	0000	beq rs1,rs2,inm12	S	si rs1==rs2 pc ← inm
0000 1001	0001	bne rs1,rs2,inm12	S	si rs1!=rs2 pc ← inm
0000 1001	0010	blt rs1,rs2,inm12	S	si rs1<rs2 pc ← inm
0000 1001	0011	bge rs1,rs2,inm12	S	si rs1≥rs2 pc ← inm
0000 1001	0100	bltu rs1,rs2,inm12	S	como blt para datos sin signo
0000 1001	0101	bgeu rs1,rs2,inm12	S	como bge para datos sin signo



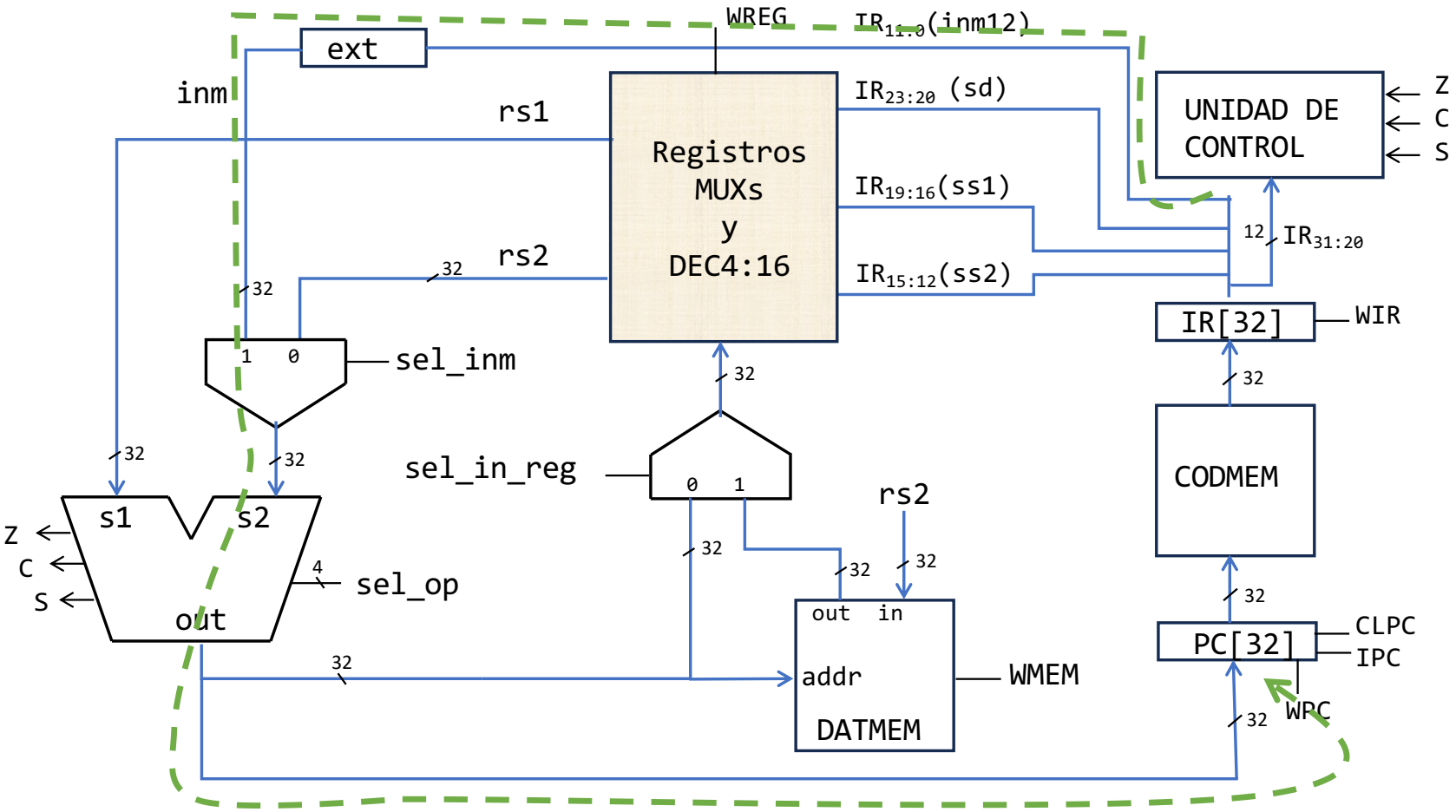
RISC Y: cambios en la unidad de datos



En rojo aparecen los cambios:

- Señales de estado Z, C y S que se obtienen en la ALU y son entradas de la unidad de control
- Bus desde la ALU a PC y señal WPC para cargar la dirección de salto.
- El bus desde IR a la unidad de control es ahora de 12 bits ya que los bits de func son necesarios.

RISC Y



La dirección de salto es transferida a PC desde IR a través del MUX y de la ALU

(operación transferencia de s2 en la alu: sel_op = 0010)

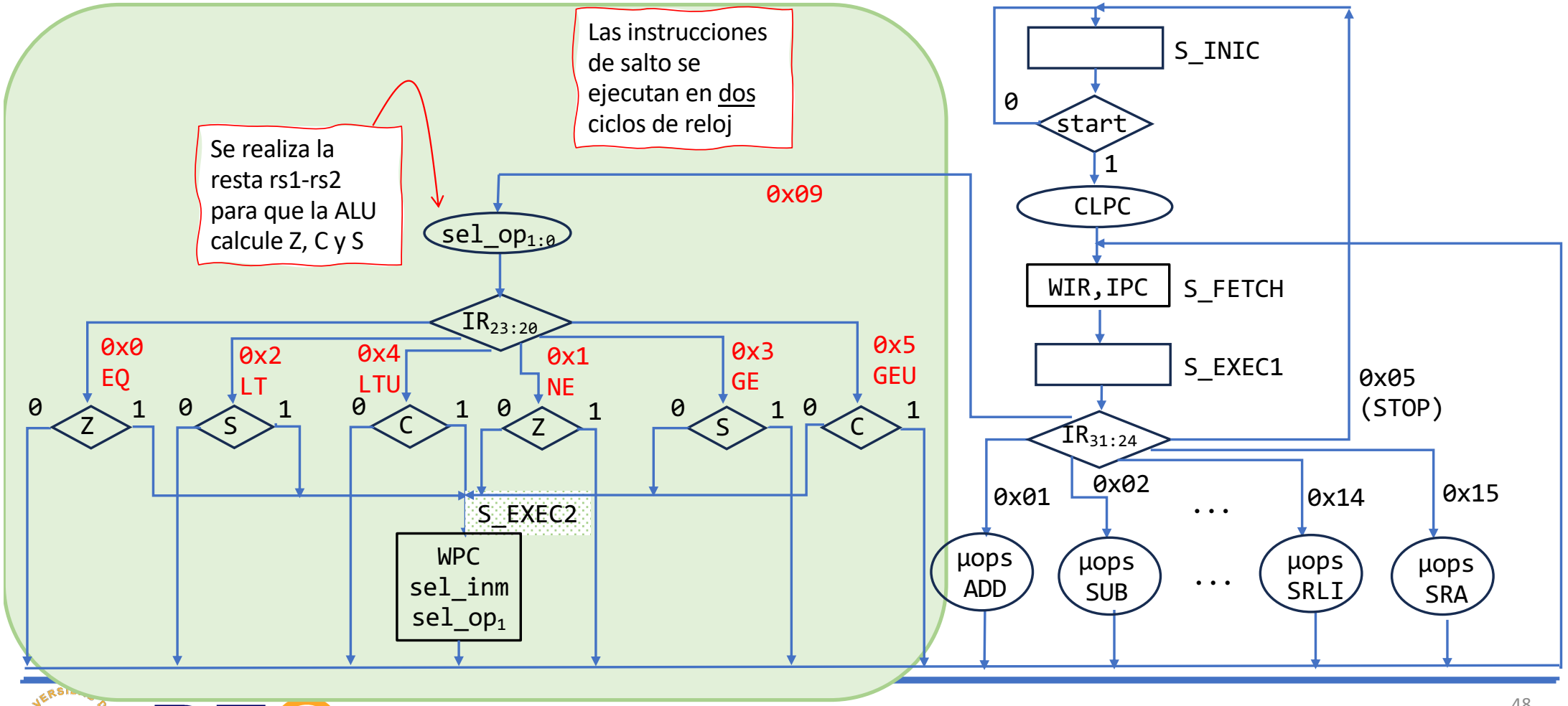
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
CODOP								func				RS1				RS2				inm12												S

RISC Y

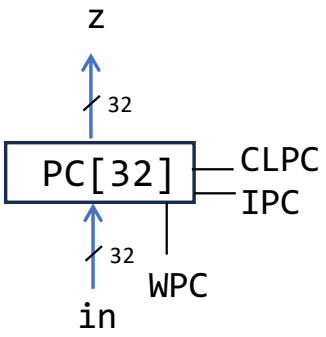
En los saltos condicionales, tras comparar los datos contenidos en rs1 y rs2 mediante una resta, se evalúan las salidas de estado de la ALU (Z, C o S según la instrucción).

