Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2343 – Arquitectura de Computadores (II/2017)

Proyecto Semestral: Entrega Práctica 03

Computador básico completo, 50 % de la entrega 03

Fecha de entrega: Lunes 23 de Octubre a las 17:00 horas

1. Objetivo

Para esta entrega tendrán que mejorar una vez más el computador básico desarrollado durante las entregas pasadas, además de modificar su *assembler* de manera que soporte todas las instrucciones y especificaciones detalladas más adelante.

En la Figura 1 se muestra el diagrama del computador básico modificado que tendrán que diseñar para esta entrega.

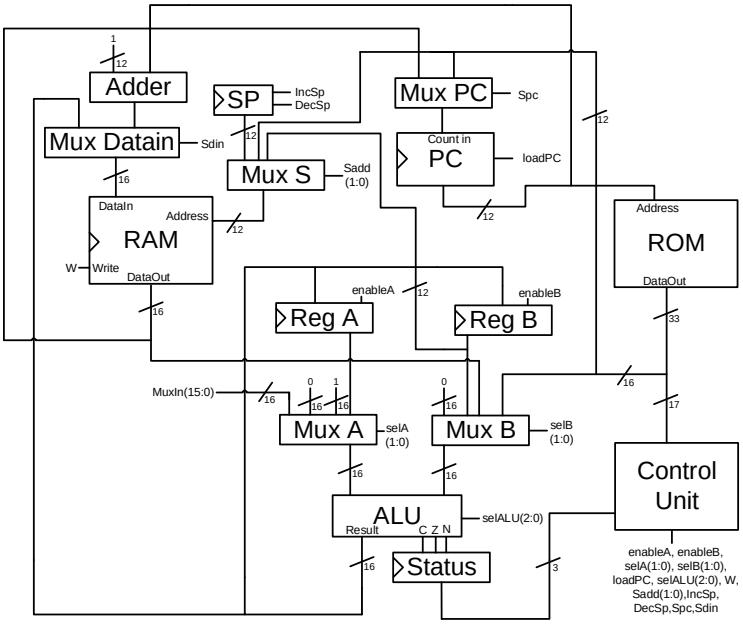

Figura 1: Computador básico.

Como se puede apreciar el computador consta de un registro *Stack Pointer*[1], un registro *Status*, un sumador *Adder*, los multiplexores *Datain*, *S*, *IN* y *PC*, además de un bus de entrada.

[1]Considere que si el contador de su implementación parte en cero, será necesario que al principio de cada programa se decremente en una unidad *SP* de manera que este parta en 111111111111.

Como se puede apreciar en la Figura 3, el contenido del bus de entrada esta determinado por el multiplexor *MuxIn*, que dependiendo de los 16 bits del literal selecionaá una entrada distinta.

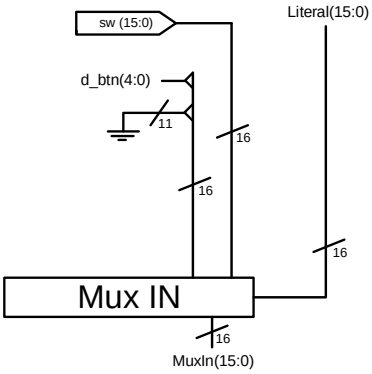

Figura 2: I/O detallado.

Para estandarizar el uso de los componentes, se reservarán los siguientes puertos para entradas:

Puerto	Input
0	Switches
1	Botones
*	Nada

Figura 3: Tabla de puertos Input.

2. Instrucciones y *assembler*

Para más detalles sobre los cambios del assembler para esta entrega, revisar el enunciado *Assembler*. A continuación se muestran las instrucciones que debe sosportar el computador:

```
MOV A,(B)     (cargar Mem[B] en A)
    B,(B)     (cargar Mem[B] en B)
    (B),A     (guarda A en Mem[B])
    (B),Lit   (guarda Lit en Mem[B])

ADD A,(B)     (guarda A+Mem[B] en A)
    B,(B)     (guarda A+Mem[B] en B)

SUB A,(B)     (guarda A-Mem[B] en A)
    B,(B)     (guarda A-Mem[B] en B)

AND A,(B)     (guarda A and Mem[B] en A)
    B,(B)     (guarda A and Mem[B] en B)

OR  A,(B)     (guarda A or Mem[B] en A)
    B,(B)     (guarda A or Mem[B] en B)
```

```
XOR A,(B)      (guarda A xor Mem[B] en A)
    B,(B)      (guarda A xor Mem[B] en B)

NOT (B),A      (guarda not A en Mem[B])

SHL (B),A      (guarda shift left A en Mem[B])

SHR (B),A      (guarda shift right A en Mem[B])

INC (B)        (incrementa Mem[B] en una unidad)

CMP A,(B)      (hace A-Mem[B])

PUSH A          Mem[SP] = A, SP--
     B          Mem[SP] = B, SP--

POP  A          SP++, A = Mem[SP]
     B          SP++, B = Mem[SP]

CALL Dir        Mem[SP] = PC, PC = Dir, SP--

RET             SP++, PC = Mem[SP]

IN   A,Lit      A = Input[Lit]
     B,Lit      B = Input[Lit]
     (B),Lit    Mem[B] = Input[Lit]
```

