Microarquitectura 03: Ciclo de Instrucción y Microprogramación

Organización del computador - FIUBA

2.^{do} cuatrimestre de 2023

Última modificación: Mon Sep 25 12:24:11 2023 -0300

Créditos

Para armar las presentaciones del curso utilizamos:

R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.

D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.

J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

El contenido de los slides está basado en las presentaciones de Patricio Moreno y de Organización del Computador I - FCEN.

Tabla de contenidos

Tabla de contenidos

Ciclo de Vida de un Programa

Ciclo de Vida de un Programa

1. Programación: Escribir un algoritmo en **lenguaje ensamblador (o assembly)**. Esto es el **código fuente**.

Ciclo de Vida de un Programa

1. Programación: Escribir un algoritmo en **lenguaje ensamblador (o assembly)**. Esto es el **código fuente**.
2. Ensamblado: Un programa llamado **Ensamblador** toma el **código fuente** y lo traduce a un **código máquina**.

Ciclo de Vida de un Programa

1. Programación: Escribir un algoritmo en **lenguaje ensamblador (o assembly)**. Esto es el **código fuente**.
2. Ensamblado: Un programa llamado **Ensamblador** toma el **código fuente** y lo traduce a un **código máquina**.
   2.1 Resuelve las directivas dirigidas al **Ensamblador**.
   2.2 Calcula el tamaño de cada instrucciones, y resuelve las etiquetas.
   2.3 Traduce las instrucciones a 0s y 1s.

Ciclo de Vida de un Programa

1. Programación: Escribir un algoritmo en **lenguaje ensamblador (o assembly)**. Esto es el **código fuente**.
2. Ensamblado: Un programa llamado **Ensamblador** toma el **código fuente** y lo traduce a un **código máquina**.
   2.1 Resuelve las directivas dirigidas al **Ensamblador**.
   2.2 Calcula el tamaño de cada instrucciones, y resuelve las etiquetas.
   2.3 Traduce las instrucciones a 0s y 1s.
3. Carga: Se le indica al **Ensamblador** una dirección inicial, y éste copia el **código máquina** en la desde esa posición en adelante. Luego, carga esa misma dirección en el **PC (Program Counter)**.

Ciclo de Vida de un Programa

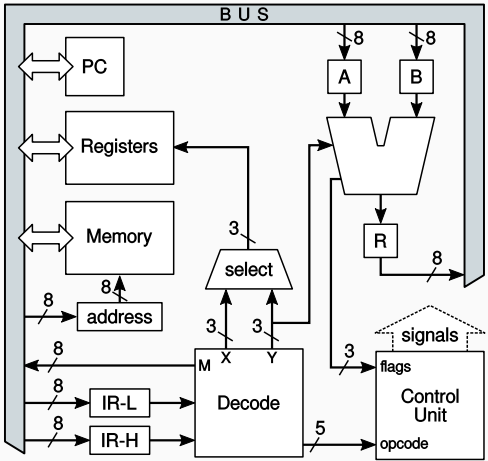
1. Programación: Escribir un algoritmo en **lenguaje ensamblador (o assembly)**. Esto es el **código fuente**.
2. Ensamblado: Un programa llamado **Ensamblador** toma el **código fuente** y lo traduce a un **código máquina**.
   2.1 Resuelve las directivas dirigidas al **Ensamblador**.
   2.2 Calcula el tamaño de cada instrucciones, y resuelve las etiquetas.
   2.3 Traduce las instrucciones a 0s y 1s.
3. Carga: Se le indica al **Ensamblador** una dirección inicial, y éste copia el **código máquina** en la desde esa posición en adelante. Luego, carga esa misma dirección en el **PC (Program Counter)**.
4. Ejecución: El **CPU** da inicio a su **ciclo de ejecución** comenzando por la posición indicada en el **PC**.

Tabla de contenidos

Tabla de contenidos

Organización

`OrgaSmall` es un procesador diseñado e implementado sobre la herramienta *Logisim*. Este cuenta con las siguientes características:



- 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.

Tabla de contenidos

Instrucciones

Instrucción	CodOp	Formato	Acción
`ADD Rx, Ry`	`00001`	A	`Rx ← Rx + Ry`
`SUB Rx, Ry`	`00011`	A	`Rx ← Rx - Ry`
`AND Rx, Ry`	`00100`	A	`Rx ← Rx and Ry`
...	...	.	...
`MOV Rx, Ry`	`01000`	A	`Rx ← Ry`
`STR [M], Rx`	`10000`	D	`Mem[M] ← Rx`
...	...	.	...
...	...	.	...
`JN M`	`10111`	C	Si `flag_N=1` entonces `PC ← M`
...	...	.	...

