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

IIC2343 Arquitectura de Computadores

Saltos y Subrutinas

1 Motivación

Para que una máquina programable pueda ser referida como un computador, es necesario agregarle la capacidad de controlar el flujo de un programa para permitir que este realice decisiones e iteraciones. Debemos incorporar este elemento tanto de nivel del hardware del computador, como a nivel de software, para poder desarrollar programas complejos y completar un computador básico funcional.

2 Salto incondicional

2.1 Instrucción de salto incondicional

El código presentado en la tabla 1, genera la serie de Fibonacci almacenando los números alternadamente en los registros A y B. Para aumentar la cantidad de números de la secuencia, con las instrucciones y el computador que llevamos construido, la única alternativa es repetir las instrucciones de suma cuantas veces lo necesitemos.

Dirección	Instrucción	Operandos	A	B
0x00	MOV	A,0	**0**	?
0x01	MOV	B,1	0	**1**
0x02	ADD	A,B	**1**	1
0x03	ADD	B,A	1	**2**
0x04	ADD	A,B	**3**	2
0x05	ADD	B,A	3	**5**
0x06	ADD	A,B	**8**	5
0x07	ADD	B,A	8	**13**

Tabla 1: Programa que genera secuencia limitada de Fibonacci.

Una alternativa a esto sería tener la capacidad de saltar a una instrucción previa de manera de poder repetirla, sin perder el valor de los registros. Para poder lograr esto, debemos agregar una instrucción que permita saltar hacia otra instrucción, indicándole la dirección de memoria donde

está almacenada. Esta instrucción se conoce como **salto incondicional** y en nuestro assembly ocuparemos el nombre `JMP` ("'jump"' o salto) para referirnos a ella.

En la tabla 2, se observa el mismo programa antes descrito, pero esta vez con la instrucción de salto incondicional. Dado que el salto ocurre siempre, **este programa nunca se detiene** y por tanto es capaz de generar la serie de Fibonacci hasta el límite del rango disponible por los registros (i.e. 255).

Dirección	Instrucción	Operandos
0x00	MOV	A,0
0x01	MOV	B,1
0x02	ADD	A,B
0x03	ADD	B,A
0x04	JMP	0x02

Tabla 2: Programa que genera secuencia "'infinita"' de Fibonacci.

Para utilizar la instrucción de salto incondicional, debemos conocer la dirección de memoria a la cual vamos a saltar, lo que en programas largos puede ser complejo. Para facilitar esto se utiliza el concepto de **label** que es un indicador que se puede agregar en una línea del código assembly para referirse a la dirección de memoria asociada a esa línea. De esta manera, podemos reescribir el programa anterior de la siguiente forma:

Dirección	Label	Instrucción	Operandos
0x00		MOV	A,0
0x01		MOV	B,1
0x02	start:	ADD	A,B
0x03		ADD	B,A
0x04		JMP	start

Tabla 3: Programa que genera secuencia "'infinita"' de Fibonacci con label para salto.

2.2 Implementación en hardware

Para implementar la instrucción de salto incondicional en hardware, se debe tener la capacidad de modificar la instrucción que se va ejecutar. La información de que instrucción se ejecuta está en el program counter, por lo que para agregar la capacidad de salto incondicional es necesario agregar la capacidad de cargar el program counter con el parámetro de la instrucción. Par esto es necesario agregar una nueva señal del control Lpc que indicará si el program counter está en modo carga $Lpc = 1$ o en modo incremento $Lpc = 0$. La figura 1 muestra el diagrama del computador incluyendo la capacidad de salto incondicional.

