Programación en lenguaje ensamblador
CODE-2

31 Marzo 2011

1. Introducción

En un principio, los programas eran codificados directamente en binario, obteniendo programas eficientes, en el sentido de que hacían exactamente lo que se indicaba en las instrucciones, ni más ni menos. Pero ésto tenía la gran desventaja de resultar poco legible, y en consecuencia, dificultaba la realización y depuración de programas que involucraran un elevado número de instrucciones.

Luego comenzó a implementarse la programación en lenguaje ensamblador, el cual básicamente asigna a cada instrucción un nemónico, permitiendo lograr programas más legibles que los hechos en binario, si bien igual de eficientes, Y pudiendo obtener un mapeo uno a uno entre un código y otro (la conversión es trivial).

Para introducir al alumno en la arquitectura de un procesador y en las nociones de la programación en ensamblador, se presenta aquí un modelo de procesador general simplificado: la Computadora Didáctico Elemental o CODE-2.

El procesador consta de una unidad aritmético lógica (ALU) que permite realizar sumas, restas, desplazamientos y la operación lógica NAND. Tiene asociado un grupo de biestables indicadores o flags de resultado (Z: Zero, S: Sign, C: Carry, V: Overflow), que dan información en base al último resultado producido por la ALU, útil para efectuar saltos condicionales.

Utiliza una memoria principal de 2^{16}celdas (palabras) de 16 bits cada una, utilizada para guardar datos e instrucciones, y un registro contador de programa (PC) que guarda la dirección de la próxima instrucción a ejecutar.

El procesador también dispone de 16 registros de propósito general: R0..RF, utilizados para guardar datos auxiliares.

formatos de instrucciones

La Computadora Didáctico Elemental CODE-2 es un procesador de tipo RISC [1], que consta de un conjunto de 16 instrucciones de 16 bits de ancho (constante, por ser RISC), que siguen uno de entre cinco formatos diferentes.

Todos los formatos están divididos en campos, y comienzan con un campo de 4 bits denominado *codop* que, tal como sugiere su nombre, corresponde al código de operación asociado a la instrucción. Esto indica que el juego de instrucciones del CODE-2 es a lo sumo de tamaño 16, tal como ya se ha señalado.

2. Formatos de datos e instrucciones

- Instrucciones que contienen sólo código de operación.

[1] RISC: Reduced Instruction Set Computer (Computadora de Conjunto de Instrucciones Reducido). Son aquellos procesadores cuyo conjunto de instrucciones proveen la funcionalidad mínima necesaria, al contrario de los CISC (Complete Instruction Set Computer).

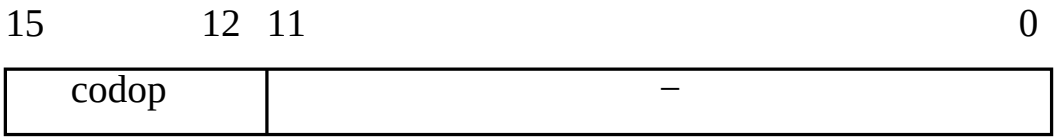

Figura 1: formato de instrucción F0.

- Instrucciones que además del campo de código de operación, tienen un campo *Rx* de 4 bits para especificar un registro, pudiendo así referenciar los registros del *R0* al *RF*.

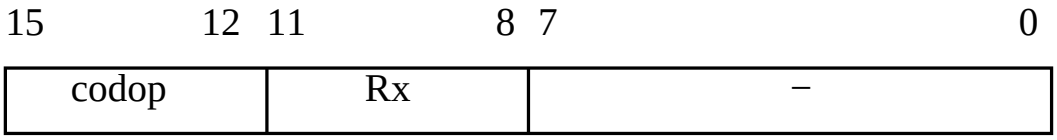

Figura 2: formato de instrucción F1.

- Instrucciones que aparte del código de operación, tienen un campo *cnd* de 4 bits para especificar una condición (en instrucciones de salto y de llamadas a subrutinas). El campo *cnd* es un campo de control, que puede pensarse como una extensión del campo *codop*.

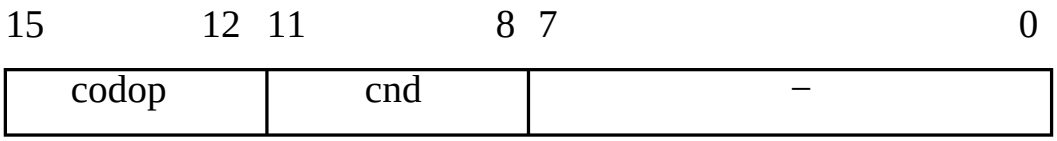

Figura 3: formato de instrucción F2.

- Instrucciones que cuentan con el campo *codop*, más un campo *Rx* de 4 bits para especificar un registro, y otro campo *v* de 8 bits para determinar un valor inmediato (dato o dirección).

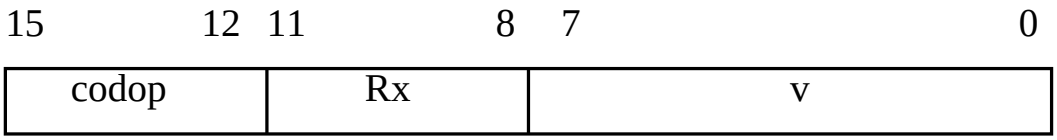

Figura 4: formato de instrucción F3.