Ver manual de referencia!!!

Formato de instrucción

Las instrucciones están codificadas en 16 bits. Los primeros 5 bits son el **opcode** de la instrucción, el resto de los bits indican los parámetros. Existen 4 posibles codificaciones de parámetros.

Formato	Codificación
A	00000 XXX YYY-----
B	00000 XXX --------
C	00000 --- MMMMMMMM
D	00000 XXX MMMMMMMM

XXX codifica el número del registro X (**Rx**), vale entre 0 y 7

YYY codifica el número del registro Y (**Ry**) o un inmediato, vale entre 0 y 7

MMMMMMMM Dirección de memoria o valor inmediato, número de 8 bits

Las posiciones indicadas con **-** deben ser cero. Las instrucciones de **opcode**: 9, 10, 11, 12, 13, 14, 15, 28, 29 y 30 son instrucciones reservadas que luego usaremos para expandir la máquina.

Tabla de contenidos

Tabla de contenidos

Ejemplo de ensamblado y carga de un programa

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R2,[et2]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

Ejemplo de ensamblado y carga de un programa

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R2,[et2]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

Tenemos que:

Ejemplo de ensamblado y carga de un programa

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R2,[et2]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

Tenemos que:

- ver cuántas palabras necesita cada instrucción

Ejemplo de ensamblado y carga de un programa

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R2,[et2]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

Tenemos que:

- ver cuántas palabras necesita cada instrucción
- calcular los valores de las etiquetas

Ejemplo de ensamblado y carga de un programa

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R2,[et2]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

Tenemos que:

- ver cuántas palabras necesita cada instrucción
- calcular los valores de las etiquetas

DB *(Define Byte)*: directiva al ensamblador que provoca que en la posición de memoria que le corresponde, aparezca el valor indicado.

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección ocupa

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

main:	LOAD R1,[et1]
	LOAD R1,[et1]
	ADD R1,R2
	STR [et1],R1
	JMP main
et1:	DB 0x07
et2:	DB 0x04

dirección ocupa

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:   LOAD R1,[et1]
        LOAD R1,[et1]
        ADD R1,R2
        STR [et1],R1
        JMP main
et1:    DB 0x07
et2:    DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
main:  LOAD R1,[et1]
       LOAD R1,[et1]
       ADD R1,R2
       STR [et1],R1
       JMP main
et1:   DB 0x07
et2:   DB 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

La etiqueta main corresponde a la dirección de memoria **0x00**.
La etiqueta et1 corresponde a la dirección de memoria **0x0A**.
La etiqueta et2 corresponde a la dirección de memoria **0x0B**.

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

PC	0000

Codifiquemos el siguiente programa y carguémoslo desde la posición de memoria 0x00:

```
LOAD R1,0x0A
LOAD R2,0x0B
ADD R1,R2
STR 0x0A,R1
JMP 0x00
DW 0x07
DW 0x04
```

dirección	ocupa
0x00	dos palabras
0x02	dos palabras
0x04	dos palabras
0x06	dos palabras
0x08	dos palabras
0x0A	una palabra
0x0B	una palabra

Traducidas las etiquetas lo codificamos y cargamos en memoria:

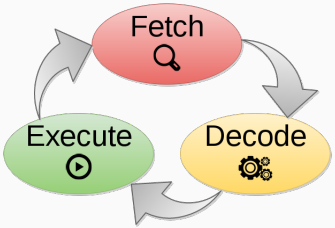
00	01	02	03	04	05	06	07	08	09	10	11
0x88	0x0A	0x89	0x0B	0x08	0x20	0x80	0x0A	0xA0	0x00	0x07	0x04

Y también cargamos la dirección inicial en el **PC**:

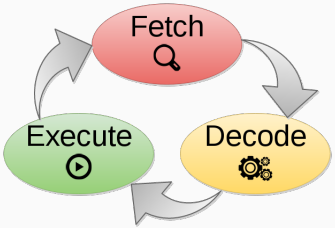
PC	0000

Tabla de contenidos

Ciclo de Instrucción

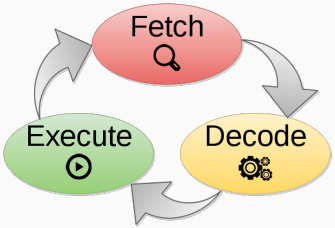


Ciclo de Instrucción



Fetch La **UC (Unidad de Control)** obtiene una instrucción de la posición de a la que apunta el **PC** **y lo incrementa** (Si es necesario: busca más palabras de la instrucción usando el **PC** e incrementándolo cada vez.)

