

# Taller de Microarquitectura

# Organización del Computador 1

Segundo Cuatrimestre 2020

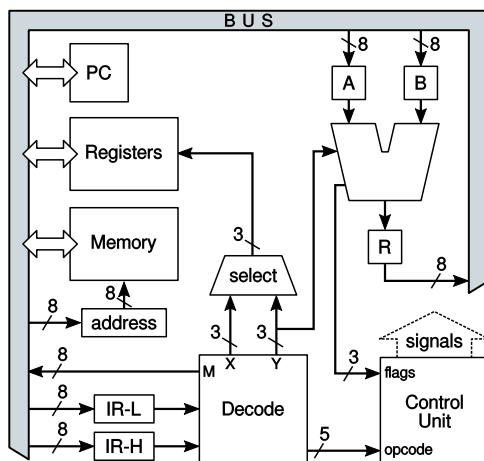
El presente taller consiste en analizar y extender una micro-arquitectura diseñada sobre el simulador *Logisim*. Se buscará codificar programas simples en ensamblador, modificar parte de la arquitectura y diseñar nuevas instrucciones.

El simulador se puede bajar desde la página <http://www.cburch.com/logisim/> o de los repositorios de Ubuntu. Requiere Java 1.5 o superior. Para ejecutarlo, teclear en una consola `java -jar logisim.jar`.

Para este taller utilizaremos mayoritariamente el modo simulación. Sobre todo las opciones de “Enable clock ticks”.

Además utilizaremos el componente memoria RAM, que inicia con todas sus posiciones en 0 y dos memorias ROM cuyo valor puede ser cargado desde un archivo desde la opción “Contents→Open...”.

## Procesador OrgaSmall



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

Se adjunta como parte de este taller las hojas de detalles del procesador OrgaSmall.

## Ejercicios

(1) **Introducción** - Recorrer la máquina y la hoja de datos, y responder:

a) ¿Cuál es el tamaño de la memoria? **256 bytes**

Son  $10 \cdot 2^{11} + 4 \cdot (2^{11} - 2^8) + 2 \cdot (2^{11} - 2^8) + 10 \cdot (2^{11} - 2^6)$  porque tenemos 10 instrucciones reservadas con 11 bits restantes.

- b) ¿Cuántas instrucciones sin operandos se podrían agregar respetando el formato de instrucción actual? Hay 10 operaciones. Las reservadas.
- c) Sin respetar el formato de instrucción, ¿cuántas instrucciones sin operandos se pueden agregar?
- d) ¿Qué tamaño tiene el PC? 8 bits, pues hay 256 posibles direcciones a las que apuntar y en binario para poder representarlas necesitaríamos  $x$  bits con  $2^x = 256$
- e) ¿Dónde se encuentra y qué tamaño tiene el IR? Se encuentra en el decodificador y tiene un tamaño de 16 bits. (no tengo enie, perdon)

#### Checkpoint 1

- (2) **Analizar** - Estudiar el funcionamiento de los circuitos indicados y responder las siguientes preguntas:

- a) PC (Contador de Programa): ¿Qué función cumple la señal `inc`?
- b) ALU (Unidad Aritmético Lógica): ¿Qué función cumple la señal `opW`?
- c) ControlUnit (Unidad de control): ¿Cómo se resuelven los saltos condicionales? Describir el mecanismo.
- d) microOrgaSmall (DataPath): ¿Para qué sirve la señal `DE_enOutImm`? ¿Qué parte del circuito indica que índice del registro a leer y escribir?

En caso de estar encendido se encarga incrementar o decrementar en 1 el PC de modo que se pueda avanzar en el programa.

En caso de estar apagado permite resetear la posición cuando `load=1`.

Permite que se actualicen los registros de los flags. El decodificador tiene en uno de sus outputs el 'opcode' que a su vez es un input del Control Unit

`DE_enOutImm` pasa la dirección de memoria que ya pasó por el IR al bus. `RB_selectIndexIn` y `RB_selectIndexOut` indican que índice del registro leer y escribir.

