¿Qué intención manifiesta un programa?

Podemos interpretar al programa como:
La **composición de instrucciones** que indican cómo ha de modificarse el **estado del procesador** en base a la semántica de ejecución asociada a éstas.
• Podemos definir al **estado del programa** como el conjunto de valores asociados tanto a los registros (propósito general, PC, registros internos) como a las distintas posiciones de memoria
• La **semántica asociada a las instrucciones** puede indicar cómo se transforman los valores de los registros asociados a datos del programa (registros de propósito general y memoria) o al control de flujo del mismo (PC, SP)

Perspectiva declarativa de un programa

```
main:   MOV R1,R3
        ADD R1,R2
        JMP main
et1:    DW 0x07
et2:    DW 0x04
```

```
main:   MOV R1,R3←
        ADD R1,R2←
        JMP main
et1:    DW 0x07
et2:    DW 0x04
```

Vemos instrucciones que modifican los datos del programa (**registros y memoria**←),

```
main:   MOV R1,R3←
        ADD R1,R2←
        JMP main
et1:    DW 0x07⇐
et2:    DW 0x04⇐
```

inicializan los mismos (**memoria**⇐),

```
main:   MOV R1,R3←
        ADD R1,R2←
        JMP main↩
et1:    DW 0x07⇐
et2:    DW 0x04⇐
```

o modifican el flujo de ejecución (**PC**←-).

```
main:   MOV R1,R3
        ADD R1,R2
        JMP main
et1:    DW 0x07
et2:    DW 0x04
```

La estructura del programa induce un conjunto posible de ejecuciones en base a su estado inicial (**PC**, **registros**, **memoria**) y las modificaciones introducidas por la lógica de control de flujo (**saltos**).

```
main:  MOV R1,R3↩
       ADD R1,R2
       JMP main
et1:   DW 0x07
et2:   DW 0x04
```

```
main:  MOV R1,R3
       ADD R1,R2↩
       JMP main
et1:   DW 0x07
et2:   DW 0x04
```

```
main:  MOV R1,R3↩
       ADD R1,R2
       JMP main
et1:   DW 0x07
et2:   DW 0x04
```

```
main:  MOV R1,R3
       ADD R1,R2↩
       JMP main
et1:   DW 0x07
et2:   DW 0x04
```

Perspectiva estructural de un programa

```
main:  MOV R1,R3
       ADD R1,R2
       JMP main
et1:   DW 0x07
et2:   DW 0x04
```

Las instrucciones se codifican y almacenan en memoria y la modificación del flujo de ejecución necesita conocer la propia estructura del programa (**emplazamiento en memoria**).

```
0x00  main:  MOV R1,R3
0x02         ADD R1,R2
0x04         JMP main(0x00)
0x06   et1:  DW 0x07
0x07   et2:  DW 0x04
```

El programa codificado y almacenado en memoria principal:

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

Recordemos que por seguir una arquitectura de **Von Neumann** tenemos en un mismo espacio de memoria tanto instrucciones (rango **0x00 - 0x05**) como datos (rango **0x06 - 0x07**).

Program Counter

¿Cómo reproducimos la idea de **ejecución secuencial u ordenada del programa**?
Con un registro de propósito particular (**PC**) que indica **de qué dirección de memoria toma la próxima instrucción (fetch).**

Perspectiva estructural de un programa

Dir.	Valor			
0x00	**01000**	001	011	xxxxx
0x02	**00001**	001	010	xxxxx
0x04	**10100**	xxx	00000000	
0x06	00000111			
0x07	00000100			

Veamos cómo la ejecución del programa actualiza el PC, **primero con la notación mnemónica de las instrucciones.**

Dir.	Valor			
0x00	**01000**	001	011	xxxxx
0x02	**00001**	001	010	xxxxx
0x04	**10100**	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x00
MOV R1,R3

Dir.	Valor			
0x00	**01000**	001	011	xxxxx
0x02	**00001**	001	010	xxxxx
0x04	**10100**	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x02
ADD R1,R2

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x04
JMP main

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x00
MOV R1,R3

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x02
ADD R1,R2

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x04
JMP main

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

Veamos cómo la ejecución del programa actualiza el PC, según la **semántica de las instrucciones.**

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x00
R1 ← R3

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x02
R1 ← R1 + R2

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x04
PC ← 0x00

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x00
R1 ← R3

Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x02
R1 ← R1 + R2

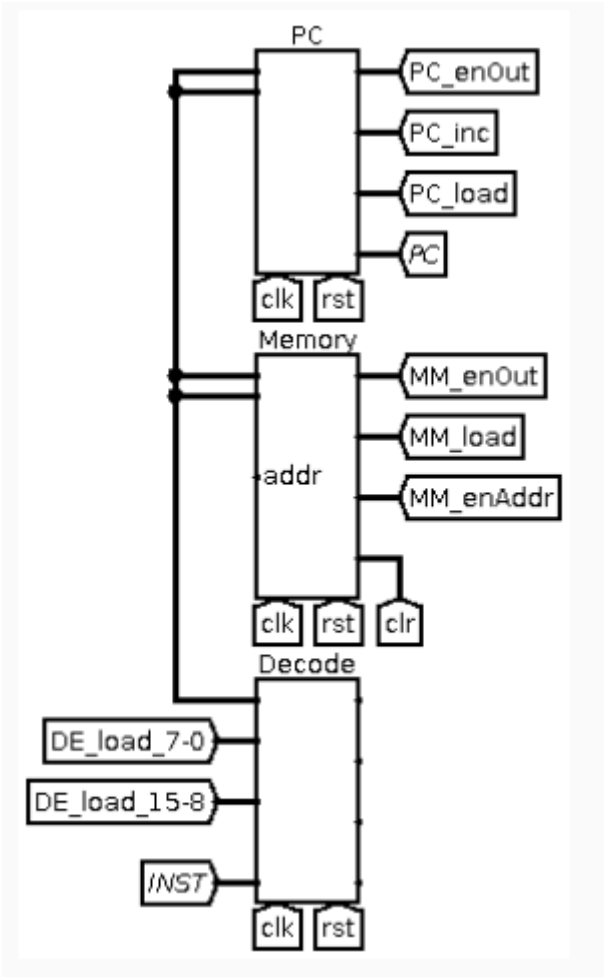
Dir.	Valor			
0x00	01000	001	011	xxxxx
0x02	00001	001	010	xxxxx
0x04	10100	xxx	00000000	
0x06	00000111			
0x07	00000100			

↩ PC: 0x04
PC ← 0x00

OrgaSmall

- Arquitectura von Neumann, memoria de datos e instrucciones compartida
- 8 registros de propósito general, R0 a R7
- 1 registro de propósito específico PC
- Tamaño de palabra de 8 bits e instrucciones de 16 bits
- Memoria de 256 palabras de 8 bits
- Bus de 8 bits
- **Diseño microprogramado**

