Práctica 4: Microarquitectura - Introducción

Segundo Cuatrimestre 2022

Organización del Computador I
DC - UBA

Introducción

Sobre la clase de hoy

Hoy vamos a ver los principios de construcción, práctica y ejemplos de microarquitectura y microprogramas relacionados. La estructura de la clase va ser la siguiente:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**
- **Ejemplos de microarquitectura y microprogramación**

Sobre la clase de hoy

Hoy vamos a ver los principios de construcción, práctica y ejemplos de microarquitectura y microprogramas relacionados. La estructura de la clase va ser la siguiente:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**
- **Ejemplos de microarquitectura y microprogramación**

Sobre la clase de hoy

Hoy vamos a ver los principios de construcción, práctica y ejemplos de microarquitectura y microprogramas relacionados. La estructura de la clase va ser la siguiente:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**
- **Ejemplos de microarquitectura y microprogramación**

Sobre la clase de hoy

Hoy vamos a ver los principios de construcción, práctica y ejemplos de microarquitectura y microprogramas relacionados. La estructura de la clase va ser la siguiente:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**
- **Ejemplos de microarquitectura y microprogramación**

Sobre la clase de hoy

Hoy vamos a ver los principios de construcción, práctica y ejemplos de microarquitectura y microprogramas relacionados. La estructura de la clase va ser la siguiente:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**
- **Ejemplos de microarquitectura y microprogramación**

El programa como objeto de estudio

¿Qué intención manifiesta un programa?

Podemos interpretar al programa como:

¿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.

¿Qué intención manifiesta un programa?

Algunos detalles a tener en cuenta en base a esta definición:

- 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)

¿Qué intención manifiesta un programa?

Algunos detalles a tener en cuenta en base a esta definición:

- 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)

¿Qué intención manifiesta un programa?

Algunos detalles a tener en cuenta en base a esta definición:

- 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

Perspectiva declarativa de un programa

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

Perspectiva declarativa de un programa

```
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**←),

Perspectiva declarativa de un programa

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

Perspectiva declarativa de un programa

```
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**←), inicializan los mismos (**memoria**⇐), o modifican el flujo de ejecución (**PC**↩).

Perspectiva declarativa de un programa

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

Perspectiva declarativa de un programa

```
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**).

Perspectiva declarativa de un programa

```
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**).

Perspectiva declarativa de un programa

```
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**).

Perspectiva declarativa de un programa

```
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**).

Perspectiva declarativa de un programa

```
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**).

Perspectiva declarativa de un programa

```
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**).

Perspectiva declarativa de un programa

```
main:   MOV R1,R3
        ADD R1,R2
        JMP main(etc)
  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**).

Perspectiva estructural

Perspectiva estructural de un programa

```
        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**).

Perspectiva estructural de un programa

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

Perspectiva estructural de un programa

Ahora veamos cómo se ve el programa codificado y almacenado en memoria principal.

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			

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			

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**?

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 tomar 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**.

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			

PC: 0x00

MOV R1,R3

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			

PC: 0x02

ADD R1,R2

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			

PC: 0x04

JMP main

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			

PC: 0x00

MOV R1,R3

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			

PC: 0x02

ADD R1,R2

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			

PC: 0x04

JMP main

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, **según la semántica de las instrucciones**.

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			

$\hookleftarrow$

PC: 0x00

R1 $\leftarrow$ R3

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			

$\hookleftarrow$

PC: 0x02

$R1 \leftarrow R1 + R2$

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			

$\hookleftarrow$

PC: 0x04

PC $\leftarrow$ 0x00

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			

$\hookleftarrow$

PC: 0x00

R1 $\leftarrow$ R3

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			

$\hookleftarrow$

PC: 0x02

$R1 \leftarrow R1 + R2$

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			

$\hookleftarrow$

PC: 0x04

PC $\leftarrow$ 0x00

Datapath

Repaso de las perspectivas del programa

En la sección anterior observamos:

- Que el programa se codifica y almacena en memoria junto con los datos (**Von Neumann**).
- Que el flujo de ejecución será secuencial salvo que lo indiquemos con las instrucciones de salto (**PC,JMP,JXX**).

Repaso de las perspectivas del programa

En la sección anterior observamos:

- Que el programa se codifica y almacena en memoria junto con los datos (**Von Neumann**).
- Que el flujo de ejecución será secuencial salvo que lo indiquemos con las instrucciones de salto (**PC,JMP,JXX**).

Repaso de las perspectivas del programa

En la sección anterior observamos:

- Que el programa se codifica y almacena en memoria junto con los datos (**Von Neumann**).
- Que el flujo de ejecución será secuencial salvo que lo indiquemos con las instrucciones de salto (**PC,JMP,JXX**).

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Para estudiar su relación con:

- Los componentes de nuestra organización
- Las señales de control necesarias
- El orden en el que éstas deben activarse

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Para estudiar su relación con:

- Los componentes de nuestra organización
- Las señales de control necesarias
- El orden en el que éstas deben activarse

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Para estudiar su relación con:

- Los componentes de nuestra organización
- Las señales de control necesarias
- El orden en el que éstas deben activarse

Repaso de las perspectivas del programa

A continuación analizaremos las siguientes unidades funcionales:

- Etapas de Fetch-Decode
- Instrucción de tipo ADD
- Instrucción de tipo STR
- Instrucciones de tipo JMP o JZ

Para estudiar su relación con:

- Los componentes de nuestra organización
- Las señales de control necesarias
- El orden en el que éstas deben activarse

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

- 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**

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

- 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**

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

- 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**

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

- 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**

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

- 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**

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

- 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**

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

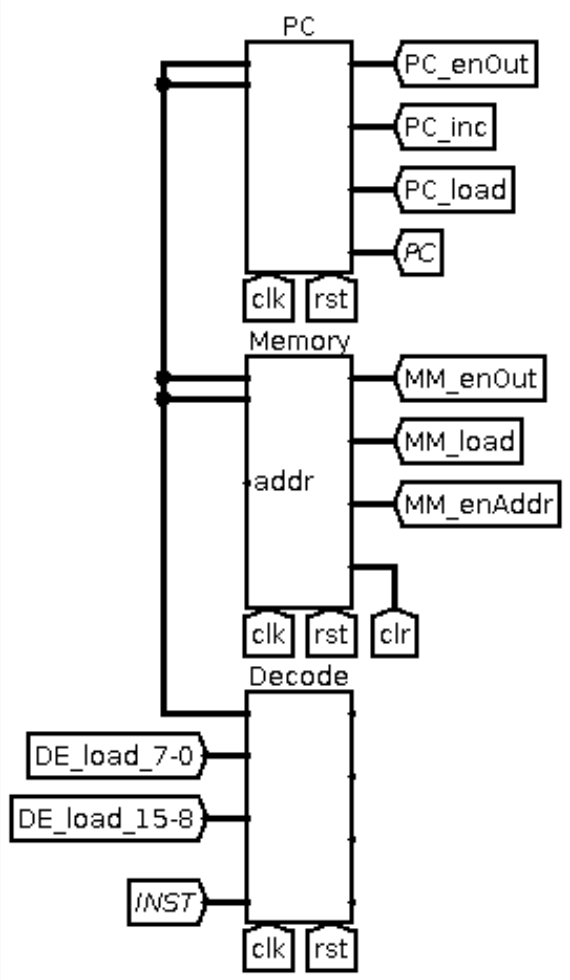
- Arquitectura *von Neumann*, memoria de datos e instrucciones compartida
- 8 registros de propósito general, `R0` a `R7`
- 1 registro de propósito específico PC
- Tamaño de palabra de 8 bits e instrucciones de 16 bits
- Memoria de 256 palabras de 8 bits
- Bus de 8 bits
- Diseño microprogramado

