Trabajo Práctico 1
Microarquitectura

Tecnología Digital II

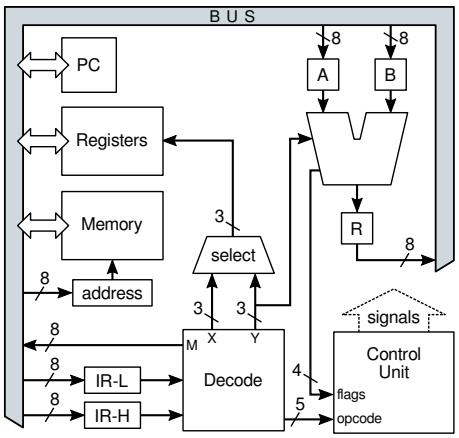
Introducción

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 domingo 13/04 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 R3, 0x01
        SET R2, 0x0A
        SET R1, 0x01
        SET R0, 0x00

        ciclo:
                ADD R3, R1
                CALL |R7|, completar
                CMP R3, R2
                JZ fin
                JMP ciclo
completar:
        PUSH |R7|, R0
        PUSH |R7|, R1
        LOAD R0, [0xA0]
        ADD R0, R1
        STR [0xA0], R0
        SET R1, 0xB0
        ADD R1, R3
        STR [R1], R0
        POP |R7|, R1
        POP |R7|, R0
        RET |R7|
fin:
        DB 0xB8
        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`?

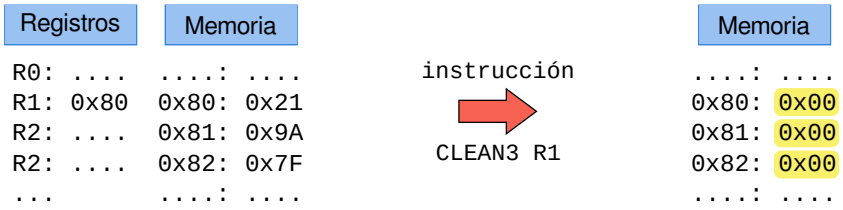
e) ¿El programa utiliza la pila?, ¿Qué datos son almacenados en la pila?

f) Describir detalladamente el funcionamiento de las instrucciones `CALL`, `JC`, `PUSH` y `LOADF`.

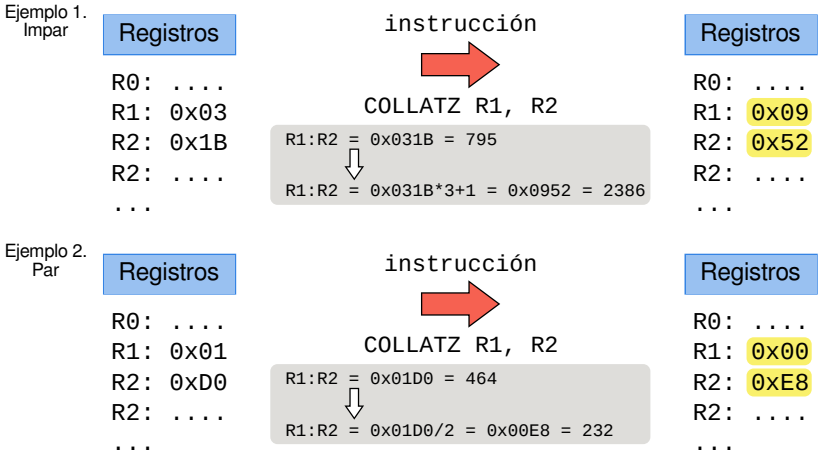
3. **Ampliando la máquina** - Agregar las siguientes nuevas instrucciones:

> 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 `CLEAN3`. Esta toma un registro como parámetro, y escribe ceros en los tres bytes consecutivos a partir de la dirección de memoria apuntada por el resgistro. Se recomienda utilizar como código de operación el `0x0E`.



b) Agregar la instrucción `COLLATZ`, que toma de dos registros como parámetro y los utiliza como un solo número sin signo. A este número le aplica un paso de la conjetura de Collatz. Si el número es par, lo divide por 2, si el número es impar, lo múltiplica por 3 y le suma 1. Si el resultado excede los 16 bits, el resultado debe ser 0. Si el resultado es 1, entonces no debe cambiar. El resultado debe ser almacenado en los dos registros pasados como parámetro. 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 `CLEAN3`, implementar la función `cleanBytes` que toma una dirección de memoria y una cantidad, y escribe ceros en todo el arreglo de memoria. Notar que la cantidad de bytes puede no ser múltiplo de 3.

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

b) Usando la instrucción `COLLATZ`, implementar la función `collatzSteps` que toma una dirección de memoria que apunta a un número de dos bytes y una cantidad de pasos. Debe aplicar la cantidad de pasos indicados en el número y escribir el resultado nuevamente en memoria.

```
collatzSteps(*p,steps)
        val = p[i] << 8 + p[i+1]
        for i=0; i<steps; i++
                if val % 2 = 0
                     val = val/2
                else
                     val = val * 3 + 1
        p[i+1] = val
        p[i] = val >> 8
```

Considerar en todos los casos que los parámetros llegan en `R0` y `R1` respectivamente. 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.