CODOP ($IR_{31:24}$)	func	Sintaxis	Condición de salto
0000 1001	0000	beq rs1,rs2,inm12	Z=1
0000 1001	0001	bne rs1,rs2,inm12	Z=0
0000 1001	0010	blt rs1,rs2,inm12	S=1
0000 1001	0011	bge rs1,rs2,inm12	S=0
0000 1001	0100	bltu rs1,rs2,inm12	C=1
0000 1001	0101	bgeu rs1,rs2,inm12	C=0

CARTA ASM (modificación a la del CS2)

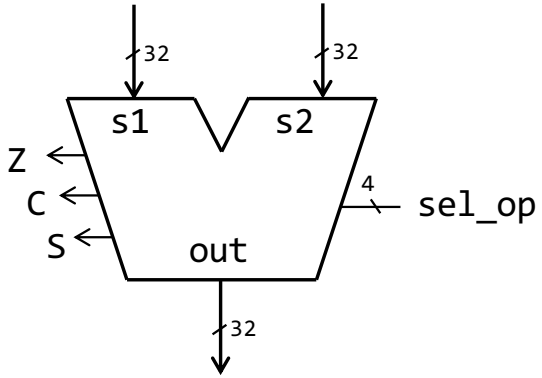


Descripción RT de los elementos que han cambiado



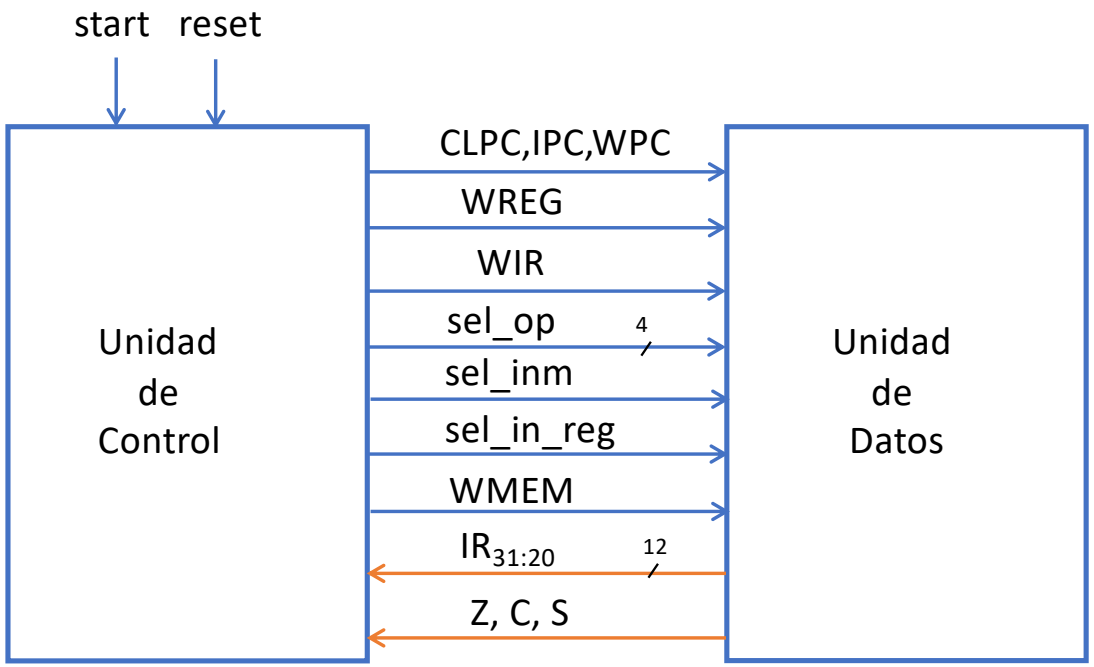
CLPC WPC IPC	operación
1 X X	PC ← 0
0 1 X	PC ← in
0 0 1	PC ← PC + 4
0 0 0	PC ← PC

z = [PC]



Salidas de estado:
C – carry (acarreo de salida)
Z – salida Zero (1 si el resultado es 0)
S – signo verdadero del resultado

RISC Y


start
reset
Unidad de Control
CLPC,IPC,WPC
WREG
WIR
sel_op
4
sel_inm
sel_in_reg
WMEM
IR31:20
12
Z, C, S
Unidad de Datos

RISC Y

Programación:

- Por similitud a RISCV (tema siguiente) nombraremos los registros R1… R15 como X1…X15 y consideraremos que x0 siempre contiene 0x0000 0000 (no puede escribirse)

- Las direcciones de la memoria son de bytes, como las palabras son de 32 bits, cada palabra tendrá una dirección múltiplo de 4 (palabras alineadas)

- Los incrementos del contador de programa (PC) en el ciclo de búsqueda serán de 4 unidades (PC←PC + 4): se apunta a la siguiente instrucción, es decir, a los siguientes 32 bits (4 bytes)

- Al escribir una palabra en memoria se asignarán los bytes menos significativos de la palabra a las direcciones más bajas (criterio Little Endian)

RISC Y

Programación:

Ejemplo: programa que escribe la palabra 0x1245ab70 en la dirección 0x20 de la memoria de datos (little endian).

POS.	PROGRAMA	OPERACIÓN
N	addi x1,x0,0x124	x1 ← 0x124
N+4	slli x1,x1,12	x1 ← x1 << 12
N+8	addi x1,x1,0x5ab	x1 ← x1 + 0x5ab
N+12	slli x1,x1,12	x1 ← x1 << 12
N+16	addi x1,x1,0x070	x1 ← x1 + 0x070
N+20	sw x1,0x20(x0)	memdat(0x20) ← x1

0x20	0x21	0x22	0x23
0x70	0xab	0x45	0x12

Las posiciones que ocupan las líneas de programa en MEMCOD se incrementan de 4 en 4

Cada línea de programa se codifica con 32 bits

El byte menos significativo de la palabra se escribe en la dirección más baja

RISC Y

Programación:

Ejemplo: programa que toma un valor N almacenado en el registro X15 y proporciona en el registro X14 el valor N + (N-1) + (N-2) + … + 1

POS	PROGRAMA	OPERACIÓN
0x0	addi x14,x0,0	x14 ← 0
0x4	add x13,x15,x0	x13 ← x15
0x8	add x14,x13,x14	x14 ← x13 + x14
0xc	addi x13,x13,0xfff	x13 ← x13 – 1
0x10	bne x13,x0,0x8	si x13 ≠ 0, pc ← 0x8
0x14	stop	

RISC Y

Programación:

Ejemplo: programa que lee un dato del registro X15 lo compara con 64 y escribe el menor de los dos en X14

POS	PROGRAMA	OPERACIÓN
0x0	addi x14,x0,0x40	x14 ← 64
0x4	blt x14,x15,0xc	salta si 64 < x15
0x8	add x14,x0,x15	x14 ← x15
0xc	stop	

RISC Y

Programación:

Ejemplo: programa que calcula y almacena en x14 de forma consecutiva los 5 primeros múltiplos del número que se indique en X15

POS	PROGRAMA	OPERACIÓN
0x0	addi x14,x0,0	x14 ← 0
0x4	addi x7,x0,5	x7 ← 5
0x8	add x14,x15,x14	x14 ← x14 + x15
0xc	addi x7,x7,-1	x7 ← x7 - 1
0x10	bne x7,x0, 0x8	salta y repite si x7 no es 0
0x14	stop	

