Trabajo Práctico 1
Microarquitectura

Tecnología Digital II

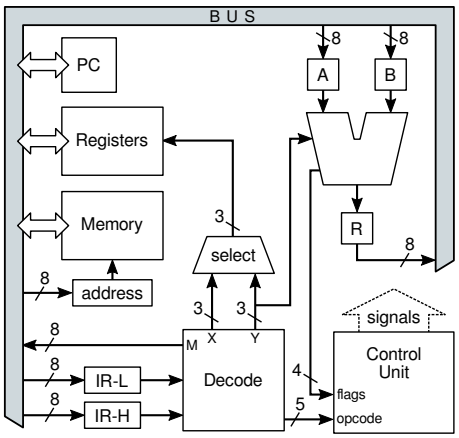
Introducción

El presente trabajo práctico consiste en analizar y extender una microarquitectura 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, ingresar en una consola:
`java -jar logisim.jar`.

El trabajo práctico debe realizarse en grupos de tres personas. Tienen dos semanas para realizar la totalidad de los ejercicios y entregar un informe en formato digital con la solución de los ejercicios. Se solicita entregar la solución de todos los ejercicios en archivos separados, junto con el informe en formato PDF. **La fecha de entrega límite es el martes 10/09 a las 23:59.**
Se solicita no realizar consultas del trabajo práctico por canales públicos. Limitar las preguntas al foro privado creado para tal fin.

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 direccionable a byte de tamaño 256 bytes.
- Bus de 8 bits.
- Diseño microprogramado.

Para poder descargar la Arquitectura OrgaSmall ir a: `https://github.com/fokerman/microOrgaSmall/`
La versión que utilizaremos se encuentra bajo el nombre `OrgaSmallWithStack`.

Ejercicios

1. **Trabajo previo** - Estudiar la documentación

   a) Leer la hoja de datos del procesador.

   b) Estudiar los circuitos que implementan la máquina.

   c) Identificar todas las instrucciones y su funcionamiento.

   d) Determinar los tamaños de las instrucciones, memoria y memoria de microinstrucciones.

   e) Analizar y estudiar el código *python* provisto junto a la máquina.

2. **Ensamblar y ejecutar** - Escribir el siguiente archivo, compilarlo y cargarlo en la memoria de la máquina:

```
main:
        SET R7, 0xFF
        SET R2, 0x50
        SET R3, 0x70
        SET R1, 0x01
        SET R0, 0x00

        aca:
                CMP R3, R2
                JZ fin
                CALL |R7|, coso2
                ADD R2, R1
                JMP aca

coso2:
        PUSH |R7|, R0
        SET R0, 0xFF
        SUB R0, R2
        STR [R2], R0
        POP |R7|, R0
        RET |R7|

fin:
        DB 0xA0
        DB 0x00

halt:
        JMP halt
```

> Para ensamblar el archivo, nombrarlo como `ejemplo.asm` y ejecutar el siguiente comando:
> `python assembler.py ejemplo.asm`
> Este comando genera un archivo `.mem` que puede ser cargado en la memoria RAM de la máquina. Además, genera un archivo `.txt` con las instrucciones en ensamblador del programa y sus direcciones de memoria para facilitar la lectura del binario.

a) Previamente a ejecutar el programa, describir con palabras el comportamiento esperado del mismo. No se debe explicar instrucción por instrucción, la idea es entender qué hace el programa y qué resultado genera.

b) Identificar la dirección de memoria de cada una de las etiquetas del programa.

c) Ejecutar e identificar de ser posible cuántos ciclos de clock son necesarios para que el programa llegue a la instrucción `JMP halt`.

d) ¿Cuántas microinstrucciones son necesarias para ejecutar la instrucción `ADD`? ¿Cuántas para la instrucción `JZ`? ¿Cuántas para la instrucción `JMP`?

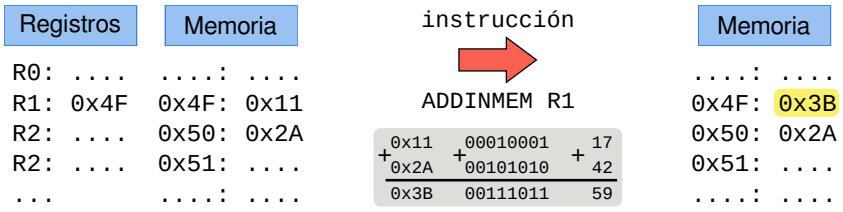
e) ¿El programa utiliza la pila?, ¿Qué datos son almacenados en la pila?

f) Describir detalladamente el funcionamiento de las instrucciones `PUSH`, `POP`, `CALL` y `RET`.