Repaso de la arquitectura de `OrgaSmall`

Tengan en cuenta que vamos a trabajar con la siguiente arquitectura (`OrgaSmall` no es `ORGA1`):

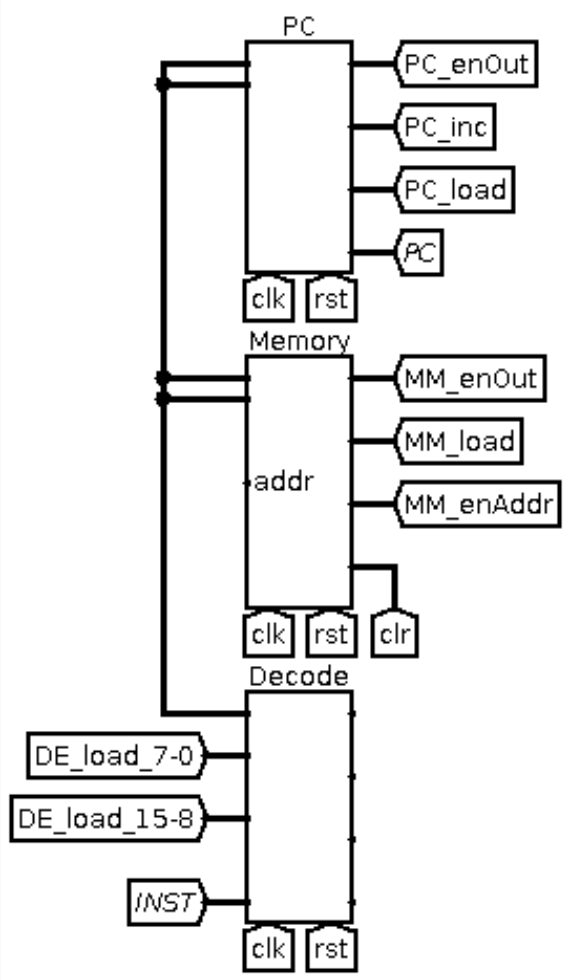
- Arquitectura *von Neumann*, memoria de datos e instrucciones compartida
- 8 registros de propósito general, `R0` a `R7`
- 1 registro de propósito específico PC
- Tamaño de palabra de 8 bits e instrucciones de 16 bits
- Memoria de 256 palabras de 8 bits
- Bus de 8 bits
- **Diseño microprogramado**

Datapath del Fetch-Decode



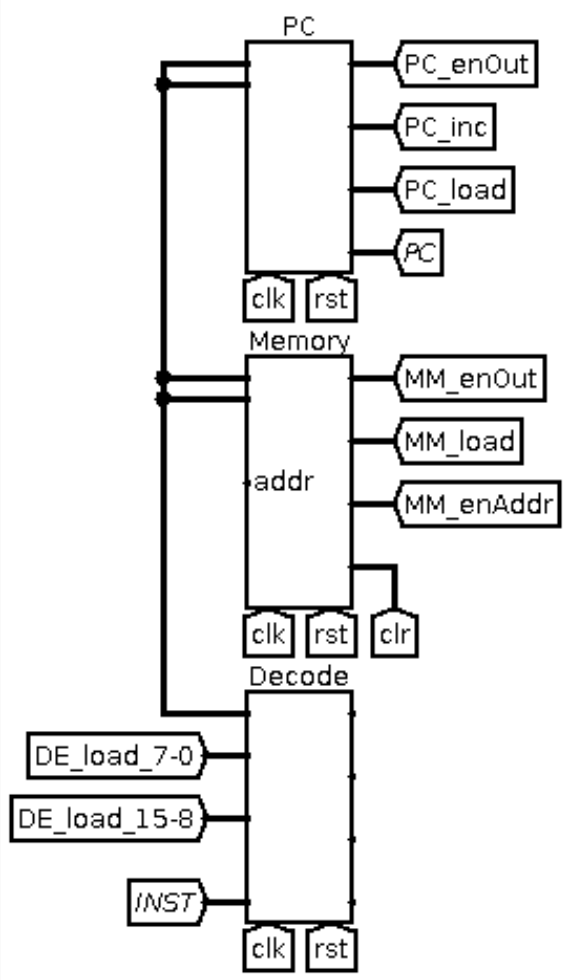
Ahora sí podemos analizar por partes el flujo de datos (data-path) que permite reproducir el mecanismo deseado de **FETCH DECODE EXECUTE**.

Datapath del Fetch-Decode



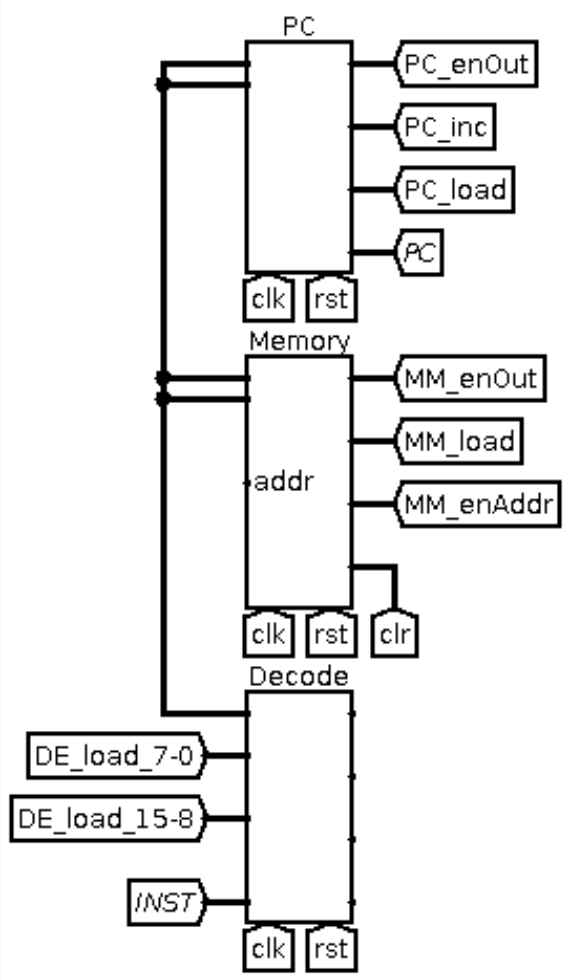
En el ciclo de Fetch-Decode participan el **PC**, la **Memoria** y la unidad de **Decode**. El **PC** no sólo encapsula un registro sino que expone también una señal de control que permite incrementarlo (en dos palabras).

Datapath del Fetch-Decode



Veamos qué secuencia de micro-instrucciones (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.

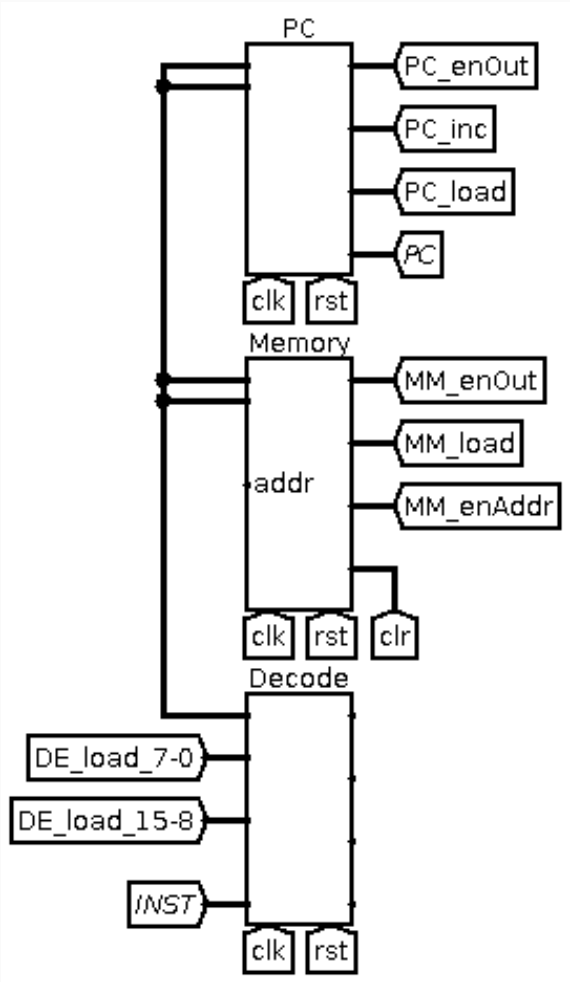
Datapath del Fetch-Decode



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

Aquí las asignaciones (:=) indican la activación de un par de señales **write/enableOut** y las declaraciones aisladas (PC_{inc}) indican la activación durante un ciclo de la señal indicada.

