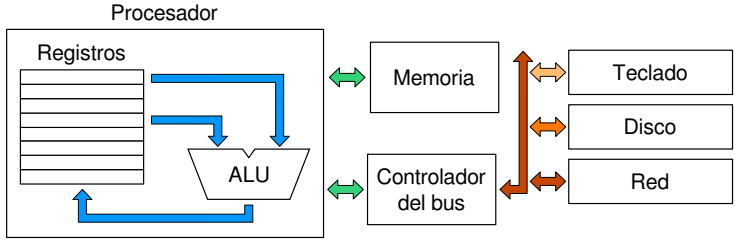
Máquina Simple

David Alejandro González Márquez

Clase disponible en: https://github.com/fokerman/computingSystemsCourse

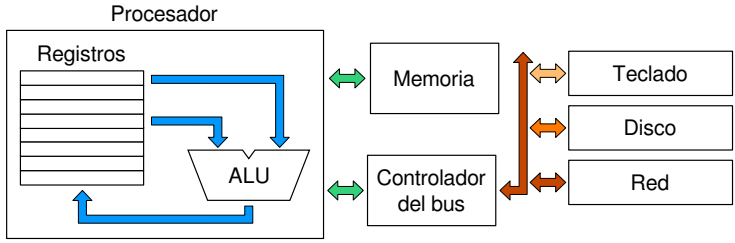
Sistema de Cómputo

Un sistema de cómputo consiste básicamente en *procesadores*, *memorias* y *dispositivos de entrada/salida* interconectados.



Sistema de Cómputo

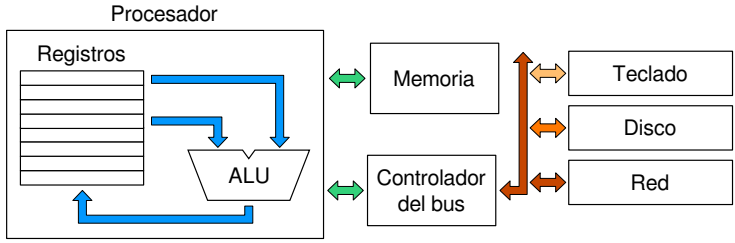
Un sistema de cómputo consiste básicamente en *procesadores*, *memorias* y *dispositivos de entrada/salida* interconectados.



- El **Procesador** o CPU (*Central Processing Unit*) es el encargado de tomar datos de la memoria o de dispositivos de entrada/salida, procesarlos y generar resultados o acciones sobre otros componentes del sistema.

Sistema de Cómputo

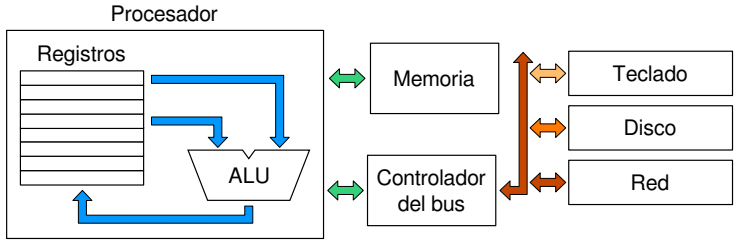
Un sistema de cómputo consiste básicamente en *procesadores*, *memorias* y *dispositivos de entrada/salida* interconectados.



- El **Procesador** o CPU (*Central Processing Unit*) es el encargado de tomar datos de la memoria o de dispositivos de entrada/salida, procesarlos y generar resultados o acciones sobre otros componentes del sistema.
- La **Memoria** se ocupa de almacenar datos para que puedan ser leídos o escritos por el procesador.

Sistema de Cómputo

Un sistema de cómputo consiste básicamente en *procesadores*, *memorias* y *dispositivos de entrada/salida* interconectados.



- El **Procesador** o CPU (*Central Processing Unit*) es el encargado de tomar datos de la memoria o de dispositivos de entrada/salida, procesarlos y generar resultados o acciones sobre otros componentes del sistema.
- La **Memoria** se ocupa de almacenar datos para que puedan ser leídos o escritos por el procesador.
- Los **dispositivos de entrada/salida** nos permiten interactuar con el mundo exterior.

Sistema de Cómputo

Utilizando los circuitos que construimos hasta ahora vamos a armar un procesador.

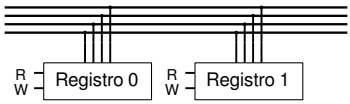
La idea es **conectar componentes** entre sí, como registros y circuitos aritméticos, por medio de cables, a los que llamaremos buses.

Los **buses**, en esta primera aproximación, serán un conjunto ordenado de cables que nos permiten llevar datos desde un componente a otro.

Pero, **¿cómo movemos datos de un componente a otro?**

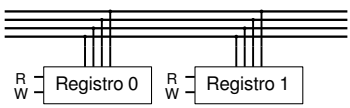
Transferencia entre registros

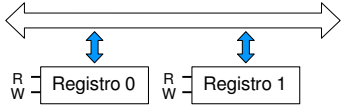
Supongamos dos registros bidireccionales conectados a un bus de 4 bits:



Transferencia entre registros

Supongamos dos registros bidireccionales conectados a un bus de 4 bits:





Este circuito puede ser representado como un **datapath**, donde se ilustra el camino que pueden hacer los datos entre los distintos componentes pasando por un bus.

Transferencia entre registros

Supongamos dos registros bidireccionales conectados a un bus de 4 bits:

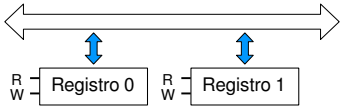
Este circuito puede ser representado como un **datapath**, donde se ilustra el camino que pueden hacer los datos entre los distintos componentes pasando por un bus.

En este ejemplo, cada registro cuenta con dos señales:

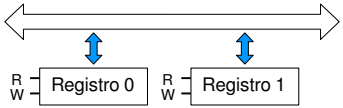
- **R**ead: Si está en 1 presenta en el bus el dato almacenado en el registro.
- **W**rite: Si está en 1 guarda en el registro el dato que esté en el bus.

Modificando el estado de las señales **R** y **W** de cada registro buscamos copiar los datos entre ambos registros.