3. Ejemplos

Considere los siguientes ejemplos, los cuales corresponden a diversos programas que su *assembler* debe ser capaz de compilar:

- Programa 1:

```
DATA:

CODE:                   // Shift left rotate

MOV B,0                 // Puntero en 0
MOV A,8000h             // 1000000000000000b a A
MOV (B),A               // Guardar numero

shl_r:
 MOV A,0                // 0 a A
 OR A,(B)               // Recuperar numero
 SHL (B),A              // Guardar shift left de numero
                        // Si carry == 1
 JCR shl_r_carry        // Recuperar bit
 JMP shl_r_end          // No hacer nada
 shl_r_carry:
  INC (B)               // Agregar el bit perdido
 shl_r_end:
  JMP shl_r             // Repetir
```

- Programa 2:

```
DATA:

 arr    5
        Ah
        1
        3
        8
        5
 n      6
 r      0

CODE:                   // Sumar arreglo

MOV B,arr               // Puntero arr a B

siguiente:
 MOV A,(n)              // Restantes a A
 CMP A,0                // Si Restantes == 0
```

```
 JEQ end                // Terminar
 DEC A                  // Restantes --
 MOV (n),A              // Guardar Restantes
 MOV A,(r)              // Resultado a A
 ADD A,(B)              // Resultado + Arr[i] a A
 MOV (r),A              // Guardar Resultado
 INC B                  // Puntero en B ++
 JMP siguiente          // Siguiente

end:
 MOV A,(r)              // Resultado a A
 JMP end
```

- Programa 3:

```
DATA:

CODE:                   // Hack al stack

MOV A,2                 // 2 a A
PUSH A                  // Guarda A
MOV A,0                 // |
NOT B,A                 // | Puntero al primero en el stack a B
INC (B)                 // Primero en el stack++
POP A                   // Recupera A incrementado

end:
 JMP end
```

- Programa 4:

```
DATA:

CODE:                   // Swap con stack

MOV A,3                 // A = 3
MOV B,5                 // B = 5

PUSH A                  // |
PUSH B                  // |
POP A                   // |
POP B                   // | Swap con Stack
```

- Programa 5:

```
DATA:

CODE:                   // Subrutinas simples

MOV A,3                 // 3 a A
MOV B,2                 // 7 a B
CALL add                // A + B a B
MOV A,1                 // 1 A A
CALL add                // A + B a B
MOV A,7                 // 7 a A
CALL sub                // A - B a B
MOV A,B                 // B a A

fin:
JMP fin

add:
 ADD B,A                // A + B a B
RET

sub:
 SUB B,A                // A - B a B
RET
```

- Programa 6:

```
DATA:

CODE:                   // Subrutinas anidadas

MOV A,7
MOV B,1

CALL resta

fin:
 JMP fin

suma:
 XOR B,A                // Bits que no generan carry a B
 PUSH B                 // Guardar bits que no generan carries
 XOR B,A                // Recuperar segundo sumando
 AND A,B                // Bits que generan carry a A
 POP B                  // Recuperar bits que no generan carries
 CMP A,0                // Si carries == 0
```

```
 JEQ suma_fin           // Terminar
 SHL A                  // Convertir bits a carries en A
 CALL suma              // Sumar carries
 suma_fin:
 MOV A,B                // Resultado a A
RET

comp2:
 NOT A                  // Negado de A a A
 INC A                  // A++
RET

resta:
 PUSH A                 // Guarda minuendo
 MOV A,B                // Sustraendo a A
 CALL comp2             // Complemento a 2 del sustraendo a A
 MOV B,A                // Complemento a 2 del sustraendo a B
 POP A                  // Recupera minuendo
 CALL suma              // Suma de minuendo y complemento a 2 del sustraendo a A
RET
```

