DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN
ESCUELA DE INGENIERÍA
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

IIC2343-2 - Arquitectura de Computadores (II/2021)

Etapa 2 del Proyecto

Presentación: Lunes 8 de Noviembre

Descripción

Su trabajo en esta etapa será exclusivamente dentro de la CPU y su ensamblador.

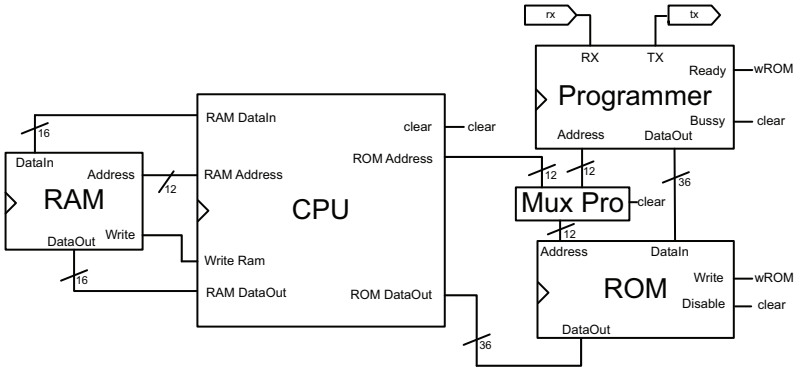

Figura 1: Diagrama parcial del computador básico de proyecto dentro del componente Basys3.

Para terminar de implementar su CPU deberán agregar las funciones de: direccionamiento indirecto, el resto de los saltos, subrutinas y stack. Los cambios a la arquitectura los pueden ver en el diagrama. Juntos con las modificaciones tendrán que extender su control unit y la codificación de su palabra, te tal modo de que su CPU pueda ejecutar las instrucciones de assembly descrito.

Programar el código de maquina manualmente en la ROM es una tarea tediosa y propensa al error. Para evitar esto, deberán hacer un ensamblador que los ayude a traducir archivos en lenguaje de assembly a su código de maquina, y escribir dicho programa directamente a la ROM por medio del programador. La especificación del ensamblador se encuentra en su propio enunciado.

Hardware

En este punto pueden usar las librerías `IEEE.std_logic_unsigned.all` y `IEEE.numeric_std.all` dentro de CPU si lo desean. Recuerden que para que el programador sea capaz de resetear el estado de los registros y contadores después de re-programar la ROM es importante que les conecten la señal clear para obtener el resultado esperado.