Datapath del Fetch-Decode



Ahora sí podemos analizar por partes el flujo de datos (datapath) que permite reproducir el mecanismo deseado de **FETCH DECODE EXECUTE.**
En el ciclo de Fetch-Decode participan el **PC**, la **Memoria** y la unidad de **Decode**. El **PC** no sólo encapsula un registro sino que expone también una señal de control que permite incrementarlo (en dos palabras).
Veamos qué secuencia de microinstrucciones (RTL) debería suceder para conseguir que la instrucción almacenada en memoria (de 2 palabras de 8 bits) se almacene en los registros internos del módulo de decodificación.

$$
\begin{array}{lcl}
MM_{Addr} & := & PC \\
DE_{H} & := & MM_{Data} \\
PC_{inc} & & \\
MM_{Addr} & := & PC \\
DE_{L} & := & MM_{Data} \\
PC_{inc} & &
\end{array}
$$

Aquí las asignaciones (:=) indican la activación de un par de señales **write/enableOut** y las declaraciones aisladas (PCinc) indican la activación durante un ciclo de la señal indicada.
Pero al escribir el microprograma de su implementación se indicarán las señales de la siguiente manera:

```
PC_enOut  MM_enAddr
MM_enOut  DE_loadH  PC_inc
PC_enOut  MM_enAddr
MM_enOut  DE_loadL  PC_inc
```

Donde cada línea se corresponde con un ciclo de reloj y los nombres declarados en cada una, con las señales de control que se activan **a la vez** (son recursos no compartidos) en cada uno de estos ciclos.
En RTL:

$$
\begin{array}{lll}
MM_{Addr} & := & PC \\
DE_H & := & MM_{Data} \\
PC_{inc} & & \\
MM_{Addr} & := & PC \\
DE_L & := & MM_{Data} \\
PC_{inc} & &
\end{array}
$$

Como señales:

```
PC_enOut  MM_enAddr
MM_enOut  DE_loadH  PC_inc
PC_enOut  MM_enAddr
MM_enOut  DE_loadL  PC_inc
```

Observemos que luego de haber cargado las dos palabras de la instrucción la estructura del componente decode descompone a la misma en partes que hacen que funcionen como señales de control.
En RTL:

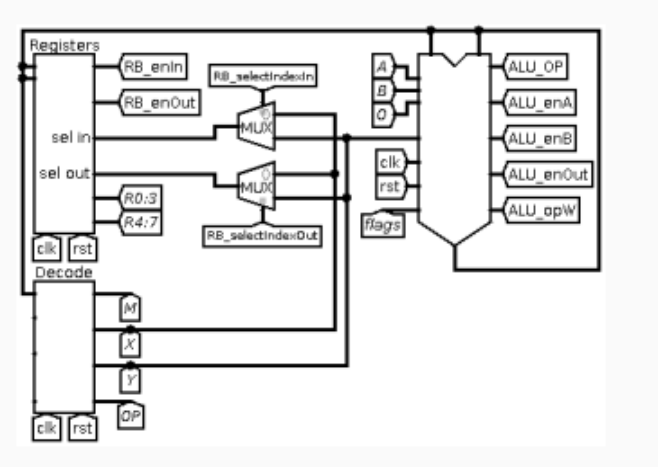
$$
\begin{array}{lll}
MM_{Addr} & := & PC \\
DE_H & := & MM_{Data} \\
PC_{inc} & & \\
MM_{Addr} & := & PC \\
DE_L & := & MM_{Data} \\
PC_{inc} & &
\end{array}
$$

Como señales:

```
PC_enOut  MM_enAddr
MM_enOut  DE_loadH  PC_inc
PC_enOut  MM_enAddr
MM_enOut  DE_loadL  PC_inc
```

• ¿El **PC** se actualiza en cada ciclo de reloj?
• ¿Cómo sabemos **qué microinstrucción** se ejecuta en cada ciclo de reloj?
• ¿Dónde se almacenan las microinstrucciones?

Datapath del ADD



En la ejecución (**EXECUTE**) del **ADD** participan la **ALU**, que resuelve la aritmética, los **Registros** y el **Decode** que indica qué registros participan.
Veamos qué secuencia de microinstrucciones (RTL) debería suceder para conseguir que se transfieran los valores de los registros indicados en la instrucción a la ALU, se realice la operación y se copie el resultado en el registro de destino.

$$\begin{aligned} ALU_A &:= R_A \\ ALU_B &:= R_B \\ ALU_{add} & \\ R_A &:= ALU_{out} \end{aligned}$$

Aquí las asignaciones a RA, RB indican no sólo la activación de un par de señales **write/enableOut** sino que indican el índice del registro de interés a partir de los bits correspondientes al operando de la instrucción (**Decode** y mux correspondiente).
El microprograma asociado es el siguiente:

```
RB_enOut   ALU_enA   RB_selectIndexOut=0
RB_enOut   ALU_enB   RB_selectIndexOut=1
ALU_OP=ADD ALU_opW
RB_enIn    ALU_enOut RB_selectIndexIn=0
```

Datapath del STR

En la ejecución (**EXECUTE**) del **STR** participan el controlador de **memoria**, los **Registros** y el **Decode** que indica el registro fuente y la dirección destino.
Veamos qué secuencia de microinstrucciones (RTL) debería suceder para conseguir que se transfieran los valores de dirección de memoria al controlador, y el valor del registro fuente a la dirección indicada.

$$\begin{aligned} M_{addr} &:= DE_{imm} \\ M_{data} &:= RA \end{aligned}$$

Notemos que son simplemente dos asignaciones pero que el controlador tiene entradas de **dirección y de datos.**
El microprograma asociado es el siguiente:

```
DE_enOutImm  MM_enAddr
RB_enOut  MM_load  RB_selectIndexOut=0
```

Observemos que la asignación desde el **Decode** al controlador de memoria se resguarda con un tri-estado por realizarse a través del recurso compartido (bus)

Saltos condicionales, ¿cómo implementarlos?