#### Checkpoint 2

- (3) **Ensamblar y correr** - Escribir en un archivo, compilar y cargar el siguiente programa:

b)

```

0x00-0x01 JMP seguir
                seguir:
0x02-0x03 SET R0, 0xFF
0x04-0x05 SET R1, 0x11

                siguiente:
0x06-0x07 ADD R0, R1
0x08-0x09 JC siguiente

                halt:
0x0A-0x0B JMP halt

```

a) Inicializa los registros y luego los suma entre si hasta que la suma no de carry. Y luego ingresa en un bucle infinito

Para ensamblar un archivo `.asm` ejecutar el comando:

```
python assembler.py NombreDeArchivo.asm
```

Esto generará un archivo `.mem` que puede ser cargado en la memoria RAM de la máquina.

- a) Antes de correr el programa, identificar el comportamiento esperado.
- b) ¿Qué lugar ocupará cada instrucción en la memoria? Detallar por qué valor se reemplazarán las etiquetas.

- c) Ejecutar y controlar ¿cuántos ciclos de clock son necesarios para que este código llegue a la instrucción `JMP halt`? **49 Clocks**
- d) ¿Cuántas microinstrucciones son necesarias para realizar el `ADD`? ¿Cuántas para el salto? **5 microinstrucciones para el `ADD` y 2 para el `JMP`**
- e) Asumiendo que el clock oscila a 32Hz. El simulador cuenta con un contador de clocks reinicializable manualmente para facilitar dicha tarea. ¿Cuánto tarda cada instrucción (en ms) y cuánto la ejecución?

**6\*1000/32 = 1875ms para el `JMP` y para `SET`,  
9\*1000/32 = 2812,5ms para el `ADD`, para el `JC`  
cuando hay carry tarda 2187,5ms y cuando no  
tarda 1875. La ejecución es eterna.**

---

Checkpoint 3

---

- (4) **Analizar** - Cada componente tiene una X sobre un cable. Para cada uno de ellos indicar qué sucedería si ese cable:
- a) Se corta
  - b) Vale siempre 0 (la cantidad de bits necesarios)
  - c) Vale siempre 1 (la cantidad de bits necesarios)

---

Checkpoint 4

---

- (5) **Programar** - Escribir en ASM un programa que calcule la suma de los primeros  $n$  números naturales. El valor resultante debe guardarse en `R1`, y se espera que el valor de  $n$  sea leído de `R0`.

---

Checkpoint 5

---

- (6) **Ampliando la máquina** - Agregar las siguientes instrucciones nuevas:

Nota1: Los siguientes ítems deben ser presentados mediante un código de ejemplo que pruebe la funcionalidad agregada.

Nota2: Tener en cuenta que si se agrega una operación, será necesario agregar el nombre mnemotécnico y el opcode en el archivo "`assembler.py`".

Para generar un nuevo *set* de micro-instrucciones, generar un archivo `.ops` y traducirlo con el comando:

```
python buildMicroOps.py NombreDeArchivo.ops
```

Esto generará un archivo `.mem` que puede ser cargado en la memoria ROM de la unidad de control.

- a) Sin agregar circuitos nuevos, agregar la instrucción **SIG**, que dado un registro aumenta su valor en 1. Esta operación **no** modifica los flags. Utilizar como código de operación el `0x09`.
- b) Sin agregar circuitos nuevos, agregar la instrucción **NEG** que obtenga el inverso aditivo de un número sin modificar los flags.  
Nota: el inverso aditivo de un número se puede obtener como `xor(XX, 0xFF)+0x01`. Utilizar como código de operación el `0x0A`.
- c) Implementar un circuito que dados dos números  $A_{7-0}$  y  $B_{7-0}$  los combine de forma tal que el resultado sea  $B_1 A_6 B_3 A_4 B_5 A_2 B_7 A_0$ .

(7) **Optativos** - Otras modificaciones interesantes:

- a)* Modificar las instrucciones **JC**, **JZ**, **JN** y **JMP** para que tomen como parámetro un registro.
- b)* Agregar las instrucciones **CALL** y **RET**. Considerar que uno de los parámetros de ambas instrucciones es un índice de registro que se utilizará como **Stack Pointer**. Además, la instrucción **CALL** toma como parámetro un inmediato de 8 bits que indica la dirección de memoria a donde saltar.
- c)* Modificar el circuito agregando las conexiones necesarias para codificar las instrucciones **STR [RX+cte5], Ry** y **LOAD Ry, [RX+cte5]**. Este par de instrucciones son modificaciones a las existentes, considerando que **cte5** es una constante de 5 bits en complemento a dos.
- d)* Agregar instrucciones que permitan leer y escribir los **flags**. Describir las modificaciones realizadas sobre el circuito.