- Programa 7:

```
DATA:

CODE:                   // Sumar inputs

MOV B,0                 // Puntero en 0
IN (B),0                // Guardar switches
IN A,2                  // Nada a A
ADD A,(B)               // Sumar inputs
IN B,8000h              // Nada a B
ADD A,B                 // Sumar inputs

MOV B,0                 // Puntero en 0
IN (B),1                // Guardar botones
ADD A,(B)               // Sumar inputs
IN (B),FFFFh            // Guardar nada
ADD A,(B)               // Sumar inputs

end:
 JMP end
```

- Programa 8:

```
DATA:

CODE:                           // Sumar switches | Velocidad de clock a "full"

PUSH B                          // Guardar puntero

input:
 CALL std_io_btn_wait           // Esperar cambio en botones
 IN (B),0                       // Ingresar arr[i]
 MOV A,(B)                      // Recuperar switches
 INC B                          // Incrementar puntero
 CMP A,0                        // Si Switches != 0
 JNE input                      // Siguiente input

POP B                           // Recuperar puntero
MOV A,0                         // Resultado = 0

sumar:
 PUSH A                         // Guardar resultado
 MOV A,(B)                      // arr[i] a A
 CMP A,0                        // Si arr[i] == 0
 JEQ sumar_fin                  // Terminar
 POP A                          // Recuperar resultado
 ADD A,(B)                      // Resultado + arr[i]
 INC B                          // Puntero++
 JMP sumar                      // Siguiente
 sumar_fin:
 POP A                          // Recuperar Resultado

end:
 JMP end

////////////////Libreria std_io/////////////////////////////////////////
                                                                      //
std_io_btn_wait:                // * en A, * en B                     //
 PUSH B                         // Guarda B                           //
 IN A,1                         // Estado actual                      //
 std_io_btn_wait_press_lp:                                           //
  IN B,1                        // Nuevo estado                       //
  CMP A,B                       // Si ==                              //
  JEQ std_io_btn_wait_press_lp  // Continuar                          //
 XOR B,A                        // Bits cambiados                     //
 std_io_btn_wait_release_lp:                                         //
  IN A,1                        // Nuevo estado                       //
  AND A,B                       // Bits an cambiados                 //
  CMP A,0                       // SI != 0                            //
  JNE std_io_btn_wait_release_lp// Continuar                         //
```

```
48  MOV A,B                         // Bits cambiados a A                   //
49  POP B                           // Recupera B                           //
50 RET                              // Retorna Bit(s) en A                  //
51                                                                          //
52 ////////////////////////////////////////////////////////////////////////////
```

4. Entrega

Deben entregar:

- El proyecto completo de la CPU, es decir, todos los archivos involucrados en su proyecto, sin las carpetas .caché, .hw, .runs o .sim (eso debe ser manejado a través del .gitignore).
- Una carpeta para el assembler, incluido un archivo ejecutable (.exe o .jar) que contenga su *assembler*.
- Un breve informe (incluyendo el número de grupo y el nombre de los integrantes) que contenga:
  - La especificación de la estructura de las instrucciones de su CPU (función de cada uno de los 33 bits de una instrucción).
  - La convención de todos los multiplexores.
  - Una tabla anexa (puede ser un .csv) con todas las instrucciones soportadas por la CPU y su implementación de acuerdo a las señales de la Control Unit.
  - Una explicación de cómo ocupar su *assembler*, detallando paso a paso cómo usar su programa para ensamblar un archivo en *assembly* y generar el archivo de salida para insertar en la ROM.
  - Un párrafo por persona que especifica qué hizo cada integrante del grupo durante la entrega y una sección que explique qué fue lo más fácil y lo más difícil para el grupo en la entrega.

La entrega del proyecto (código, informe y adicionales) es por medio del repositorio en Github tal que el Lunes 23 de Octubre a las 17:00 horas deben tener en la rama Master todos los archivos correspondientes a su grupo. El día de la entrega deben llegar con sus archivos listos para mostrar a su ayudante asignado.

Entregas atrasadas serán **penalizadas con 0.5 puntos** por cada hora (o fracción) de atraso.

Para entrega 2 y 3: El día anterior a la entrega, se subirán los archivos para la evaluación tanto en la página del curso como en un issue en el Syllabus del curso. Son 6 archivos que corresponden a algoritmos en para evaluar. Deben generar los archivos .bit para cada uno de los ellos. Luego, durante la entrega presencial, se van a probar esos algoritmos y se pedirá la compilación completa (transformar el .txt a instrucciones de la ROM, insertar estas instrucciones en el computador y generar el bitstream en Vivado) de uno de ellos. Toda acotación especial se debe incluir en el informe de cada entrega o en un README.