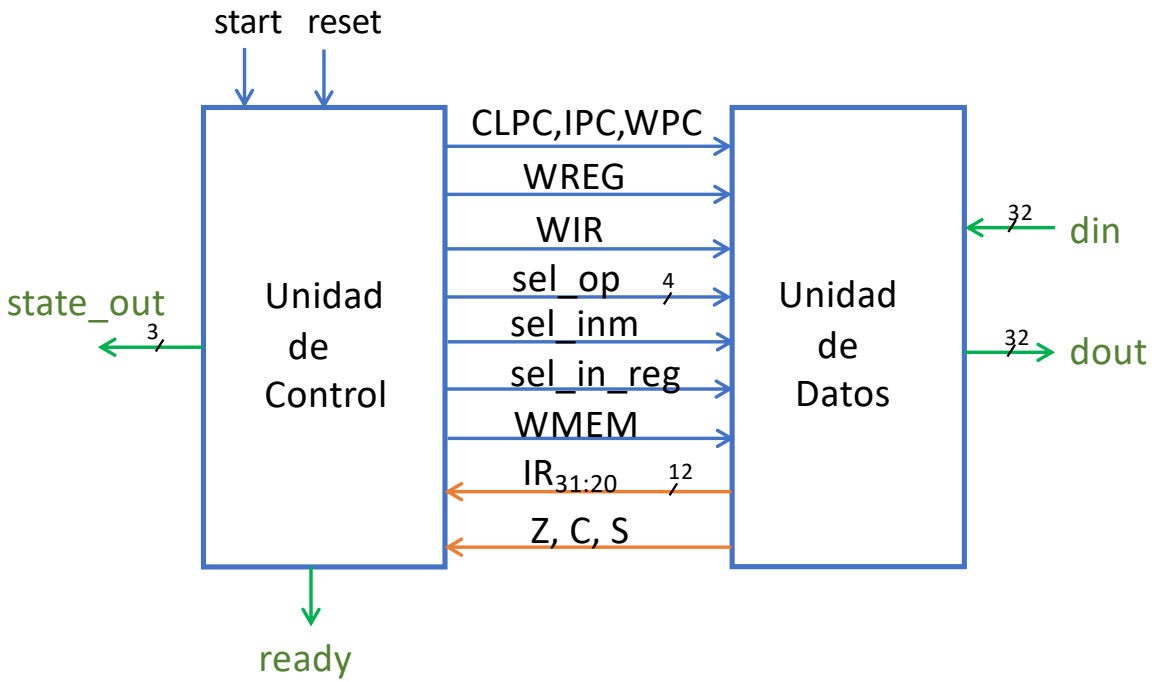
RISC Y → versión para programación en placa

En la descripción Verilog añadiremos ciertos cambios que permiten obtener propiedades interesantes.

1) Buses *din* y *dout*: din se conectará a la entrada de X15 y dout a la salida de X14, lo que permitirá tener entrada y salida muy básica
(los conectaremos a los *switches* y *displays* de la placa respectivamente)

2) Bus *state_out* que permite monitorizar el estado de la Ucontrol (los conectaremos a los leds de la placa)

3) Señal *ready*, que pondrá la u. de control a 1 mientras esté en el estado de espera (nos permitirá hacer chequeos de actividad)



4) Los buses de dirección de las memorias DATMEM y CODMEM se han definido con menos de 32 bits, ya que no necesitamos tanto espacio de memoria.

RISC Y: archivo globals.vh

```
`define __RYGLOBALS
//// General definitions
`define DATA_START 12'h0
`define DATA_END 12'h0ff
```

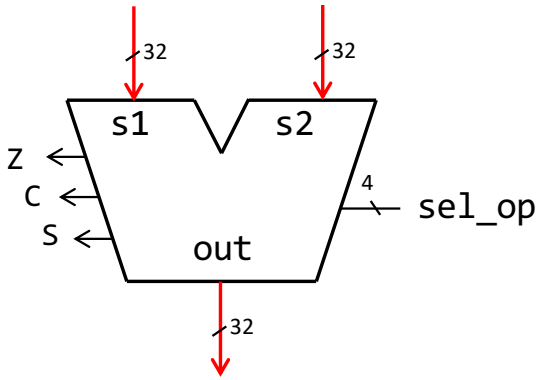
En el archivo globals.vh definimos macros para los valores binarios de sel_op

```
//// Assembly operation codes
`define ADD 8'h01
`define SUB 8'h02
`define ADDI 8'h03
`define LUI 8'h04
`define STOP 8'h05
`define LW 8'h06
`define SW 8'h07
`define B 8'h09
`define XOR 8'h0a
`define OR 8'h0b
`define AND 8'h0c
`define XORI 8'h0d
`define ORI 8'h0e
`define ANDI 8'h0f
`define SLL 8'h10
`define SRL 8'h11
`define SRA 8'h12
`define SLLI 8'h13
`define SRLI 8'h14
`define SRAI 8'h15
```

```
//// Registers
`define X0 4'h0
`define X1 4'h1
`define X2 4'h2
`define X3 4'h3
`define X4 4'h4
`define X5 4'h5
`define X6 4'h6
`define X7 4'h7
`define X8 4'h8
`define X9 4'h9
`define X10 4'ha
`define X11 4'hb
`define X12 4'hc
`define X13 4'hd
`define X14 4'he
`define X15 4'hf
```

```
//// ALU operation codes
`define ALU_ADD 4'd0
`define ALU_SUB 4'd1
`define ALU_TR2 4'd2
`define ALU_XOR 4'd3
`define ALU_OR 4'd4
`define ALU_AND 4'd5
`define ALU_SLL 4'd6
`define ALU_SRL 4'd7
`define ALU_SRA 4'd8

//// Status register flag positions in funcs
`define EQ 4'd0
`define NE 4'd1
`define LT 4'd2
`define GE 4'd3
`define LTU 4'd4
`define GEU 4'd5
```

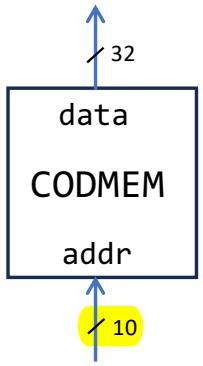
RISC Y: descripción Verilog (ALU)



En el archivo globals.vh definimos macros para los valores binarios de sel_op

```
`include "globals.vh"
module alu #( parameter BW = 32 )(
   input  wire [BW-1:0] s1,
   input  wire [BW-1:0] s2,
   input  wire [3:0] sel_op,
   output reg  [BW-1:0] out,
   output reg C, Z, S
);
    reg V,N;
    always @* begin
      out = 'b0;
      {C, Z, S} = 'b0;
      case(sel_op)
      `ALU_ADD:   out = s1 + s2;
      `ALU_SUB: begin
         out = s1 - s2;
         C =   ~s1[BW-1] & s2[BW-1]   |  s2[BW-1] & out[BW-1]
             | ~s1[BW-1] & out[BW-1];
         V =    s1[BW-1] & ~s2[BW-1] & ~out[BW-1]
             | ~s1[BW-1] & s2[BW-1] & out[BW-1];
         Z = ~|out;
         N = out[BW-1];
         S = V ^ N;
      end
      `ALU_TR2:   out = s2;
      `ALU_XOR:  out = s1 ^ s2;
      `ALU_OR:    out = s1 | s2;
      `ALU_AND: out = s1 & s2;
      `ALU_SLL:   out = s1 << s2[4:0];
      `ALU_SRL:  out = s1 >> s2[4:0];
      `ALU_SRA: out = $signed(s1) >>> s2[4:0];
       //out = {{32{s1[31]}}, s1} >> s2[4:0];

      default: out = 'bx;
      endcase
   end

endmodule
```

RISC Y: descripción Verilog (CODMEM)



Reducimos el tamaño de la memoria para facilitar su implementación.

```
`include "globals.vh"
module code_mem (
   input  wire [ 9:0] addr,
   output wire [31:0] data
   );

   reg [31:0] code[0:1023];    // memory array
   integer i;

   assign data = code[addr];

   initial begin
      // inicialización a 0
      for (i=0; i<1024; i=i+1)
         code[i] = 'b0;
```

```
// Contenido (programa)

code['h0] = {`ADDI,   `X14,  `X0,   4'd0,   12'h000};  // I
code['h1] = {`ADD,    `X13,  `X15,  `X0,    12'd0};    // R
code['h2] = {`ADD,    `X14,  `X13,  `X14,   12'd0};    // R
code['h3] = {`ADDI,   `X13,  `X13,  4'd0,   12'hfff};  // I
code['h4] = {`B,      `NE,   `X13,  `X0,    12'h008};  // S
code['h5] = {`STOP,   24'd0};
   end
endmodule
```