Necesitamos implementar los saltos condicionales, veamos cómo sería su expresión en RTL:

$$\begin{aligned} IF \quad & Z = 1 \\ & PC := DE_{imm} \end{aligned}$$

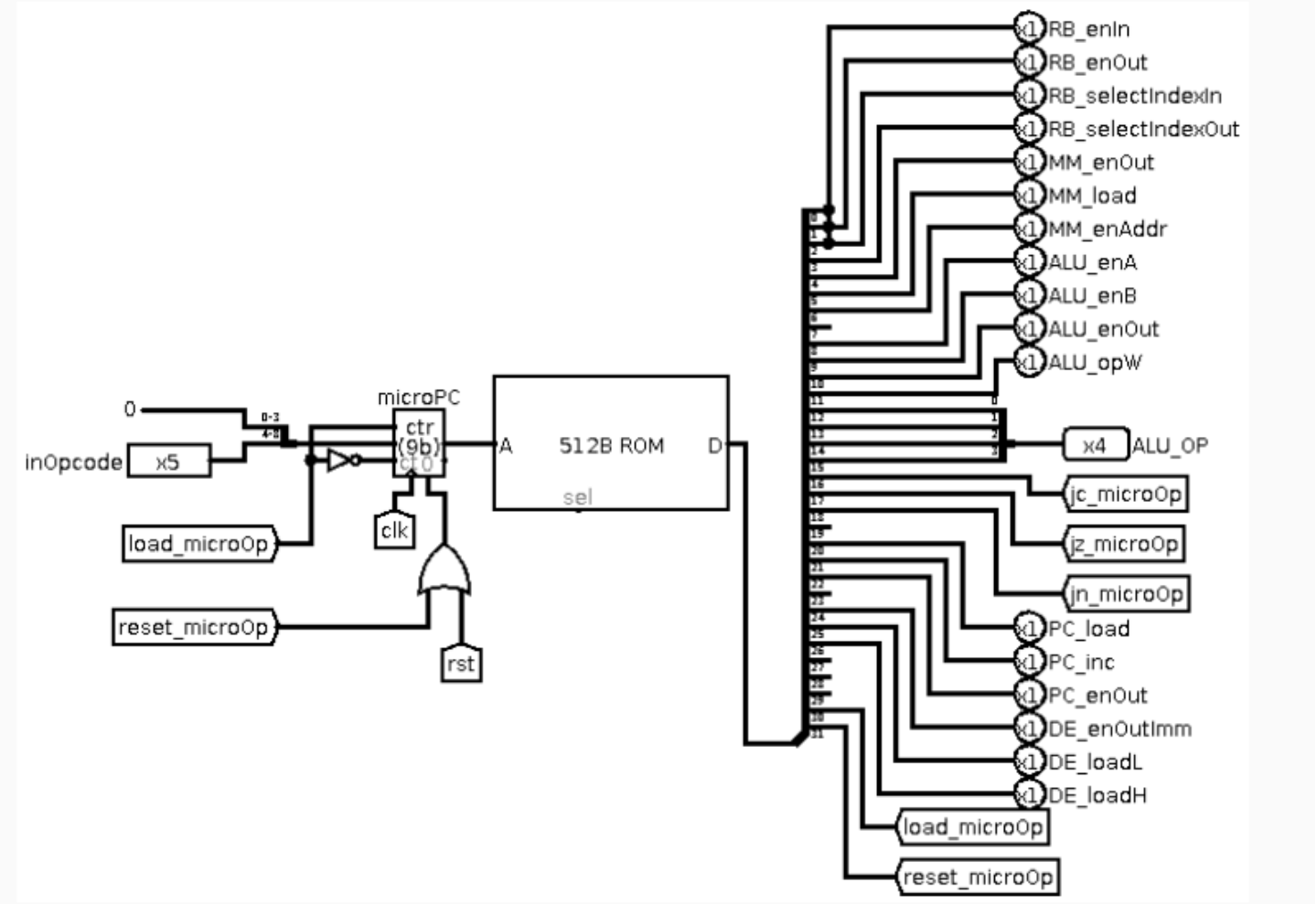
¿Cómo conseguimos que la asignación se haga sólo si se cumple la condición (Z = 0)?
Tenemos que definir primero cómo implementar el mecanismo que ejecuta microinstrucciones.

Micro PC

Así como existe un registro de propósito específico que indica de qué dirección de memoria tomar la próxima instrucción (**PC**), existe otro registro interno que indica cuál es la microinstrucción que va a ser ejecutada en el siguiente ciclo de reloj, el **Micro PC.**
¿A qué dirección de memoria hace referencia?

Unidad de control

Los microprogramas que permiten ejecutar las acciones asociadas con cada instrucción de nuestro lenguaje (**ASM**) se ejecutan dentro de un componente llamado **unidad de control** que cuenta con una memoria interna donde almacena la codificación de los microprogramas.
Esta memoria está compuesta por palabras que en nuestro caso son de **32 bits**, se acceden a través de **direcciones de 9 bits y cada bit dentro de una palabra determina el valor de una señal de control dentro de nuestra organización.**
Por eso, sus microprogramas escritos como conjunción de señales se traducen en una serie de palabras de 32 bits.
Vamos presentar el diagrama de la unidad de control, donde podemos observar:
• Que el **micro PC** es un contador, ya que en cada ciclo de reloj su comportamiento por defecto es incrementar en uno la posición de memoria a ser leída.
• Las salidas de control que pueden modificar el comportamiento del **micro PC** (**load_microOp**, **reset_microOp**).
• Que las salidas están cableadas a los bits de cada palabra en la memoria interna.
• Que la entrada de **inOpCode** llega del Decode y se extiende con ceros en su parte baja y sobreescribe el valor del **micro PC** si se habilita la señal **load_microOp**.



Saltos condicionales y unidad de control

Los contenidos de la memoria de la unidad de control son **las microinstrucciones en las direcciones indicadas** por las etiquetas y accedidas a través de la dirección **A** según se encuentran en el archivo **microCode.ops**.

⇒

Los contenidos se compilan de su declaración mnemónica (como listas de señales) a las palabras de 32 bits de acuerdo a las señales que deben activarse a la salida **D**, según se encuentran en el archivo **microCode.mem**.


```
00000:
PC_enOut MM_enAddr
MM_enOut      DE_loadH
PC_inc
PC_enOut MM_enAddr
MM_enOut      DE_loadL
PC_inc
load_microOp
reset_microOp
...
```