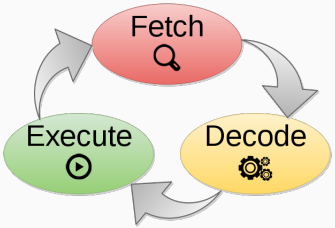
Ciclo de Instrucción



Fetch La **UC (Unidad de Control)** obtiene una instrucción de la posición de a la que apunta el **PC** **y lo incrementa** (Si es necesario: busca más palabras de la instrucción usando el **PC** e incrementándolo cada vez.)

Decode La **UC** decodifica la instrucción.

Ciclo de Instrucción



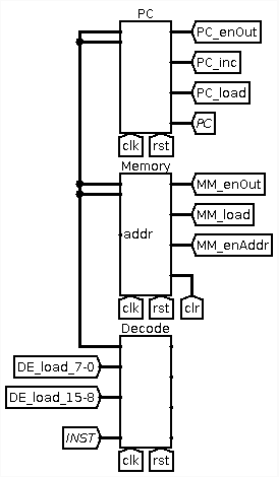
Fetch La **UC (Unidad de Control)** obtiene una instrucción de la posición de a la que apunta el **PC** **y lo incrementa** (Si es necesario: busca más palabras de la instrucción usando el **PC** e incrementándolo cada vez.)

Decode La **UC** decodifica la instrucción.

Execute La **UC** ejecuta la instrucción.

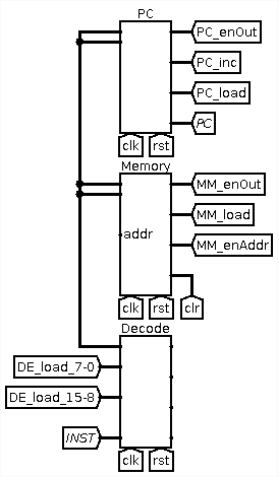
Tabla de contenidos

Datapath del Fetch-Decode



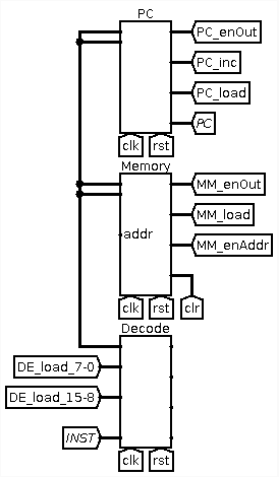
Ahora sí podemos analizar por partes el flujo de datos (datapath) que permite reproducir el mecanismo deseado de **FETCH DECODE EXECUTE**.

Datapath del Fetch-Decode



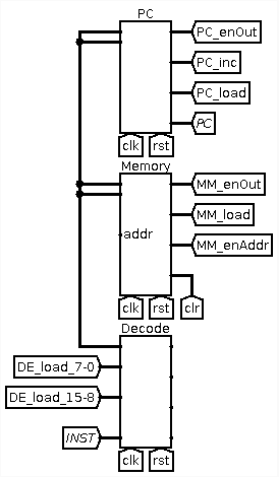
En el ciclo de Fetch-Decode participan el **PC**, la **Memoria** y la unidad de **Decode**. El **PC** no sólo encapsula un registro sino que expone también una señal de control que permite incrementarlo (en una palabra).

Datapath del Fetch-Decode



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.

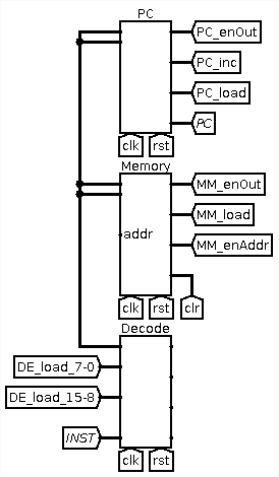
Datapath del Fetch-Decode



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

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.