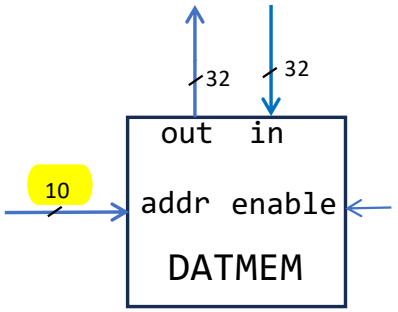
Las direcciones reales del procesador son siempre múltiplos de 4, aquí aparecen 0,1,2,3… porque los dos LSB se fijan a 0

En el archivo globals.vh definimos macros para escribir el código máquina, cada uno de ellos es sustituido por su equivalente con ceros y unos

RISC Y: descripción Verilog (DATMEM)

```
`include "globals.vh"

module data_mem (
   input  wire clk,
   input  wire enable,
   input  wire WMEM,
   input  wire [11:0] addr,
   input  wire [31:0] in,
   output wire [31:0] out
   );

   localparam max_waddr = (`DATA_END+1)/4 - 1;
 //número de palabras-1 a partir de las direcciones de byte, por tanto, dir máxima de palabra

   reg [31:0] mem[0:max_waddr];
// RAM 1024x32 4kiB    Hay 1024 palabras (con 12 bits de addr)
   wire [9:0] waddr = addr[11:2];
// word address   las direcciones de palabra solo tienen 10 bits
```

```
  // RAM read/write
   always @(posedge clk)
      if (enable && WMEM)
            mem[waddr] <= in;

   // Asynchronous read
   assign out = enable ? mem[waddr] : 'b0;

endmodule
```

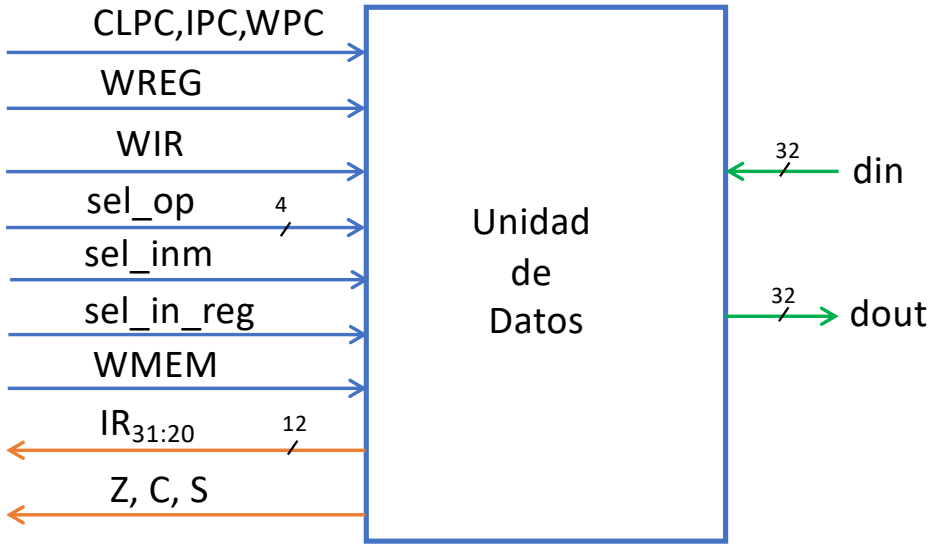


Reducimos el tamaño de la memoria para facilitar su implementación.

Las direcciones del procesador son siempre múltiplos de 4.

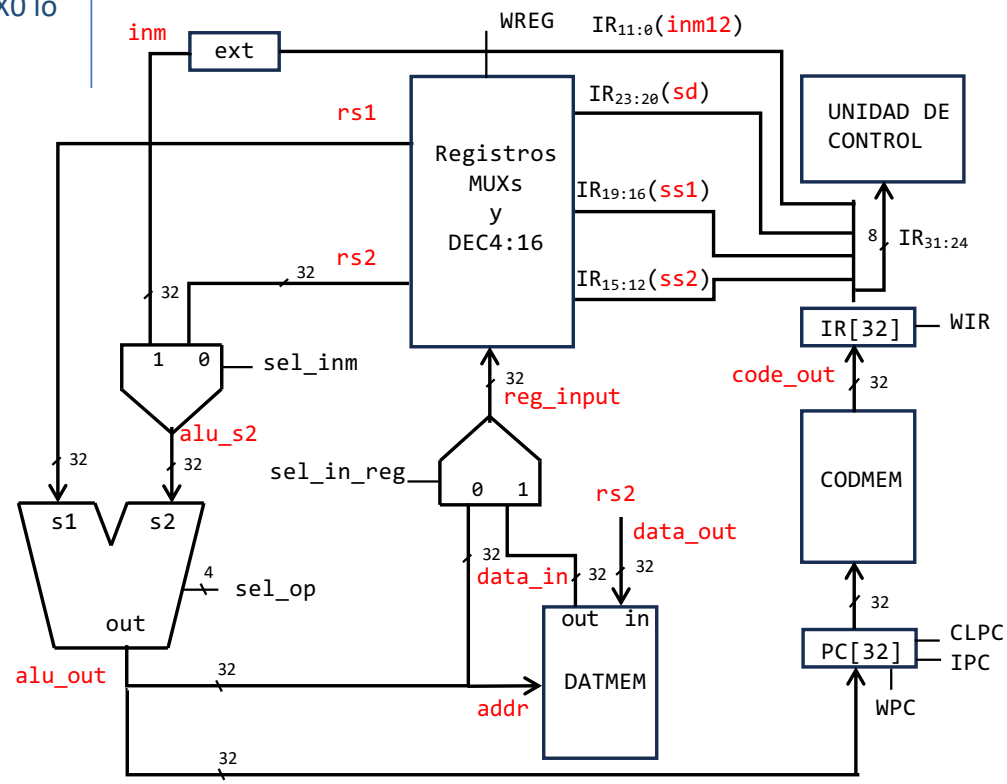
RISC Y: descripción Verilog (unidad de datos)

```
`include "globals.vh"

module data_unit(
   input  wire clk,
   input  wire CLPC,
   input  wire IPC,
   input  wire WPC,
   input  wire WREG,
   input  wire WIR,
   input  wire sel_in_reg,
   input  wire sel_imm,
   input  wire [3:0] sel_op,
   input  wire enable,        //habilitación DE MEMDAT
   input  wire WMEM,
   input  wire [31:0] din,   // conexión con placa

   output wire [ 7:0] codop,   // para los bits IR31:24
   output wire [ 3:0] func,    // para los bits IR23:20
   output wire [31:0] dout,     //conexión con placa
   output wire Z,C,S
   );
```



RISC Y: descripción Verilog (unidad de datos)