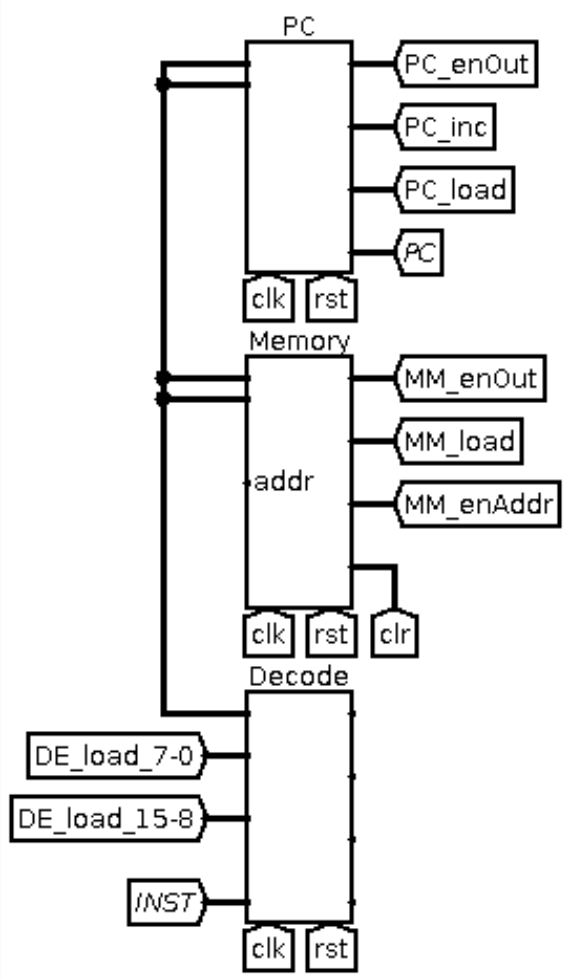
Datapath del Fetch-Decode



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
```

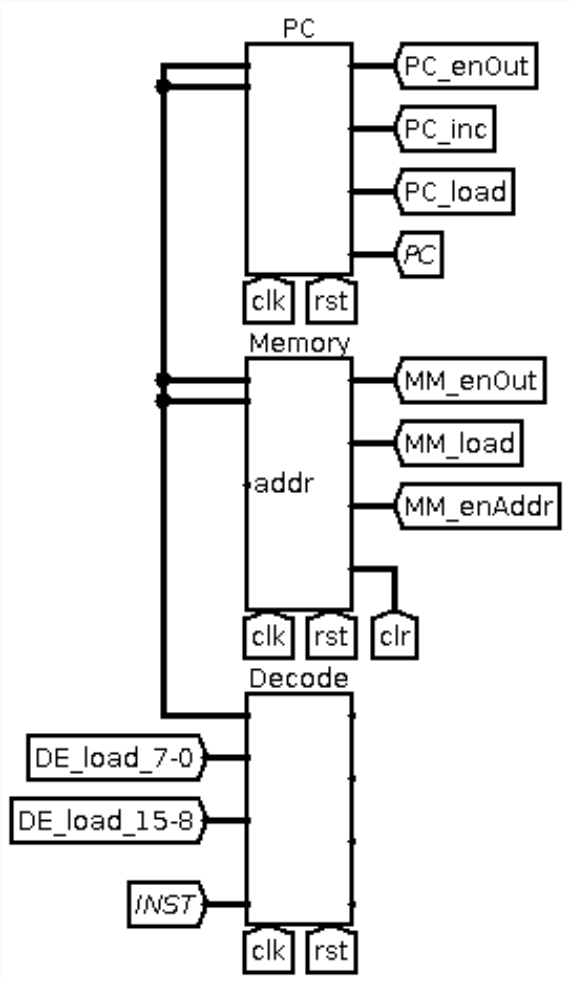
Datapath del Fetch-Decode



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.

Datapath del Fetch-Decode



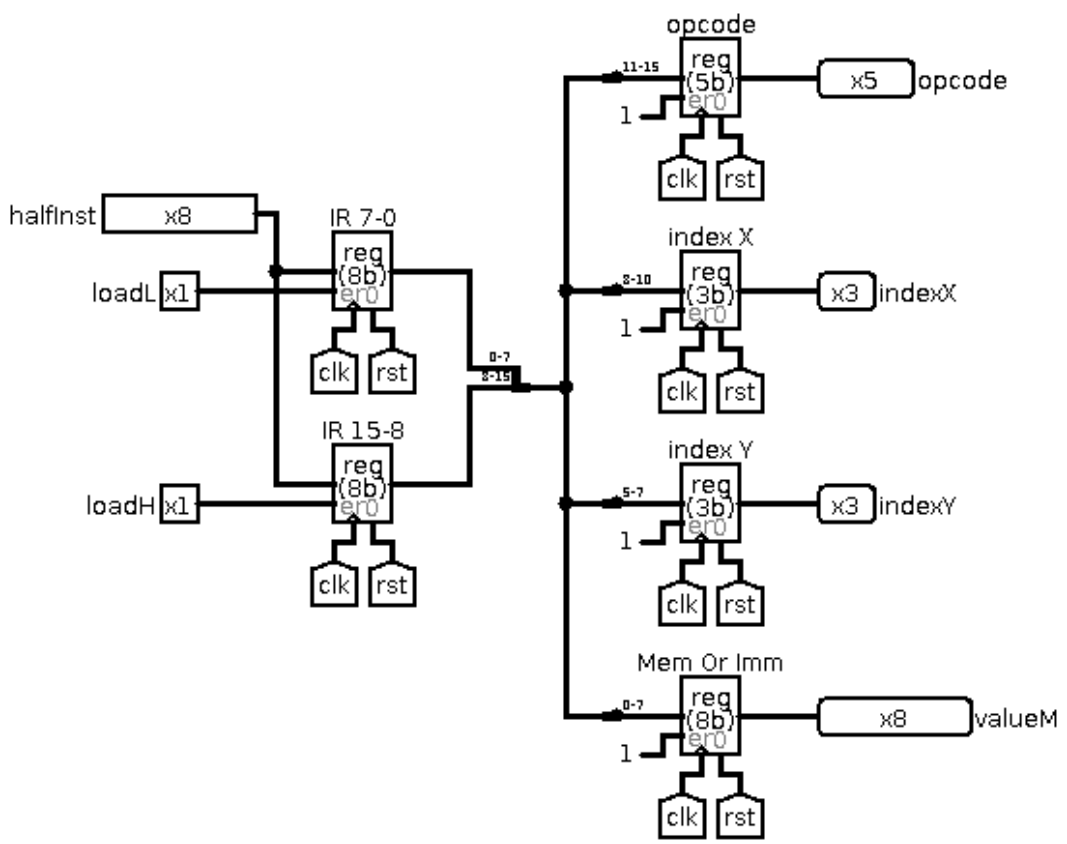
En RTL:

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

Como señales:

PC_enOut MM_enAddr
MM_enOut DE_loadH PC_inc
PC_enOut MM_enAddr
MM_enOut DE_loadL PC_inc

Datapath del Fetch-Decode



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.

Datapath del Fetch-Decode

En RTL:

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

Como señales:

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

Habiendo visto como se micro-programan las etapas de fetch y decode, pensemos:

- ¿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 Fetch-Decode

En RTL:

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

Como señales:

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

Habiendo visto como se micro-programan las etapas de fetch y decode, pensemos:

- ¿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 Fetch-Decode

En RTL:

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

Como señales:

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

Habiendo visto como se micro-programan las etapas de fetch y decode, pensemos:

- ¿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 Fetch-Decode

En RTL:

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

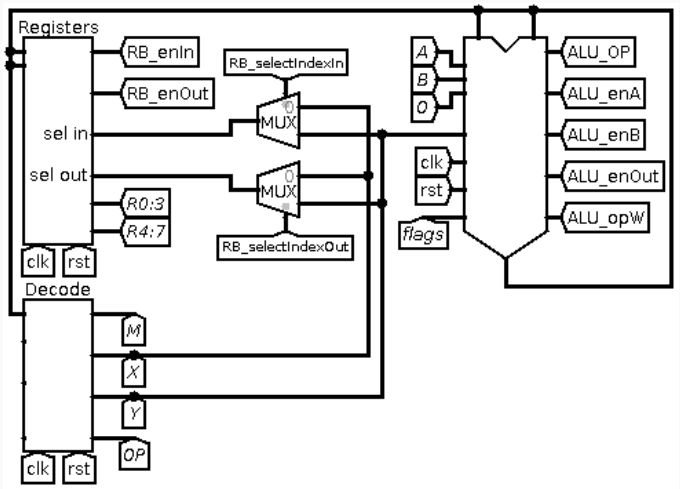
Como señales:

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

Habiendo visto como se micro-programan las etapas de fetch y decode, pensemos:

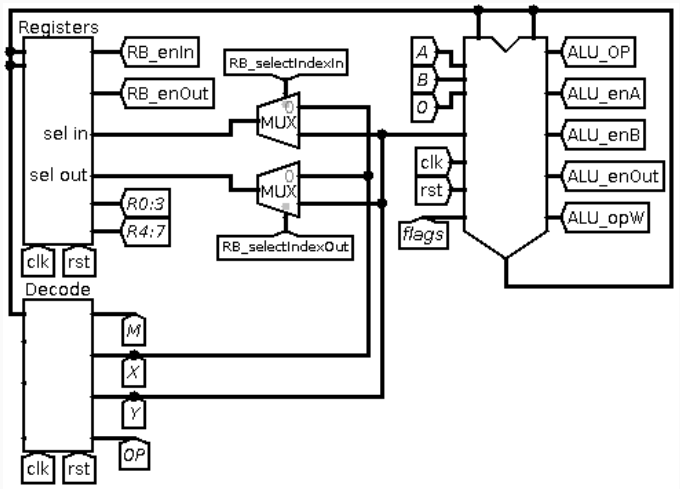
- ¿El **PC** se actualiza en cada ciclo de reloj?
- ¿Cómo sabemos **qué microinstrucción** se ejecuta en cada ciclo de reloj?
- ¿Dónde se almacenan las microinstrucciones?

Datapath del ADD



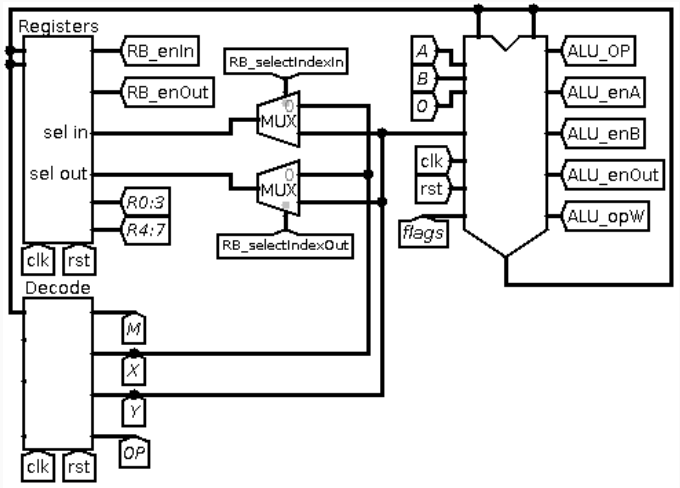