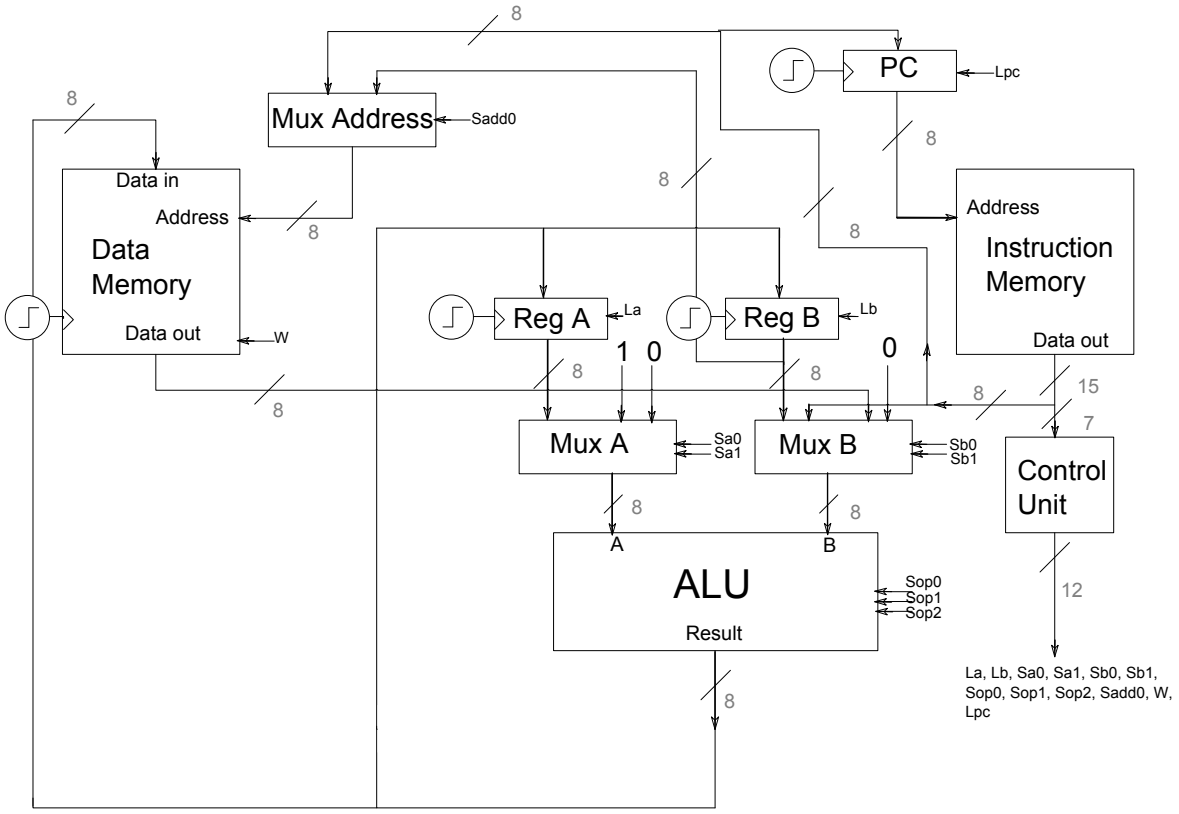

Figure 1: Computador con salto incondicional.

3 Salto condicional

3.1 Instrucciones de salto condicional para comparación

El salto incondicional no es suficiente para entregar la capacidad de decisión al computador. Para lograr esto es necesario agregar la opción de saltar condicionalmente, dependiendo de alguna condición. Un primer tipo de instrucciones de salto condicional corresponde a instrucciones que salten dependiendo del resultado de una comparación aritmética: $a == b$, $a! = b$, $a > b$, $a < b$,$a \geq b$,$a \leq b$.

Para lograr esto, es necesario primero agregar una instrucción que permita ejecutar la comparación, para lo cual es necesario reescribir las comparaciones aritméticas de la siguiente forma $a - b == 0$, $a - b! = 0$, $a - b > 0$, $a - b < 0$,$a - b \geq 0$,$a - b \leq 0$. Se observa que en todos los casos, la operación de comparación corresponde a restar el valor de a con el de b, por lo cual agregaremos la instrucción `CMP A,B` que ejecuta la resta entre los registros A y B y **no** almacena el resultado.

El siguiente paso luego de ejecutar la comparación es identificar si el resultado fue 0 o no, para los casos de igualdad, y si fue negativo o no, para los casos de mayor y menor que. De esta forma, se agregarán las siguientes instrucciones:

- `JEQ`: "'Jump equal"', salta en caso de igualdad $a == b$ es decir cuando el resultado de la ultima operación fue cero ($Z = 1$).
- `JNE`: "'Jump not equal"', salta en caso de desigualdad $a! = b$ es decir cuando el resultado de

la ultima operación fue distinto cero ($Z = 0$).

- `JGT`: "'Jump greater than"', salta en caso de que $a > b$ es decir cuando el resultado de la ultima operación no fue cero ni negativo ($Z = 0, N = 0$).
- `JLT`: "'Jump less than"', salta en caso de de que $a < b$ es decir cuando el resultado de la ultima operación fue negativo ($N = 1$).
- `JGE`: "'Jump greater or equal than"', salta en caso de $a \geq b$ es decir cuando el resultado de la ultima operación no fue negativa ($N = 0$).
- `JLE`: "'Jump less or equal than"', salta en caso de $a \leq b$ es decir cuando el resultado de la ultima operación fue cero o negativa ($Z = 1, N = 1$).

La tabla con las instrucciones en detalle se presenta a continuación:

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,(B)	A-Mem[B]		
	A,Lit	A-Lit		CMP A,0
	A,(Dir)	A-Mem[Dir]		CMP A,(label)
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=0	JGT label
JLT	Dir	PC = Dir	N=1	JLT label
JGE	Dir	PC = Dir	N=0	JGE label
JLE	Dir	PC = Dir	Z=1 o N=1	JLE label

Tabla 4: Instrucciones de salto condicional para comparación.

Con estas instrucciones, ya es posible desarrollar programas más complejos como por ejemplo un programa simple de multiplicación, como se observa en la tabla 5.

Dirección	Label	Instrucción	Operandos
0x00		MOV	A,3
0x01		MOV	B,5
0x02	mult:	ADD	B,5
0x03		SUB	A,1
0x04		CMP	A,1
0x05		JNE	mult

Tabla 5: Programa que multiplica 3×5.

3.2 Condition codes para comparación

Para poder ejecutar las comparaciones anteriores se observa que se requiere saber si el resultado de la operación anterior fue cero o si fue negativa. Esta es información que se puede obtener de la ALU, y se conoce como **condition codes** o códigos de condición:

- Zero (**Z**): El código de condición cero (Z por su nombre en ingles) se puede obtener a partir del resultado de la ALU, haciendo un **or** entre todos los bits y luego un **not** de la salida del or. De esta forma, sólo se obtendrá una salida final de 1 cuando todos los bits del resultado de la ALU sean 0.
- Negative (**N**): El código de condición negativo se puede obtener de manera simple, tomando el bit más significativo del resultado. Dado que las operaciones se realizan en complemento a 2, si el bit más significativo es 1, será un número negativo, si es 0, positivo.

3.3 Instrucciones de salto condicional por excepción

Un tipo particular de instrucciones de salto condicional son aquellas cuando ocurre algún tipo de caso de excepción. En particular nos interesa saber cuando la operación realizada resultó en un número que sobrepasa la representación válida por el tamaño del resultado. Hay dos casos a considerar: en operaciones de número si signos, cuando ocurre un **carry** se está indicando que el resultado real no es el que queda almacenado en la salida resultado; en operaciones de número con signo, cuando ocurre un **overflow** es decir cuando una operación que debía resultar en un número positivo resulta en un número negativo o viceversa. Las instrucciones para saltar en estos casos son:

- `JCR`: "'Jump carry"', salta en caso de que ocurra un carry ($C = 1$).
- `JOV`: "'Jump overflow"', salta en caso de que ocurra un overflow ($V = 1$).