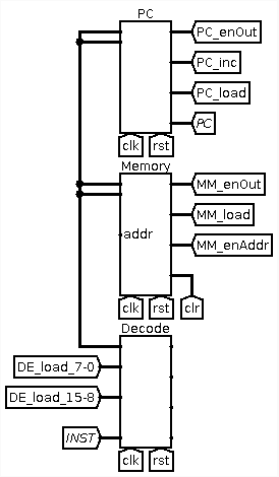
Datapath del Fetch-Decode



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
```

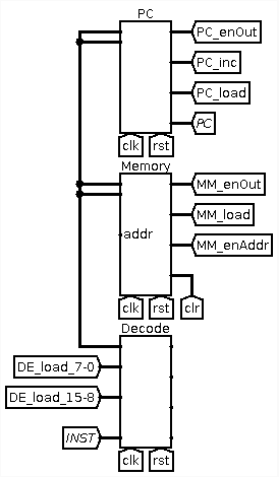
Datapath del Fetch-Decode




```
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, son 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

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 microprograman las etapas de fetch y decode, pensemos:

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 microprograman las etapas de fetch y decode, pensemos:

- ¿El **PC** se actualiza en cada ciclo de reloj?

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 microprograman 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?

Datapath del Fetch-Decode

En RTL:

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

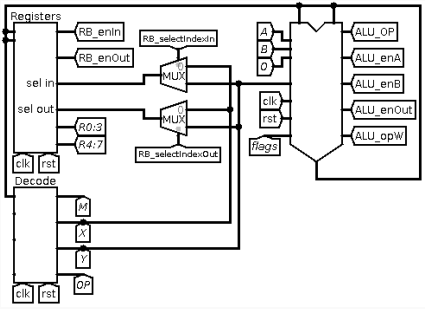
Como señales:

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

Habiendo visto como se microprograman las etapas de fetch y decode, pensemos:

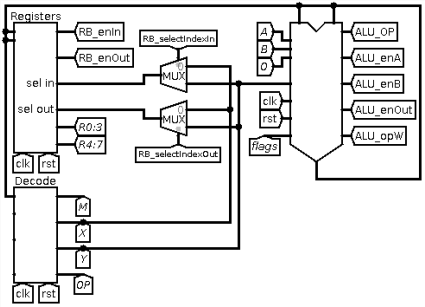
- ¿El **PC** se actualiza en cada ciclo de reloj?
- ¿Cómo sabemos **qué microinstrucción** se ejecuta en cada ciclo de reloj?
- ¿Dónde se almacenan las microinstrucciones?

Datapath del ADD



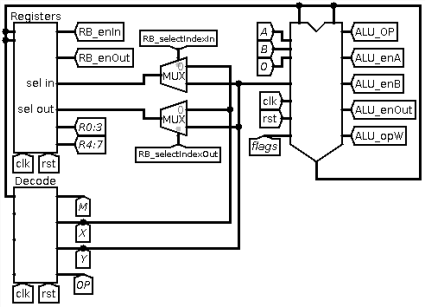
En la ejecución (**EXECUTE**) del **ADD** participan la **ALU**, que resuelve la aritmética, los **Registros** y el **Decode** que indica qué registros participan.

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.

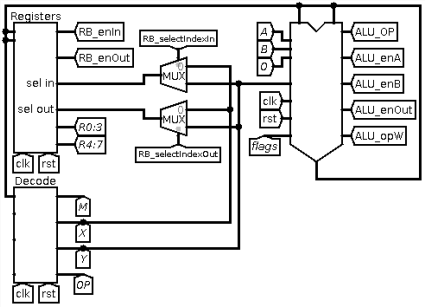
Datapath del ADD



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

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 multiplexor correspondiente).

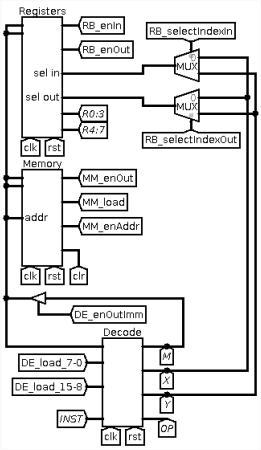
Datapath del ADD



El microprograma asociado es el siguiente:

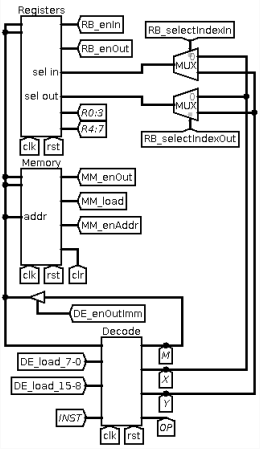
```
RB_enOut  ALU_enA  RB_selectIndexOut=0
RB_enOut  ALU_enB  RB_selectIndexOut=1
ALU_OP=ADD ALU_opW
RB_enIn   ALU_enOut RB_selectIndexIn=0
```