En la ejecución (**EXECUTE**) del **ADD** participan la **ALU**, que resuelve la aritmética, los **Registros** y el **Decode** que indica qué registros participan.

Datapath del ADD



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.

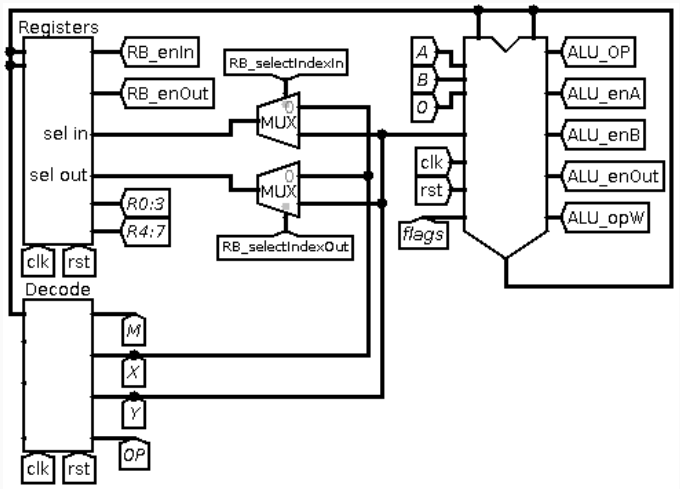
Datapath del ADD



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

Aquí las asignaciones a R_A, R_B 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).

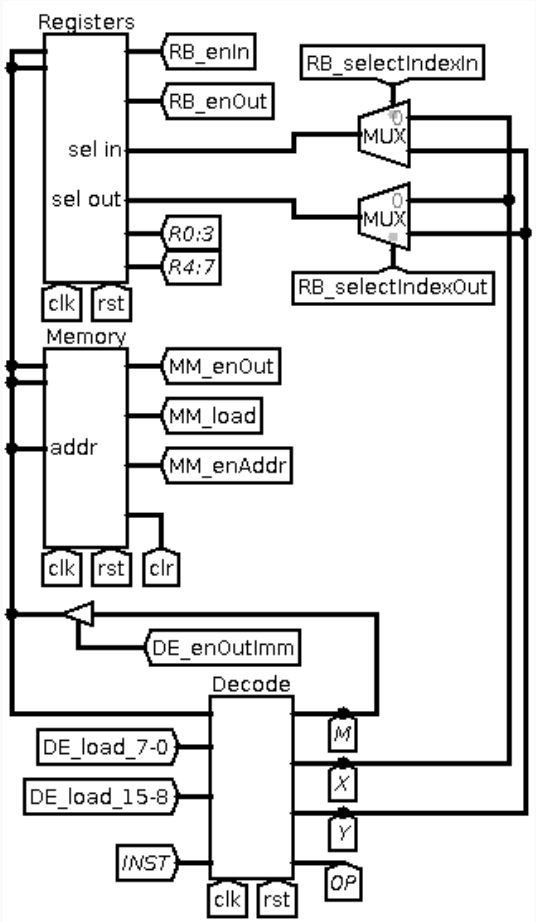
Datapath del ADD



El microprograma asociado es el siguiente:

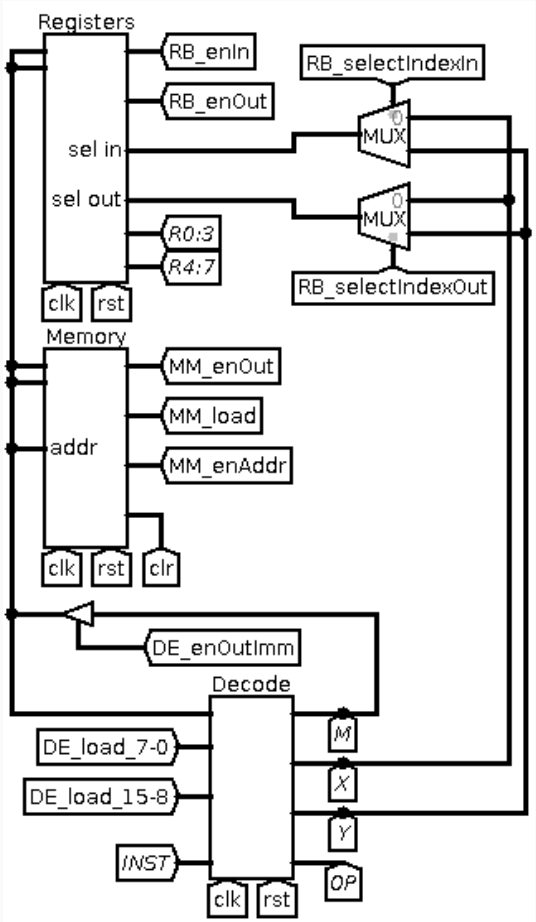
RB_enOut ALU_enA RB_selectIndexOut=0
RB_enOut ALU_enB RB_selectIndexOut=1
ALU_OP=ADD ALU_opW
RB_enIn ALU_enOut RB_selectIndexIn=0

Datapath del STR



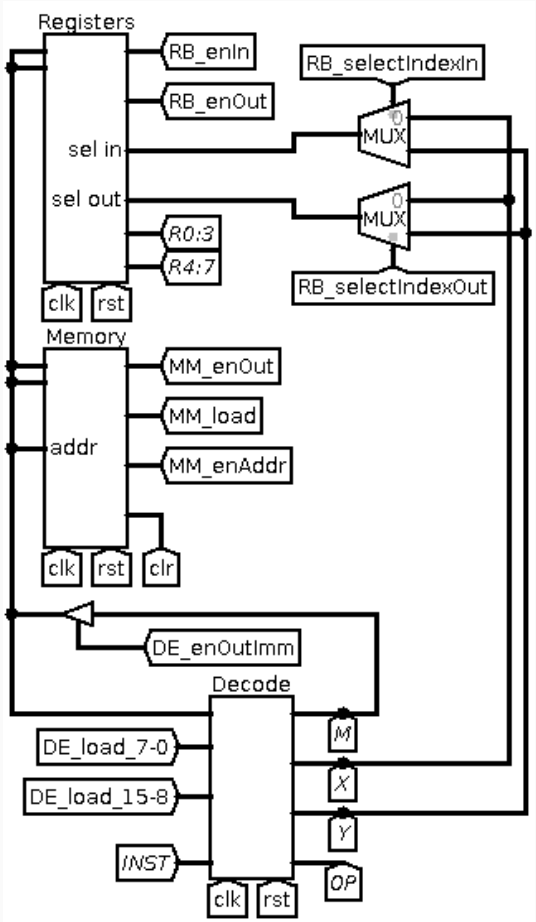
En la ejecución (**EXECUTE**) del **STR** participan el controlador de **memoria**, los **Registros** y el **Decode** que indica el registro fuente y la dirección destino.

Datapath del STR



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.

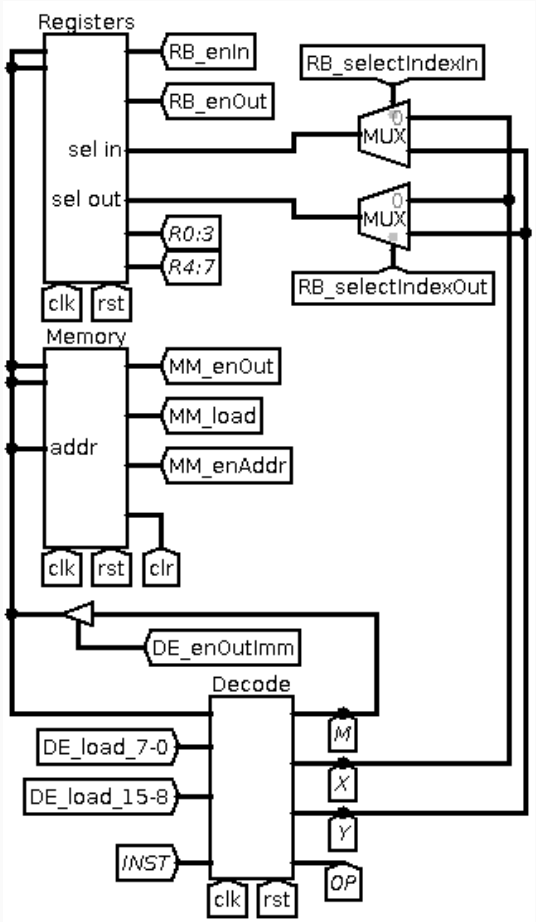
Datapath del STR



$$M_{addr} := DE_{imm}$$
$$M_{data} := RA$$

Notemos que son simplemente dos asignaciones pero que el controlador tiene entradas **de dirección y de datos**.

Datapath del STR



El microprograma asociado es el siguiente:

DE_enOutImm MM_enAddr
RB_enOut MM_load RB_selectIndexOut=0

Observemos que la asiganción 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:

Saltos condicionales, ¿cómo implementarlos?

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

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

Saltos condicionales, ¿cómo implementarlos?

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

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

¿Cómo conseguimos que la asignación se haga sólo si se cumple la condición ($Z = 0$)?

Saltos condicionales, ¿cómo implementarlos?

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