El detalle se las instrucciones se presenta a continuación:

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label

Tabla 6: Instrucciones de salto condicional por excepción.

Con estas instrucciones es posible implementar programas que identifiquen cuando ocurrió un carry o un overflow y que realicen algún mecanismo que maneje estas condiciones excepcionales, como se observa en las tablas 7 y 8.

3.4 Condition Codes para excepciones

Al igual que en los casos de comparación, se necesita que la ALU entregue la información de carry u overflow:

Dirección	Label	Instrucción	Operandos
0x00	start:	MOV	A,0
0x01		MOV	B,1
0x02	acum:	ADD	A,B
0x03		JOV	exc
0x04		JMP	acum
0x05	exc:	JMP	start

Tabla 7: Programa que maneja excepción de overflow.

Dirección	Label	Instrucción	Operandos
0x00	start:	MOV	A,0
0x01		MOV	B,1
0x02	acum:	ADD	A,B
0x03		JC	exc
0x04		JMP	acum
0x05	exc:	JMP	start

Tabla 8: Programa que maneja excepción de carry.

- Carry (**C**): el bit de carry es simplemente la salida "'carry out"' del sumador restador, ya que solo con estas operaciones puede ocurrir un carry.
- Overflow (**V**): el bit de overflow es más complejo que el carry, ya que ocurre en distintas circunstancias dependiendo del signo de las entradas, la operación y de la salida. A continuación se presentan los casos posibles para que ocurra un overflow:

Operación	A	B	Resultado	Ejemplo (1 byte)
$A+B$	≥ 0	≥ 0	< 0	$127+4=-125$
$A+B$	< 0	< 0	≥ 0	$-127+-4=125$
$A-B$	≥ 0	< 0	< 0	$127--4=-125$
$A-B$	< 0	≥ 0	≥ 0	$-127-4=125$

Tabla 9: Posibles casos de overflow.

Para implementar eso, es necesario agregar un circuito combinacional a la ALU que dependiendo de las entradas, operación y salidas entregue un 1 o un 0 para indicar el overflow.

3.5 Implementación en hardware

La implementación en hardware del salto condicional requiere, como se señaló anteriormente, que la ALU entregue los 4 bits de los condition codes, agregando las compuertas descritas previamente. Además de esto, dado que el salto se realiza con las condiciones ocurridas en la operación anterior, es necesario almacenar los condition codes en un registro, habitualmente denominado **status register**.

Este registro se conecta a su vez a la unidad de control, la cual se encargará de determinar si se activa la señal *Lpc* dependiendo si se cumple o no la condición asociada a la instrucción de salto correspondiente.

El diagrama del computador con la capacidad de salto condicional se muestra a continuación:

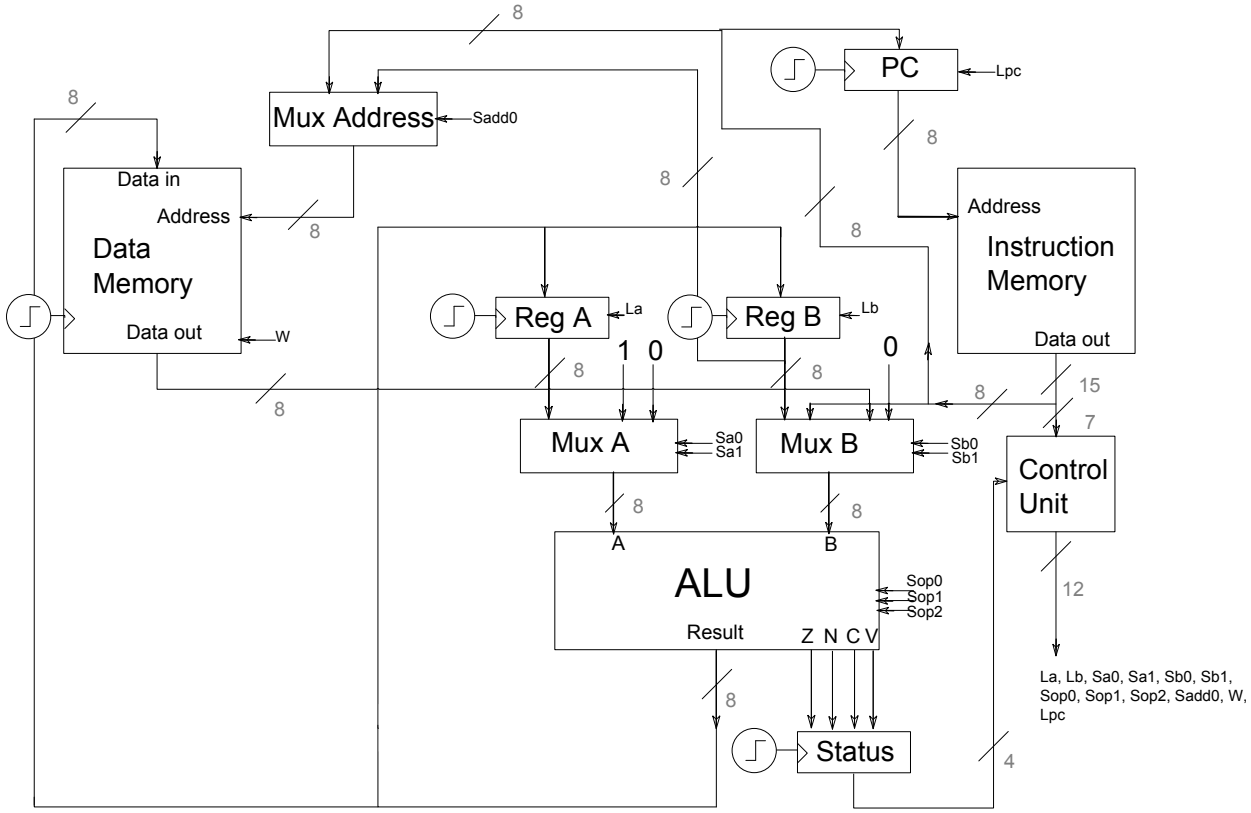

Figure 2: Computador con salto condicional.