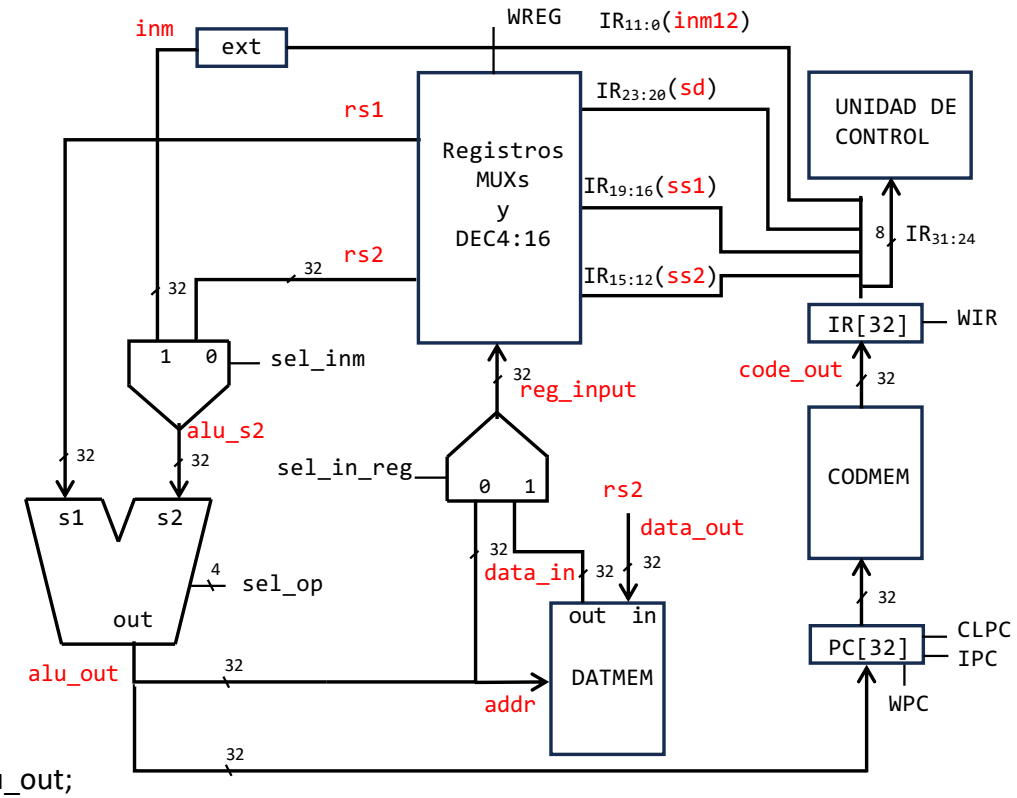
```
   reg [11:0] pc;              // contador de programa
   reg  [31:0] ir;             // registro IR
   reg  [31:0] regs [1:15];    // banco de registros: por qué del 1 al 15? -> porque el X0 lo
fijamos a 0

   wire [31:0] code_out;        // salida de codmem
   wire [ 3:0] ss1, ss2;        // selección de los registros fuente
   wire [ 3:0] sd;              // selección del registro destino
   wire [31:0] reg_input        //entrada al banco de registros
   wire [11:0] imm12;
   wire [31:0] imm;             // imm12 extendido en signo
   wire [31:0] rs1, rs2;        // registros fuente
   wire [31:0] alu_s2;          // entrada s2 de la ALU (la única que cambia según la
instrucción)
   wire [31:0] alu_out;         // ALU output

// memory interface
   wire [11:0] addr;            // memory address
   wire  [31:0] data_in;        // data desde memdat hacia los registros
   wire [31:0] data_out;        // data desde rs2 hacia memdat
```



RISC Y: descripción Verilog (unidad de datos)

```
//contador de programa PC
 always @(posedge clk)
      if(CLPC)
         pc <= 'b0;
      else if (IPC)
         pc <= pc + 4;
      else if (WPC)
         pc <= alu_out;

// memoria de código CODMEM
code_mem code_mem (
      .addr(pc[11:2]),
      .data(code_out)
   );

// registro de instrucciones IR
   always @(posedge clk)
      if (WIR)
         ir <= code_out;
```

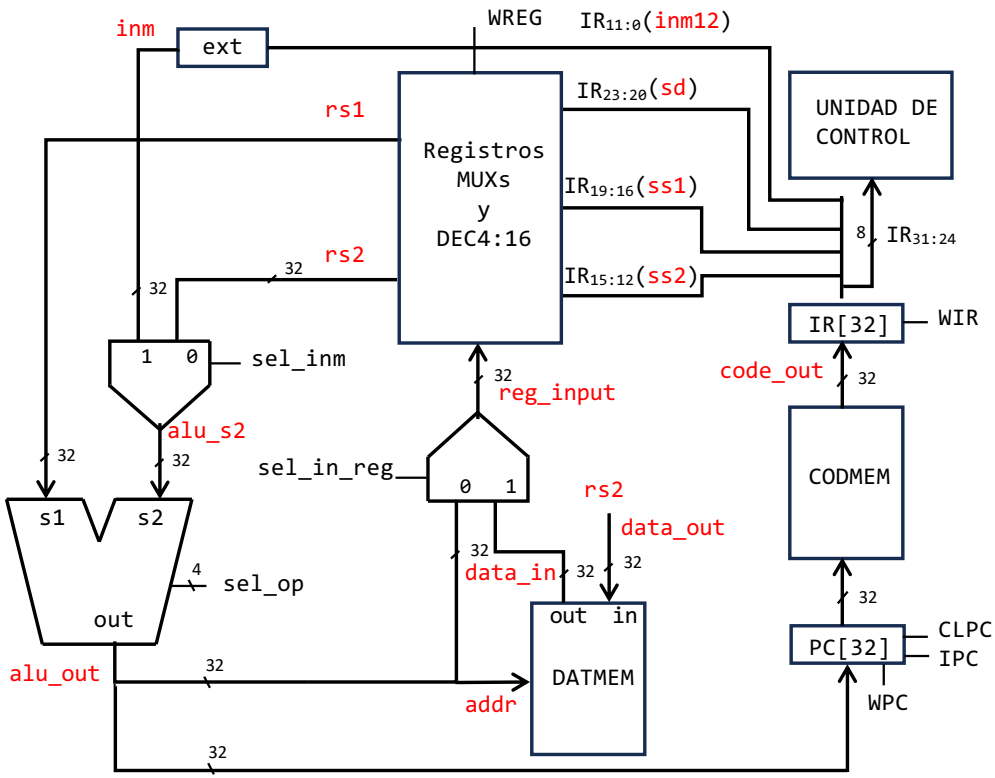
```
assign  codop = ir[31:24];
assign  sd = ir[23:20];
assign  ss1 = ir[19:16];
assign  ss2 = ir[15:12];
assign  imm12 = ir[11:0];
assign  imm20 = ir[19:0];
assign  func = ir[23:20];

assign imm = {{20{imm12[11]}}, imm12};

assign addr = alu_out[11:0];
assign data_out = rs2;

assign rs1 = (ss1 == 4'd0) ? 32'd0 : regs[ss1];
// pone a 0 el registro x0, no deja escribirlo
assign rs2 = (ss2 == 4'd0) ? 32'd0 : regs[ss2];
// pone a 0 el registro x0, no deja escribirlo

//MUX de la ALU
assign alu_s2 = sel_imm ? imm : rs2;

assign reg_input =  sel_in_reg  ? data_in : alu_out;
```



RISC Y: descripción Verilog (unidad de datos)

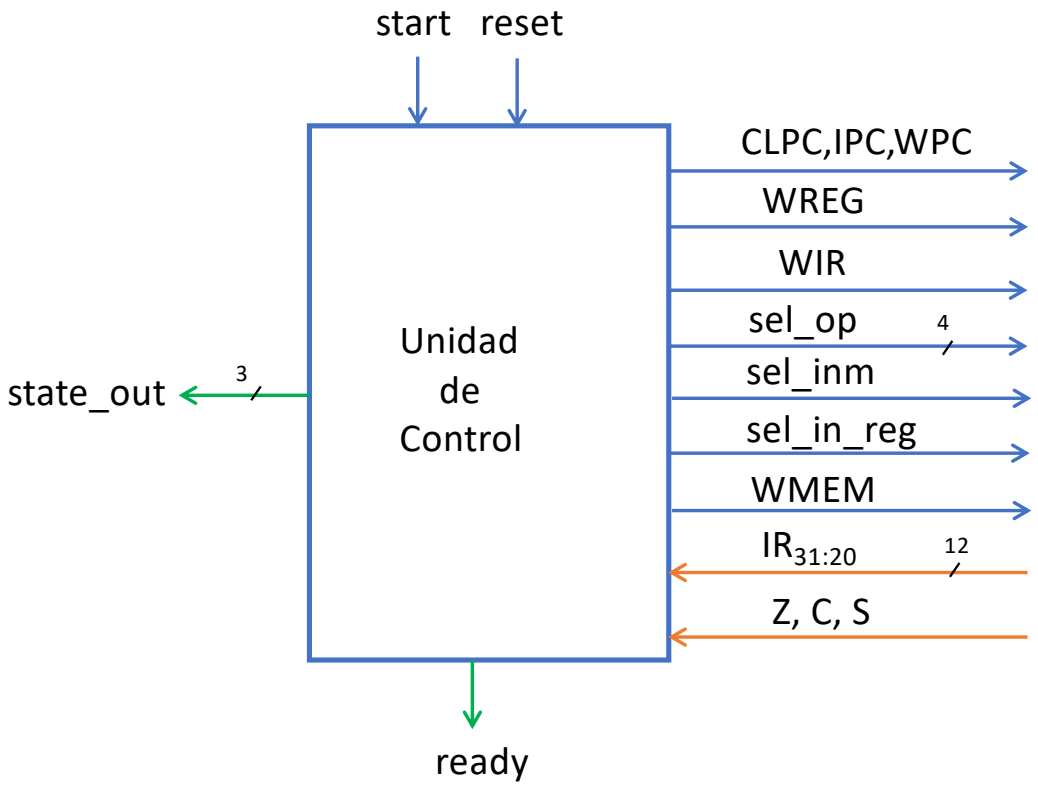
```
// conexiones al exterior
always @(posedge clk)
      if(WREG)
         regs[sd] <= reg_input;
      else
         regs[15] <= din;

assign dout = regs[14];

// instancia de la alu
 alu #(.BW(32)) alu (
      .s1(rs1),
      .s2(alu_s2),
      .sel_op(sel_op),
      .out(alu_out),
      .C(C),.Z(Z),.S(S)
   );
```

```
//instancia de DATMEM
data_mem data_mem(
      .clk(clk),
      .enable(enable),
      .WMEM(WMEM),
      .addr(addr),
      .in(data_out),
      .out(data_in)
   );

endmodule
```