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

¿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.

Unidad de control

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**.

Unidad de control

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.

Unidad de control

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 sobre escribe el valor del **micro PC** si se habilita la señal **load_microOp**.

Unidad de control

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 sobre escribe el valor del **micro PC** si se habilita la señal **load_microOp**.

Unidad de control

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 sobre escribe el valor del **micro PC** si se habilita la señal **load_microOp**.

Unidad de control

Vamos presentar el diagrama de la unidad de control, donde podemos observar:

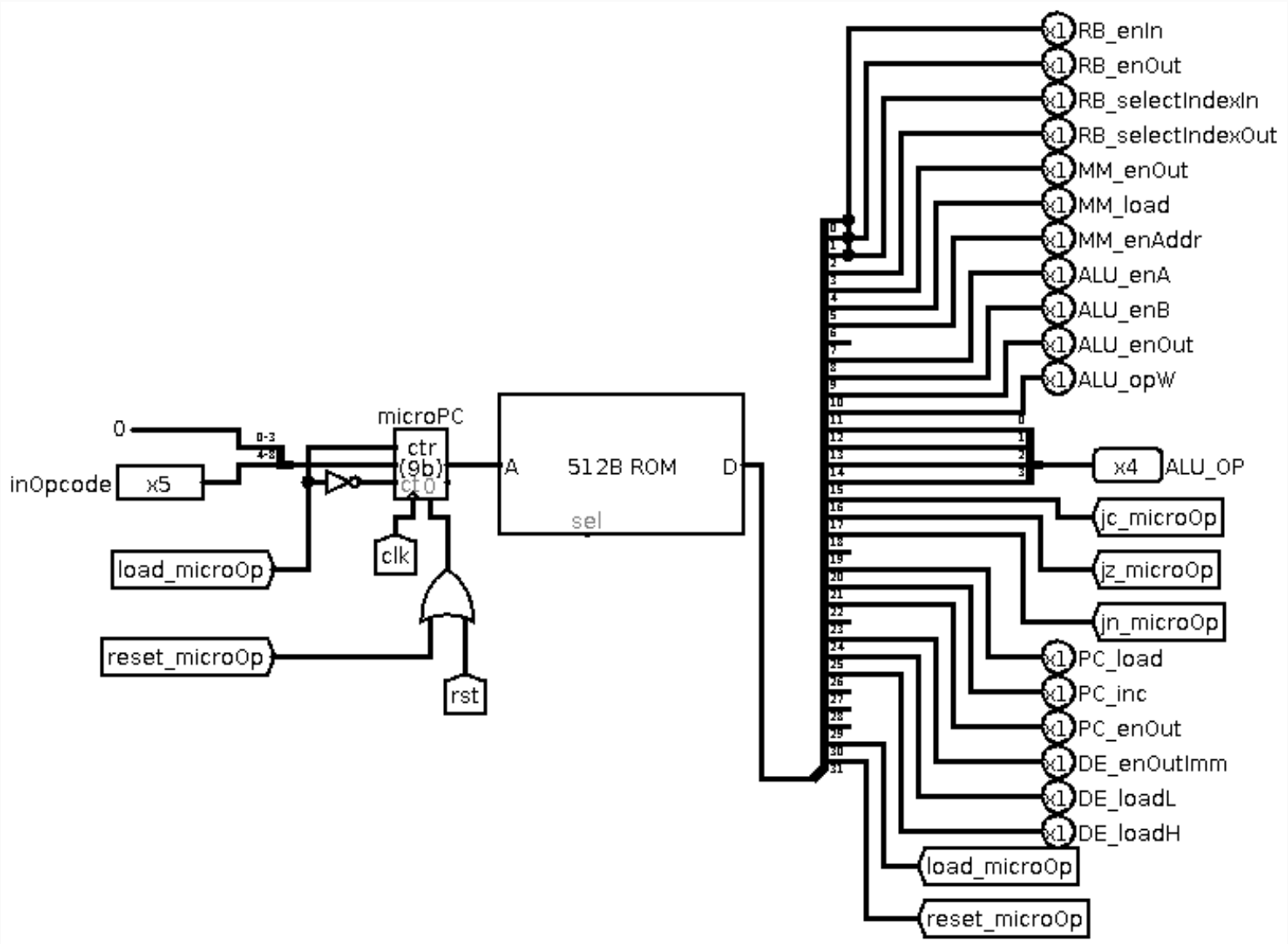
- Que el **micro PC** es un contador, ya que en cada ciclo de reloj su comportamiento por defecto es incrementar en uno la posición de memoria a ser leída.
- Las salidas de control que pueden modificar el comportamiento del **micro PC** (**load_microOp, reset_microOp**).
- Que las salidas están cableadas 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 sobre escribe el valor del **micro PC** si se habilita la señal **load_microOp**.

Unidad de control

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 sobre escribe el valor del **micro PC** si se habilita la señal **load_microOp**.

Unidad de control



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**.

$\Rightarrow$

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**.

Saltos condicionales y unidad de control

=

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
...

$\Rightarrow$

00400040
04200010
00400040
02200010
40000000
80000000
...

Saltos condicionales y unidad de control

Recapitulando, queríamos implementar JZ:

Saltos condicionales y unidad de control

Recapitulando, queríamos implementar JZ:

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