4 Resumen instrucciones de salto

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CMP	A,B	A-B		
	A,Lit	A-Lit		CMP A,0
JMP	Dir	PC = Dir		JMP end
JEQ	Dir	PC = Dir	Z=1	JEQ label
JNE	Dir	PC = Dir	Z=0	JNE label
JGT	Dir	PC = Dir	N=0 y Z=0	JGT label
JLT	Dir	PC = Dir	N=1	JLT label
JGE	Dir	PC = Dir	N=0	JGE label
JLE	Dir	PC = Dir	Z=1 o N=1	JLE label
JCR	Dir	PC = Dir	C=1	JCR label
JOV	Dir	PC = Dir	V=1	JOV label

Tabla 10: Instrucciones de salto del computador básico.

5 Subrutinas

Supongamos que se desea escribir un código en assembly para calcular el producto punto entre un vector (a, b) y un vector (c, d). Por definición, el producto punto se calcula como: $(a, b) \cdot (c, d) = ac+bd$. Para realizar la función entonces necesitamos realizar dos multiplicaciones, un pseudocódigo en lenguaje de alto nivel para esta operación se muestra a continuación:

```
byte[] vector1 = new byte[]{2,3};
byte[] vector2 = new byte[]{4,5};
byte prodPunto = 0;

prodPunto += mult(vector1[0], vector2[0]);
prodPunto += mult(vector1[1], vector2[1]);
```

Como se vio anteriormente, con el computador básico y las instrucciones vistas ya es posible desarrollar un programa que multiplique dos números como el siguiente:

```
DATA:
    var1    3
    var2    4
    res      0
    i        0
CODE:
    start:  MOV  A,(res)
            ADD  A,(var2)
            MOV  (res),A
            MOV  A,(i)
            ADD  A,1
            MOV  (i),A
            MOV  B,(var1)
            CMP  A,B
            JLT  start
```

Una opción para utilizar este código en el cálculo del producto punto sería reescribirlo dos veces, para las dos multiplicaciones. El problema de esto es que por un lado, debemos adaptar el código a cada multiplicación, y además escribirlo dos veces, es decir, desperdiciar espacio en al memoria de instrucciones. Una mejor alternativa sería poder reutilizar el mismo código de la multiplicación para las dos multiplicaciones del producto punto sin tener que reescribirlo, es decir poder llamar al trozo de código de la multiplicación durante la ejecución del programa del producto punto, lo que se conoce como una **subrutina**.

Existen tres elementos necesarios para poder implementar una subrutina: poder entregarle **parámetros**, poder recibir un **valor de retorno** y poder hacer la **llamada a la subrutina**, es decir poder saltar a la subrutina y volver luego de que termine a una **dirección de retorno** que apunte a la siguiente instrucción del programa.

```
DATA:
      vector1      2
                   3
      vector2      4
                   5
      prodPunto    0

CODE:
      ADD prodPunto, MULT(vector1[0],vector2[0])
      ADD prodPunto, MULT(vector1[0],vector2[0])
```

5.1 Parámetros

Para poder entregarle parámetros a la subrutina, es necesario almacenarlos en algún lugar al cual está pueda acceder. En el computador básico existen dos lugares donde se pueden almacenar datos: **registros** y **variables en memoria**.

- **Registros**: una primera opción para pasar parámetros es almacenarlos en los registros y que luego la subrutina se encargue de obtenerlos. Para lograr esto, se debe saber a priori que registros se ocuparán para que parámetros, de manera de cargar los que efectivamente ocupará la subrutina. Aunque está implementación es simple, la principal desventaja está en el número limitado de registros, que impide entregar más parámetros que los registros disponibles.
- **Variables**: otra opción para el paso de parámetros es almacenarlos en memoria y que la subrutina los obtenga de ahí. Al igual que en el caso de los registros, se debe saber a priori que variables se ocuparán para que parámetros, de manera de cargar las que efectivamente ocupará la subrutina. La ventaja respecto a los registros están en la mayor disponibilidad de espacio.

5.2 Valor de retorno

Al igual que con los parámetros, el valor de retorno debe ser almacenado en algún lugar que pueda ser accedido tanto por la subrutina, es decir, **registros** y **variables en memoria**.

- **Registros**: al igual que con los parámetros, para ocupar registros como valor de retorno hay que saber a priori que registro ocupará la subrutina. La diferencia con los parámetros es que como el retorno es sólo un valor, es factible realizar esto, a pesar de tener pocos registros.
- **Variables**: otra opción para el retorno es ocupar una variable en memoria. Al igual que en el caso de los registros, se debe saber a priori que variables se ocuparán para que parámetros, de manera de cargar las que efectivamente ocupará la subrutina.