- Instrucciones que contienen 4 campos de 4 bits cada uno: *codop*, Rx (registro destino del resultado), *Ry* (registro que contiene el primer operando) y Rz (registro que contiene el segundo operando).

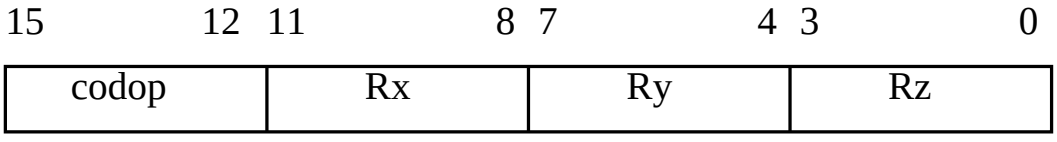

Figura 5: formato de instrucción F4.

Para la representación de los datos se utiliza el sistema CB(2,16), obteniendo un rango de representación de $\left[-\frac{2^{16}}{2}, \frac{2^{16}}{2}-1\right] = [-32768, 32767]$.

3. Repertorio de instrucciones

Cada instrucción tiene un nombre que facilita su uso (nemónico), sin necesidad de recordar el código de operación, y puede tener entre cero y tres parámetros, dependiendo del formato de instrucción que siga.

Conociendo la frecuencia del reloj (F) del procesador y el número de ciclos de reloj (n_i) que demanda la instrucción i, puede obtenerse el tiempo que tarda en ejecutarse. Luego, sumando lo que tarda cada instrucción de un programa, puede saberse el tiempo que demanda la ejecución del mismo.

$t_i = \frac{n_i}{F} = n_i \times T$; T : $periodo de reloj o tiempo de ciclo$

Previo a explicar las 16 instrucciones que constituyen el lenguaje máquina de CODE-2, es conveniente hacer algunas aclaraciones acerca de la notación que se utilizará:

- $d \leftarrow o$: el contenido del elemento *o* (origen) se copia al elemento *d* (destino).
- R0,...,RF: registros.
- M(x): contenido de la posición x de memoria.
- PC: registro contador de programa.
- OPv: contenido del puerto de salida v.
- IPv: contenido del puerto de entrada v.
- Rx[0]: bit menos significativo del registro x.
- Rx[15]: bit más significativo del registro x.
- Rx[7..0]: byte menos significativo del registro x.
- Rx[15..8]: byte más significativo del registro x.

LD (Load - Cargar)

Esta instrucción carga un registro *Rx* con el contenido de una posición de memoria (que se obtiene de la suma del contenido del registro *RD* y el valor del campo *v*[2]). Sigue el formato F3, y su código de operación es 0_h (0000_b).

Esta instrucción tiene tres parámetros: dos explícitos (*Rx* y *v*) y uno implícito (el registro de dirección *RD*). Esto dificulta su comprensión, pero ha sido necesario realizar de este modo la carga desde memoria, pues sólo puede accederse a la memoria direccionando por palabra (y en este caso las palabras son de 16 bits).

$$LD \quad Rx, v \quad \Rightarrow \quad Rx \leftarrow M(RD + v)$$

ST (Store - Almacenar)

Esta instrucción almacena el contenido del registro *Rx* en memoria (en la posición que se obtiene de sumar el contenido del registro *RD* con el valor del campo *v*). Su código de operación es $1_h(0001_b)$. Sigue el formato F3, aunque no lo parezca, y ésto se debe a que los parámetros se han colocado al revés[3].

$$ST \quad v, Rx \quad \Rightarrow \quad M(RD + v) \leftarrow Rx$$

LLO (Load Low - Carga Inmediata Baja)

Esta instrucción carga el contenido del byte menos significativo del registro *Rx* con el valor del campo *v*, poniendo a 00_hel byte más significativo. Sigue el formato F3, y su código de operación es $2_h(0010_b)$.

[2] Se realiza un direccionamiento base indexado.

[3] Se ha hecho de este modo para seguir un estándar del ensamblador, que indica que el operando de destino de la instrucción se coloca primero entre sus parámetros.

$LLO \quad Rx, v \quad \Rightarrow \quad Rx[15.,8] \leftarrow 00h; \quad Rx[7.,0] \leftarrow v$

LLI (Load High - Carga Inmediata Alta)

Esta instrucción carga el contenido del byte más significativo del registro *Rx* con el valor del campo *v*, pero en este caso, el byte menos significativo conserva su valor original. Sigue el formato F3, y su código de operación es $3_h(0011_b)$.

$LLI \quad Rx\ , v \quad \Rightarrow \quad Rx[15.,8] \leftarrow v$

IN (Entrada)

Carga el contenido del puerto de entrada *v* (IPv) en el registro Rx. Sigue el formato F3, y su código de operación es $4_h(0100_b)$.

$IN \quad Rx, v \quad \Rightarrow \quad Rx \leftarrow IPv$

OUT (Salida)

Lleva el contenido del registro *Rx* al puerto de salida *v*. Sigue el formato F3, y su código de operación es $5_h(0101_b)$.