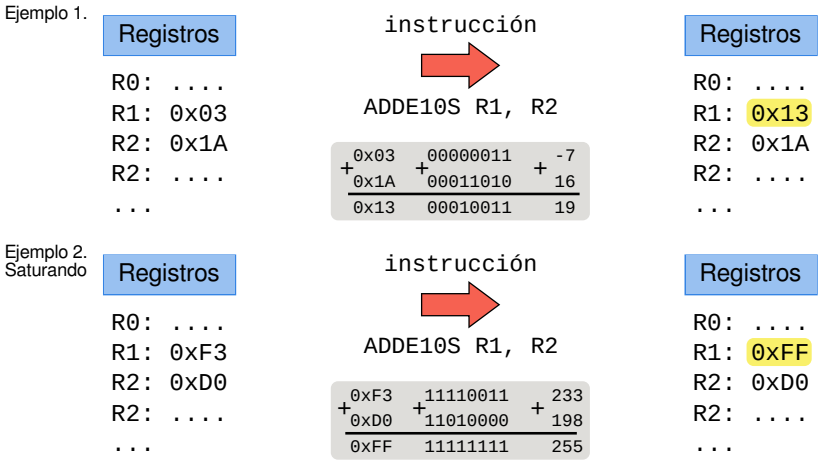
3. **Ampliando la máquina** - Agregar las siguientes nuevas instrucciones:

> Para generar un nuevo *set* de microinstrucciones, generar un archivo `.ops` y traducirlo a señales con el siguiente comando:
> `python buildMicroOps.py NombreDeArchivo.ops`
> Este 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 `ADDINMEM` que suma dos datos consecutivos en memoria y almacena el resultado en memoria nuevamente. La instrucción toma un registro que contiene la dirección de memoria del primer dato a sumar. El resultado se debe almacenar en la dirección del primer dato sumado. Se recomienda utilizar como código de operación el `0x0E`.



b) Agregar la instrucción `ADDE10S`, que toma de dos números en notación exceso-10 y los suma. Si el resultado es mayor que el máximo representable, entonces retorna el máximo representable. Toma como parámetros dos registros y el resultado lo almacena en el primero de ellos. Para implementar esta instrucción se debe modificar el circuito de la ALU para lo operación `14`. Bajo esté código se debe agregar la operación de la ALU que realice la suma descripta. Se recomienda utiliza como código de operación el `0x0F`.



Las instrucciones deben modificar **SOLO** lo indicado. No pueden modificar otros registros ni posiciones de memoria. Además para poder construir programas usando las nuevas instrucciones, deben modificar el archivo `assembler.py` dando soporte a las nuevas instrucciones.

4. **Programar** - Escribir en ASM las siguientes funciones:

a) Usando la instrucción `ADDINMEM`, implementar la función `processArray` que toma un arreglos de enteros positivos en memoria y los suma de a pares en memoria, pisando el primer elemento de cada par con el resultado. Considerar que el tamaño del arreglo siempre es par.

```
processArray(*p,size)
        for i=0; i<size/2; i=i+2
                p[i] = p[i] + p[i+1]
```

b) Usando la instrucción `ADDE10S`, implementar la función `sumE10S` que toma un par de arreglos de números en notación exceso 10 y suma los elementos entre sí. El resultado de cada operación se almacena en ambos arreglos por cada elemento sumado.

```
sumE10S(*p,*t,size)
        for i=0; i<size; i++
                r = sumExceso10Sat(p[i], t[i])
                p[i] = r
                t[i] = r
```

Considerar en todos los casos que los parámetros llegan en `R0`, `R1` y `R2`. Además las funciones no deben alterar ningún registro.
Para este ejercicio se proporciona un conjunto de archivos base donde pueden completar la implementación de sus funciones. Estos archivos pueden ser modificados para complementar su entrega con otros ejemplos de datos de entrada.