Datapath del STR



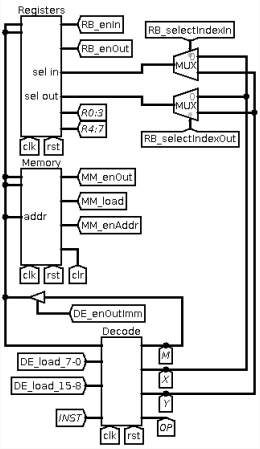
En la ejecución (**EXECUTE**) del **STR** participan el controlador de **memoria**, los **Registros** y el **Decode** que indica el registro fuente y la dirección destino.

Datapath del STR



Veamos qué secuencia de microinstrucciones (RTL) debería suceder para conseguir que se transfiera la dirección de memoria al controlador de memoria, y el valor del registro fuente a la dirección indicada.

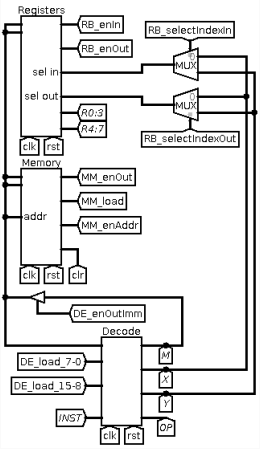
Datapath del STR



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

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 ($Z = 1$)?

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 ($Z = 1$)?

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

Unidad de control

Vamos a 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.

Unidad de control

Vamos a 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**).

Unidad de control

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

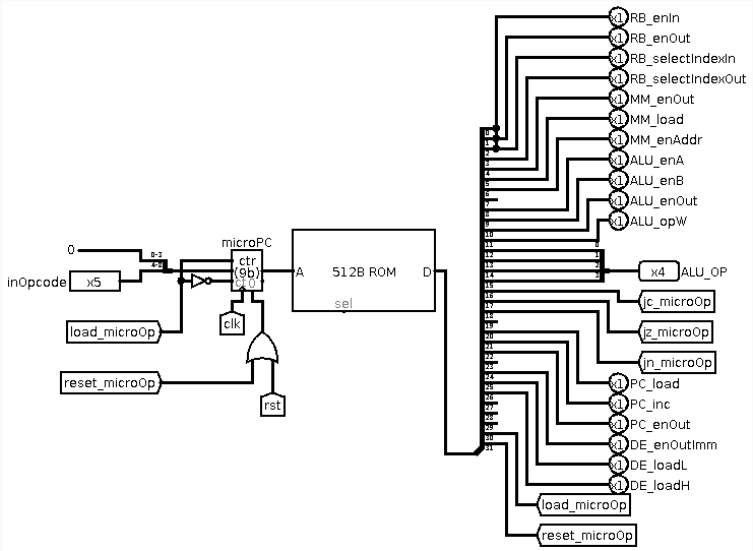
- Que el **micro PC** es un contador, ya que en cada ciclo de reloj su comportamiento por defecto es incrementar en uno la posición de memoria a ser leída.
- Las salidas de control que pueden modificar el comportamiento del **micro PC** (**load_microOp, reset_microOp**).
- Que las salidas (señales de control) están cableadas a los bits de cada palabra en la memoria interna.

Unidad de control

Vamos a 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 (señales de control) 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**.

⇒

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
...
```

⇒

```
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:

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

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 ($Z = 1$)?

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 ($Z = 1$)?

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:

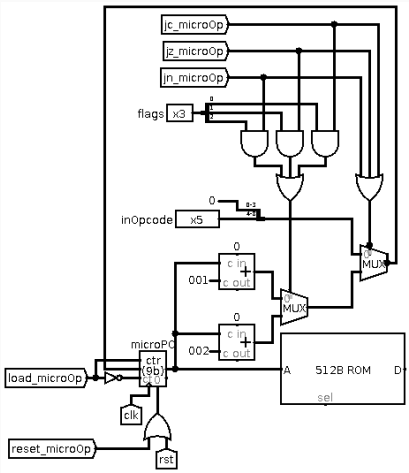
$$\begin{array}{lll} IF & Z = 1 & \\ & PC & := \quad DE_{imm} \end{array}$$

¿Cómo conseguimos que la asignación se haga sólo si se cumple la condición ($Z = 1$)?