$OUT \quad v, Rx \quad \Rightarrow \quad OPv \leftarrow Rx$

ADDS (Sumar)

Carga el contenido del registro *Rx* con la suma de los contenidos de los registros *Ry* y *Rz*. Sigue el formato F4, y su código de operación es $6_h(0110_b)$.

Según el resultado de la operación se activarán los biestables indicadores Z (si el resultado de la suma es 0), S (si el resultado de la suma es negativo), C (si la suma produjo acarreo) y V (si se produjo rebalse al sumar).

$ADDS \quad Rx, Ry, Rz \quad \Rightarrow \quad Rx \leftarrow Ry + Rz$

SUBS (Restar)

Carga el contenido del registro *Rx* con la resta de los contenidos de los registros *Ry* y *Rz*. Sigue el formato F4, y su código de operación es $7_h(0111_b)$.

Según el resultado de la operación se activarán los biestables indicadores Z, S, C y V.

$SUBS \quad Rx, Ry, Rz \quad \Rightarrow \quad Rx \leftarrow Ry - Rz$

NAND (Not AND - Operación lógica NAND)

Carga el registro *Rx* con el resultado de aplicar la operación NAND bit a bit entre los contenidos de los registros *Ry* y *Rz*. Sigue el formato F4, y su código de operación es $8_h(1000_b)$.

Según el resultado de la operación se activarán los biestables indicadores Z y S (C y V permanecen iguales).

$NAND \quad Rx, Ry, Rz \quad \Rightarrow \quad Rx \leftarrow \overline{Ry.Rz}$

Cabe notar que en base a esta única instrucción y utilizando las leyes del álgebra booleana, pueden realizarse las restantes operaciones lógicas: AND, OR, NOT, NOR y XOR.

SHL (Shift Left - Desplazar a izquierda)

Desplaza los bits del registro *Rx* una posición a la izquierda. Sigue el formato F1, y su código de operación es $9_h(1001_b)$.

El flag C se carga con el bit más significativo de *Rx*, mientras que el bit menos significativo se pone a 0. El

valor del biestable Z será 1 dependiendo de si el nuevo valor de *Rx* es 0000_h, y S valdrá 1 sólo si el nuevo bit más significativo de *Rx* vale 1 (V permanece igual).

$SHL \quad Rx \quad \Rightarrow \quad C \leftarrow Rx[15]; Rx[15..,1] \leftarrow Rx[14..,0]; Rx[0] \leftarrow 0$

Puede apreciarse fácilmente que con esta instrucción puede multiplicarse por dos el valor contenido en el registro *Rx*.

SHR (Shift Rigth - Desplazar a derecha)

Desplaza los bits del registro *Rx* una posición a la derecha. Sigue el formato F1, y su código de operación es $A_h(1010_b)$.

El biestable indicador C se carga con el bit menos significativo de *Rx*, mientras que el bit más significativo se pone a 0, por lo tanto el flag S valdrá 0. El valor del biestable Z será 1 dependiendo de si el nuevo valor de *Rx* es 0000_h, V permanece igual.

$SHR \quad Rx \quad \Rightarrow \quad C \leftarrow Rx[0]; Rx[14..,0] \leftarrow Rx[15..,1]; Rx[15] \leftarrow 0$

Podría suponerse que esta instrucción permitiría entonces dividir por dos el valor contenido en el registro *Rx*, por simetría con la instrucción anterior. Pero ésto no es cierto, pues hay que considerar que para ello habría que lograr la extensión de signo al desplazar.

SHRA (Shift Rigth Arithmetic - Desplazamiento aritmético a derecha)

Desplaza los bits del registro *Rx* una posición a la derecha. Sigue el formato F1, y su código de operación es $B_h(1011_b)$.

El biestable indicador C se carga con el bit menos significativo de *Rx*, mientras que el bit más significativo conserva su valor (por ende, S también). El valor del biestable Z será 1 dependiendo de si el nuevo valor de *Rx* es 0000_h(V permanece igual).

$SHRA \quad Rx \quad \Rightarrow \quad C \leftarrow Rx[0]; Rx[14..,0] \leftarrow Rx[15..,1]$

Aquí sí puede verse que haciendo un desplazamiento aritmético a derecha sobre el registro *Rx* se logra dividir al contenido del mismo por dos. Esto se debe a que al hacer el desplazamiento no se introduce un 0 en lugar del bit más significativo de *Rx*, sino que se deja el valor presente (*extensión de signo*).

B- (Branch - Ramificación o Salto)

Provoca un salto a la instrucción almacenada en la dirección de memoria que indica el registro *RD*[4]. Sigue el formato F2[5], y su código de operación es $C_h(1100_b)$. El valor del campo *cnd* determina la condición de salto: 0_h implica un salto incondicional (BR, es independiente del valor de los biestables), 1_h ante salto si Z=1 (BZ), 2_h para salto si S=1 (BS), 3_h indica salto si C=1 (BC) y 4_h ante salto si V=1 (BV). Entonces, pese a tratarse de una misma instrucción, a cada tipo de salto se le asocia un nemónico diferente.

