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

IIC2343 - Arquitectura de Computadores (II/2024)

Etapa 2 del Proyecto

Descripción

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

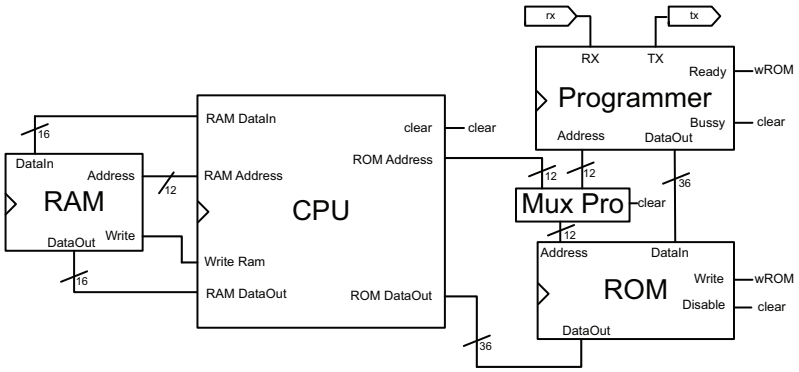

Figura 1: Diagrama parcial del computador básico de proyecto dentro del componente Basys3.

Para terminar de implementar su CPU, deberán agregar las funciones de: direccionamiento indirecto, subrutinas y *stack*. Los cambios a la arquitectura los pueden ver en el diagrama. Juntos con las modificaciones anteriores, tendrán que extender su *control unit* y la codificación de sus palabras para que su CPU pueda ejecutar las instrucciones del assembly descrito.

Adicionalmente, deberán terminar de programar su ensamblador, el que los ayudará a traducir archivos de lenguaje Assembly a su código de máquina y, luego, escribir dicho programa directamente a la ROM por medio del programador. La especificación del ensamblador se encuentra en su propio enunciado en la sección de la etapa 2.