Saltos condicionales y unidad de control

Recapitulando, queríamos implementar JZ:

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

¿Cómo conseguimos que la asignación se haga sólo si se cumple la condición ($Z = 0$)?

Saltos condicionales y unidad de control

Recapitulando, queríamos implementar JZ:

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

¿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**.

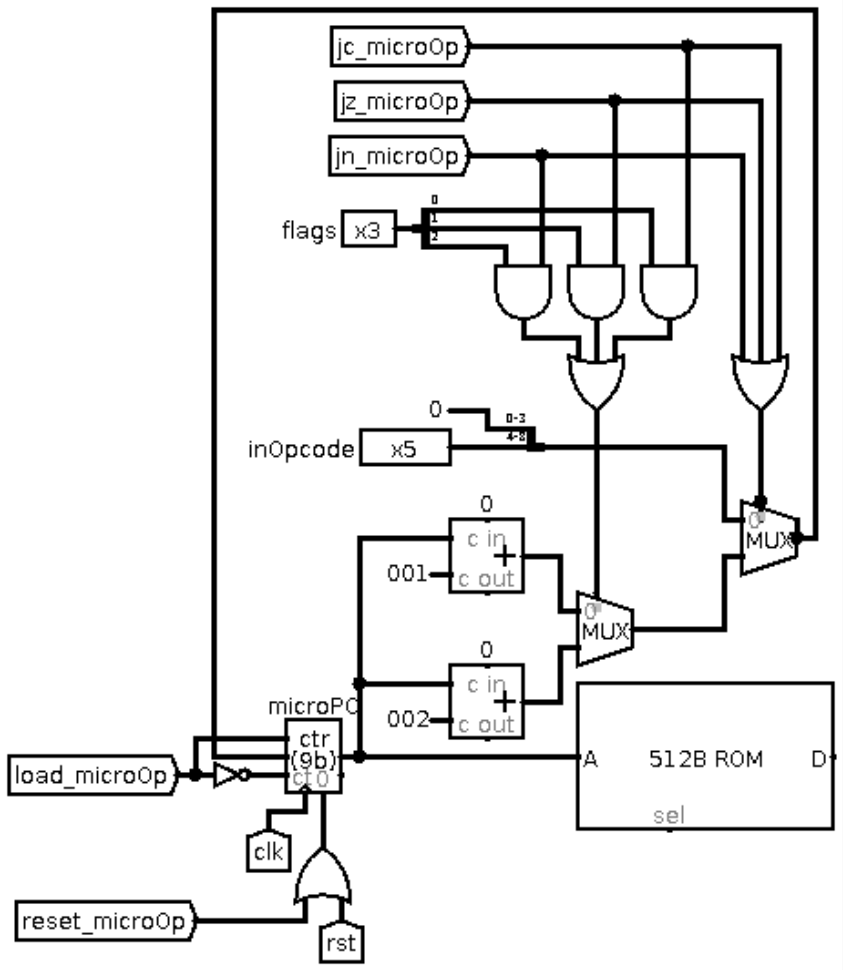
Saltos condicionales y unidad de control

Recapitulando, queríamos implementar JZ:

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

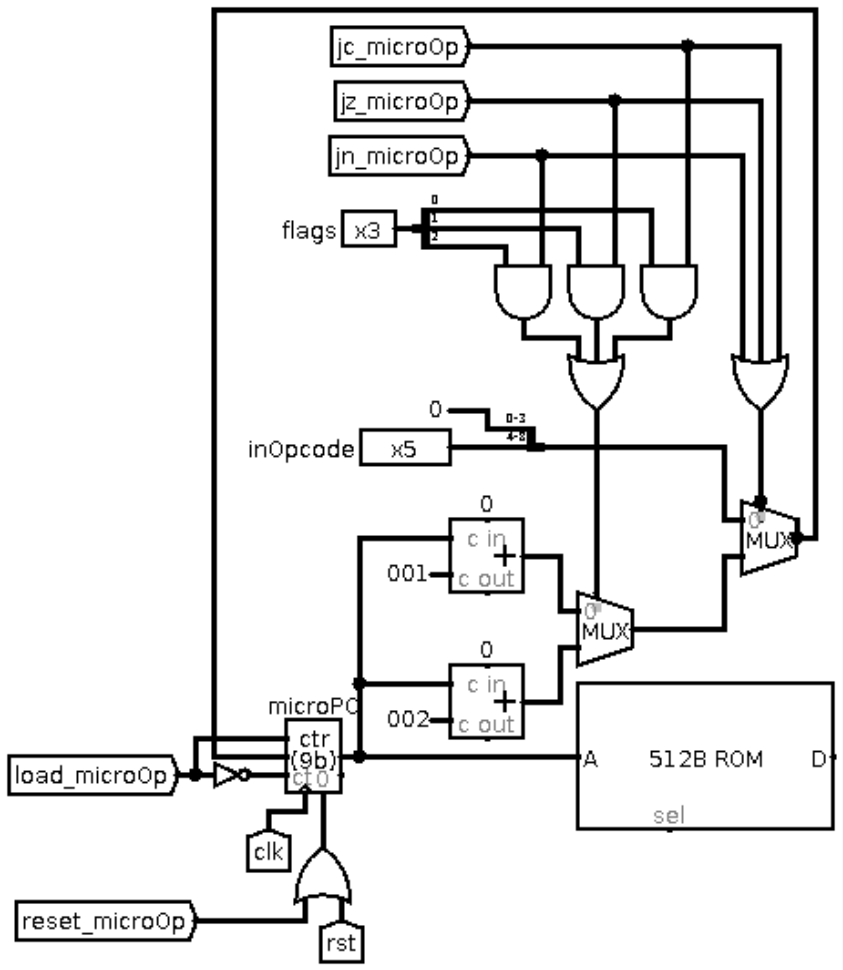
¿Cómo conseguimos que la asignación se haga sólo si se cumple la condición ($Z = 0$)?
Vamos a sobreescribir el valor del **micro PC** sólo si se encuentra habilitada la señal correspondiente de la **ALU**.
Veamos la propuesta.

Unidad de control



Observemos lo siguiente:

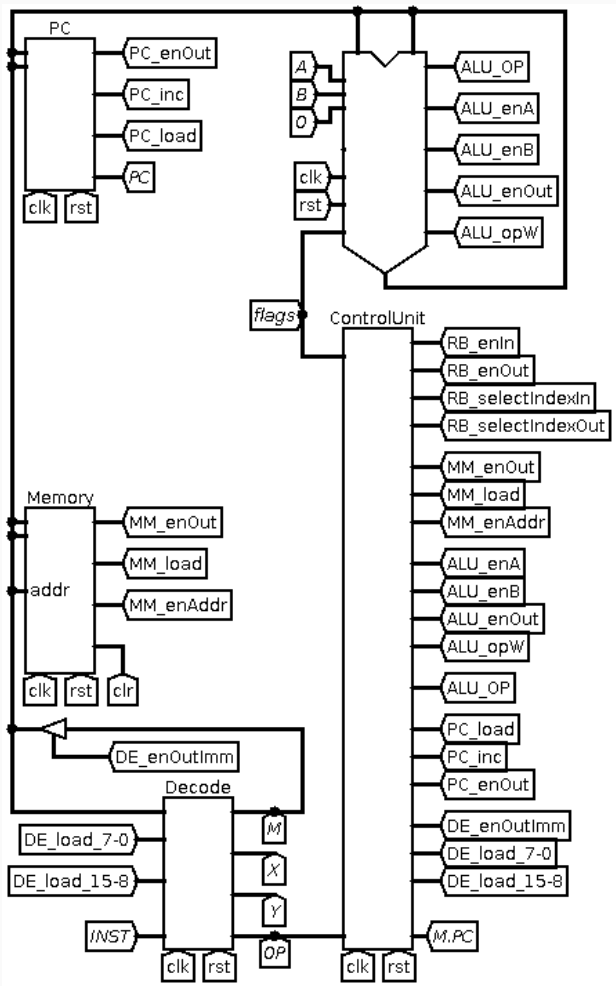
Unidad de control



Observemos lo siguiente:

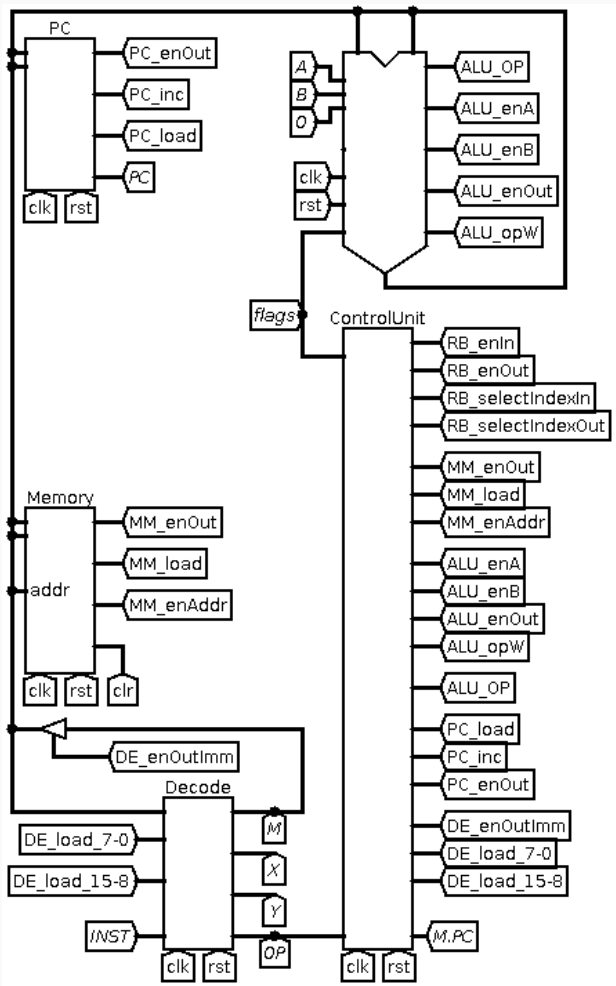
El **micro PC** se puede sobreescribir con la señal de **load_microOp** en conjunto con dos selectores de multiplexores: **el de la derecha indicando si se sobreescribe por una nueva instrucción o flags; y el de la izquierda indicando si se incrementa el micro PC en 1 (flag habilitado) o 2 (flag en cuestión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

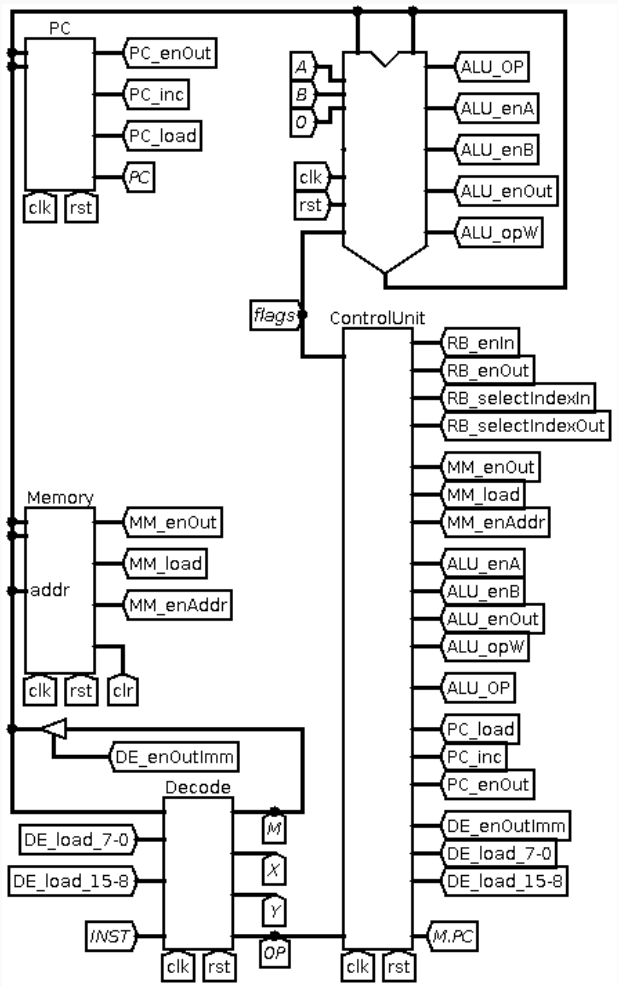
Datapath del JZ



Ya podemos notar que la **unidad de control** participa en todos los casos.

$$IF \quad Z = 0$$
$$PC \quad := \quad DE_{imm}$$

Datapath del JZ



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?

Volviendo a las preguntas

¿Qué función cumple la señal reset_microOp aquí?

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

Volviendo a las preguntas

¿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 microprogamas 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
```