Transferencia entre registros



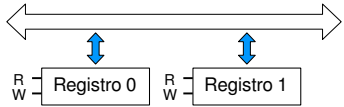
Transferencia entre registros



Copiar el dato almacenado en el Registro 0 al Registro 1:

clock	reg0.R	reg0.W	reg1.R	reg1.W
⌝⌞	1	0	0	1

Transferencia entre registros



Copiar el dato almacenado en el Registro 0 al Registro 1:

clock	reg0.R	reg0.W	reg1.R	reg1.W
⊓	1	0	0	1

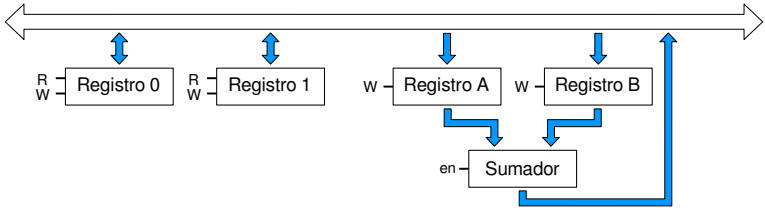
Copiar el dato almacenado en el Registro 1 al Registro 0:

clock	reg0.R	reg0.W	reg1.R	reg1.W
⊓	0	1	1	0

Es decir, **modificando** las señales de los componentes podemos hacer **operaciones**.

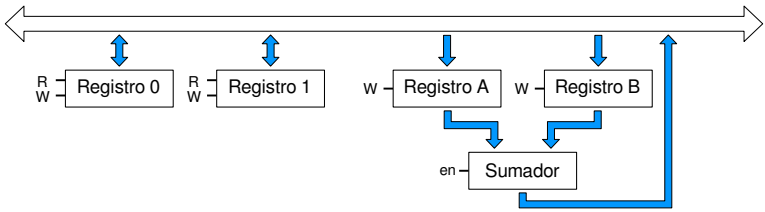
Operaciones con registros

Supongamos el siguiente *datapath*:



Operaciones con registros

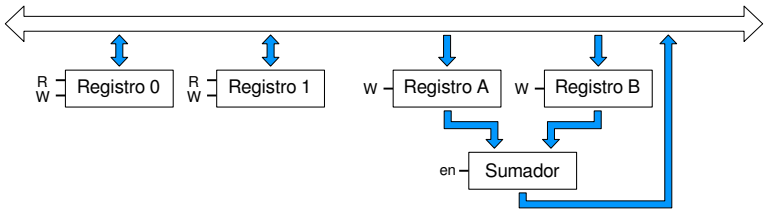
Supongamos el siguiente *datapath*:



Agregamos dos registros denominados A y B, con solo una señal **W**.
Los conectamos tal que el dato almacenado por ambos registros llegue a un sumador.
La tarea del sumador es sumar los datos de ambos registros.
Además, presenta el resultado en el bus si la señal de **en**able está en 1.

Operaciones con registros

Supongamos el siguiente *datapath*:

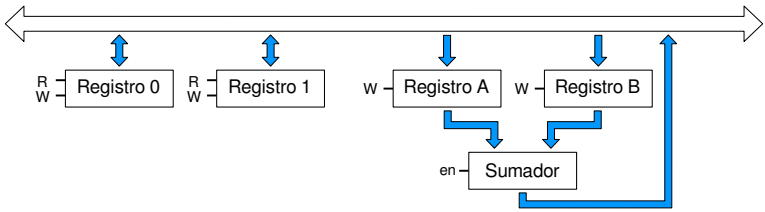


Agregamos dos registros denominados A y B, con solo una señal **W**.
Los conectamos tal que el dato almacenado por ambos registros llegue a un sumador.
La tarea del sumador es sumar los datos de ambos registros.
Además, presenta el resultado en el bus si la señal de **en**able está en 1.

¿Cuál es la secuencia de operaciones para sumar los registros 0 y 1,
y guardar el resultado de la operación en el registro 0?

Operaciones con registros

Supongamos el siguiente *datapath*:

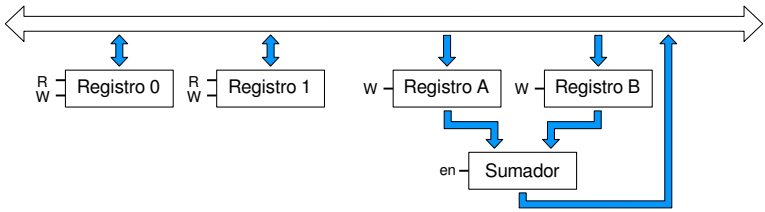


Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0:

clock	reg0.R	reg0.W	reg1.R	reg1.W	regA.W	regB.W	sum.en

Operaciones con registros

Supongamos el siguiente *datapath*:

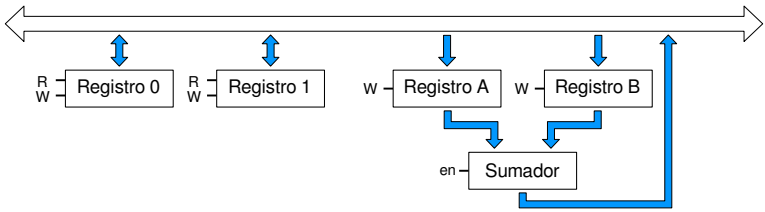


Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0:

clock	reg0.R	reg0.W	reg1.R	reg1.W	regA.W	regB.W	sum.en
⌐└	1	0	0	0	1	0	0

Operaciones con registros

Supongamos el siguiente *datapath*:

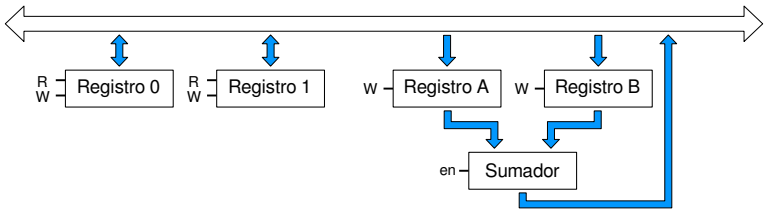


Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0:

clock	reg0.R	reg0.W	reg1.R	reg1.W	regA.W	regB.W	sum.en
⌐	1	0	0	0	1	0	0
⌐	0	0	1	0	0	1	0

Operaciones con registros

Supongamos el siguiente *datapath*:



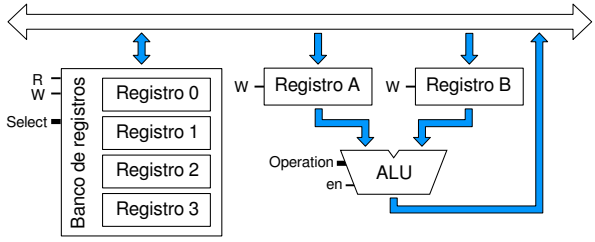
Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0:

clock	reg0.R	reg0.W	reg1.R	reg1.W	regA.W	regB.W	sum.en
⌐	1	0	0	0	1	0	0
⌐	0	0	1	0	0	1	0
⌐	0	1	0	0	0	0	1

¿Cómo hacer si queremos agregar más registros y hacer otras operaciones?