$BR \quad \Rightarrow \quad PC \leftarrow RD$
$BZ \quad \Rightarrow \quad si Z = 1, entonces PC \leftarrow RD$
$BS \quad \Rightarrow \quad si S = 1, entonces PC \leftarrow RD$
$BC \quad \Rightarrow \quad si C = 1, entonces PC \leftarrow RD$
$BV \quad \Rightarrow \quad si V = 1, entonces PC \leftarrow RD$

CALL- (Llamada a subrutina)

Dado que CODE-2 permite un manejo simple de pila en memoria, pueden implementarse llamados a subrutinas, permitiendo anidamiento. Pero la administración de la pila queda a cargo del programador, quien deberá fijar su tamaño y cuidar de no solapar esa región de memoria con instrucciones y/o datos de programas.

El crecimiento de la pila se da desde determinada dirección de memoria (base de la pila, depende del tamaño que se le asigna a la misma) hacia las direcciones de memoria más bajas (en general, la dirección del máximo

[4] Previamente a ejecutar la instrucción de salto, debe cargarse el registro RD con la dirección de salto.

[5] Como el último byte de la instrucción es ignorado por la unidad de control, puede cargarse en él cualquier valor.

tope de la pila es la 0000_h, pues es común ubicar la pila al inicio de memoria).

La instrucción CALL- provoca una llamada a la subrutina almacenada en la dirección de memoria que indica el registro *RD*. Sigue el formato F2, y su código de operación es $D_h(1101_h)$. El valor del campo *cnd* determina la condición de salto a subrutina: 0_himplica una llamada incondicional (CALLR, es independiente del valor de los biestables), 1_hante llamada si Z=1 (CALLZ), 2_hpara salto a subrutina si S=1 (CALLS), 3_h indica salto a subrutina si C=1 (CALLC) y 4_h ante llamada si V=1 (CALLV). También existen diferentes nemónicos asociados a cada tipo de llamada a subrutina.

$$\begin{array}{lcl} CALLR & \Rightarrow & PC \leftarrow RD \\ CALLZ & \Rightarrow & si\, Z = 1, entonces\, PC \leftarrow RD \\ CALLS & \Rightarrow & si\, S = 1, entonces\, PC \leftarrow RD \\ CALLC & \Rightarrow & si\, C = 1, entonces\, PC \leftarrow RD \\ CALLV & \Rightarrow & si\, V = 1, entonces\, PC \leftarrow RD \end{array}$$

Antes de cargar en el contador de programa PC la dirección de salto ($PC \leftarrow RD$), debe guardarse la dirección de retorno en la pila en memoria (apilar dirección actual del PC). Para ésto se utiliza el registro *RE*, que actúa como puntero al tope de pila (implícito), haciendo:
$RE \leftarrow RE - 1; M(RE) \leftarrow PC$.
La primer instrucción incrementa en uno el tamaño de la pila, haciendo que el puntero al tope de la misma (RE) apunte a la posición de memoria inmediatamente inferior. Por su parte, la segunda instrucción apila la dirección de retorno. Esto es realizado por el procesador cada vez que se ejecuta una instrucción CALL-.

RET (Retorno de subrutina)

Se incluye al final de una subrutina, provocando el retorno a la instrucción siguiente al CALL- que la invocó. Sigue el formato F0, y su código de operación es $E_h(1110_b)$. Se cambia el valor del PC por el contenido del tope de la pila (indicado por *RE*), y se actualiza el valor del puntero de pila (desapilar).

$$RET \quad \Rightarrow \quad PC \leftarrow M(RE); RE \leftarrow RE + 1$$

HALT (Parar)

Es útil para indicar el fin de un programa, o por ejemplo para detener el procesador una vez emitido un resultado por un puerto de salida, a la espera de una confirmación de que fue visualizado. Sigue el formato F0, y su código de operación es $F_h(1111_b)$.
Provoca que CODE-2 entre en *estado de espera*, deteniendo su funcionamiento antes de que realice la búsqueda de próxima instrucción. En definitiva, detiene al procesador[6].

$$HALT \quad \Rightarrow \quad detiene\, el\, procesador$$

4. Modos de Direccionamiento

Un modo de direccionamiento describe la forma en que el procesador determina la ubicación de un operando o la dirección destino de un salto. Básicamente, existen siete modos de direccionamiento:

- Direccionamiento implícito: el operando se encuentra en un registro implícito, y ésto está implícitamente especificado en el código de operación. Las instrucciones que lo utilizan son, en consecuencia, de menor tamaño, tornándose más legibles. Pero se pierde flexibilidad, dado que hay que recordar cargarle el valor necesario al registro implícito, previamente a ejecutar la instrucción.
  Ejemplos: en instrucciones de la forma F2 (B- y CALL-, que utilizan a RD como registro implícito); y en instrucciones RET y CALL-, que utilizan el registro RE para manipular llamado a subrutinas.

[6] Esto se hace sólo a fines didácticos. En los procesadores reales el procesador no se detiene a menos que se le deje de suministrar corriente eléctrica.

