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

IIC2343 - Arquitectura de Computadores (I/2025)

Etapa 2 del Proyecto

Descripción

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

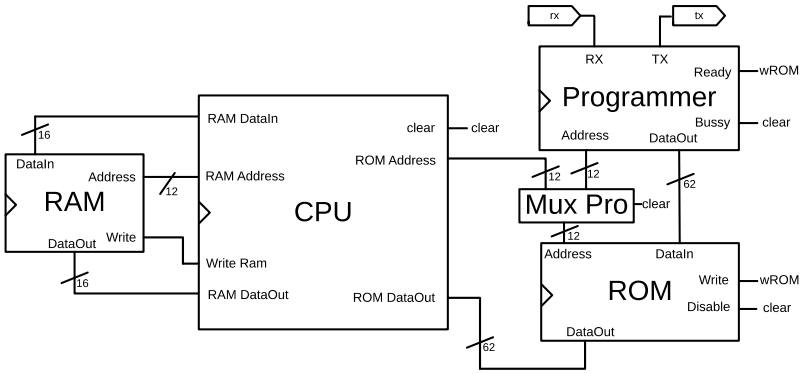

Figura 1: Diagrama parcial del computador básico de proyecto dentro del componente Basys3.

En esta etapa, expandirá el componente `Register File` a uno llamado `Super Register File`. Luego, lo conectará con una memoria de datos, una memoria de instrucciones y una `Control Unit` diseñada por usted para implementar **la primera versión de un computador programable**.

Los cambios a la arquitectura general se pueden observar en el diagrama correspondiente, mientras que las modificaciones del `Register File` y `Super Register File` se pueden observar en la siguiente figura. En particular, es importante notar que el contador `PC` **ya no se encuentra dentro del `Register File`, sino que dentro del `Super Register File`**. Es decir, debe **sacarlo** del componente original y utilizarlo **solo dentro** de `Super Register File`.

Adicionalmente, se le proveerá un ensamblador para que pueda probar código *Assembly* en su máquina programable. A partir de esta herramienta se evaluará la correctitud de su implementación.