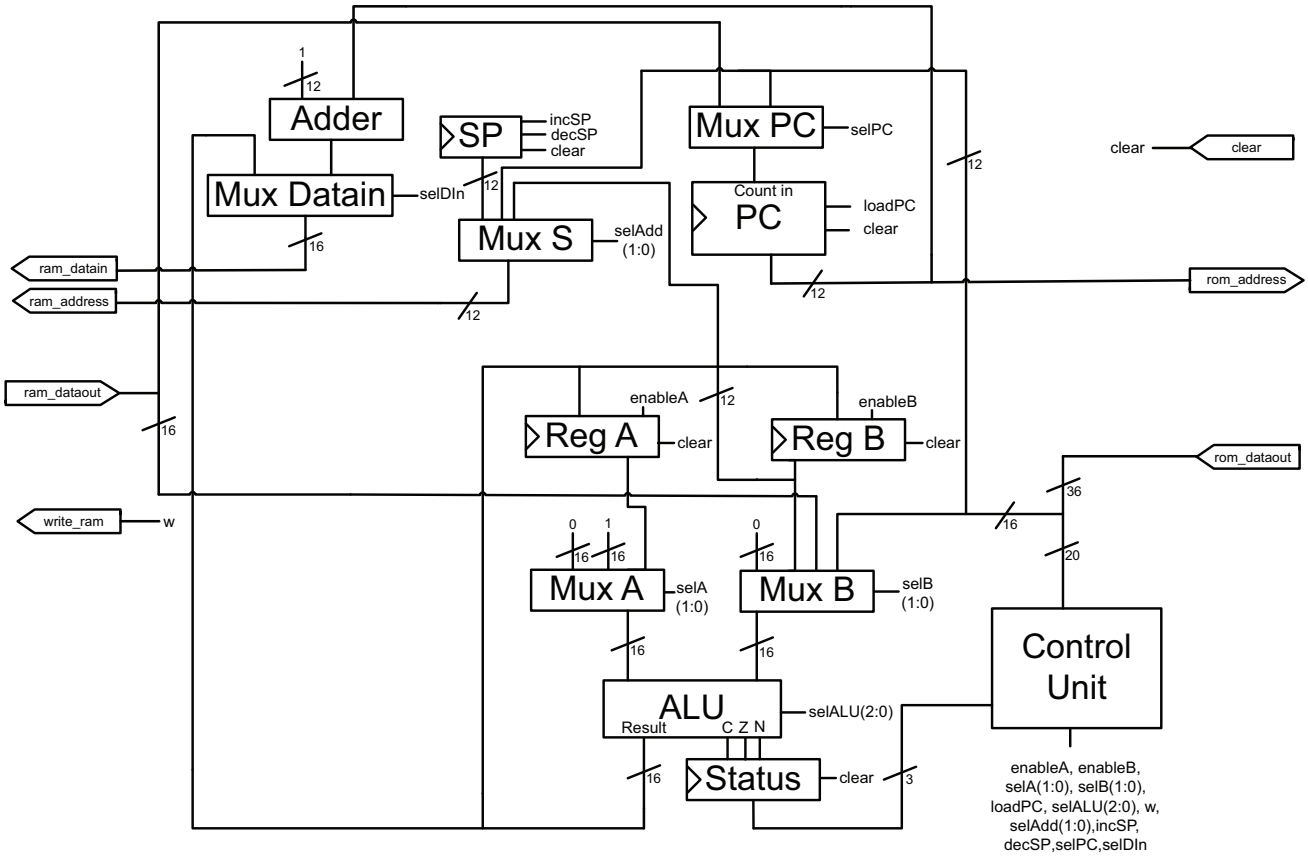

Figura 2: Diagrama interno de la CPU del computador básico de proyecto.

Componentes a agregar

- Un registro contador para la dirección del *stack* de 16 bits (SP).
- Un multiplexor para la carga del PC (MUX pc) que seleccionen entre dos entradas, cada una de 16 *bits*.
- Un multiplexor para la entrada de la RAM (MUX datain) que seleccionen entre dos entradas, cada una de 16 *bits*.
- Un multiplexor para la dirección de la RAM (MUX s) que seleccionen entre tres entradas, cada una de 12 *bits*.
- Un sumador (Adder) que incrementa el valor de 12 bits del PC en 1 y entrega un resultado en 16 bits *bits*.

Software

En está etapa usarán su ensamblador para traducir y transferir distintos programas en lenguaje de assembly a su computador para poder ejecutarlos. Escriban los programas que estimen convenientes para probar su arquitectura, al final del enunciado hay algunos ejemplos.

Presentación

El Lunes 8 de Noviembre deberán mostrar y cargar con su ensamblador con una serie de test para verificar su arquitectura. Nuevamente tendrán explicar cuales fueron las decisiones de diseño que realizaron. Deberán entregar:

- Un comprimido del proyecto en Vivado
- Un comprimido con el ejecutable de su ensamblador.
- Un breve informe con la descripción de la estructura de su palabra y una tabla que muestre cada instrucción implementada y el código de máquina en su arquitectura.

Assembly

Etapa 1

MOV	A,B	guarda B en A
	B,A	guarda A en B
	A,Lit	guarda un literal en A
	B,Lit	guarda un literal en B
	A,(Dir)	guarda Mem[Dir] en A
	B,(Dir)	guarda Mem[Dir] en B
	(Dir),A	guarda A en Mem[Dir]
	(Dir),B	guarda B en Mem[Dir]
ADD SUB AND OR XOR	A,B	guarda A op B en A
	B,A	guarda A op B en B
	A,Lit	guarda A op literal en A
	B,Lit	guarda A op literal en B
	A,(Dir)	guarda A op Mem[Dir] en A
	B,(Dir)	guarda A op Mem[Dir] en B
	(Dir)	guarda A op B en Mem[Dir]
NOT SHL SHR	A	guarda op A en A
	B,A	guarda op A en B
	(Dir),A	guarda op A en Mem[Dir]
INC	A	incrementa A en una unidad
	B	incrementa B en una unidad
	(Dir)	incrementa Mem[Dir] en una unidad
DEC	A	decrementa A en una unidad
CMP	A,B	hace A-B
	A,Lit	hace A-Lit
	A,(Dir)	hace A-Mem[Dir]
JMP	Ins	carga Ins en PC
JEQ	Ins	carga Ins en PC si en el status Z = 1
JNE	Ins	carga Ins en PC si en el status Z = 0
NOP		no hace cambios