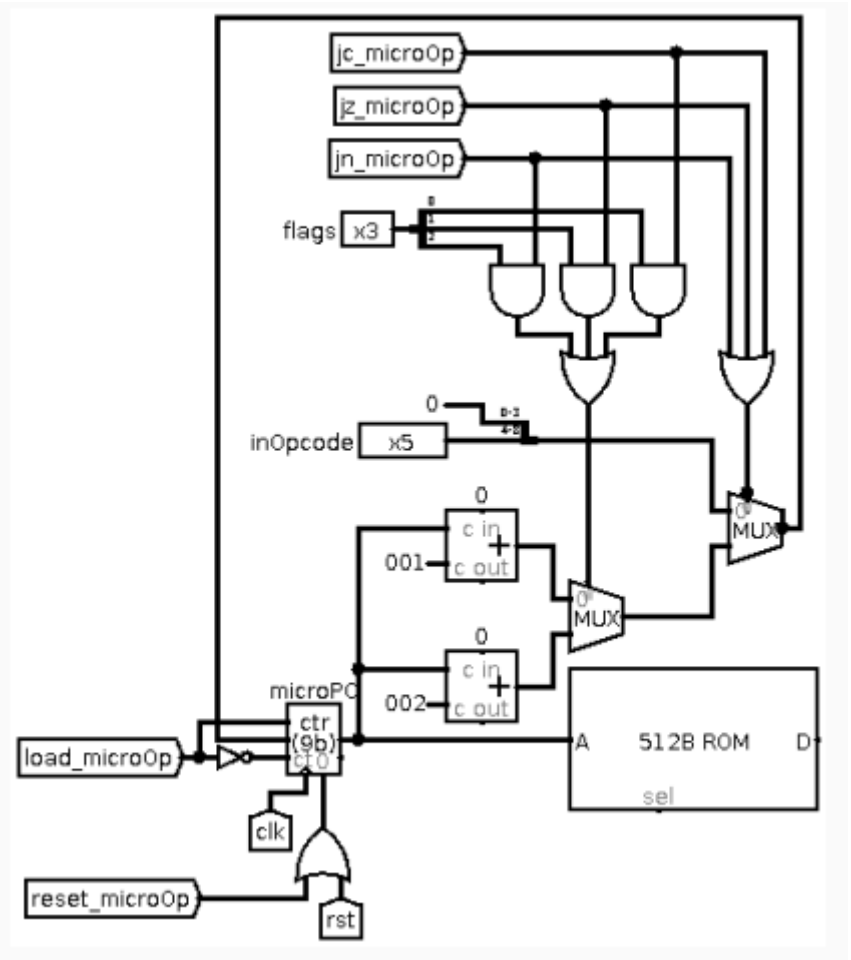
⇒

```
00400040
04200010
00400040
02200010
40000000
80000000
...
```

Recapitulando, queríamos implementar JZ:

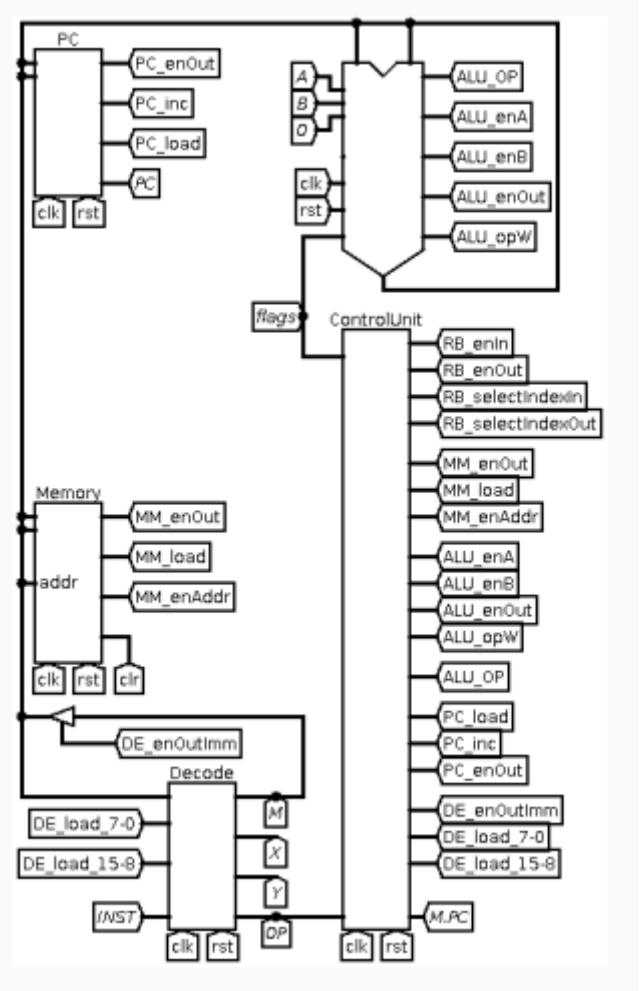
$$\begin{array}{lll} IF & Z = 1 & \\ & PC & := & DE_{imm} \end{array}$$

¿Cómo conseguimos que la asignación se haga sólo si se cumple la condición (Z = 0)? Vamos a sobreescribir el valor del **micro PC** sólo si se encuentra habilitada la señal correspondiente de la **ALU**. Veamos la propuesta.



Observemos lo siguiente:
El **micro PC** se puede sobrescribir con la señal de **load_microOp** en conjunto con dos selectores de multiplexores: **el de la derecha indicando si se sobreescribe por una nueva instrucción o flags; y el de la izquierda indicando si se incrementa el micro PC en 1 (flag habilitado) o 2 (flag en cuestión deshabilitado).**

Datapath del JZ



En la ejecución (**EXECUTE**) del **JZ** participan el controlador de **memoria**, el **PC** que puede o no ser sobreescrito, el **Decode** que indica valor con el cual podría actualizarse el **PC**, y la **ALU** cuyos flags determinan si el salto se realiza.
Ya podemos notar que la **unidad de control** participa en todos los casos.

$$\begin{array}{lll} IF & Z = 0 & \\ & PC & := & DE_{imm} \end{array}$$

El microprograma asociado deja en evidencia los mecanismos necesarios para implementar el salto:

```
JZ_microOp load_microOp
reset_microOp
DE_enOutImm PC_load
reset_microOp
```

¿Qué función cumple la señal reset microOp aquí?

```
JZ_microOp load_microOp
reset_microOp
DE_enOutImm PC_load
reset_microOp
```

Esto tiene que ver con cómo ubicamos a los microprogramas en la memoria.

Estructura del microprograma

En el comienzo del código de los microprogramas encontrarán esto:

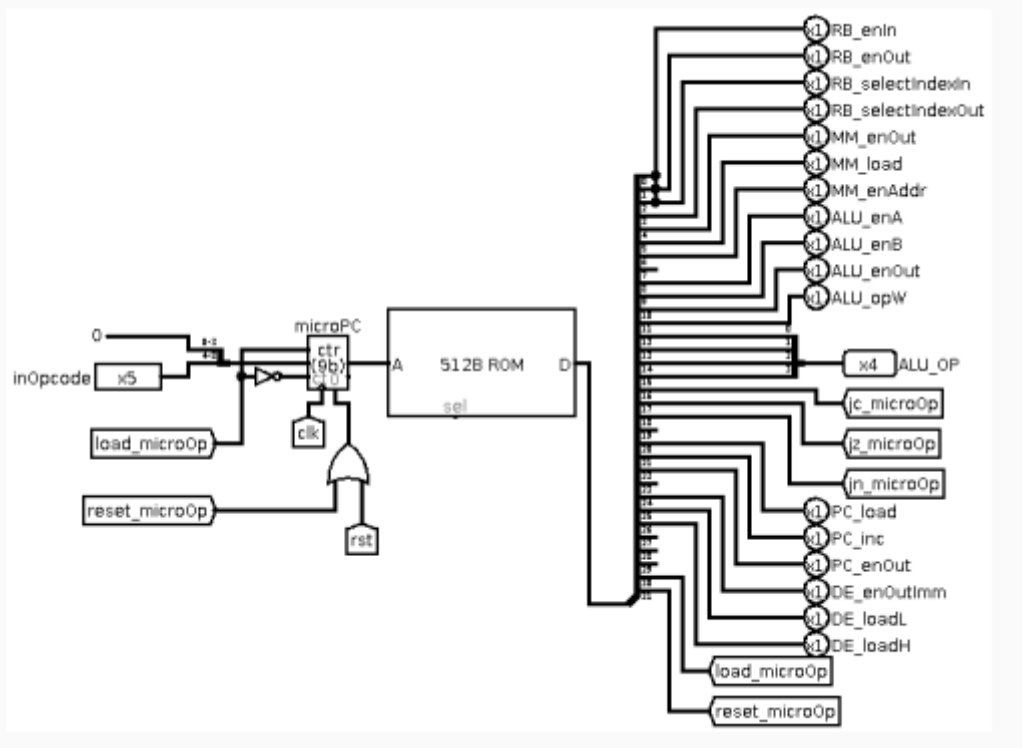
```
00000:
PC_enOut  MM_enAddr
MM_enOut  DE_loadH  PC_inc
PC_enOut  MM_enAddr
MM_enOut  DE_loadL  PC_inc
load_microOp
reset_microOp

00001:  ; ADD
RB_enOut   ALU_enA   RB_selectIndexOut=0
RB_enOut   ALU_enB   RB_selectIndexOut=1
ALU_OP=ADD ALU_opW
RB_enIn    ALU_enOut RB_selectIndexIn=0
reset_microOp
```

Las etiquetas indican el valor de los cinco bits más significativos en los que se ubica cada bloque de microcódigo, completando los bits más bajos con ceros.

Observemos que estos valores se corresponden con los códigos de operación de las instrucciones.

Como cada microprograma termina con un **reset_microOp**, que vuelve el **micro PC** a cero, se cierra el ciclo al regresar al **FETCH** luego del **EXECUTE** de cada instrucción.

Unidad de control



Observemos que: La forma en la que se indican las posiciones de los microprogramas y cómo se compilan es consistente con el funcionamiento de la unidad de control (**load_microOp**, **reset_microOp**, **inOpCode**).

• ¿El PC se actualiza en cada ciclo de reloj?
• ¿Cómo sabemos qué microinstrucción se ejecuta en cada ciclo de reloj?
• ¿Dónde se almacenan las microinstrucciones?

Microarquitectura del CPU

¡Ejercicios!

Ejercicio 1 - Máquina Orga1

1) Diseñar el camino de datos de la arquitectura de la máquina ORGA1, suponiendo que se encuentra resuelta la decodificación y el acceso a memoria de la máquina. No dibujar la unidad de control para simplificar el diagrama. Se cuentan con los siguientes circuitos:
▸ Una ALU con 2 registros de 16 bits (ALU_IN1 y ALU_IN2) que usa de entradas y 5 registros que usa de salida: ALU_OUT de 16 bits y (ALU_Z, ALU_N, ALU_C y ALU_V) de 1 bit. Sus señales de control son:

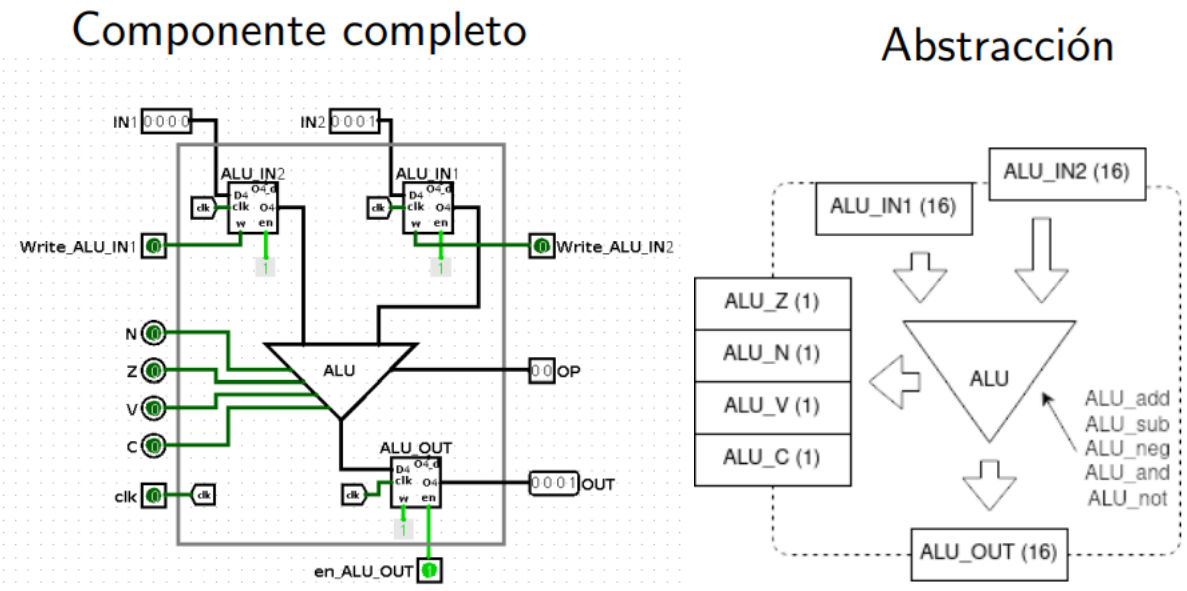
Señal	Efecto
ALU_{add}	ALU_OUT := ALU_IN1 + ALU_IN2
ALU_{sub}	ALU_OUT := ALU_IN1 - ALU_IN2
ALU_{neg}	ALU_OUT := - ALU_IN1
ALU_{and}	ALU_OUT := ALU_IN1 AND ALU_IN2
ALU_{not}	ALU_OUT := NOT ALU_IN1

▸ Un extensor de signo complemento a 2 (SIGN_EXT) con un registro de entrada de 8 bits (EXT_IN) y un registro de salida de 16 bits (EXT_OUT). Sus señales de control son:

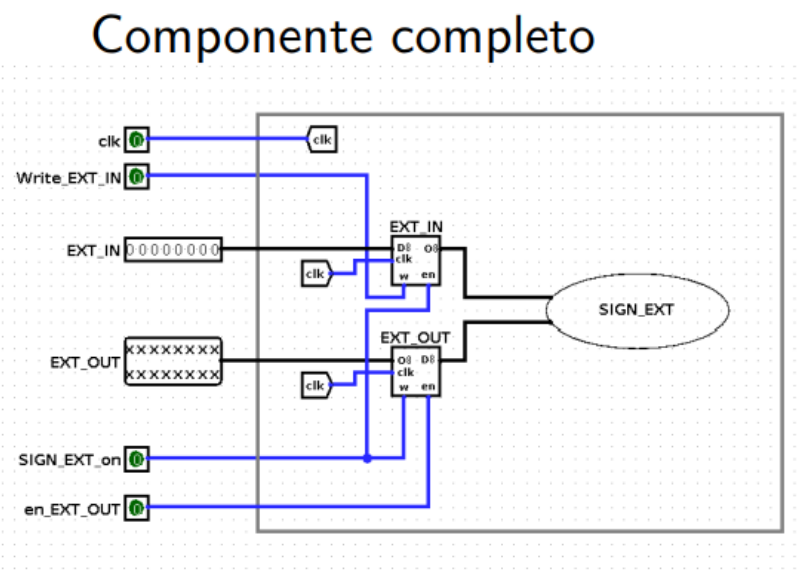
Señal	Efecto
$SIGN\_EXT_{on}$	activa la operación de extensión de signo de 8 *bits* a 16 *bits*

Datapath

Ejemplo de un componente ALU.



Ejemplo de un componente Extensor de signo.



Microprogramas

- Datapath
- Lenguaje de microinstrucciones (RTL)

Vamos a construir el micro con:
- Componentes
- Registros
- Líneas/buses

El lenguaje y Notación: Componentes

- Definimos **componentes** como circuitos con entradas, salidas y señales.
- Las **señales** son entradas que modifican el comportamiento de los circuitos.
- Las señales **se activarán** según como indique el *microprograma*.
- Se simbolizan con óvalos en el datapath.

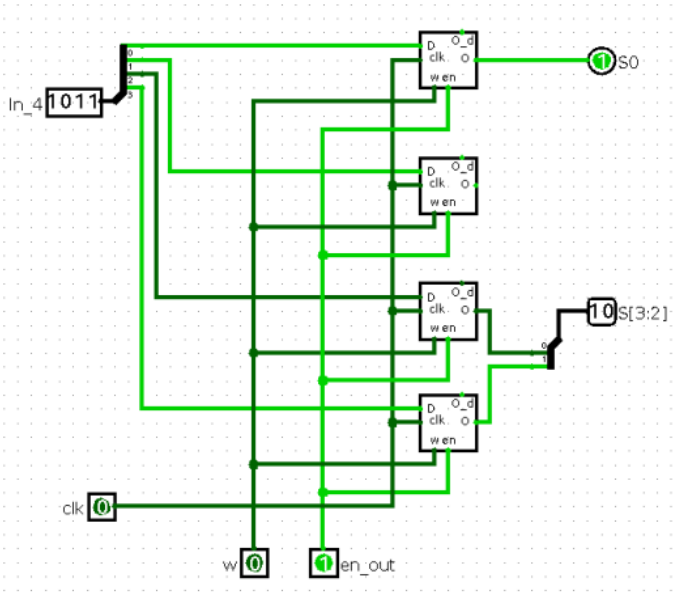
El lenguaje y Notación: Registros

- Existen registros que almacenan conjuntos de valores.
- Los registros pueden ser usados por completo o parte de ellos.
- Se simbolizan con rectángulos en el datapath.

Ejemplos
  - ALU IN1: Registro ALU IN1
  - R0[0]: Bit 0 del registro R0
  - R2[3:2]: Del bit 2 al bit 3 del registro R2

El lenguaje y Notación: Registros - O en circuito...

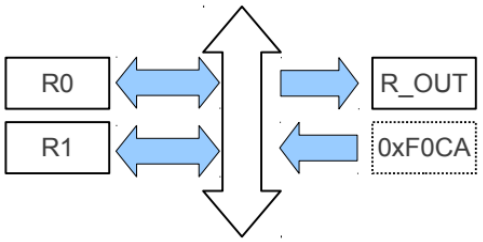
- R0[0]
- R2[3:2]



El lenguaje y Notación: Líneas

- Los datos se mueven por caminos (líneas).
- Podemos mover un dato de un registro a otro si hay un camino directo entre ellos.
- Podemos asignar un valor constante almacenado a un registro.
- Se simbolizan con flechas en el datapath.

Ejemplos



	¿vale?	
R1:= R0		

	¿vale?	
R1:= R0	✓	Copia el contenido de R0 en R1
R_OUT:=R0		

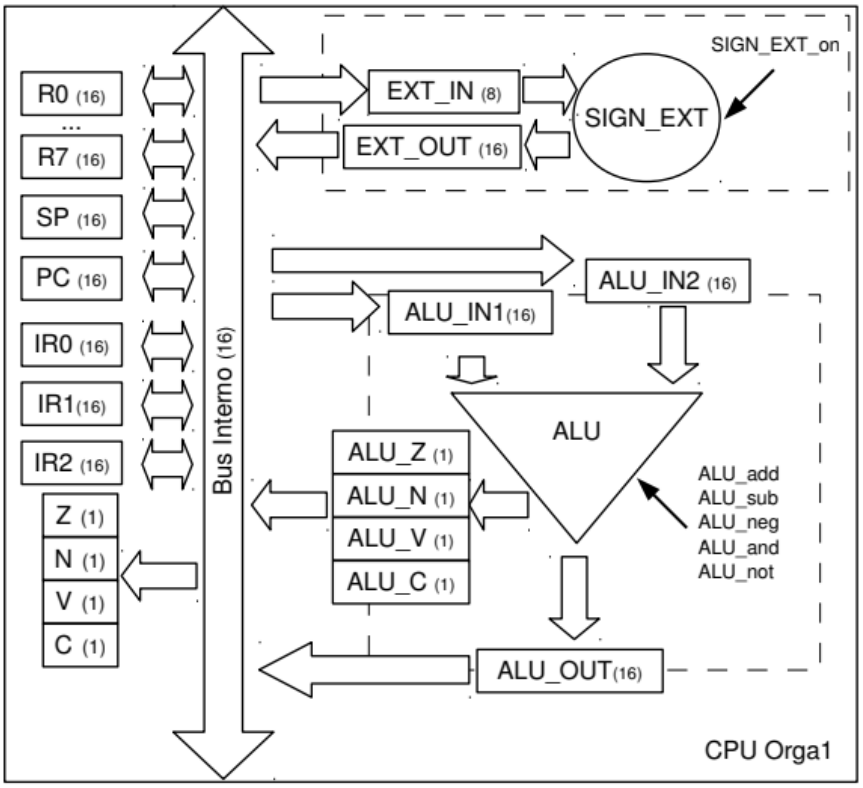
	¿vale?	
R1:= R0	✓	Copia el contenido de R0 en R1
R_OUT:=R0	✓	Copia el contenido de R0 en R_OUT
R0:=0xF0CA		