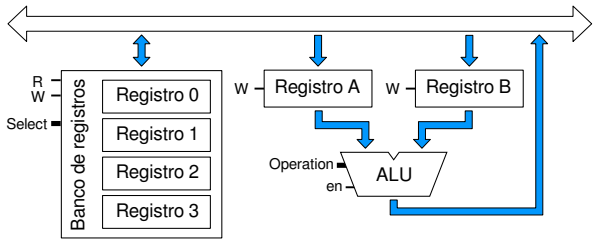
Operaciones con registros

Supongamos el siguiente *datapath*:



Operaciones con registros

Supongamos el siguiente *datapath*:

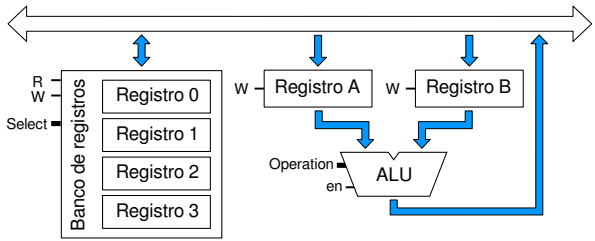


Para tener múltiples registros usamos un *banco de registros*.
Cuenta una señal que nos permite **seleccionar** el registro con el que buscamos operar.

Al sumador lo remplazamos por una *ALU*, es decir, un circuito combinatorio que nos permite realizar distintas operaciones, tanto lógicas como aritméticas.

Operaciones con registros

Supongamos el siguiente *datapath*:



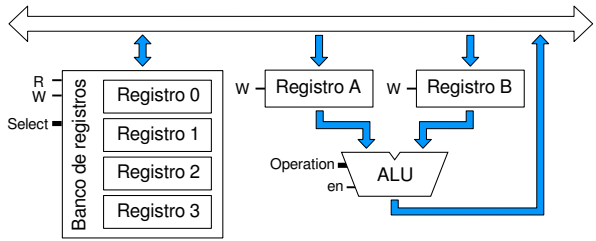
Para tener múltiples registros usamos un *banco de registros*.
Cuenta una señal que nos permite **seleccionar** el registro con el que buscamos operar.

Al sumador lo remplazamos por una *ALU*, es decir, un circuito combinatorio que nos permite realizar distintas operaciones, tanto lógicas como aritméticas.

¿Qué secuencia de señales necesitamos para realizar
la operación anterior con este nuevo datapath?

Operaciones con registros

Supongamos el siguiente *datapath*:

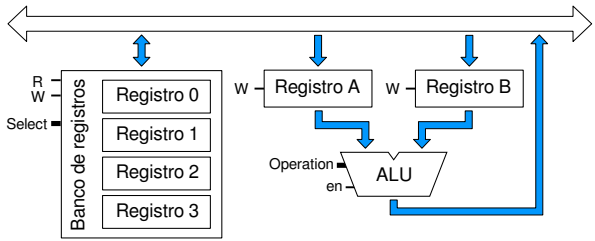


Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0.s

	regBank			regA	regB	ALU	
clock	select	R	W	W	W	Operation	en

Operaciones con registros

Supongamos el siguiente *datapath*:

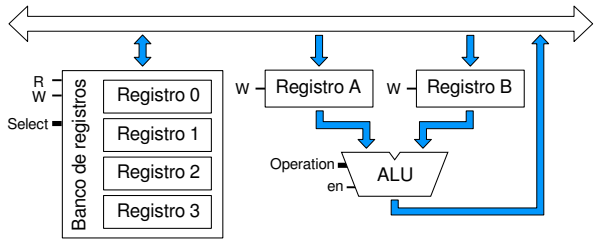


Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0.s

	regBank			regA	regB	ALU	
clock	select	R	W	W	W	Operation	en
⌝⌞	00	1	0	1	0		0

Operaciones con registros

Supongamos el siguiente *datapath*:

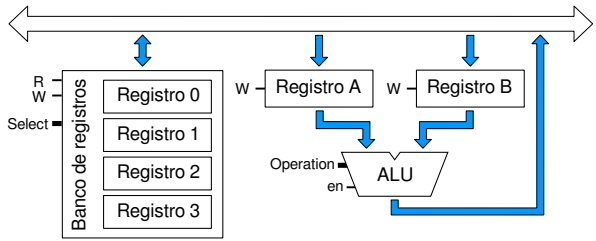


Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0.s

	regBank			regA	regB	ALU	
clock	select	R	W	W	W	Operation	en
⌐	00	1	0	1	0		0
⌐	01	1	0	0	1		0

Operaciones con registros

Supongamos el siguiente *datapath*:



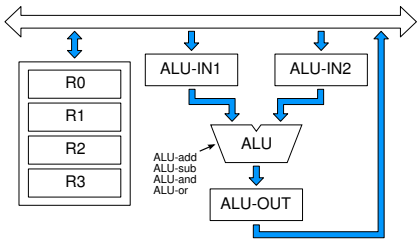
Sumar los datos almacenados en el Registro 0 y en el Registro 1, y guardar el resultado en el Registro 0.s

	regBank			regA	regB	ALU	
clock	select	R	W	W	W	Operation	en
⌝⌞	00	1	0	1	0		0
⌝⌞	01	1	0	0	1		0
⌝⌞	00	0	1	0	0	+	1

Microarquitectura: Lenguaje y Notación

Para simplificar el estudio de los sistemas de cómputo, vamos a utilizar un lenguaje para expresar la transferencia entre registros.

- Definimos **componentes** como circuitos con entradas, salidas y señales.
- Las señales son entradas que modifican el **comportamiento** de los circuitos.
- Las señales se activan según como indique el **microprograma**.
- Estos serán listas de asignaciones entre registros y activación de señales que se realizan por cada ciclo de **clock**.