RISC Y: descripción Verilog (unidad de control)

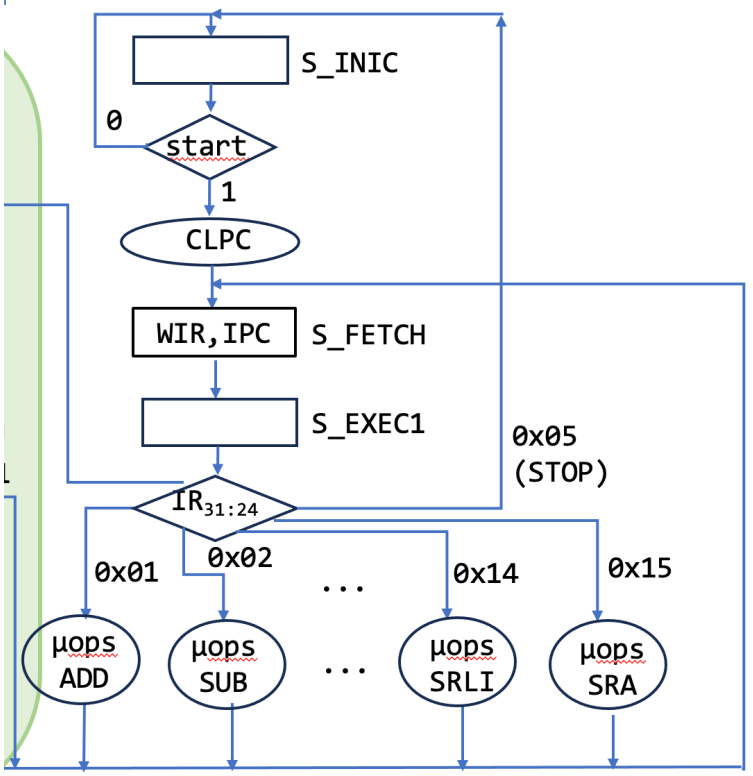
```
`include "globals.vh"
module control_unit (
   // External signals
   input  wire clk,              // clock (flanco de subida)
   input  wire reset,             // reset (sincrono activo en
alto)
   input  wire start,            // start operation
   input  wire C,Z,S,
   output reg  ready,            // ready output indicator
   output wire [2:0] state_out,    // conexión a placa

   // Data unit signals
   input  wire [7:0] codop,      // para los bits IR31:24
   input wire [ 3:0] func,       // para los bits IR23:20
   output reg  [3:0] sel_op,
   output reg  CLPC,
   output reg  IPC,
   output reg  WPC,
   output reg  WIR,
   output reg  WREG,
   output reg  sel_in_reg,
   output reg  sel_imm,
   output reg  enable,
   output reg  WMEM );
```



RISC Y: descripción Verilog (unidad de control)

```
localparam [2:0] INIC = 0,     //estados de la carta
                 FETCH = 1,
                 EXEC1 = 2,
                 EXEC2 = 3;

reg [2:0] state, next_state;

assign state_out = state;  //conexión a placa

always @(posedge clk)
      if (reset == 1'b1)
         state <= INIC;
      else
         state <= next_state;

always @* begin
next_state = 'bx;          //valores por defecto
ready = 1'b0; sel_op = 'b0;
CLPC = 1'b0; IPC = 1'b0; WPC = 1'b0;
WIR = 1'b0; WREG = 1'b0; sel_in_reg = `ALU_OUT;
sel_imm = 1'b0; enable = 1'b0; WMEM = 1'b0;
```

```
case (state)
       INIC : begin
        ready = 1'b1;
        if (start) begin
           CLPC = 1'b1;
           next_state = FETCH;
        end else begin
           next_state = INIC;
        end
      end
      FETCH: begin
        WIR = 1'b1;
        IPC = 1'b1;
        next_state = EXEC1;
      end
      EXEC1: begin
        next_state = FETCH;      // por defecto
      case(codop)
        `ADD: begin
           sel_op = `ALU_ADD;
           WREG = 1'b1;
        end
```



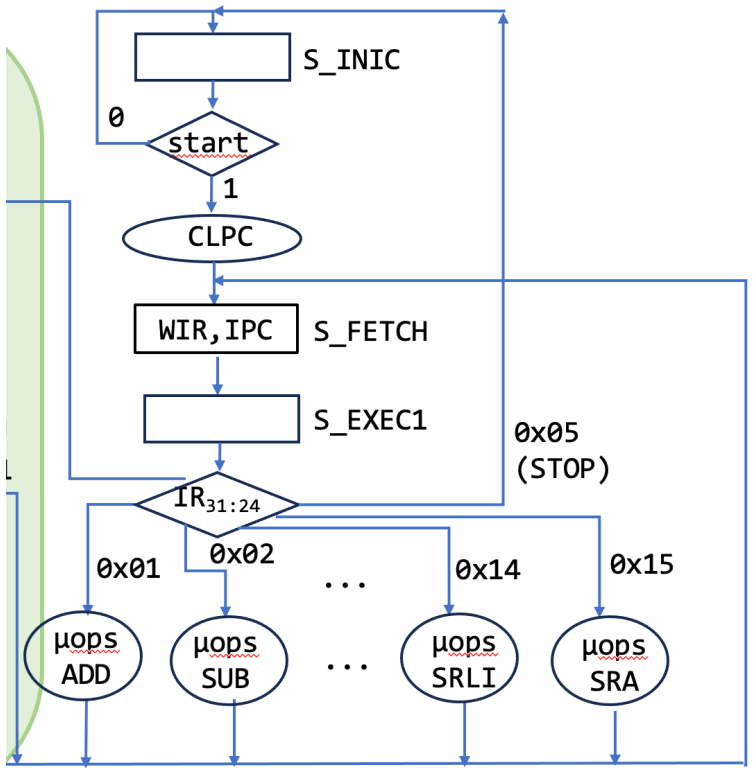
usamos las macros definidas en globals.vh

RISC Y: descripción Verilog (unidad de control)

```
`SUB: begin
   sel_op = `ALU_SUB;
   WREG = 1'b1;
end
`ADDI: begin
   sel_imm = 1'b1;
   sel_op = `ALU_ADD;
   WREG = 1'b1;
end
`SW: begin
   sel_imm = 1'b1;
   sel_op = `ALU_ADD;
   enable = 1'b1;
   WMEM = 1'b1;
end
`LW: begin
   sel_imm = 1'b1;
   sel_op = `ALU_ADD;
   enable = 1'b1;
   sel_in_reg = 1'b1;
   WREG = 1'b1;
end
```