Hardware

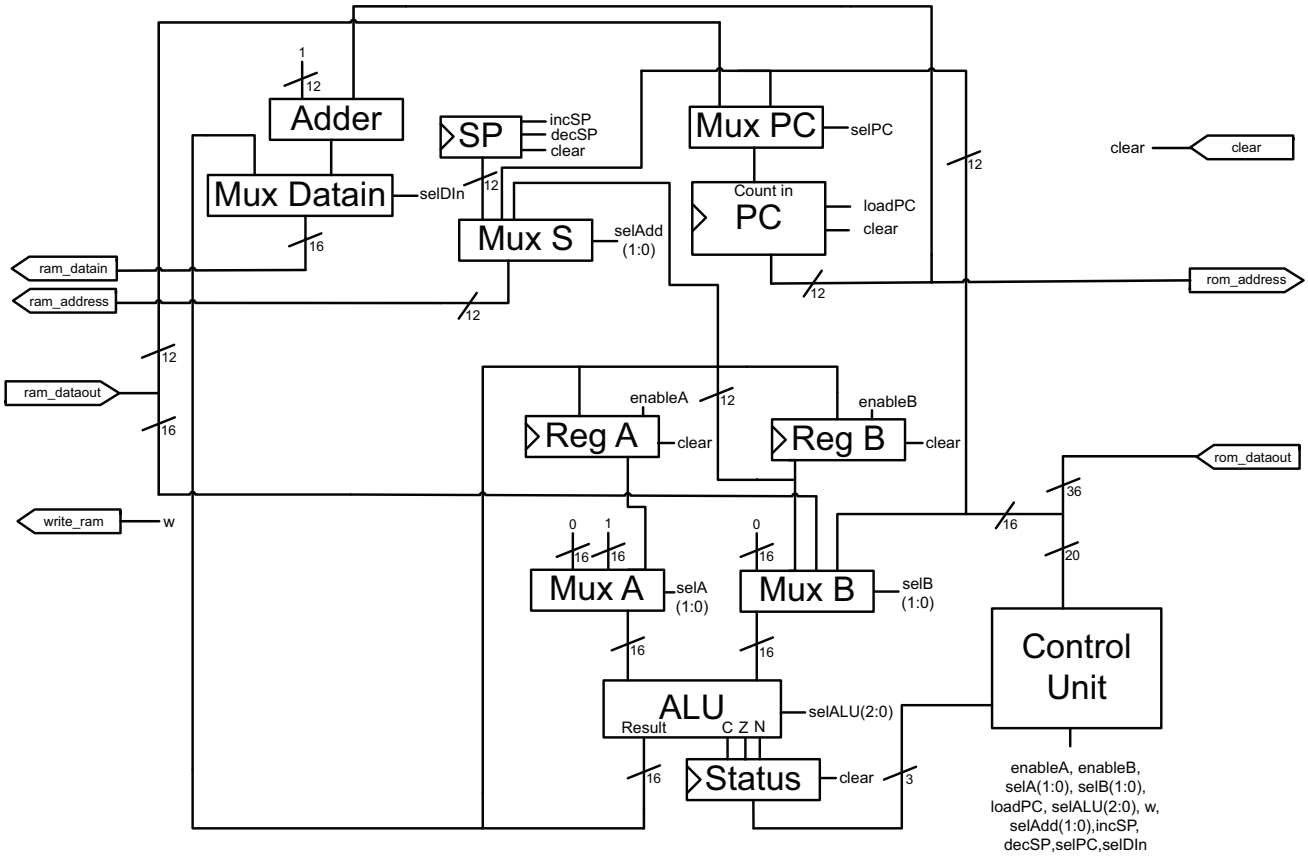

Figura 2: Diagrama interno de la CPU del computador básico de proyecto.

Componentes a agregar

- Un **registro contador** `SP` para la dirección del puntero al *stack* de al menos 12 *bits*.
- Un **multiplexor** `MUX PC` para la carga del `PC` que seleccione una de dos entradas, cada una de 12 *bits*.
- Un **multiplexor** `MUX Datain` para la entrada de datos de la `RAM` que seleccione una de dos entradas, cada una de 16 *bits*.
- Un **multiplexor** `MUX S` para la dirección de la `RAM` que seleccione una de 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*.

Software

En esta etapa, usarán su ensamblador para traducir y transferir distintos programas en lenguaje de Assembly a la ROM de su computador y, de ese modo, poder ejecutarlos. Escriban los programas que estimen convenientes para probar su arquitectura. Al final del enunciado, se encuentran algunos ejemplos.

No está de más mencionar que estos son, como se dijo, **ejemplos** y no son *tests*. Debido a esto, que les funcionen estos ejemplos no asegura que su computador sea 100 % funcional.

Presentación

Se aceptarán envíos de *commits* hasta **el lunes 4 de noviembre a las 14:30 horas**, momento en el que se recolectarán todos los repositorios. **Se tendrá que presentar en los computadores del laboratorio en no más de 10 minutos** el día de laboratorio correspondiente a su sección durante esa misma semana. Esto se llevará a cabo con la descarga del comprimido de su repositorio que el equipo docente le proveerá. El día de presentación deberán mostrar y cargar con su ensamblador un *test* para verificar el funcionamiento correcto de su arquitectura. Nuevamente, tendrán que explicar cuáles fueron las decisiones de diseño que realizaron y subir al repositorio de su grupo en *GitHub* lo siguiente:

- Su proyecto en Vivado.
- El ejecutable (o archivo, en caso de usar Python o algún lenguaje similar) de su ensamblador.

Puntaje

La nota será calculada utilizando 3 *tests*, que asumen todas las instrucciones de la entrega pasada como realizadas. Cada uno de estos *tests* tiene un puntaje distinto. Deben lograrse de manera consecutiva (es decir, si su grupo no logra pasar el *test* 1, no se probará el *test* 2). Los *tests* 1 y 2 por sí solos debe lograrse en su totalidad para obtener su puntaje asociado. Por otro lado, si no se logra pasar el *test* 3 en su totalidad, se probarán *sub-tests* de este, en donde se asignará puntaje parcial según ejecución.

Es importante destacar que, tal como en la entrega anterior, se ejecutará un *test* básico antes de estos donde se probarán instrucciones que **deben** estar funcionando. En caso de no pasar este *test*, no se evaluarán los siguientes dado que fallarán directamente.

Los *tests*, junto con sus puntajes, son los siguientes:

- ***Test* 1: Punteros (1 punto)**
- ***Test* 2: Direccionamiento indirecto (1 punto)**
- ***Test* 3: Manejo de *stack* (4 puntos)**

Cálculo de nota entrega: $PuntajeObtenido + 1$

Es importante mencionar que el uso de *process* en componentes que no requieren sincronía significará **la evaluación de toda la entrega con nota mínima.**

Recuperación de puntaje y requisitos

Durante las semanas previas a la presentación de la entrega, se realizarán salas de ayuda en el horario de laboratorio para que trabaje en el proyecto y resuelva dudas con sus ayudantes. Si su grupo termina con un 100 % de asistencia en estas instancias, se le permitirá **recuperar hasta la mitad del puntaje descontado en la evaluación**. Para que la asistencia sea considerada, **al menos una**

persona del grupo debe estar presente **durante los dos bloques de laboratorio** que correspondan a su sección. **Solo se aceptará asistencia al laboratorio de su sección**, es decir, cualquier persona que intente asistir a un laboratorio que no le corresponde no será admitido y su asistencia no será registrada.