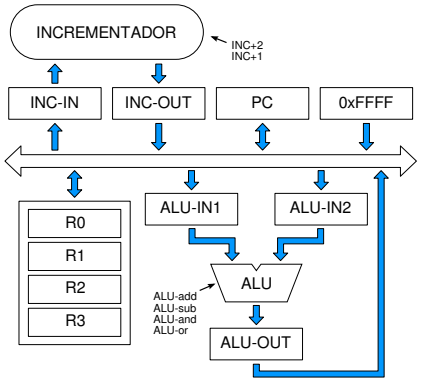
Ejemplo: Sumar el valor en el registro R0 con R1 y guardar el resultado en R0.

Microprograma

```
ALU-IN1 := R0
ALU-IN2 := R1
ALU-add
R0 := ALU-OUT
```

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*:



Podemos además tener **otros componentes** con sus propias señales como `Incrementador` con las señales `INC+2` e `INC+1` y sus propios registros.

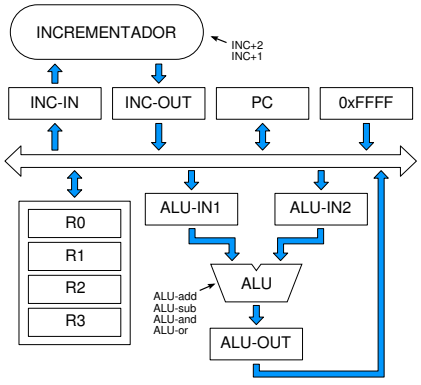
Pueden existir registros de **propósito específico** como PC o registros que proveen una constante como `0xFFFF`.

Podemos mover datos si hay un **camino directo** entre ellos.

	¿vale?	
`R1 := R0`		
`R0 := ALU-OUT`		
`R2 := 0xFFFF`		
`ALU-OUT := R0`		
`R0 := ALU-IN1`		
`R1 := 0x35`		

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*:



Podemos además tener **otros componentes** con sus propias señales como `Incrementador` con las señales `INC+2` e `INC+1` y sus propios registros.

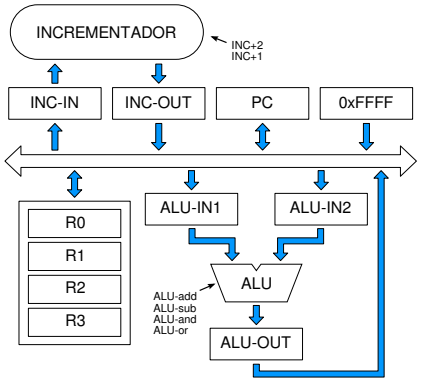
Pueden existir registros de **propósito específico** como PC o registros que proveen una constante como `0xFFFF`.

Podemos mover datos si hay un **camino directo** entre ellos.

	¿vale?	
`R1 := R0`	✓	Copia `R0` en `R1`
`R0 := ALU-OUT`		
`R2 := 0xFFFF`		
`ALU-OUT := R0`		
`R0 := ALU-IN1`		
`R1 := 0x35`		

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*:



Podemos además tener **otros componentes** con sus propias señales como `Incrementador` con las señales `INC+2` e `INC+1` y sus propios registros.

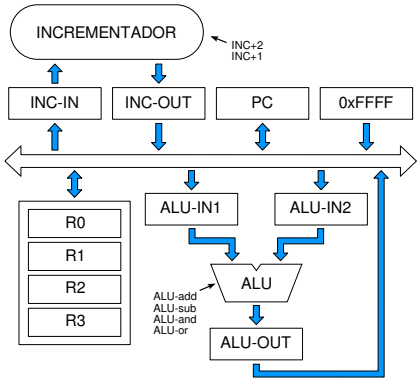
Pueden existir registros de **propósito específico** como PC o registros que proveen una constante como `0xFFFF`.

Podemos mover datos si hay un **camino directo** entre ellos.

	¿vale?	
`R1 := R0`	✓	Copia `R0` en `R1`
`R0 := ALU-OUT`	✓	Copia `R0` en `ALU-OUT`
`R2 := 0xFFFF`		
`ALU-OUT := R0`		
`R0 := ALU-IN1`		
`R1 := 0x35`		

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*:



Podemos además tener **otros componentes** con sus propias señales como `Incrementador` con las señales `INC+2` e `INC+1` y sus propios registros.

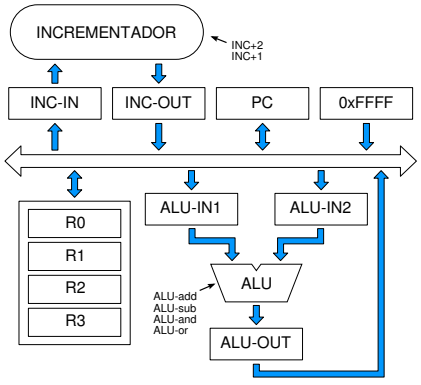
Pueden existir registros de **propósito específico** como PC o registros que proveen una constante como `0xFFFF`.

Podemos mover datos si hay un **camino directo** entre ellos.

	¿vale?	
`R1 := R0`	✓	Copia `R0` en `R1`
`R0 := ALU-OUT`	✓	Copia `R0` en `ALU-OUT`
`R2 := 0xFFFF`	✓	Copia la constante `0xFFFF` a `R2`
`ALU-OUT := R0`		
`R0 := ALU-IN1`		
`R1 := 0x35`		

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*:



Podemos además tener **otros componentes** con sus propias señales como `Incrementador` con las señales `INC+2` e `INC+1` y sus propios registros.

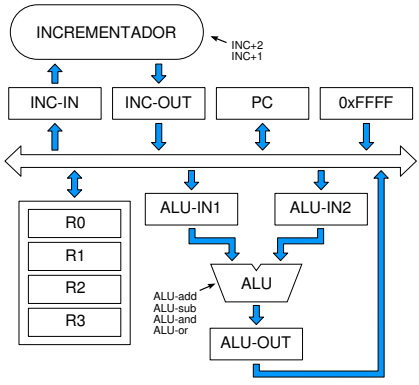
Pueden existir registros de **propósito específico** como PC o registros que proveen una constante como `0xFFFF`.

Podemos mover datos si hay un **camino directo** entre ellos.