Etapa 2

MOV	A,(B)	guarda Mem[B] en A
	B,(B)	guarda Mem[B] en B
	(B),A	guarda A en Mem[B]
	(B),Lit	guarda Lit en Mem[B]
ADD SUB AND OR XOR	A,(B)	guarda A op Mem[B] en A
	B,(B)	guarda A op Mem[B] en B
NOT SHL SHR	(B),A	guarda op A en Mem[B]
INC	(B)	incrementa Mem[B] en una unidad
CMP	A,(B)	hace A-Mem[B]
JGT	Ins	carga Ins en PC si en el status N = 0 y Z = 0
JGE	Ins	carga Ins en PC si en el status N = 0
JLT	Ins	carga Ins en PC si en el status N = 1
JLE	Ins	carga Ins en PC Ins si en el status N = 1 o Z = 1
JCR	Ins	carga Ins en PC Ins si en el status C = 1
PUSH	A	guarda A en Mem[SP] y decrementa SP
	B	guarda B en Mem[SP] y decrementa SP
POP	A	incrementa SP y luego guarda Mem[SP] en A
	B	incrementa SP y luego guarda Mem[SP] en B
CALL	Ins	guarda PC+1 en Mem[SP], carga Ins en PC y decrementa SP
RET		incrementa SP y luego carga Mem[SP] en PC

Ejemplos

- Programa 1:

```
DATA:
CODE:           // Canten 'La Farolera':
MOV A,2         // 2
MOV B,2         // y 2
ADD A,B         // son 4
NOP             // 4
NOP             // y 2
NOP             // son
ADD A,2         // 6
NOP             //
NOP             // 6
NOP             // y 2
ADD A,B         // son 8
MOV B,A         // y 8
ADD A,B         // 16
NOP             //
NOP             //
NOP             // A = 10h , B =8h
```

- Programa 2:

```
DATA:

CODE:           // Swaps

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

MOV (0),A       // |
MOV A,B         // |
MOV B,(0)       // | Swap con MOV y variable auxiliar

SUB A,B         // A = 2

XOR A,B         // |
XOR B,A         // |
XOR A,B         // | Swap con XOR
```

- Programa 3:

```
DATA:           // Variables a sumar

 a 5
 b Ah

CODE:           // Sumar variables

MOV A,0         // 0 a A
ADD A,(a)       // A + a a A
ADD A,(b)       // A + b a A
MOV B,A         // Resultado a B

end:
 DEC A          // A--
JMP end
```

- Programa 4:

```
DATA:

 a E5h          // 11100101b
 b B3h          // 10110011b
 bits 0b

CODE:           // Contar bits en 1 compartidos

MOV A, ( a)     // a a A
AND A , ( 1d )  // A & b a A
JMP loop        // Empieza desde loop

bit:
 INC  (2h)      // bits ++
 loop:
  CMP A ,0b     // Si A == 0
  JEQ  end      // Terminar
  SHR A         // Si A >> 1 genera carry
  JCR  bit      // Siguiente desde bit
                // Si no
  JMP  loop     // Siguiente desde loop

end:
 MOV A,(10b)    // Resultado a A
 JMP end
```

- Programa 5:

```
DATA:

 varA 8
 varB 3

CODE:           // Restar sin SUB ni ADD:

MOV A,(varB)    // varB a A
NOT (varB),A    // A Negado a varB
INC (varB)      // Incrementar varB

suma:

MOV A,(varA)    // Resultado a A

end:
 NOP
 JMP end
```

- Programa 6:

```
DATA:

CODE:           // No debe saltar

JMP start

error:
 MOV A,FFh      // FFh a A
 JMP error
```

```
start:
 MOV B,1
 MOV A,B
 INC A
 CMP A,B
 JEQ error

 INC B
 CMP A,2
 JNE error

 MOV (0),A
 INC B
 CMP A,2
 JGT error
 CMP A,(0)
 JGT error

 INC B
 INC (0)
 CMP A,(0)
 JGE error

 INC B
 CMP A,2
 JLT error

 CMP A,1
 JLT error
 INC B
 DEC A
 CMP A,0
 JLE error

 INC B
 SHL A
 JCR error

 SUB A,3
 JCR error

 MOV A,11h      // 11h a A
```

- Programa 7:

```
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 8:

```
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 9:

```
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 10:

```
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 11:

```
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 12:

```
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
```