La recuperación del puntaje, en caso de cumplir con los criterios de asistencia, se realizará durante la próxima sala de ayuda que corresponda a su sección (semana del 11 de noviembre). En esta, debe mostrar que arregló todos errores de su entrega original. El código de la recuperación deberá ser presentado el día de su laboratorio correspondiente. Este deberá incluir los arreglos de esta entrega, pero **nada posterior a ella**.

Assembly

Etapa 1

MOV	A,B	Guarda el valor de B en A
	B,A	Guarda el valor de A en B
	A,Lit	Guarda un literal Lit en A
	B,Lit	Guarda un literal Lit en B
	A,(Dir)	Guarda el valor de Mem[Dir] en A
	B,(Dir)	Guarda el valor de Mem[Dir] en B
	(Dir),A	Guarda el valor de A en Mem[Dir]
	(Dir),B	Guarda el valor de B en Mem[Dir]
ADD SUB AND OR XOR	A,B	Guarda el resultado de A op B en A
	B,A	Guarda el resultado de A op B en B
	A,Lit	Guarda el resultado de A op Lit en A
	B,Lit	Guarda el resultado de A op Lit en B
	A,(Dir)	Guarda el resultado de A op Mem[Dir] en A
	B,(Dir)	Guarda el resultado de A op Mem[Dir] en B
	(Dir)	Guarda el resultado de A op B en Mem[Dir]
NOT SHL SHR	A	Guarda el resultado de op A en A
	B,A	Guarda el resultado de op A en B
	(Dir),A	Guarda el resultado de op A en Mem[Dir]
INC	A	Incrementa el valor de A en una unidad
	B	Incrementa el valor de B en una unidad
	(Dir)	Incrementa el valor de Mem[Dir] en una unidad
DEC	A	Decrementa el valor de A en una unidad
CMP	A,B	Ejecuta la instrucción SUB A,B sin actualizar el valor de A
	A,Lit	Ejecuta la instrucción SUB A,Lit sin actualizar el valor de A
	A,(Dir)	Ejecuta la instrucción SUB A,(Dir) sin actualizar el valor de A
JMP	Ins	Carga la dirección de la instrucción Ins en PC
JEQ	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple Z = 1
JNE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple Z = 0
NOP		No hace cambios

Etapa 2

MOV	A,(B)	Guarda el valor de Mem[B] en A
	B,(B)	Guarda el valor de Mem[B] en B
	(B),A	Guarda el valor de A en Mem[B]
	(B),Lit	Guarda un literal Lit en Mem[B]
ADD SUB AND OR XOR	A,(B)	Guarda el resultado de A op Mem[B] en A
	B,(B)	Guarda el resultado de A op Mem[B] en B
NOT SHL SHR	(B),A	Guarda el resultado de op A en Mem[B]
INC	(B)	Incrementa el valor de Mem[B] en una unidad
CMP	A,(B)	Ejecuta la instrucción SUB A,(B) sin actualizar el valor de A
JGT	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 0 y Z = 0
JGE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 0
JLT	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 1
JLE	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple N = 1 o Z = 1
JCR	Ins	Carga la dirección de la instrucción Ins en PC si en Status se cumple C = 1
PUSH	A	Guarda el valor de A en Mem[SP] y decrementa SP en una unidad
	B	Guarda el valor de B en Mem[SP] y decrementa SP en una unidad
POP	A	Incrementa SP en una unidad y luego guarda el valor de Mem[SP] en A
	B	Incrementa SP en una unidad y luego guarda el valor de Mem[SP] en B
CALL	Ins	Guarda PC+1 en Mem[SP], carga la dirección de la instrucción Ins en PC y decrementa SP en una unidad
RET		Incrementa SP en una unidad y luego carga el valor de 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)   // varA a B
 XOR B,(varB)   // Suma de bits a B
 AND A,(varB)   // Carries de bits a A
 SHL A          // Shift Carries
 MOV (varB),A   // Carries a varB
 MOV (varA),B   // Suma a varB
 CMP A,0        // Carries > 0
 JNE suma       // Volver a sumar

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
```