	¿vale?	
`R1 := R0`	✓	Copia `R0` en `R1`
`R0 := ALU-OUT`	✓	Copia `R0` en `ALU-OUT`
`R2 := 0xFFFF`	✓	Copia la constante `0xFFFF` a `R2`
`ALU-OUT := R0`	✗	`ALU-OUT` es de solo lectura
`R0 := ALU-IN1`		
`R1 := 0x35`		

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*:



Podemos además tener **otros componentes** con sus propias señales como `Incrementador` con las señales `INC+2` e `INC+1` y sus propios registros.

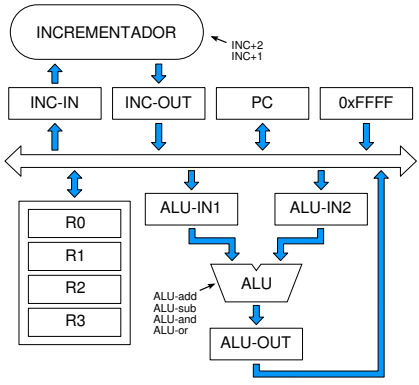
Pueden existir registros de **propósito específico** como PC o registros que proveen una constante como `0xFFFF`.

Podemos mover datos si hay un **camino directo** entre ellos.

	¿vale?	
`R1 := R0`	✓	Copia `R0` en `R1`
`R0 := ALU-OUT`	✓	Copia `R0` en `ALU-OUT`
`R2 := 0xFFFF`	✓	Copia la constante `0xFFFF` a `R2`
`ALU-OUT := R0`	×	`ALU-OUT` es de solo lectura
`R0 := ALU-IN1`	×	`ALU-IN1` es de solo escritura
`R1 := 0x35`		

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*:



Podemos además tener **otros componentes** con sus propias señales como `Incrementador` con las señales `INC+2` e `INC+1` y sus propios registros.

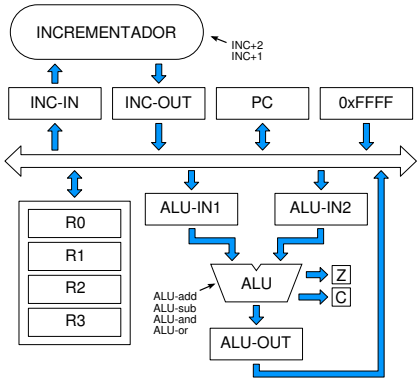
Pueden existir registros de **propósito específico** como PC o registros que proveen una constante como `0xFFFF`.

Podemos mover datos si hay un **camino directo** entre ellos.

	¿vale?	
`R1 := R0`	✓	Copia `R0` en `R1`
`R0 := ALU-OUT`	✓	Copia `R0` en `ALU-OUT`
`R2 := 0xFFFF`	✓	Copia la constante `0xFFFF` a `R2`
`ALU-OUT := R0`	×	`ALU-OUT` es de solo lectura
`R0 := ALU-IN1`	×	`ALU-IN1` es de solo escritura
`R1 := 0x35`	×	No se dispone de esa constante

Microarquitectura: Lenguaje y Notación

Supongamos el siguiente *datapath*.



Todos los registros de 16 bits, excepto Z y C de 1 bit.

Todos los registros tienen un **tamaño**, al igual que los buses. No es posible asignar registros de distinto tamaño.

Sin embargo, podemos usar un solo bit de un registro para tomar decisiones.

Ejemplos:

Microprograma 1

```
if ALU_OUT[15] = 1
    INC-IN := PC
    INC+2
    PC := INC-OUT
else
    INC-IN := PC
    INC+1
    PC := INC-OUT
```

Microprograma 2

```
if Z = 1
    INC-IN := PC
    INC+2
    PC := INC-OUT
else
    INC-IN := PC
    INC+1
    PC := INC-OUT
```

Para tomar múltiples bits es posible usar la notación de rangos, por ejemplo: `R0[3:0]`

Unidad de Control

Existe un componente más dentro del **datapath** encargado de modificar el valor de las señales de los distintos componentes para realizar alguna acción.

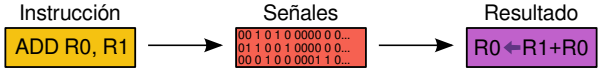
Este componente se denomina **Unidad de Control**.

Unidad de Control

Existe un componente más dentro del **datapath** encargado de modificar el valor de las señales de los distintos componentes para realizar alguna acción.

Este componente se denomina **Unidad de Control**.

Funciona como un **secuenciador**: una vez identificada la tarea a realizar, genera una secuencia de señales (*microinstrucciones*) para realizar la acción correspondiente.
Esta secuencia de señales la podemos suponer fija y almacenada dentro de una memoria.



A cada **instrucción** le corresponde una secuencia de señales.
El proceso para convertir una instrucción en una secuencia de señales se denomina **decodificación**.

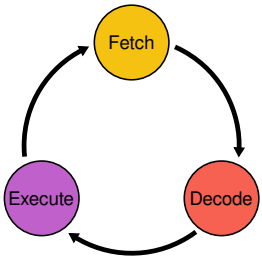
Ciclo de Instrucción

Los programas son secuencias de **instrucciones** que los procesadores pueden interpretar y ejecutar. Pero, ¿cómo hacen esta tarea?

Ciclo de Instrucción

Los programas son secuencias de **instrucciones** que los procesadores pueden interpretar y ejecutar. Pero, ¿cómo hacen esta tarea?

Realizan tres pasos todo el tiempo por cada instrucción a ejecutar.

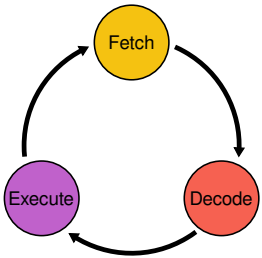


- **Fetch**: Lectura desde la memoria de la instrucción a ejecutar. Esta puede ser de tamaño fijo o de tamaño variable.

Ciclo de Instrucción

Los programas son secuencias de **instrucciones** que los procesadores pueden interpretar y ejecutar. Pero, ¿cómo hacen esta tarea?

Realizan tres pasos todo el tiempo por cada instrucción a ejecutar.

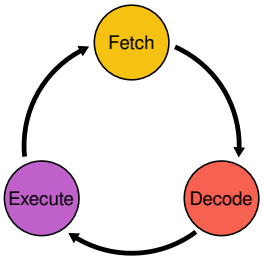


- **Fetch**: Lectura desde la memoria de la instrucción a ejecutar. Esta puede ser de tamaño fijo o de tamaño variable.
- **Decode**: Se interpreta la instrucción, identificando la acción que debe ser realizada (opcode) y los operandos. Puede ser necesario hacer múltiples accesos a memoria para cargar los operandos a registros.