- Direccionamiento inmediato: el operando forma parte de la instrucción misma. Es rápido y flexible, pero ocupa muchos bits en la instrucción.
  *Ejemplos: el valor del campo **v** en instrucciones LLO, LLI, LD y ST.*

- Direccionamiento de registro: la instrucción guarda los bits correspondientes al número de registro que contiene el operando a utilizar. Es flexible porque no impone limitaciones respecto a qué número de registro puede utilizarse. Es rápido porque basta con indicar el registro para acceder al operando completo.
  Ejemplos: en instrucciones que siguen el formato F4.

- Direccionamiento directo: la instrucción contiene la dirección de memoria o el número de registro en que se encuentra el operando.
  CODE-2 no cuenta con este tipo de direccionamiento, pero lo emula.
  Ejemplos: en instrucciones que siguen los formatos F1 y F4, y además LLO y LLI.

- Direccionamiento indirecto: la instrucción indica la posición de memoria o el número de registro en que se encuentra la dirección de memoria del operando. Permite acceder a toda la memoria, pero es lento, y hace compleja su programación.
  CODE-2 tampoco cuenta con este tipo de direccionamiento, no obstante, lo emula.
  Ejemplos: lo utilizan las instrucciones ST y LD.

- Direccionamiento base indexado: la dirección base se encuentra almacenada en un registro, y en la instrucción se encuentra el desplazamiento (o viceversa). La dirección efectiva se obtiene sumando el valor inmediato presente en la instrucción con el valor contenido en el registro. Relaciona los direccionamientos inmediato e implícito.
  Ejemplos: en instrucciones LD y ST.

- Direccionamiento de pila: constituye un caso particular de direccionamiento implícito. Se accede al operando a través del registro de pila *RE*.
  Ejemplo: en instrucciones RET y CALL-, que utilizan el registro RE para manipular llamado a subrutinas.

5. Microinstrucciones

Ante la ejecución de una instrucción, el procesador debe ejecutar determinadas microinstrucciones que la efectivicen. Esto se realiza internamente, y el mapeo de una instrucción en una o más microinstrucciones depende de la complejidad de la instrucción que las origina.

Previo a ejecutar una instrucción, la misma debe ser traída de memoria, y almacenada en el *registro de instrucción* (RI, o IR). La posición de memoria desde la cual se obtendrá la próxima instrucción a ejecutar está almacenada en el registro *contador de programas* (PC). Lo cual implica que luego de "traer" de memoria la próxima instrucción, deberá incrementarse en 1 el contenido del PC, para "apuntar" a la siguiente instrucción a cargar en el RI.

El proceso anterior es conocido como *fetching*, o *búsqueda de próxima instrucción*, y antecede a todo conjunto de microinstrucciones que se ejecute para realizar una instrucción.

FETCHING:

$RI \leftarrow M(PC) \quad \Rightarrow \quad carga\ en\ RI\ el\ contenido\ de\ la\ memoria\ en\ la\ direccion\ PC$

$PC \leftarrow PC + 1 \quad \Rightarrow \quad incrementa\ en\ 1\ la\ direccion\ contenida\ en\ PC$

Luego del Fetching, el procesador determina de qué instrucción se trata mirando los primeros 4 bits del registro de instrucción (RI[15..12]). A continuación, se detallan las microinstrucciones correspondientes al juego de instrucciones del CODE-2:

LD:

Considerando que esta instrucción sigue el formato F3, el registro destino de la carga es el indicado por RI[11..8], y el valor de desplazamiento en memoria con respecto a RD se encuentra en el byte menos significativo de RI (RI[7..0]).

$R(RI[11.,8]) \leftarrow M(RD + RI[7.,0])$

ST:

Esta instrucción sigue el mismo formato que la anterior, por lo que el registro que contiene el dato a almacenar en memoria es el indicado por RI[11..8], mientras que el valor de desplazamiento en memoria con respecto a RD se encuentra en el byte menos significativo de RI (RI[7..0]).

$M(RD + RI[7.,0]) \leftarrow R(RI[11.,8])$

LLO:

Siguiendo el formato F3, el registro en el que se va a almacenar el dato inmediato *v* está dado por RI[11..8], y el dato en cuestión se indica en el byte menos significativo de RI (RI[7..0]).

Se carga la parte baja de tal registro con el dato inmediato *v*, y la parte alta del mismo con 0s.

$R(RI[11.,8])[7.,0] \leftarrow RI[7.,0]$

$R(RI[11.,8])[15.,8] \leftarrow 0$

LLI:

El registro en el que se va a cargar el valor inmediato *v* está dado por RI[11..8], mientras que *v* se indica en el byte menos significativo de RI (RI[7..0]).

Se carga la parte alta de tal registro con el dato inmediato *v*, y la parte baja del mismo permanece intacta.

$R(RI[11.,8])[15.,8] \leftarrow RI[7.,0]$

IN:

Tiene formato F3, motivo por el cual el contenido del puerto de entrada dado por *v* (valor presente en el byte menos significativo de RI), se copiará al registro indicado por RI[11..8].

$R(RI[11.,8]) \leftarrow IP(RI[7.,0])$

OUT:

Dado el formato F3, en RI[7..0] se tiene el puerto de salida en el que se hará presente el contenido del registro indicado en RI[11..8].