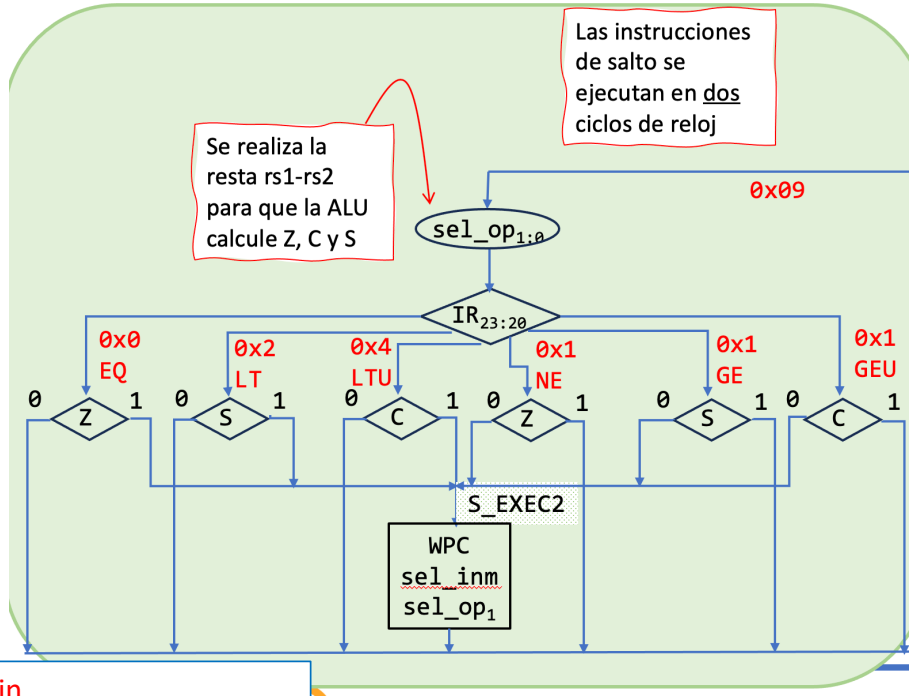
```
`OR: begin
   sel_op = `ALU_OR;
   WREG = 1'b1;
end
`AND: begin
   sel_op = `ALU_AND;
   WREG = 1'b1;
end
`XORI: begin
   sel_imm = 1'b1;
   sel_op = `ALU_XOR;
   WREG = 1'b1;
end
`ORI: begin
   sel_imm = 1'b1;
   sel_op = `ALU_OR;
   WREG = 1'b1;
end
`ANDI: begin
   sel_imm = 1'b1;
   sel_op = `ALU_AND;
   WREG = 1'b1;
end
```

```
`SLL: begin
   sel_op = `ALU_SLL;
   WREG = 1'b1;
end
`SRL: begin
   sel_op = `ALU_SRL;
   WREG = 1'b1;
end
`SRA: begin
   sel_op = `ALU_SRA;
   WREG = 1'b1;
end
`SLLI: begin
   sel_imm = 1'b1;
   sel_op = `ALU_SLL;
   WREG = 1'b1;
end
`SRLI: begin
   sel_imm = 1'b1;
   sel_op = `ALU_SRL;
   WREG = 1'b1;
end
`SRAI: begin
   sel_imm = 1'b1;
   sel_op = `ALU_SRA;
   WREG = 1'b1;
end
```



RISC Y: descripción Verilog (unidad de control)

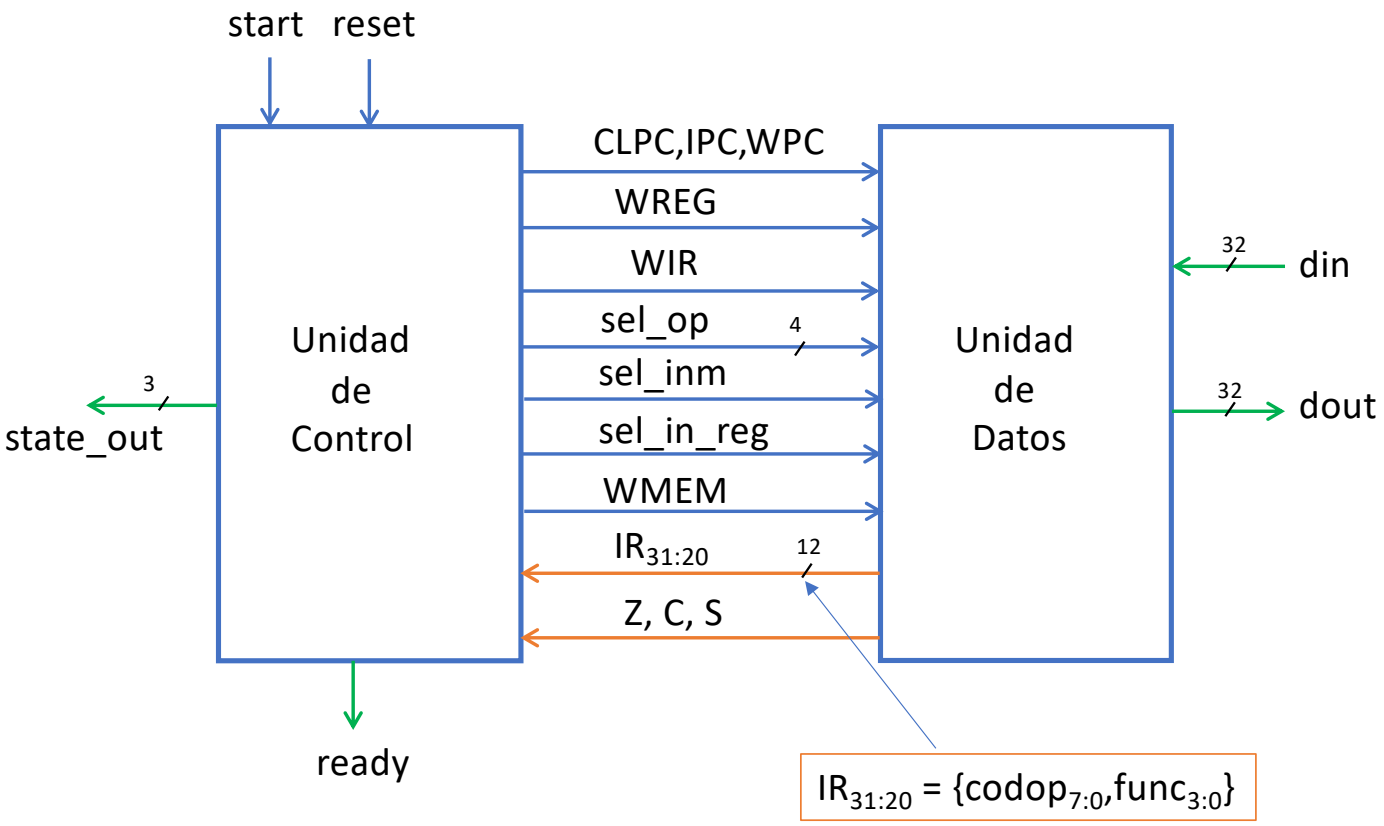
```
`B: begin
    sel_op = `ALU_SUB;
    if (func == `EQ) begin
      if (Z == 1'b1) begin
        next_state = EXEC2;
      end
    end
    if (func == `NE)
      if (Z == 1'b0) begin
        next_state = EXEC2
      end
    end
    if (func == `LT) begin
      if (C == 1'b1) begin
        next_state = EXEC2;
      end
    end
    if (func == `GE)
      if (C == 1'b0) begin
        next_state = EXEC2;
      end
    end
    if (func == `LT) begin
      if (S == 1'b1) begin
        next_state = EXEC2
      end
    end
    if (func == `GE) begin
      if (S == 1'b0) begin
        next_state = EXEC2;
      end
    end
    if (func == `LTU) begin
      if (C == 1'b1) begin
        next_state = EXEC2;
      end
    end
    if (func == `GEU) begin
      if (C == 1'b0) begin
        next_state = EXEC2;
      end
    end
    default:  next_state = INIC;
    endcase
end
```




```
EXEC2: begin
        next_state = FETCH;
        sel_imm = 1'b1;
        sel_op = `ALU_TR2;
        WPC = 1'b1;
      end
    endcase
  end
endmodule
```