Ciclo de Instrucción

Los programas son secuencias de **instrucciones** que los procesadores pueden interpretar y ejecutar. Pero, ¿cómo hacen esta tarea?

Realizan tres pasos todo el tiempo por cada instrucción a ejecutar.

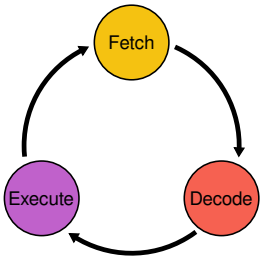


- **Fetch**: Lectura desde la memoria de la instrucción a ejecutar. Esta puede ser de tamaño fijo o de tamaño variable.
- **Decode**: Se interpreta la instrucción, identificando la acción que debe ser realizada (opcode) y los operandos. Puede ser necesario hacer múltiples accesos a memoria para cargar los operandos a registros.
- **Execute**: Se ejecuta la operación. Puede ser una operación aritmética o lógica, un acceso a memoria o movimiento entre registros, incluso la configuración de algún registro dentro del procesador.

Ciclo de Instrucción

Los programas son secuencias de **instrucciones** que los procesadores pueden interpretar y ejecutar. Pero, ¿cómo hacen esta tarea?

Realizan tres pasos todo el tiempo por cada instrucción a ejecutar.



- **Fetch**: Lectura desde la memoria de la instrucción a ejecutar. Esta puede ser de tamaño fijo o de tamaño variable.
- **Decode**: Se interpreta la instrucción, identificando la acción que debe ser realizada (opcode) y los operandos. Puede ser necesario hacer múltiples accesos a memoria para cargar los operandos a registros.
- **Execute**: Se ejecuta la operación. Puede ser una operación aritmética o lógica, un acceso a memoria o movimiento entre registros, incluso la configuración de algún registro dentro del procesador.

Pero, **¿cuál es la próxima instrucción a ejecutar?**

Program Counter

Los procesadores poseen un registro especial denominado **Program Counter** o **PC**.

Este registro guarda una dirección de **memoria**.

Program Counter

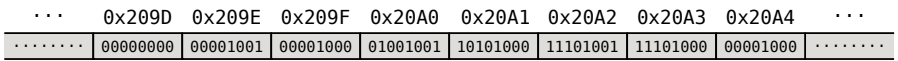
Los procesadores poseen un registro especial denominado **Program Counter** o **PC**.

Este registro guarda una dirección de **memoria**.

Memoria

La memoria es el componente que nos permite almacenar programas y datos. Está organizada para ser accedida mediante direcciones. Una dirección de memoria identifica una unidad direccionable de un tamaño fijo. Por ejemplo, 1 byte.

Ejemplo:



Memoria direccionable a byte, es decir, cada dirección de memoria apunta a un byte.

Program Counter

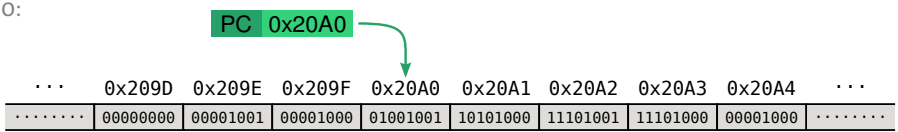
Los procesadores poseen un registro especial denominado **Program Counter** o **PC**.

Este registro guarda una dirección de **memoria**.

Memoria

La memoria es el componente que nos permite almacenar programas y datos. Está organizada para ser accedida mediante direcciones. Una dirección de memoria identifica una unidad direccionable de un tamaño fijo. Por ejemplo, 1 byte.

Ejemplo:



Memoria direccionable a byte, es decir, cada dirección de memoria apunta a un byte.

El PC apunta a la dirección `0x20A0`, donde se encuentra almacenado el valor `01001001`b.

Program Counter

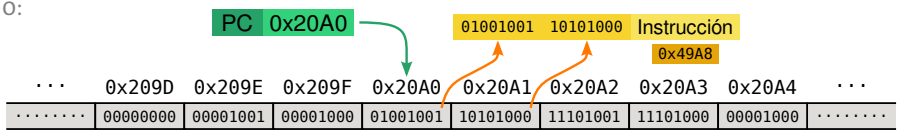
Los procesadores poseen un registro especial denominado **Program Counter** o **PC**.

Este registro guarda una dirección de **memoria**.

Memoria

La memoria es el componente que nos permite almacenar programas y datos. Está organizada para ser accedida mediante direcciones. Una dirección de memoria identifica una unidad direccionable de un tamaño fijo. Por ejemplo, 1 byte.

Ejemplo:



Memoria direccionable a byte, es decir, cada dirección de memoria apunta a un byte.

El PC apunta a la dirección `0x20A0`, donde se encuentra almacenado el valor `01001001`b.
Si cada instrucción ocupa 2 bytes,

Program Counter

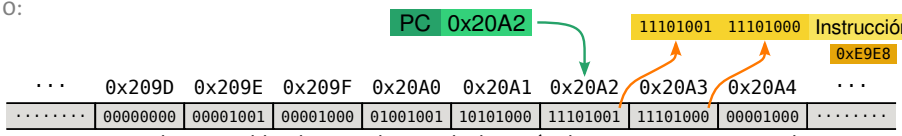
Los procesadores poseen un registro especial denominado **Program Counter** o **PC**.

Este registro guarda una dirección de **memoria**.

Memoria

La memoria es el componente que nos permite almacenar programas y datos. Está organizada para ser accedida mediante direcciones. Una dirección de memoria identifica una unidad direccionable de un tamaño fijo. Por ejemplo, 1 byte.

Ejemplo:



Memoria direccionable a byte, es decir, cada dirección de memoria apunta a un byte.

El PC apunta a la dirección `0x20A0`, donde se encuentra almacenado el valor `01001001`b.
Si cada instrucción ocupa 2 bytes, la próxima instrucción a ejecutar es `0x20A2`.