$OP(RI[7.,0]) \leftarrow R(RI[11.,8])$

ADDS:

Siguiendo el formato F4, en RI[11..8] se encuentra indicado el registro destino de la suma, y en RI[7..4] y RI[3..0] se tienen los números de registros en que residen los operandos.

$R(RI[11.,8]) \leftarrow R(RI[7.,4]) + R(RI[3.,0])$

SUBS:

Dado que sigue el formato F4, los operandos destino y origen de la resta se encuentran en las mismas posiciones dentro del RI.

$R(RI[11.,8]) \leftarrow R(RI[7.,4]) - R(RI[3.,0])$

NAND:

Como sigue el formato F4, los operandos destino y origen se encuentran en las mismas posiciones dentro del RI.

$R(RI[11.,8]) \leftarrow R(RI[7.,4])\ NAND\ R(RI[3.,0])$

SHL:

Sigue el formato de instrucción F1, en consecuencia, el registro cuyo valor será desplazado a izquierda, es el indicado en RI[11..8].

El bit más significativo de tal registro se almacenará en el flag de Carry, los restantes se desplazarán un lugar a izquierda, y se guardará un 0 en lugar del bit menos significativo del mismo.

$C \leftarrow R(RI[11.,8])[15]$

$R(RI[11.,8])[15.,1] \leftarrow R(RI[11.,8])[14.,0]$

$R(RI[11.,8])[0] \leftarrow 0$

SHR:

Sigue el formato de instrucción F1, en consecuencia, el registro cuyo valor será desplazado a derecha, es el indicado en RI[11..8].

El bit de menor peso de tal registro cargará el flag de Carry, los restantes se desplazarán un lugar a derecha, y se guardará un 0 en el lugar del bit más significativo.

$C \leftarrow R(RI[11.,8])[0]$

$R(RI[11.,8])[14.,0] \leftarrow R(RI[11.,8])[15.,1]$

$R(RI[11.,8])[15] \leftarrow 0$

SHRA:

Se tienen las mismas consideraciones que para la instrucción anterior, a excepción de que se conserva el valor del bit más significativo del registro a desplazar.

$C \leftarrow R(RI[11.,8])[0]$

$R(RI[11.,8])[14.,0] \leftarrow R(RI[11.,8])[15.,1]$

BR: como el salto es incondicional, simplemente se copia el contenido de RD en PC.

$PC \leftarrow RD$
BZ: se copia el contenido de RD en PC, sólo si Z = 1.

$If\ Z\ then\ PC \leftarrow RD$
BS: se copia el contenido de RD en PC, sólo si S = 1.

$If\ S\ then\ PC \leftarrow RD$
BC: se copia el contenido de RD en PC, sólo si C = 1.

$If\ C\ then\ PC \leftarrow RD$
BV: se copia el contenido de RD en PC, sólo si V = 1.

$If\ V\ then\ PC \leftarrow RD$

Siendo que todas las instrucciones de salto (B-) utilizan como dirección de salto un operando implícito (el registro RD), su valor debe cargarse en el PC, para que la próxima instrucción a buscar de memoria sea la que reside en la dirección indicada por la dirección de salto.

CALLR: el salto a subrutina es incondicional; se apila la dirección de retorno y luego se copia el contenido de RD en PC.

$RE \leftarrow RE - 1$
$M(RE) \leftarrow PC$
$PC \leftarrow RD$

CALLZ: se apila la dirección de retorno y luego se copia el contenido de RD en PC, sólo si Z = 1.

$If\ Z\ then$

$RE \leftarrow RE - 1$
$M(RE) \leftarrow PC$
$PC \leftarrow RD$

CALLS: se apila la dirección de retorno y luego se copia el contenido de RD en PC, sólo si S = 1.

$If\ S\ then$

$RE \leftarrow RE - 1$
$M(RE) \leftarrow PC$
$PC \leftarrow RD$

CALLC: se apila la dirección de retorno y luego se copia el contenido de RD en PC, sólo si C = 1.

$If\ C$

$RE \leftarrow RE - 1$
$M(RE) \leftarrow PC$
$PC \leftarrow RD$

CALLV: se apila la dirección de retorno y luego se copia el contenido de RD en PC, sólo si V = 1.

$If\ V$

$RE \leftarrow RE - 1$
$M(RE) \leftarrow PC$
$PC \leftarrow RD$

Las instrucciones de salto a subrutina (CALL-) utilizan como dirección de salto un operando implícito (el registro RD); su valor debe almacenarse en el PC, para que la próxima instrucción a buscar de memoria sea la que reside en la dirección indicada por la dirección de la subrutina. Pero previamente debe apilarse en memoria, en la región reservada para la pila, la dirección de retorno, contenida en PC.

RET:

Utiliza un operando implícito (el registro RE), que contiene la dirección de memoria en que reside la dirección de retorno de subrutina. La misma es cargada en el PC, y se desapila incrementando el contenido de RE (es decir, dejando el tope de pila en una posición mayor de memoria[7]).

$PC \leftarrow M(RE)$

$RE \leftarrow RE + 1$

HALT:

No tiene asociada microinstrucción alguna.