	¿vale?	
R1:= R0	✓	Copia el contenido de R0 en R1
R_OUT:=R0	✓	Copia el contenido de R0 en R_OUT
R0:=0xF0CA	✓	Copia la constante 0xF0CA a R0
R0:=R_OUT		

	¿vale?	
R1:= R0	✓	Copia el contenido de R0 en R1
R_OUT:=R0	✓	Copia el contenido de R0 en R_OUT
R0:=0xF0CA	✓	Copia la constante 0xF0CA a R0
R0:=R_OUT	×	El registro R_OUT es de solo escritura
R1:=x		

	¿vale?	
R1:= R0	✓	Copia el contenido de R0 en R1
R_OUT:=R0	✓	Copia el contenido de R0 en R_OUT
R0:=0xF0CA	✓	Copia la constante 0xF0CA a R0
R0:=R_OUT	×	El registro R_OUT es de solo escritura
R1:=x	×	No es posible asignar una constante *cualquiera*

Ejercicio 1 - Máquina Orga1 - Solución



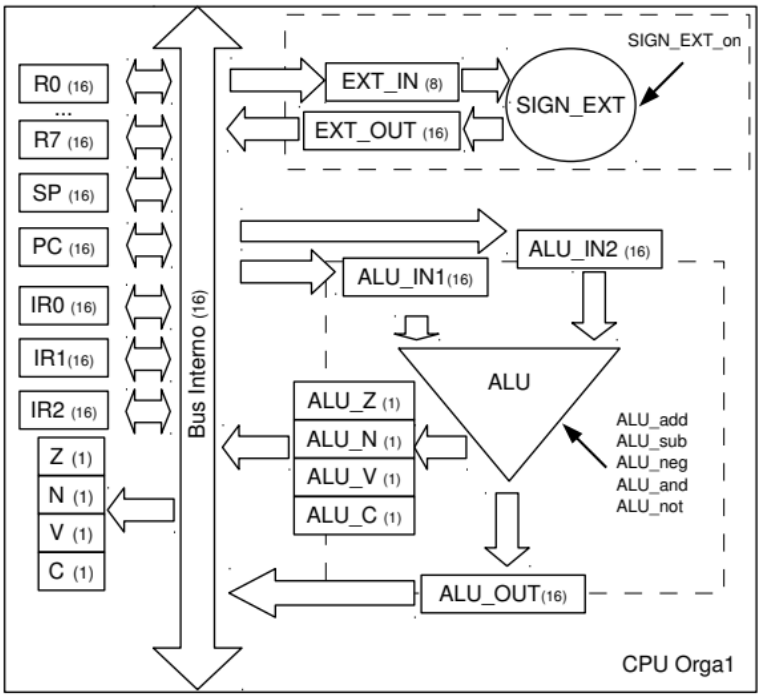
- 16 bits: R0, .., R7, SP, PC, IR0, IR1, IR2, EXT_OUT, ALU_IN1, ALU_IN2, ALU_OUT
- 8 bits: EXT_IN
- 1 bit: Z, N, V, C, ALU Z, ALU N, ALU V, ALU C
- Bus interno: 16 líneas.
- Los flags están conectados a las 4 líneas menos significativas del bus.
- El registro EXT_IN está conectado a las 8 líneas menos significativas del bus.

Ejercicio 1 - Máquina Orga1

2) Indicar cuál es la secuencia de señales (o microoperaciones) que debe realizar la unidad de control para ejecutar las siguientes instrucciones:

- MOV R5, R1
- AND R7, R1
- JE 0xFF

Ejercicio 1 - Máquina Orga1 - Solución



Ejercicio 1 - Solución: Secuencias de microoperaciones

- MOV R5,R1
  1. R5 := R1
- AND R7, R1
  1. ALU_IN1 := R7
  2. ALU_IN2 := R1
  3. ALU_{and}
  4. R7 := ALU_OUT
  5. Z := ALU_Z
  6. N := ALU_N
  7. C := ALU_C
  8. V := ALU_V
- JE 0xFF
  1. IF Z = 1
  2. EXT_IN := IR0[7:0]
  3. SIGN_EXT_on
  4. ALU_IN_1 := PC
  5. ALU_IN_2 := EXT_OUT
  6. ALU_add
  7. PC := ALU_OUT

Ejercicio 2 - Máquina Orga1

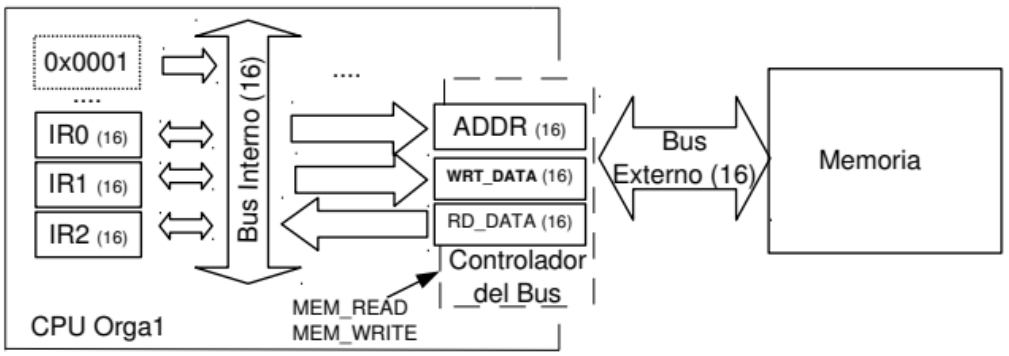
Se cuenta con una memoria con palabra y direccionamiento de 16 bits. El CPU se comunicará con ella utilizando un controlador de bus. Este es un dispositivo que posee 2 registros de entrada de 16 bits (ADDR, WRT DATA) y 1 de salida de 16 bits (RD DATA). Sus señales de control son:

▸ MEM WRITE: Activa la microoperación de escritura del contenido del registro WRT DATA en la dirección de memoria indicada por el ADDR

▸ MEM READ: Activa la microoperación de lectura del contenido de la dirección de memoria indicada por el ADDR, colocando el valor en el registro RD DATA.