Program Counter

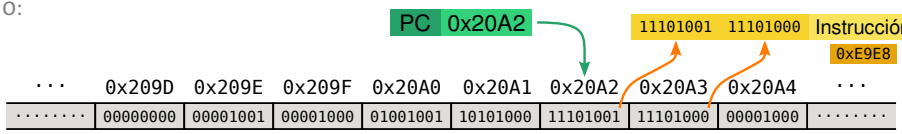
Los procesadores poseen un registro especial denominado **Program Counter** o **PC**.

Este registro guarda una dirección de **memoria**.

Memoria

La memoria es el componente que nos permite almacenar programas y datos. Está organizada para ser accedida mediante direcciones. Una dirección de memoria identifica una unidad direccionable de un tamaño fijo. Por ejemplo, 1 byte.

Ejemplo:

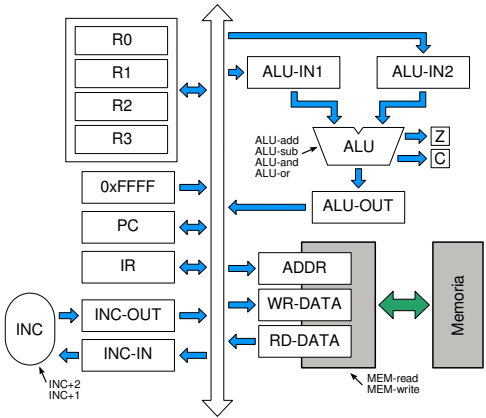


Memoria direccionable a byte, es decir, cada dirección de memoria apunta a un byte.

El PC apunta a la dirección `0x20A0`, donde se encuentra almacenado el valor `01001001`b.
Si cada instrucción ocupa 2 bytes, la próxima instrucción a ejecutar es `0x20A2`.
Y ahora, **¿cómo leemos instrucciones?**

Componente de memoria

Supongamos el siguiente *datapath*.

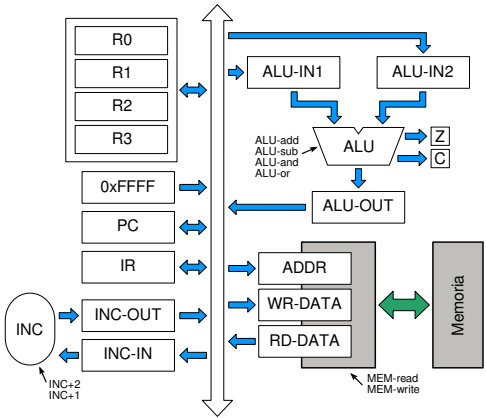


Todos los registros de 16 bits, excepto Z y C de 1 bit.

La **interfaz** con la memoria la vamos a modelar como un dispositivo más, que se conecta mediante un **bus externo** a la memoria.

Componente de memoria

Supongamos el siguiente *datapath*.



Todos los registros de 16 bits, excepto Z y C de 1 bit.

La **interfaz** con la memoria la vamos a modelar como un dispositivo más, que se conecta mediante un **bus externo** a la memoria.

Para acceder a la memoria tenemos tres registros:

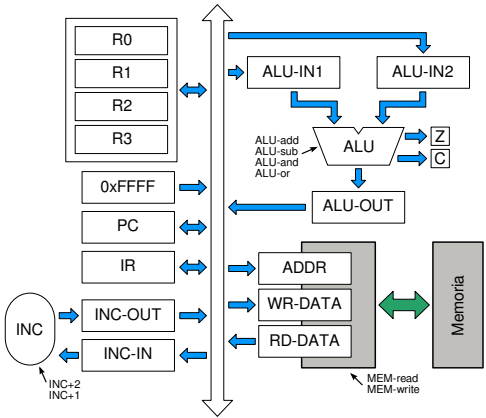
ADDR: Contiene la dirección a leer o escribir.

WR-DATA: Contiene el dato a escribir en memoria.

RD-DATA: Contiene el dato leído de memoria.

Componente de memoria

Supongamos el siguiente *datapath*.



Todos los registros de 16 bits, excepto Z y C de 1 bit.

La **interfaz** con la memoria la vamos a modelar como un dispositivo más, que se conecta mediante un **bus externo** a la memoria.

Para acceder a la memoria tenemos tres registros:

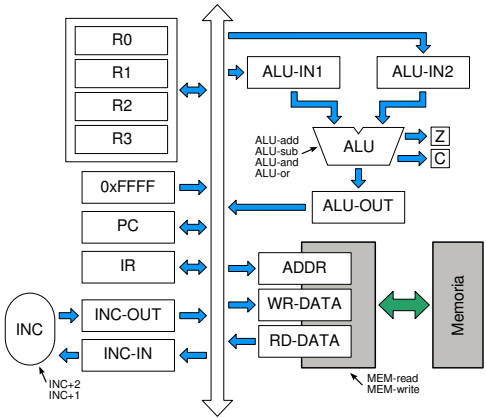
ADDR: Contiene la dirección a leer o escribir.
WR-DATA: Contiene el dato a escribir en memoria.
RD-DATA: Contiene el dato leído de memoria.

Y contamos con dos señales.

MEM-read: Lee un dato de la dirección `ADDR` y copia su contenido en `RD-DATA`.
MEM-write: Escribe el dato `WR-DATA` en la dirección `ADDR`.

Componente de memoria

Supongamos el siguiente *datapath*.



Todos los registros de 16 bits, excepto Z y C de 1 bit.

La **interfaz** con la memoria la vamos a modelar como un dispositivo más, que se conecta mediante un **bus externo** a la memoria.

Para acceder a la memoria tenemos tres registros:

ADDR: Contiene la dirección a leer o escribir.
WR-DATA: Contiene el dato a escribir en memoria.
RD-DATA: Contiene el dato leído de memoria.

Y contamos con dos señales.

MEM-read: Lee un dato de la dirección ADDR y copia su contenido en RD-DATA.
MEM-write: Escribe el dato WR-DATA en la dirección ADDR.

Ejemplos:

Microprograma 1

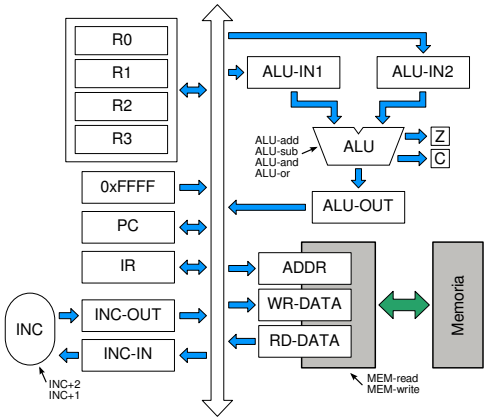
```
ADDR := PC
MEM-read
IR := RD-DATA
```

Microprograma 2

```
ADDR := R0
WR-DATA := R1
MEM-write
```

Componente de memoria

Supongamos el siguiente *datapath*.