Hardware

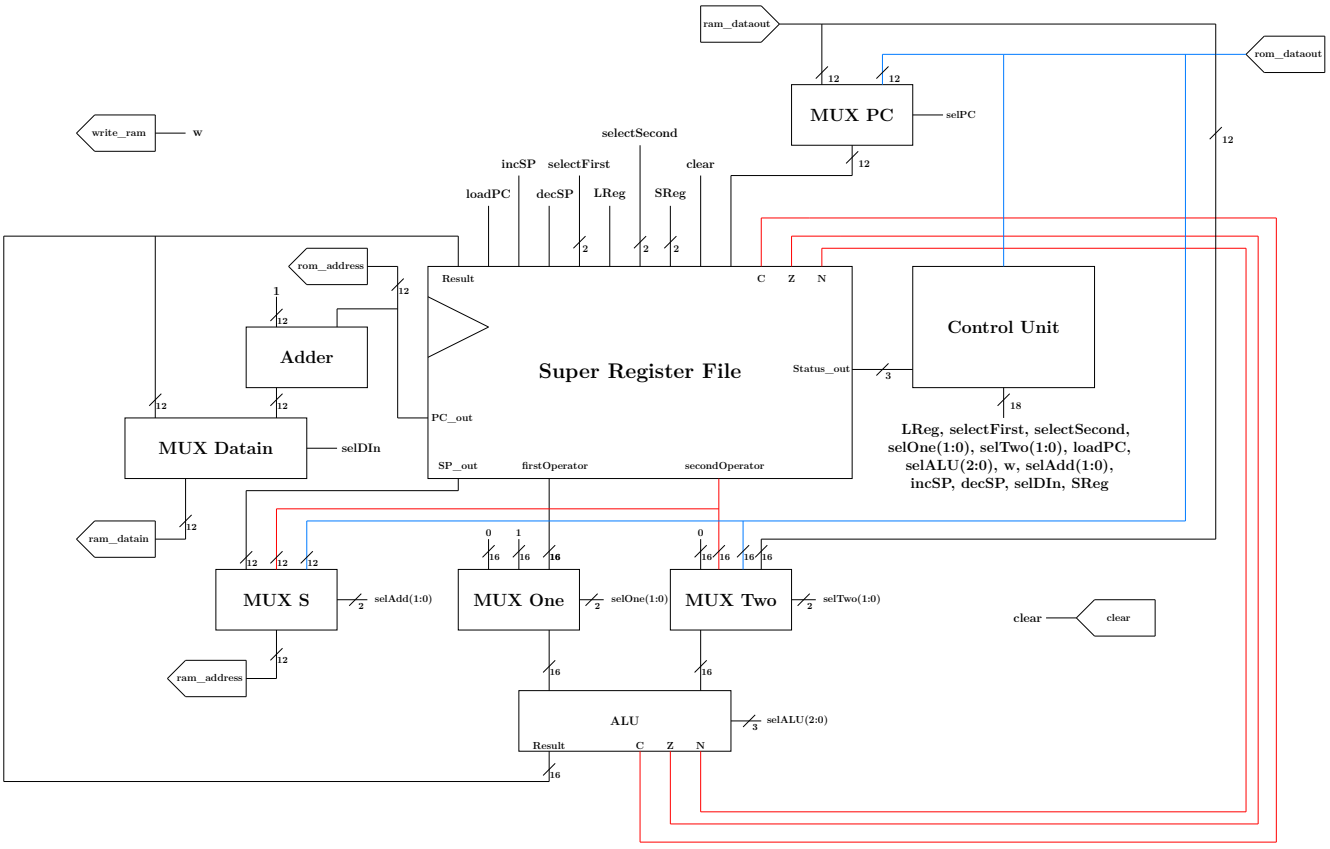

Figura 2: Diagrama interno de la CPU del computador básico de proyecto.

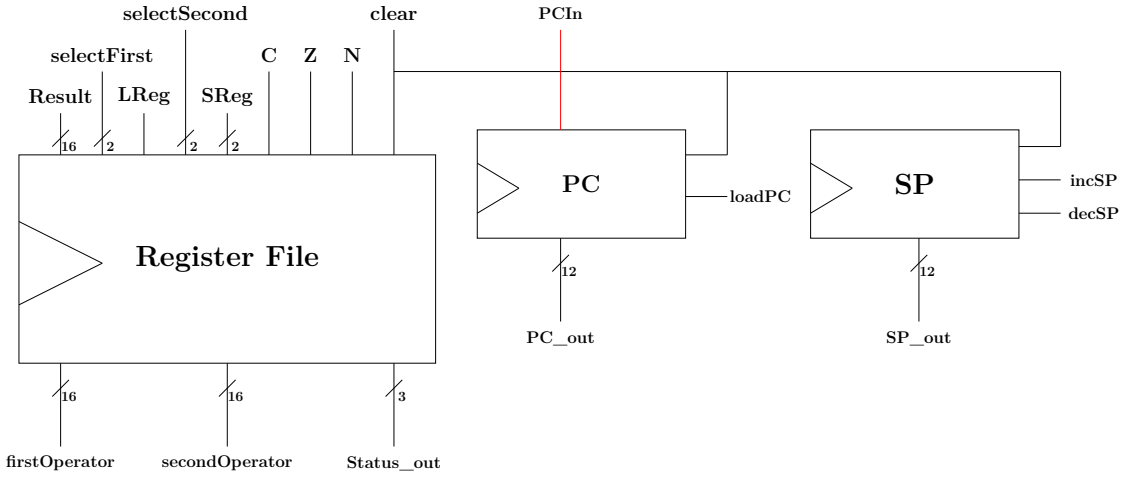

Figura 3: Diagrama interno del componente `Super Register File`.

Componentes a agregar

- El conjunto de registros `Register File` de la primera entrega **sin el registro contador `PC` en su interior**.
- Un **conjunto de registros** `Super Register File` implementado con **lógica combinacional**.
- Un **registro contador** `PC` para la dirección del puntero a la **instrucción en ejecución** de al menos 12 *bits* instanciado **dentro de `Super Register File`**.
- Un **registro contador** `SP` para la dirección del puntero al *stack* de al menos 12 *bits*.
- Una **unidad de control** `Control Unit` implementada con **lógica combinacional**.
- Un **registro contador** `SP` para la dirección del puntero al *stack* de al menos 12 *bits*.
- Un **multiplexor** `MUX PC` para la carga de `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 una unidad y entrega un resultado en 16 *bits*.

Software

En esta etapa, usará un ensamblador para traducir y transferir distintos programas en lenguaje de *Assembly* a la ROM de su computador y, de ese modo, poder ejecutarlos. Escriba los programas que estime convenientes para probar su arquitectura. Al final del enunciado, se encuentran algunos ejemplos. No está de más mencionar que estos son, como se señaló, **ejemplos** y no son *tests*. Debido a esto, el funcionamiento correcto de estos ejemplos no asegura que su computador sea 100 % funcional.