5.3 Llamada y dirección de retorno

Además de poder entregar parámetros y recibir retorno del parte de la subrutina, debemos de alguna forma pasar de la ejecución del código principal a la ejecución del código de esta. Supongamos el siguiente código que realiza el producto punto y ocupa paso de parámetro y retorno ocupando variables:

```
DATA:
      vector1     2
                  3
      vector2     4
                  5
      prodPunto   0
      var1        0
      var2        0
      res         0
      i           0

CODE:
                  MOV A, (vector1)
```

```
                    MOV (var1), A
                    MOV A, (vector2)
                    MOV (var2), A
                    //Llamar a subrutina y retornar
                    MOV A, (prodPunto)
                    ADD A, (res)
                    MOV (prodPunto), A

                    MOV B, vector1
                    INC B
                    MOV A, (B)
                    MOV (var1), A
                    MOV B, vector2
                    INC B
                    MOV A, (B)
                    MOV (var2), A
                    //Llamar a subrutina y retornar
                    MOV A, (prodPunto)
                    ADD A, (res)
                    MOV (prodPunto), A

                    JMP end

mult:
start:      MOV  A,(res)
            ADD  A,(var2)
            MOV  (res),A
            MOV  A,(i)
            ADD  A,1
            MOV  (i),A
            MOV  B,(var1)
            CMP  A,B
            JLT  start

end:
```

El primer paso para poder acceder al código de la subrutina es que se encuentre en la memoria de instrucciones. En el ejemplo anterior se observa que el código de la subrutina se incluyó al final, y se agregó un label `mult` para indicar el inicio de la subrutina. Llamar a la subrutina, entonces, se puede hacer simplemente saltando al label correspondiente con una instrucción `JMP mult`. El problema está en el retorno: al finalizar la ejecución de la subrutina, ¿cómo se sabe a donde volver? No es posible agregar otro JMP, ya que no se puede saber a que dirección se va a retornar (depende de donde se llamo a la subrutina). Para poder realizar correctamente el retorno es necesario **almacenar el valor del program counter** antes de realizar la llamada, para luego volver a cargarlo luego de retornar de la subrutina. En realidad, lo que nos interesa es guardar el valor del program counter incrementado en 1, dado que al volver de la subrutina queremos ejecutar la siguiente instrucción, no la actual.

Para lograr la llamada a la subrutina, entonces, debemos agregar el soporte de hardware necesario que permita almacenar el valor del program counter incrementado en 1 en memoria (al hacer la llamada) y para poder cargarlo desde memoria (al retornar), lo que se observa en la figura 3.

Con esta modificación de hardware podemos agregar dos instrucciones para realizar el llamado a subrutina y retorno de ellas: la instrucción `CALL dir` la cual almacena el PC en memoria y salta al label *dir* donde estará almacenada la subrutina; y la instrucción `RET` la cual se debe agregar al final de la subrutina y salta de vuelta a la dirección almacenada previamente del PC.

Con estas dos instrucciones se puede reescribir el código completo del programa que calcula el producto punto de la siguiente forma:

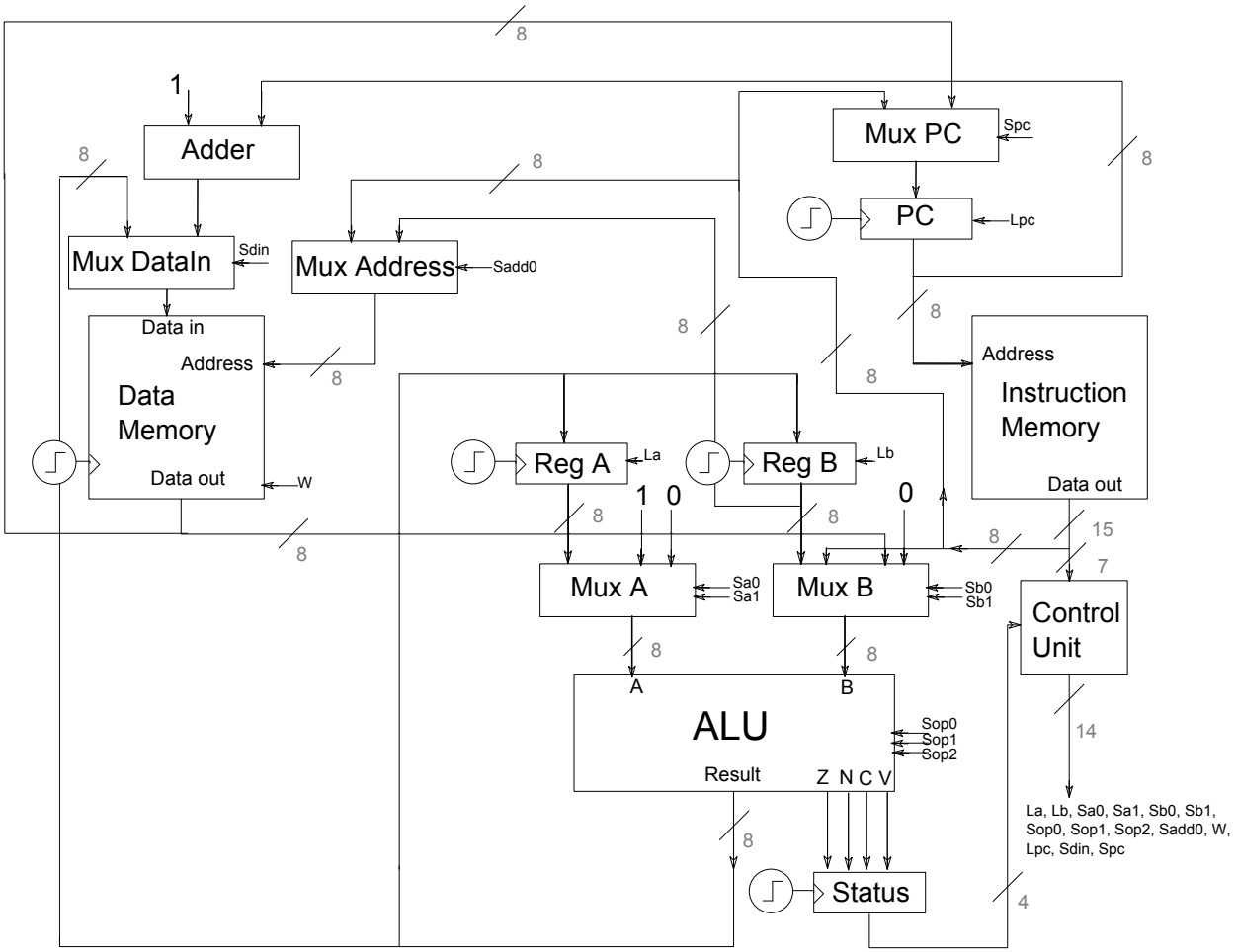

Figure 3: Almacenamiento del program counter

```
DATA:
      vector1     2
                  3
      vector2     4
                  5
      prodPunto   0
      var1        0
      var2        0
      res         0
      i           0

CODE:
                  JMP  init
      mult:
                  MOV A, 0
                  MOV (res), A
                  MOV A, 0
                  MOV (i), A
      start:      MOV  A,(res)
                  ADD  A,(var2)
                  MOV  (res),A
                  MOV  A,(i)
```

```
                ADD  A,1
                MOV  (i),A
                MOV  B,(var1)
                CMP  A,B
                JLT  start
                RET

init:           MOV A, (vector1)
                MOV (var1), A
                MOV A, (vector2)
                MOV (var2), A
                CALL mult
                MOV A, (prodPunto)
                ADD A, (res)
                MOV (prodPunto), A

                MOV B, vector1
                INC B
                MOV A, (B)
                MOV (var1), A
                MOV B, vector2
                INC B
                MOV A, (B)
                MOV (var2), A
                CALL mult
                MOV A, (prodPunto)
                ADD A, (res)
                MOV (prodPunto), A
```

6 Almacenamiento de la dirección de retorno

6.1 Stack pointer

Para completar el manejo de subrutinas falta aún un elemento: en que parte de la memoria se guarda el valor del PC al hacer el llamado a la subrutina. Una opción es que el programador se encargara de indicar una dirección (por ejemplo almacenándola en el registro B de direccionamiento indirecto) previo a la llamada y previo al retorno. El problema de esto es que agrega complejidad a la programación.

Para solucionar este problema, se agrega un nuevo registro al computador denominado **stack pointer** o SP el cual comienza cargado con el valor de la **última dirección de la memoria**. La idea será entonces ocupar esa dirección para realizar la carga del program counter al ejecutar la instrucción `CALL`. Luego, al llamar a la instrucción `RET` se leerá el valor de memoria apuntado por el stack pointer y se cargará en el program counter para volver al flujo normal del programa.