5. Puntajes

A continuación se describe el puntaje asociado a cada item:

La entrega consistirá en mostrar el output de diferentes tests en en el display. El display, al igual que en la entrega anterior, deberá mostrar los 8 bits menos significativos de cada registro. Entonces, se evaluará que la secuencia mostrada sea correcta. Si es correcta, se obtiene el puntaje completo, en caso contrario no se obtiene puntaje.

Cada ítem de la tabla está asociado a un ponderador. En la entrega cada ítem se pude evaluar entre 0 y 1 y ese valor se multiplcia con el ponderador asociado. En el caso del descuento y bonus, se puede evaluar entre 0 y 0,5.

0.75	0.75	0.75	0.75	1	1	0.5	0.5
Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Formalidad	Informe

Figura 4: Tabla de puntajes.

6. Contacto

Cualquier pregunta sobre el proyecto, ya sean de enunciado, contenido o sobre aspectos administrativos deben comunicarse con los ayudantes creando issues en el Syllabus del Github del curso o directamente con los ayudantes:

- Francesca Lucchini: flucchini@uc.cl
- Felipe Pezoa: fipezoa@uc.cl
- Hernán Valdivieso: hfvaldivieso@uc.cl
- Luis Leiva: lileiva@uc.cl

7. Evaluación

Cada entrega del proyecto se evaluará de forma grupal y se ponderará por un porcentaje de coevaluación para calcular la nota de cada alumno.

Dado lo anterior, dentro de las primeras **24 horas** posteriores a cada entrega, **todos los alumnos** deberán completar de forma **individual y obligatoria** el formulario web que los ayudantes pondrán a su disposición, repartiendo un máximo de 4 puntos (aunque cambia según la cantidad de estudiantes en el grupo; el máximo corresponde al número de compañeros), con hasta un decimal, entre sus compañeros y una diferencia máxima entre el mayor y menor puntaje de 1. La suma de todos los puntos obtenidos por el integrante, *sp*, será utilizada para el cálculo de la nota de cada entrega, lo que puede hacer que este repruebe el curso.

La nota de cada entrega se calcula de la siguiente forma (con un máximo de 7,5):

$$NotaEntrega_{individual} = \text{mín}(k_g \times NotaEntrega_{grupal}, NotaEntrega_{grupal} + 0,5)$$

donde,

$$k_g = \frac{sp+3}{7}$$

y *sp* puede variar de acuerdo a la cantidad de estudiantes que conforman el grupo.

Los alumnos que no cumplan con enviar la coevaluación en el plazo asignado tendrán un **descuento de 0.5 puntos** en su nota de la entrega correspondiente.

8. Integridad académica

Los alumnos de la Escuela de Ingeniería de la Pontificia Universidad Católica de Chile deben mantener un comportamiento acorde a la Declaración de Principios de la Universidad. En particular, se espera que mantengan altos estándares de honestidad académica. Cualquier acto deshonesto o fraude académico está prohibido; los alumnos que incurran en este tipo de acciones se exponen a un Procedimiento Sumario. Es responsabilidad de cada alumno conocer y respetar el documento sobre Integridad Académica publicado por la Dirección de Docencia de la Escuela de Ingeniería.

Específicamente, para los cursos del Departamento de Ciencia de la Computación, rige obligatoriamente la siguiente política de integridad académica. Todo trabajo presentado por un alumno para los efectos de la evaluación de un curso debe ser hecho individualmente por el alumno, sin apoyo en material de terceros. Por "trabajo" se entiende en general las interrogaciones escritas, las tareas de programación u otras, los trabajos de laboratorio, los proyectos, el examen, entre otros. Si un alumno copia un trabajo, obtendrá nota final 1,1 en el curso y se solicitará a la Dirección de Docencia de la Escuela de Ingeniería que no le permita retirar el curso de la carga académica semestral. Por "copia" se entiende incluir en el trabajo presentado como propio partes hechas por otra persona.

Obviamente, está permitido usar material disponible públicamente, por ejemplo, libros o contenidos tomados de Internet, siempre y cuando se incluya la referencia correspondiente.

Lo anterior se entiende como complemento al Reglamento del Alumno de la Pontificia Universidad Católica de Chile. Por ello, es posible pedir a la Universidad la aplicación de sanciones adicionales especificadas en dicho reglamento.