1. Extender el camino de datos de la arquitectura de la máquina ORGA1. No dibujar la unidad de control para simplificar el diagrama.
2. ¿Qué componentes del camino de datos se encuentran dentro del CPU y fuera de él?
3. Indicar cuál es la secuencia de señales (o microoperaciones) que debe realizar la unidad de control para ejecutar las siguientes instrucciones:
   - ▸ MOV R2, [R5]
   - ▸ MOV [0xFF00], [0xFF01]
4. Como Tarea: Describa la secuencia de microoperaciones que realiza la unidad de control para realizar un fetch de una instrucción de la máquina Orga1.

Ejercicio 2 - Máquina Orga1: Solución



Ejercicio 2 - Solución: secuencias de microoperaciones

- MOV R2, [R5]
  1. ADDR := R5
  2. MEM_READ
  3. R2 := RD_DATA

- MOV [0xFF00], [0xFF01]
  1. ADDR := IR2
  2. MEM_READ
  3. WRT_DATA := RD_DATA
  4. ADDR := IR1
  5. MEM_WRITE

Ejercicio 3 La computadora STACK1 es una máquina de pila con direccionamiento a byte, tamaño de palabra de 16 bits y direcciones de memoria de 12 bits. Trabaja con aritmética complemento a 2 de 16 bits. Posee el siguiente set de instrucciones:

Instrucción	CodOp	Significado
PUSH [M]	0000	push [M]
POP [M]	0001	[M] := pop
ADD	0010	push(pop+pop)
SUB	0011	push(pop-pop)
JUMP	0100	PC := pop (sólo los 12 bits menos significativos)
SKIP_N	0101	ignora la próxima instrucción si top es < 0
SKIP_Z	0110	ignora la próxima instrucción si top es 0
SKIP_GE	0111	ignora la próxima instrucción si top es $>= 0$

El formato de instrucción de STACK1 es el que sigue:

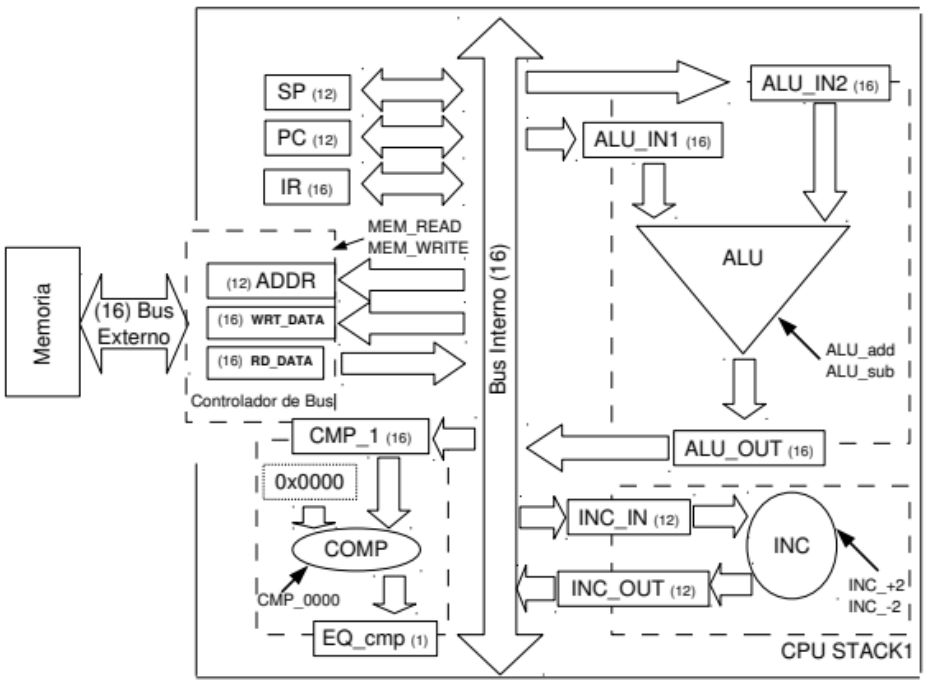
CodOp	Dirección
4 bits	12 bits

1) Definir el camino de datos y la organización del CPU de STACK1 para soportar la implementación de, al menos, estas instrucciones.
2) Describa la secuencia de microoperaciones que realiza la unidad de control para realizar un *fetch* de una instrucción.
3) Implementar las siguientes instrucciones:

I. JUMP
II. SKIP_Z
III. PUSH [0xFAC]
IV. ADD

Ejercicio 3: Solución

Se utiliza un circuito incrementador con 2 señales: $\texttt{INC\_+2}$ que suma 2 a la entrada, y $\texttt{INC\_-2}$ que resta 2 a la entrada.
ADDR, INC_IN, INC_OUT, SP y PC son registros de 12 bits.
IR, CMP_1, ALU_IN1, ALU_IN2, ALU_OUT, WRT_DATA y RD_DATA y los buses INTERNO y EXTERNO son de 16 bits.
EQ_CMP es de 1 bit. Las 12 líneas de los registros correspondientes están conectadas a las líneas menos significativas del BUS.



Ejercicio 3 - Solución: fetch

1. ADDR := PC
2. MEM_READ
3. IR := RD_DATA // cargo el IR
4. INC_IN := PC
5. $\texttt{INC\_+2}$
6. PC := INC_OUT // incremento PC

Ejercicio 3 - Solución: secuencia de microoperaciones

- JUMP
  1. INC_IN := SP
  2. $\texttt{INC\_+2}$
  3. SP := INC_OUT
  4. ADDR := SP
  5. MEM_READ
  6. PC := RD_DATA[11:0]

- SKIP_Z
  1. INC_IN := SP
  2. $\texttt{INC\_+2}$
  3. ADDR := INC_OUT
  4. MEM_READ
  5. CMP_1 := RD_DATA
  6. CMP_0000
  7. if EQ_cmp = 1
  8. INC_IN := PC
  9. $\texttt{INC\_+2}$
  10. PC := INC_OUT
  11. endif

- PUSH [X]
  1. ADDR := IR[11:0]
  2. MEM_READ
  3. WRT_DATA := RD_DATA
  4. ADDR := SP
  5. MEM_WRITE
  6. INC_IN := SP
  7. INC_{-2}
  8. SP := INC_OUT

- ADD
  1. INC_IN := SP
  2. INC_{+2}
  3. SP := INC_OUT
  4. ADDR := SP
  5. MEM_READ
  6. ALU_IN1 := RD_DATA // primer operando
  7. INC_IN := SP
  8. INC_{+2}
  9. SP := INC_OUT
  10. ADDR := SP
  11. MEM_READ
  12. ALU_IN2 := RD_DATA // segundo operando
  13. ALU_{add}
  14. WRT_DATA := ALU_OUT
  15. ADDR := SP
  16. MEM_WRITE // push resultado
  17. INC_IN := SP
  18. INC_{-2}
  19. SP := INC_OUT