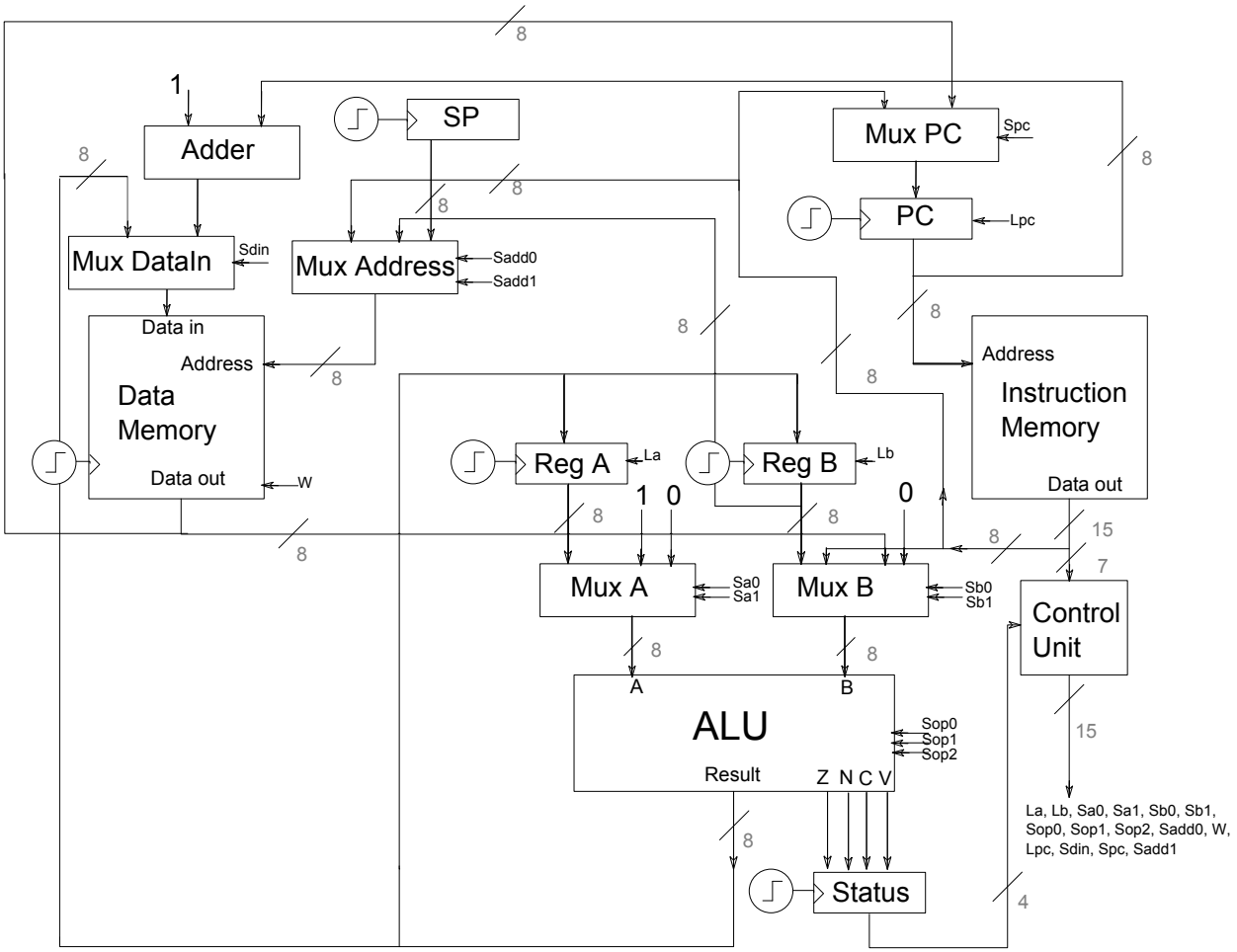

Figure 4: Registro stack pointer agregado al computador básico

6.2 Subrutinas anidadas

El tener un registro con un valor fijo para almacenar la dirección de retorno simplifica el manejo de llamadas a subrutinas simples. Sin embargo, persiste aún un problema, que se puede observar en un código como el siguiente:

```
                ...
                ...
                CALL subrutina1
                ...
                ...
subrutina1:
                CALL subrutina2
                ...
                RET
                ...
                ...
subrutina2:      ...
                ...
                RET
```

En el código se llama a una subrutina y **dentro de ésta** se llama a una segunda subrutina. ¿Que pasa con el valor de la primera dirección de retorno? Al hacer el llamado a la *subrutina*2 estamos escribiendo en el SP la dirección de la RAM donde se almacenará la dirección de retorno de la subrutina. El problema es que con esto perdemos el valor de la dirección de retorno asociada a la *subrutina*1.

Para solucionar esto necesitamos tener más espacio de almacenamiento para poder ir "'apilando"' las direcciones de retorno de posibles llamdas anidadas de subrutinas. Dado que el stack pointer ya está direccionando la memoria RAM es lógico querer armar esta pila de valores en la misma RAM. Para lograr esto, se le agrega al stack pointer la capacidad de incrementar y decrementar su valor (como un contador up/down), agregando dos nuevas señales de control (figura 5).

De esta forma, al hacer el llamado a la instrucción `CALL subrutina1`, además de guardar en Mem[SP] el valor del PC + 1, **se debe decrementar en 1 el valor del SP**. De está manera, la siguiente instrucción de llamada a subrutina, `CALL subrutina1` va a guardar el valor en la posición Mem[SP - 1] y por tanto no sobreescribirá la dirección de retorno. Adicionalmente a esto, la instrucción de retorno `RET` se encargará ahora además de almacenar el valor de Mem[SP] en el PC, de **incrementar el valor del SP** asegurándose de volver a la posición correcta de éste.

Hacer estas modificaciones tiene, sin embargo, un costo involucrado. Las instrucciones `CALL` y `RET` tienen ahora que realizar diversas funciones en un orden específico. Para el caso de `CALL` estás son:

1. Guardar PC + 1 en la posición actual del SP
2. Decrementar en 1 el SP
3. Guardar la dirección de la subrutina en PC

Si el SP actualiza el valor decrementado en flanco de subida, es posible realizar estas tres operaciones en un sólo clock: mientras se esté en el ciclo del clock, el valor PC + 1 se calcula y queda en la entrada de datos de la memoria esperando a almacenarse, el SP estará entregando como salida su valor actual y la dirección de la subrutina estará a la entrada del PC. Cuando ocurra el próximo flanco, la memoria va a almacenar el valor de PC + 1, el SP va a decrementar y el PC va a almacenar el valor de la dirección.

Para el caso de la instrucción `RET`, sin embargo, la situación es distinta. Los pasos a ejecutar son:

1. Incrementar en 1 el SP
2. Guardar el valor de memoria apuntado por el SP incrementado en PC

El problema está que en este caso es necesario esperar que el SP se incremente antes de cargar el PC con el valor de memoria. Si el incremento del SP ocurre en flanco de subida, no es posible realizar estas dos operaciones en un mismo ciclo. La solución para esto es incorporar la noción de instrucciones de múltiples ciclos: la instrucción `RET` se debe ejecutar en dos ciclos, en un primer ciclo se incrementa el SP, y en un segundo se guarda el valor de memoria apuntado por SP en PC. De esta forma el assembler al momento de transformar la instrucción RET en lenguaje de máquina, debe preocuparse de gener dos instrucciones que implementen las señales de control para los dos pasos.