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.

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
```

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

Estructura del microprograma

En el comienzo del código de los microprogramas encontrarán esto:

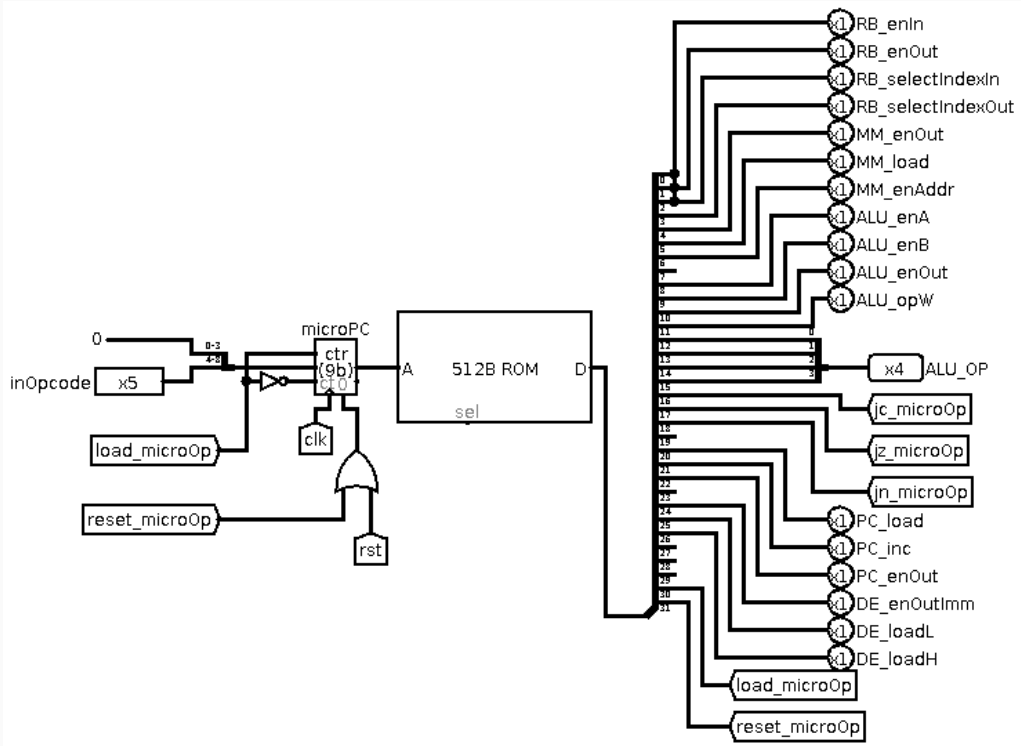
```
00000:
PC_enOut  MM_enAddr
MM_enOut  DE_loadH  PC_inc
PC_enOut  MM_enAddr
MM_enOut  DE_loadL  PC_inc
load_microOp
reset_microOp