[7] Recordar que el crecimiento de la pila es hacia las posiciones más bajas de la memoria.

6. Realización de programas - Estrategias

Puesta a 0 y a 1 de un registro

Los valores 0 y 1 son comúnmente usados en la realización de programas, para comparar, para llevar un contador de iteraciones, etc. Por ésto es útil almacenarlos en registros, para poder operar.

Ejemplos:

$LLO \quad R0, 00_h \quad \Rightarrow \quad R0 \leftarrow 0000_h$

$LLO \quad R1, 01_h \quad \Rightarrow \quad R1 \leftarrow 0001_h$

Claro que del mismo modo podría almacenarse cualquier otro valor en un registro, por ejemplo para ejecutar instrucciones que demandan la utilización de la ALU.

Copiar el contenido de un registro en otro

Puede hacerse utilizando una instrucción de suma ADDS entre el registro a copiar y un registro cuyo contenido sea 0000_h, y almacenándose el resultado en el registro destino.

Ejemplo:

$ADDS \quad R5, R2, R0 \quad \Rightarrow \quad R5 \leftarrow R2\,(R2+0)$

Detectar si un número es 0 o si es negativo

Una forma de lograrlo es utilizando una instrucción de suma ADDS entre el registro que contiene el número en cuestión *Rx* y un registro cuyo contenido sea 0000_h, y almacenando el resultado en el mismo *Rx*. Así, se cargan los biestables indicadores dependiendo del resultado de la suma; y en particular, será Z=1 sii *Rx*=0000_h. Esto permite efectuar un salto, detectada tal condición[8].

Ejemplo:

$ADDS \quad R5, R5, R0 \quad \Rightarrow \quad R5 \leftarrow R5\,(R5+0); Z \leftarrow 1\, si\, R5 = \mathbf{0}$

$BZ \quad \Rightarrow \quad si\, Z = 1\, entonces\, saltar\, a\, la\, direccion\, contenida\, en\, RD$

Ejecutando un salto según el flag S (BS) permitirá detectar si el número es negativo (se habrá cargado S con el bit más significativo del número).

Comparar dos números

Realizando la resta (en el registro *Rx*) entre ambos números, previamente almacenados en registros, se cargan los biestables indicadores dependiendo del resultado: si son iguales, *Rx*=0000_h, entonces Z=1; si el primero es menor que el segundo, el bit más significativo de *Rx* será igual a 1, entonces S=1; y si no se cumplió ninguna de las condiciones anteriores, el primer número es mayor que el segundo.

Ejemplo:

$SUBS\, Rx, Ry, Rz \Rightarrow Rx \leftarrow Ry - Rz; Z \leftarrow 1, si\, Ry = Rz; S \leftarrow 1\, si\, Ry < Rz$

//Cargar RD con la dirección de salto en caso que Ry sea igual a Rz

$BZ \quad \Rightarrow \quad si\, Z = 1\, entonces\, saltar\, a\, la\, direccion\, contenida\, en\, RD$

//Cargar RD con la dirección de salto en caso que Ry sea menor que Rz

$BS \quad \Rightarrow \quad si\, S = 1\, entonces\, saltar\, a\, la\, direccion\, contenida\, en\, RD$

//Cargar RD con la dirección de salto en caso que Ry sea mayor que Rz

$BR \quad \Rightarrow \quad salto\, incondicional\, a\, la\, direccion\, contenida\, en\, RD$

Contadores ascendentes y descentes

Habiendo almacenado el factor de incremento (decremento) en un registro, y dado un valor inicial al contador, puede incrementarse (decrementarse) el mismo, efectuando la suma (resta) entre el contenido del registro que almacena el valor del contador y el factor.

Ejemplo:

$LLO \quad R1, 04_h \quad \Rightarrow \quad R1 \leftarrow 4;\ factor\, de\, incremento$

$ADDS \quad Rx, Rx, R1 \quad \Rightarrow \quad Rx\, cuenta\, ascendentemente\, de\, a\, 4$

$SUBS \quad Rx, Rx, R1 \quad \Rightarrow \quad Rx\, cuenta\, descendiendo\, de\, a\, 4$

Recorrido de un arreglo

Si se desea operar con los datos de un arreglo en memoria, conociendo su dirección de comienzo y fin, puede hacerse lo siguiente:

[8] Haciendo $SUBS \quad r5, r5, r0$ se obtendría idéntico resultado.

- Almacenar en registros las direcciones de comienzo y fin.
- Llevar un contador sobre las posiciones del arreglo, que comience en la posición inicial del mismo.
- (loop) Guardar la dirección de finalización del programa en RD, para saltar en caso de haber recorrido el arreglo completo.
- Chequear si el contador sobrepasa los límites del arreglo; de ser así, es el fin del programa, caso contrario, se continúa la ejecución del mismo.
- Almacenar el dato del arreglo en un registro, para poder operar con con él.
- Incrementar el contador.
- Se vuelve al inicio del bucle (salto a loop).