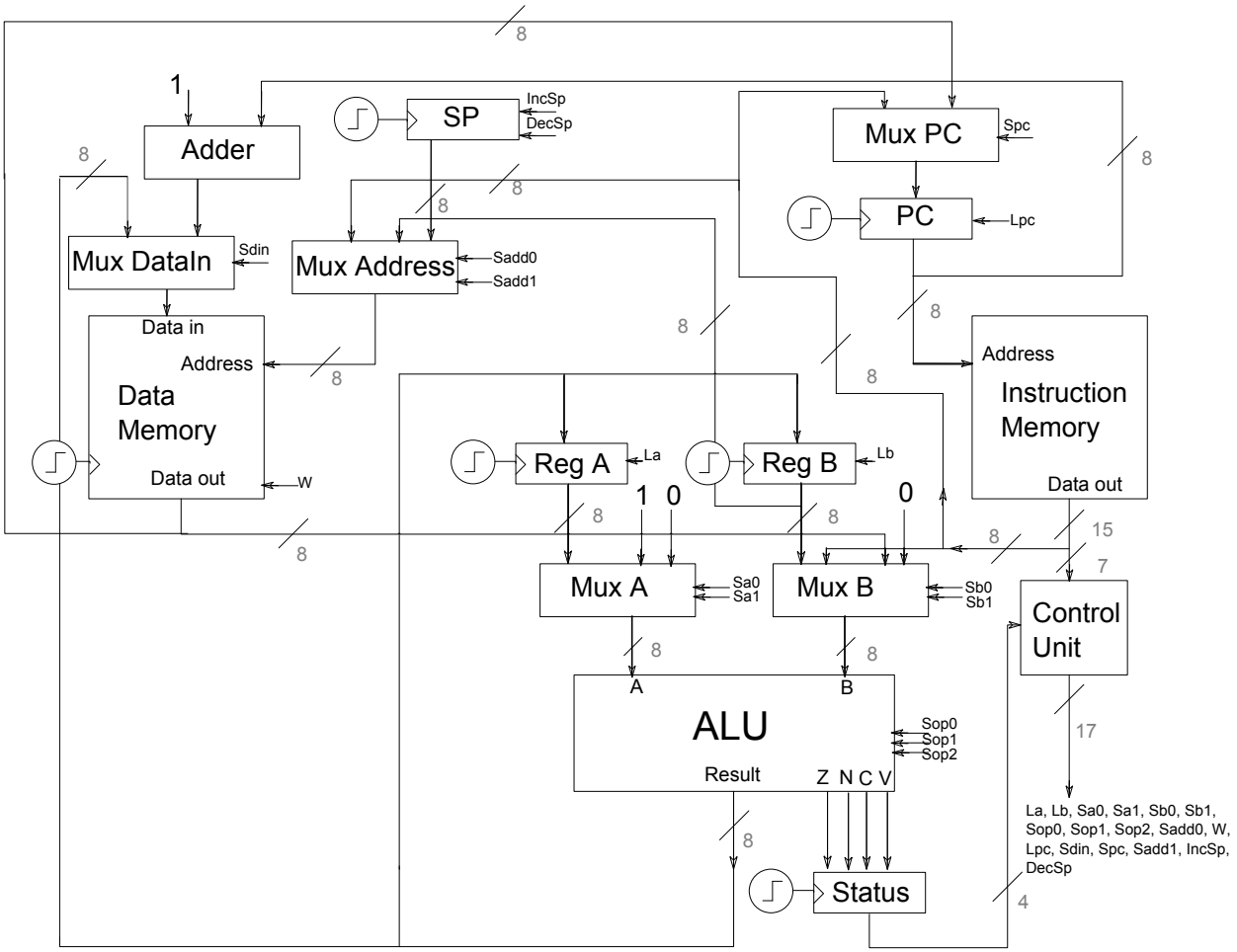

Figure 5: Agregadas señales de incremento y decremento al stack pointer

6.3 Stack de uso general

El segmento de memoria que es apuntado por el stack pointer se conoce como **stack** o pila ya que se puede pensar como una estrucura en la cual se van apilando valores (las direcciones de retorno) y luego se van sacando desde el que está mas arriba hacia abajo. El stack puede ser usado también como mecanismo para guardar otros valores en la memoria además de la dirección de retorno.

En particular, en ciertas circunstancias es útil poder guardar el valor de los registros de la CPU antes de hacer un salto a una subrutina. Por ejemplo, en el siguiente trozo de programa:

```
        ...
        ...
        MOV  A,5
        MOV  B,3
        CALL subrutina
        ADD  A,B
        ...
        ...
```

luego de retornar de la llamada a la subrutina, es imposible saber que valores tienen los registros

A y B, ya que pueden haber sido modificados por ésta. Para permitir guardar los valores de los registros y no perder su valores, se agregan dos instrucciones: `PUSH Reg` que permiten almacenar un registro en el stack, en la posición indicada por el stack pointer decrementando luego el valor de éste; y `POP Reg` que permite obtener desde el stack el valor apuntado por el stack pointer, almacenarlo en el registro indicado, e incrementar el SP.

De esta forma el ejemplo anterior podríamos reescribirlo así para no perder los valores de los registros:

```
...
...
MOV  A,5
MOV  B,3
PUSH A
PUSH B
CALL subrutina
POP  B
POP  A
ADD  A,B
```

Para la instrucción `POP`, ocurre el mismo problema que con la instrucción `RET` y por tanto es necesario implementarla en dos pasos.

6.4 Instrucciones de subrutinas y stack

A continuación se detallan todas las instrucciones de manejo de subrutina y stack agregadas al computador básico:

Instrucción	Operandos	Operación	Condiciones	Ejemplo de uso
CALL	Dir	Mem[SP] = PC + 1, SP- -, PC = Dir		CALL func
RET		SP++		-
		PC = Mem[SP]		-
PUSH	A	Mem[SP] = A, SP- -		-
PUSH	B	Mem[SP] = B, SP- -		-
POP	A	SP++		-
		A = Mem[SP]		-
POP	B	SP++		-
		B = Mem[SP]		-

7 Ejercicios

- ¿Como se podría implementar en el computador básico la opción de que este avise luego de realizar una operación cuando el resultado es par o impar?
- Implemente el siguiente código ocupando el assembly del computador básico visto hasta esta clase:

```
int b = 2;
for (int a=0; a<10; a++)
   b += 5;
```

- Implemente el siguiente código ocupando el assembly del computador básico visto hasta esta clase:

```
int b = 2;
int a = 3;
if (a < b)
   b = 5;
 else if(b >= a)
   a = 3;
 else
   a = b;
```

- Generalice el ejemplo del producto punto para que permita calcularlo entre dos vectores de largo arbitrario. Asuma que el largo de los vectores estará almacenado en una variable `n`.
- Escriba una subrutina que calcule la potencia n de un número x, es decir el valor de x^n, para dos variables `x` y `n` almacenadas en la memoria de datos. Utilice la subrutina de multiplicación para realizar su cálculo.
- Escriba un programa que calcula el cuadrado de binomio $(a + b)^2$ para dos variables `a` y `b` almacenadas en la memoria de datos. Utilice las subrutinas de multiplicación y potencia.

8 Referencias e información adicional

- Hennessy, J.; Patterson, D.: Computer Organization and Design: The Hardware/Software Interface, 4 Ed., Morgan-Kaufmann, 2008. Chapter 3: Arithmetic for Computers, Chapter 4: The processor.