Trabajo Práctico 1
Microarquitectura

Tecnología Digital II

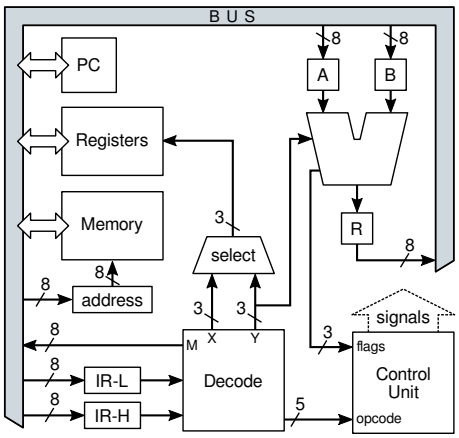
Introducción

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. **La fecha de entrega límite es el viernes 10 de septiembre.**
Se solicita no realizar consultas del trabajo práctico por los foros públicos. Limitar las preguntas al foro privado creado para tal fin. Tener en cuenta que algunos de los ítems están marcados como optativos, es decir, que pueden optar por resolverlos o no. El hecho de realizarlos no implica mayor calificación final, pero pueden ser tenidos en cuenta en un caso al límite de la aprobación.

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/`

Ejercicios

1. **Introducción** - Leer la hoja de datos y responder:

   a) ¿Cúal es el tamaño de la memoria en cantidad de bytes?

   b) ¿Cúantas instruciones sin operandos se podrían agregar al formato de instrucción?

   c) ¿Qué tamaño tiene el `PC`?

   d) ¿Dónde se encuentra y qué tamaño tiene el `IR`?

   e) ¿Cual es el tamaño de la memoria de microinstrucciones? ¿Cúal es su unidad direccionable?

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 comple 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 cuál índice del registro a leer y escribir?

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

```
inicio:
SET R0, 0x10
SET R1, 0x00
SET R2, 0x01

ciclo:
ADD R1, R2
CMP R0, R1
JZ fin
JMP ciclo
fin:

halt:
JMP halt
```

> Para ensamblar el archivo, nombrarlo como `ejemploContador.asm` y ejecutar el siguiente comando:
> `python assembler.py ejemploContador.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 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 JMP? y ¿Cuántas para la instrucción JZ?

4. **Programar** - Escribir en ASM los siguientes programas:

   a) Escribir un programa que calcule la suma de los primeros n números naturales.

```
sumN(n)
  a = 0
  while n != 0
    a = a + n
    n = n - 1
  return a
```

b) Calcular el máximo común divisor.

```
mcd(a, b)
  while a != b
    if a > b
      a = a - b
    else
      b = b - a
  return a
```

c) **[Optativo]** Calcular el i-ésimo número de Fibonacci.

```
fib(n)
  a=1
  b=1
  for i in 0 to n:
    tmp = a + b
    a = b
    b = tmp
  return a
```

Considerar en todos los casos que los parámetros llegan en `R0` y `R1` o solo en `R0`. El resultado debe ser guardado en `R4`.

5. **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 `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(XXXX, 0xFFFF) + 0x0001`. Utilizar como código de operación el 0x0A.

b) **[Optativo]** Modificando el circuito de la `ALU`, agregar la instrucción `COM` que combine dos números A_{7-0} y B_{7-0}, tal que el resultado sea B_1 A_6 B_3 A_4 B_5 A_2 B_7 A_0. Utilizar como código de operación el 0x09.

Nota: cada ítem debe ser presentado con un código de ejemplo que pruebe la funcionalidad agregada.