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
```

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

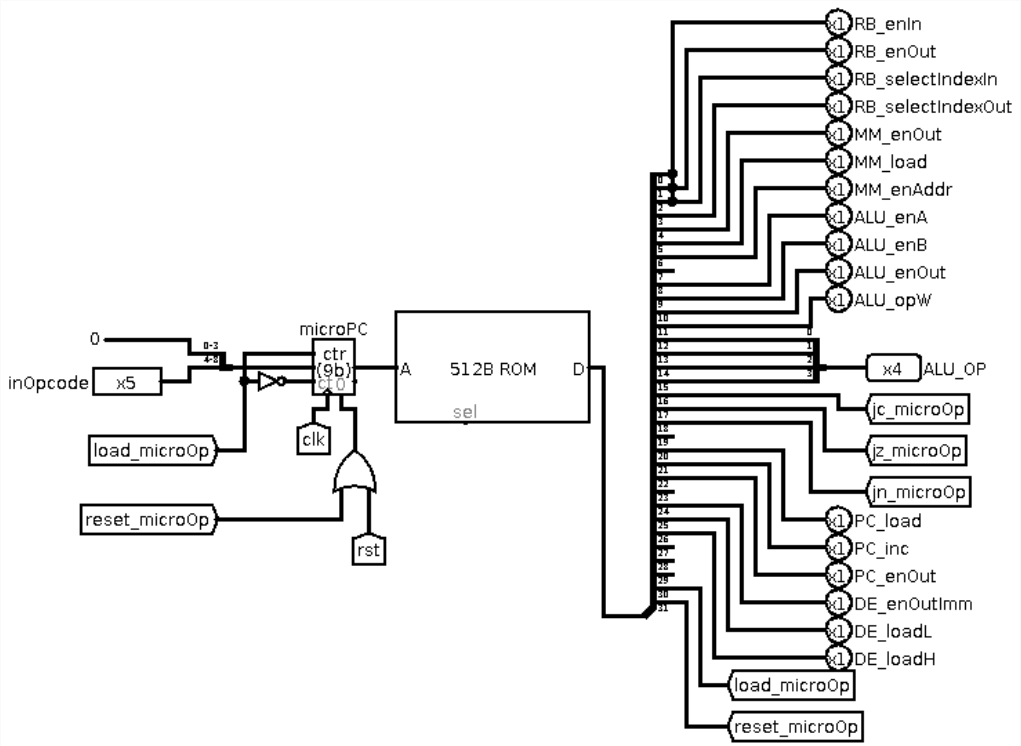
Observemos que:



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**).



Cierre de la introducción

Volviendo a las preguntas

Ya deberíamos poder responder a estas preguntas:

- ¿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?

Volviendo a las preguntas

Ya deberíamos poder responder a estas preguntas:

- ¿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?

Volviendo a las preguntas

Ya deberíamos poder responder a estas preguntas:

- ¿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?

Volviendo a las preguntas

Ya deberíamos poder responder a estas preguntas:

- ¿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?

Repaso de la clase de hoy

Hasta ahora vimos:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**

Repaso de la clase de hoy

Hasta ahora vimos:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**

Repaso de la clase de hoy

Hasta ahora vimos:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**

Repaso de la clase de hoy

Hasta ahora vimos:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**

Repaso de la clase de hoy

Hasta ahora vimos:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**

Y nos falta:

- **Ejemplos de microarquitectura y microprogramación**

Repaso de la clase de hoy

Hasta ahora vimos:

- **Distintas perspectivas sobre el programa como objeto, un repaso**
- **Descomposición del datapath según sus funciones** (Fetch-Decode, aritmética, saltos)
- **Constitución de la Unidad de Control**

Y nos falta:

- **Ejemplos de microarquitectura y microprogramación**

Pasemos a una breve ronda de preguntas antes de los ejercicios.