Vamos a sobreescribir el valor del **micro PC** sólo si se encuentra habilitada la señal correspondiente de la **ALU**.

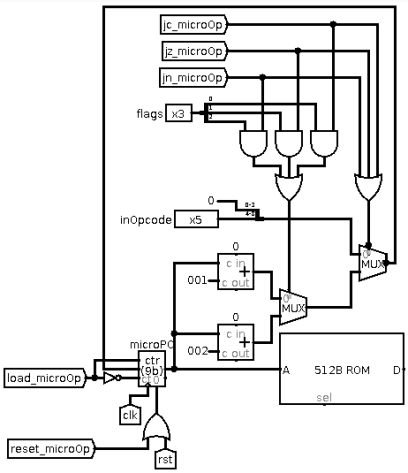
Veamos la propuesta.

Unidad de control



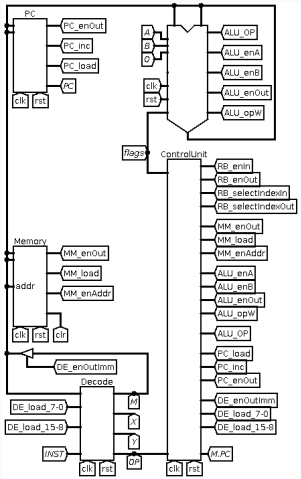
Observemos lo siguiente:

Unidad de control



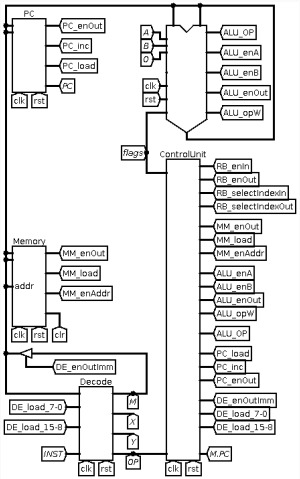
Observemos lo siguiente:
El **micro PC** se puede sobreescribir con la señal **load_microOp** en conjunto con dos selectores de multiplexores: **el de la derecha indicando si se sobreescribe por una nueva instrucción o por los flags; y el de la izquierda indicando si se incrementa el micro PC en 1 (flag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

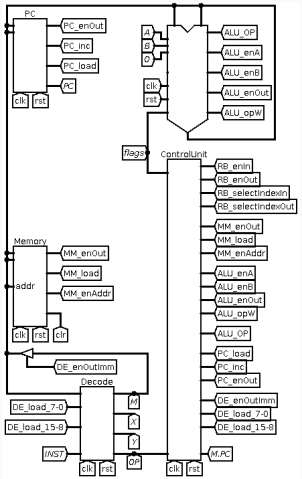
Datapath del JZ



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

$$IF \quad Z = 1$$
$$PC \quad := \quad DE_{imm}$$

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:

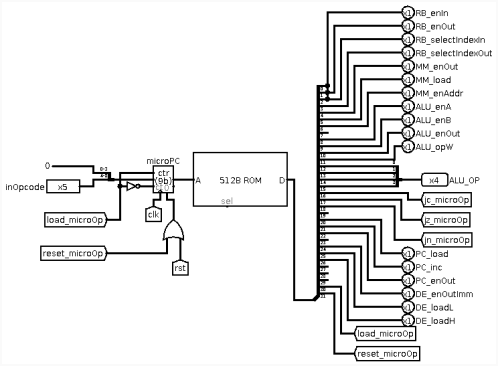
```
00000:
PC_enOut MM_enAddr
MM_enOut DE_loadH PC_inc
PC_enOut MM_enAddr
MM_enOut DE_loadL PC_inc
load_microOp
reset_microOp

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 regresando al **FETCH** luego del **EXECUTE** de cada instrucción.

Unidad de control

Observemos que:

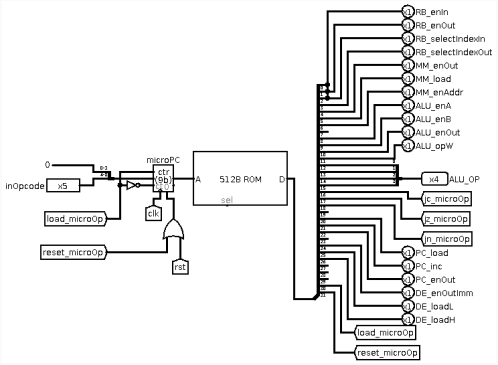
.



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**).



Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`