Todos los registros de 16 bits, excepto Z y C de 1 bit.

La **interfaz** con la memoria la vamos a modelar como un dispositivo más, que se conecta mediante un **bus externo** a la memoria.

Para acceder a la memoria tenemos tres registros:

ADDR: Contiene la dirección a leer o escribir.
WR-DATA: Contiene el dato a escribir en memoria.
RD-DATA: Contiene el dato leído de memoria.

Y contamos con dos señales.

MEM-read: Lee un dato de la dirección ADDR y copia su contenido en RD-DATA.
MEM-write: Escribe el dato WR-DATA en la dirección ADDR.

Ejemplos:

Microprograma 1

```
ADDR := PC
MEM-read
IR := RD-DATA
```

Microprograma 2

```
ADDR := R0
WR-DATA := R1
MEM-write
```

Y ahora, **¿cómo escribimos instrucciones?**

Lenguaje ensamblador

Llamamos **lenguaje de máquina** al conjunto de instrucciones que el procesador puede interpretar y ejecutar.
Llamamos **lenguaje ensamblador** a las instrucciones del lenguaje de máquina escritas como texto.

Lenguaje ensamblador

Llamamos **lenguaje de máquina** al conjunto de instrucciones que el procesador puede interpretar y ejecutar.
Llamamos **lenguaje ensamblador** a las instrucciones del lenguaje de máquina escritas como texto.

Las instrucciones en lenguaje ensamblador se deben traducir a instrucciones en lenguaje de máquina. La aplicación encargada de hacer esta traducción se denomina **ensamblador**.

holamundo.asm

```
section .data
  msg: DB 'Hola Mundo', 10
  largo EQU $ - msg

  global _start
section .text
  _start:
    mov rax, 4      ; funcion 4
    mov rbx, 1      ; stdout
    mov rcx, msg    ; mensaje
    mov rdx, largo ; longitud
    int 0x80
    mov rax, 1      ; funcion 1
    mov rbx, 0      ; codigo
    int 0x80
```

ensamblador

holamundo

```
holamundo:     file format elf64-x86-64


Disassembly of section .text:

0000000000401000 <_start>:
  401000: b8 04 00 00 00         mov    $0x4,%eax
  401005: bb 01 00 00 00         mov    $0x1,%ebx
  40100a: 48 b9 00 20 40 00 00  movabs $0x402000,%rcx
  401011: 00 00 00
  401014: ba 0b 00 00 00         mov    $0xb,%edx
  401019: cd 80                  int    $0x80
  40101b: b8 01 00 00 00         mov    $0x1,%eax
  401020: bb 00 00 00 00         mov    $0x0,%ebx
  401025: cd 80                  int    $0x80
```

Organización vs Arquitectura

Arquitectura

Organización

Organización vs Arquitectura

Arquitectura

Componentes **visibles** al programador.

- Identificación de registros.
- Conjunto de instrucciones (*Instruction Set Architecture*).
- Modos de acceso a memoria.
- Mecanismos de entrada/salida.

Organización

Organización vs Arquitectura

Arquitectura

Componentes **visibles** al programador.

- Identificación de registros.
- Conjunto de instrucciones (*Instruction Set Architecture*).
- Modos de acceso a memoria.
- Mecanismos de entrada/salida.

Organización

Implementación de una arquitectura.

- Señales de la unidad de control.
- Implementación de microinstrucciones o hardware específico.
- Cantidad y uso de unidades funcionales.
- Tecnología de integración.

Organización vs Arquitectura

Arquitectura

Componentes **visibles** al programador.

- Identificación de registros.
- Conjunto de instrucciones (*Instruction Set Architecture*).
- Modos de acceso a memoria.
- Mecanismos de entrada/salida.

Organización

Implementación de una arquitectura.

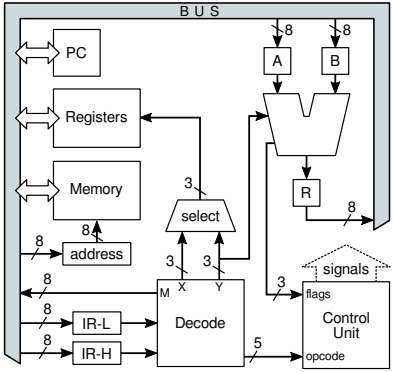
- Señales de la unidad de control.
- Implementación de microinstrucciones o hardware específico.
- Cantidad y uso de unidades funcionales.
- Tecnología de integración.

Hasta el momento, estudiamos la **organización** de un sistema de computo.
Ahora vamos a estudiar su **arquitectura**.

Arquitectura OrgaSmall

Para ilustrar el funcionamiento de un sistema de computo, vamos a presentar la arquitectura *OrgaSmall*.

Esta arquitectura fue diseñada con fines didácticos y su objetivo es poner de manifiesto todos los conceptos básicos de la organización de un procesador con una arquitectura simple.



- 8 registros de propósito general, R0 a R7.
- 1 registro de propósito específico PC.
- Unidad direccionable de 8 bits (1 byte) e instrucciones de 16 bits (2 bytes).
- Memoria de 256 bytes.
- Bus de 8 bits.
- Diseño microprogramado.

Bibliografía

- Tanenbaum, “Organización de Computadoras. Un Enfoque Estructurado”, 4ta Edición, 2000.
  - **Capítulo 2 - Organización de los Sistemas de Computadora** - Páginas 39-45
  - **Capítulo 4 - El nivel de Microarquitectura** - Páginas 203-215

- Null, “Essentials of Computer Organization and Architecture”, 5th Edition, 2018.
  - **Chapter 4 - MARIE: An Introduction to a Simple Computer**
    - 4.2 CPU Basics and Organization
    - 4.3 The Bus
    - 4.6 Memory Organization and Adressing
    - 4.8 MARIE
    - 4.9 Instruction Processing

Referencias

- Especificación de la Arquitectura OrgaSmall
  `https://github.com/fokerman/microOrgaSmall/`

¡Gracias!

Recuerden leer los comentarios adjuntos
en cada clase por aclaraciones.