Ejemplo: se quiere obtener la suma de los elementos de un arreglo (se almacenará en R4), cuyas direcciones de comienzo y fin están almacenadas en los registros RA y RB respectivamente. Se utilizará el registro R2 como contador para recorrer el arreglo. Se ha almacenado en R1 el valor 0001_h *como factor de incremento del contador, y en R0 se guardó el* 0000_h.

100_h : $ADDS \quad R2, RA, R0 \quad \Rightarrow \quad inicializa\ R2\ con\ el\ valor\ de\ RA$

101_h : $ADDS \quad R4, R0, R0 \quad \Rightarrow \quad inicializa\ la\ suma\ R4\ en\ 0$

102_h : $LLO \quad RD, 0D_h$

103_h : $LLI \quad RD\ , 01_h \quad \Rightarrow \quad guardan\ en\ RD\ la\ dir\ de\ fin\ de\ programa$

104_h : $SUBS \quad RF, RB, R2 \Rightarrow \quad RF \leftarrow RB - R2; si\ R2 > RB, S \leftarrow 1;\ fin\ arreglo$

105_h : $BS \quad \Rightarrow \quad si\ S = 1, salta\ a\ 10D_h$

106_h : $ADDS \quad RD, R2, R0 \quad \Rightarrow \quad RD \leftarrow R2$

107_h : $LD \quad R3, 00_h \quad \Rightarrow \quad R3 \leftarrow M(RD + 00_h)$

108_h : $ADDS \quad R4, R4, R3 \Rightarrow \quad R4 \leftarrow R4 + R3$

109_h : $ADDS \quad R2, R2, R1 \Rightarrow \quad incrementa\ en\ 1\ R2$

$10A_h$: $LLO \quad RD, 02_h$

$10B_h$: $LLI \quad RD\ , 01_h \quad \Rightarrow \quad guardan\ en\ RD\ la\ dir\ de\ inicio\ del\ loop$

$10C_h$: $BR \quad \Rightarrow \quad salto\ incondicional\ a\ 102_h$

$10D_h$: $HALT \quad \Rightarrow \quad fin\ del\ programa$

Chequeo de paridad de un número

Una opción consiste en ver si el bit menos significativo del número es 0 (lo que indica que es par) ó 1 (entonces es impar), para lo cual puede hacerse: guardar en un registro *Rx* el resultado de hacer NAND entre el número en cuestión y 0001_h, guardar la máscara $FFFF_h$en otro registro *Ry* y, por último, comparar los contenidos de ambos registros. Si son iguales, entonces NAND(número,0001_h)=$FFFF_h$, lo que indica que el último bit del número es 0. Si el número es impar (el último bit es 1) el resultado de NAND(número, 0001_h) será $FFFE_h$.

Consideraciones: el número está almacenado en el registro R2. Se utilizará el registro R3 para almacenar la máscara $FFFF_h$. *Se ha guardado en R1 el valor* 0001_h*para efectuar la operación NAND.*

151_h : $LLO \quad R3, FF_h$

152_h : $LLI \quad R\,3, FF_h \quad \Rightarrow \quad carga\,en\,R3\,la\,mascara\,FFFF_h$

153_h : $NAND \quad R4, R2, R1 \quad \Rightarrow \quad R4 \leftarrow \overline{R2.R1}$

154_h : $LLO \quad RD, 5B_h$

155_h : $LLI \quad RD\,, 01_h \quad \Rightarrow \quad cargan\,en\,RD\,dir\,de\,salto\,si\,par; RD \leftarrow 15B_h$

156_h : $SUBS \quad R5, R4, R3 \quad \Rightarrow \quad R5 \leftarrow R4 - R3; si\,R4 = R3, Z \leftarrow 1$

157_h : $BZ \quad \Rightarrow \quad si\,Z = 1, salta\,a\,15B_h$

158_h : $LLO \quad RD, 01_h$

159_h : $LLI \quad RD\,, 02_h \quad \Rightarrow \quad cargan\,en\,RD\,dir\,de\,salto\,si\,impar; RD \leftarrow 201_h$

$15A_h$: $BR \quad \Rightarrow \quad salto\,incondicional\,a\,201_h$

O bien, desplazando el número (guardado en el registro *R*x) a derecha, el flag C se cargaría con el bit menos significativo. Luego, si C=1 el número es impar, entonces se salta a determinada dirección de memoria, almacenada previamente en *RD* (siguiendo el ejemplo anterior, 201_h). De lo contrario, se almacenará en *RD* la dirección de salto ante número par ($15B_h$).

$1FA_h$: $LLO \quad RD, 01_h$

$1FB_h$: $LLI \quad RD\,, 02_h \quad \Rightarrow \quad guardan\,en\,RD\,la\,dir\,de\,salto\,si\,impar$

$1FC_h$: $SHR \quad Rx \quad \Rightarrow \quad C \leftarrow Rx[0]$

$1FD_h$: $BC \quad \Rightarrow \quad si\,C = 1, salta\,a\,201_h$

$1FE_h$: $LLO \quad RD, 5B_h$

$1FF_h$: $LLI \quad RD\,, 01_h \quad \Rightarrow \quad guardan\,en\,RD\,la\,dir\,de\,salto\,si\,par$

200_h : $BR \quad \Rightarrow \quad salto\,incondicional\,a\,15B_h$