En la siguiente tabla, se describe cómo las palabras en la ROM son codificadas para representar las instrucciones. El ensamblador entregado se encargará de leer las instrucciones en *Assembly*, codificarlas y enviarlas al computador programable. En caso de que ninguno de los bits de la palabra de la ROM esté activo para la selección de un multiplexor, estos deben seleccionar una de sus señales de entrada de forma arbitraria a través de `with others` en Vivado. A continuación, se comparten algunos ejemplos:

- 00000000000000000000000000000000000010010100001001001000000001
  Guarda en el registro `C` el resultado de la operación `C + B`. En *Assembly*: `ADD C,B`.
- 00000000000101110000000000000000000100100100000000000100000001
  Guarda el literal 23 en el registro `D`. En *Assembly*: `MOV D,23`.
- 00000000000011111010010000000000000000000000000000000000000000
  Salta a la dirección 0x0F asociada a `label` si se cumple la condición de un salto por ser Menor. En *Assembly*: `JLT label`.
- 00000000000000000000000000010100101001000100000000010000000001
  Escribe en la dirección de memoria `Mem[SP]` el valor del registro `C` y decrementa el valor contenido en `SP` en una unidad. En *Assembly*: `PUSH C`.

Bits	Acción
0	1 cuando la operación de la ALU es Sumar
1	1 cuando la operación de la ALU es Restar
2	1 cuando la operación de la ALU es `AND`
3	1 cuando la operación de la ALU es `OR`
4	1 cuando la operación de la ALU es `XOR`
5	1 cuando la operación de la ALU es `NOT`
6	1 cuando la operación de la ALU es *shift right*
7	1 cuando la operación de la ALU es *shift left*
8	1 cuando se carga un registro
9	1 cuando el primer operando es `A`
10	1 cuando el primer operando es `B`
11	1 cuando el primer operando es `C`
12	1 cuando el primer operando es `D`
13	1 cuando el segundo operando es `A`
14	1 cuando el segundo operando es `B`
15	1 cuando el segundo operando es `C`
16	1 cuando el segundo operando es `D`
17	1 cuando se carga el registro `A`
18	1 cuando se carga el registro `B`
19	1 cuando se carga el registro `C`
20	1 cuando se carga el registro `D`
21	1 cuando en el `MuxOne` se selecciona el primer operando de `Super Register File`
22	1 cuando en el `MuxOne` se selecciona el Uno
23	1 cuando en el `MuxOne` se selecciona el Cero
24	1 cuando en el `MuxTwo` se selecciona el segundo operando de `Super Register File`
25	1 cuando en el `MuxTwo` se selecciona el Cero
26	1 cuando en el `MuxTwo` se selecciona el Literal
27	1 cuando en el `MuxTwo` se selecciona la salida de la RAM
28	1 cuando el `MuxDataIn` se selecciona la salida de la ALU
29	1 cuando el `MuxDataIn` se selecciona `PC+1`
30	1 cuando se habilita la escritura en la RAM
31	1 cuando en el `MuxS` se selecciona el Literal
32	1 cuando en el `MuxS` se selecciona el segundo operando de `Super Register File`
33	1 cuando en el `MuxS` se selecciona `SP`
34	1 cuando se incrementa el `SP`
35	1 cuando se decrementa el `SP`
36	1 cuando hay un salto incondicional
37	1 cuando hay un salto por Igualdad
38	1 cuando hay un salto por Desigualdad
39	1 cuando hay un salto por ser Mayor
40	1 cuando hay un salto por ser Mayor o Igual
41	1 cuando hay un salto por ser Menor
42	1 cuando hay un salto por ser Menor o Igual
43	1 cuando hay un salto por un Acarreo
44	1 cuando en el `MuxPC` se selecciona el Literal
45	1 cuando en el `MuxPC` se selecciona la salida de la RAM
61 al 46	Valor del Literal de 16 bits

Tabla 1: Tabla de palabras de control.

Conectividad con la placa

Para esta entrega, en Canvas contará con un archivo `Basys3.vhd`, el que deberá reemplazar por el existente en su proyecto. Este tiene todas las conexiones con la CPU ya establecidas para la interacción con la FPGA. Lo que muestra el *display* de la placa dependerá de un segmento de la señal `sw`, cuyo comportamiento se describe en la siguiente tabla:

`sw(15:12)`	Selección	Tipo
000	`firstOperator`	Salida
001	`secondOperator`	Salida
010	`ram_datain`	Salida
011	`ram_address`	Salida
100	`rom_dataout`	Salida
101	`rom_address`	Salida

Por otro lado, la velocidad del reloj será dada por el componente `ClockDivider`. Este ya se encuentra definido en el proyecto, y su velocidad será manipulada por otro segmento de la señal `sw`, cuyo comportamiento se describe a continuación.