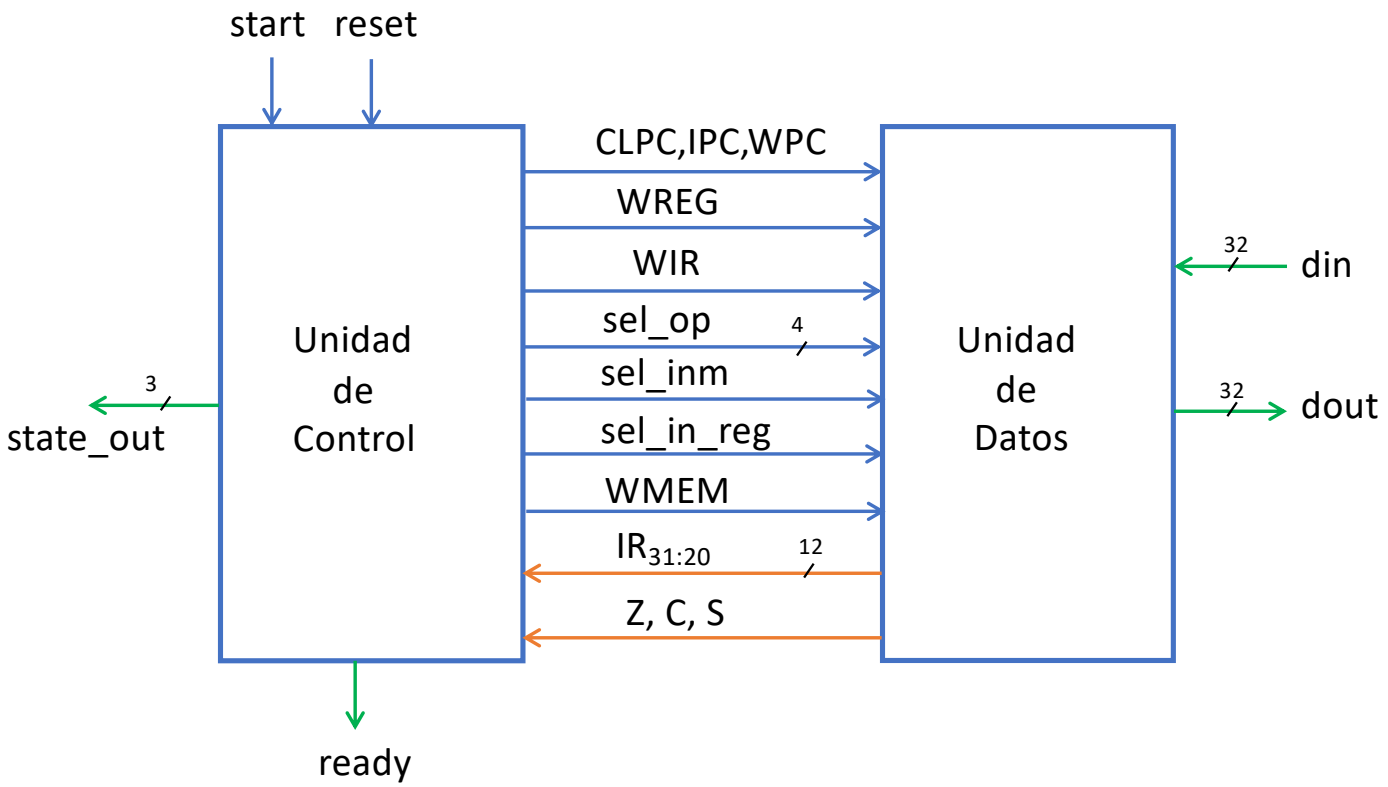
RISC Y: sistema digital

```
module riscy (
   input  wire clk,
   input  wire reset,
   input  wire start,
   output wire ready,

   input  wire [31:0] din,
   output wire [31:0] dout,
   output wire [ 2:0] state_out
   );

   // señales de conexión de las instancias UD-UC
   wire [7:0] codop;
   wire [3:0] func;
   wire [3:0] sel_op;
   wire sel_in_reg;
   wire CLPC, IPC, WPC, WIR, WREG, sel_imm,
   enable, WMEM, C,Z,S;
```



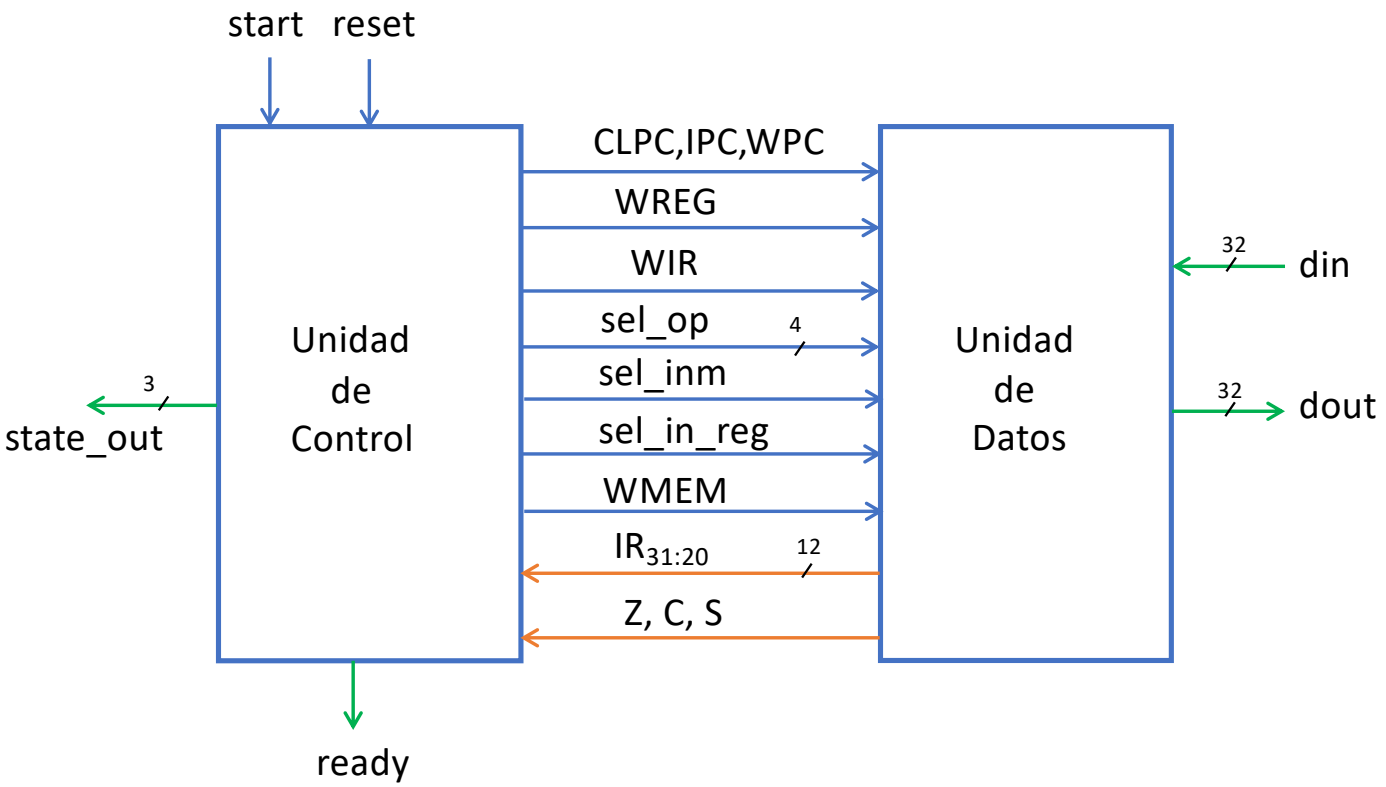
RISC Y: sistema digital

```
// instancia de la unidad de control
  control_unit control_unit (
     .clk(clk),
     .reset(reset),
     .start(start),
     .ready(ready),
     .codop(codop),
     .func(func),
     .sel_op(sel_op),
     .CLPC(CLPC),
     .IPC(IPC),
     .WPC(WPC),
     .WIR(WIR),
     .WREG(WREG),
     .sel_in_reg(sel_in_reg),
     .sel_imm(sel_imm),
     .C(C),
     .S(S),
     .Z(Z),
     .state_out(state_out),
     .enable(enable),
     .WMEM(WMEM)
  );
```



RISC Y: sistema digital

```
// instancia de la unidad de datos
  data_unit data_unit (
      .clk(clk),
      .sel_op(sel_op),
      .CLPC(CLPC),
      .IPC(IPC),
      .WPC(WPC),
      .WIR(WIR),
      .WREG(WREG),
      .sel_in_reg(sel_in_reg),
      .sel_imm(sel_imm),
      .codop(codop),
      .func(func),
      .C(C),
      .S(S),
      .Z(Z),
      .din(din),
      .dout(dout),
      .enable(enable),
      .WMEM(WMEM)
   );
endmodule
```



Arquitectura RISC Y