`sw(1:0)`	Selección	Tipo
00	`Instantáneo - 25 MHz`	Entrada
01	`Rápido - 8 MHz`	Entrada
10	`Normal - 2 MHz`	Entrada
11	`Lento - 0.5 MHz`	Entrada

Cabe destacar que, a diferencia de la entrega anterior, todos los componentes se conectarán a la misma señal *clock* proveniente del componente `ClockDivider`, asegurando sincronía entre los registros y la memoria de datos en cada flanco de subida.

Finalmente, **no se utilizarán botones de la placa en esta entrega**.

Presentación

Se aceptarán envíos de *commits* hasta **el martes 13 de mayo a las 13:30 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 el ensamblador los *tests* 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* su proyecto en Vivado.

Puntaje

La nota será calculada utilizando 4 *tests*. Cada uno de estos debe correr de manera consecutiva, es decir, si su grupo no logra pasar el *test* 1, no se probará el *test* 2). Esto se debe a que el *test* `i+1` hará uso de las instrucciones del *test* `i`. Si no se logra pasar un *test* en su totalidad, no obstante, se probarán *sub-tests* de este, en donde se asignará puntaje parcial según su ejecución.

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

- ***Test* 1: Operaciones de la ALU sobre registros y literales (2 puntos)**
- ***Test* 2: Lectura y escritura en memoria (1.5 puntos)**
- ***Test* 3: Saltos incondicionales y condicionales (1.5 puntos)**
- ***Test* 4: Subrutinas y manejo de memoria *stack* (1 punto)**

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 admitida 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 20 de mayo). En esta, debe mostrar que arregló **todos** los errores de su entrega original. El código de la recuperación se presentará el día de su laboratorio correspondiente, donde deberá incluir los arreglos de esta entrega, pero **nada posterior a ella**.

Assembly

MOV	R1,R2	Guarda el valor de R2 en R1
	R1,Lit	Guarda un literal Lit en R1
	R1,(Dir)	Guarda el valor de Mem[Dir] en R1
	(Dir),R1	Guarda el valor de R1 en Mem[Dir]
	R1,(R2)	Guarda el valor de Mem[R1] en R2
	(R2),R1	Guarda el valor de R1 en Mem[R2]
	(R2),Lit	Guarda un literal Lit en Mem[R2]
ADD SUB AND OR XOR	R1,R2	Guarda el resultado de R1 op R2 en R1
	R1,Lit	Guarda el resultado de R1 op Lit en R1
	R1,(Dir)	Guarda el resultado de R1 op Mem[Dir] en A
	(Dir)	Guarda el resultado de A op B en Mem[Dir]
	R1,(R2)	Guarda el resultado de R1 op Mem[R2] en R1
NOT SHL SHR	R1	Guarda el resultado de op R1 en R1
	(Dir),R1	Guarda el resultado de op R1 en Mem[Dir]
	(R2),R1	Guarda el resultado de op R1 en Mem[R2]
INC	R1	Incrementa el valor de R1 en una unidad
	(Dir)	Incrementa el valor de Mem[Dir] en una unidad
	(R1)	Incrementa el valor de Mem[R1] en una unidad
DEC	R1	Decrementa el valor de R1 en una unidad
CMP	R1,R2	Ejecuta la instrucción SUB R1,R1 sin actualizar el valor en ningún registro
	R1,Lit	Ejecuta la instrucción SUB R1,Lit sin actualizar el valor en ningún registro
	R1,(Dir)	Ejecuta la instrucción SUB R1,(Dir) sin actualizar el valor en ningún registro
	R1,(R2)	Ejecuta la instrucción SUB R1,(R2) sin actualizar el valor en ningún registro
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
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
NOP		No hace cambios
PUSH	R1	Guarda el valor de R1 en Mem[SP] y decrementa SP en una unidad
POP	R1	Incrementa SP en una unidad y en el siguiente ciclo guarda el valor de Mem[SP] en R1
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 en el siguiente ciclo carga el valor de Mem[SP] en PC

Ejemplos

- Programa 1:

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

- Programa 2:

```
DATA:

CODE:           // Swaps

MOV A,3         // A = 3
MOV C,5         // C = 5

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

SUB A,C         // A = 2

XOR A,C         // |
XOR C,A         // |
XOR A,C         // | 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 D,A         // Resultado a D

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 C,3                 // C = 3
MOV D,5                 // D = 5

PUSH C                  // |
PUSH D                  // |
POP C                   // |
POP D                   // | 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
```