Resumen de la arquitectura ARMv8

Organización del Computador - Prácticos 6,7,8 y 9

Pedro Villar

13 de mayo de 2024

Operaciones aritméticas - Suma y resta

Intrucción	Sintaxis	Significado
ADD	ADD X0, X1, X2	X0 = X1 + X2
SUB	SUB X0, X1, X2	X0 = X1 - X2
ADDI	ADDI X0, X1, #3	X0 = X1 + 3
SUBI	SUBI X0, X1, #3	X0 = X1 - 3

En las instrucciones `ADDI` y `SUBI` el tercer operando es un valor inmediato, se representa con #.

#1 Ejemplo de suma y resta

Supongamos que las variables `a,b,c,d,e` están almacenadas en los registros `X0,X1,X2,X3,X4` respectivamente.

```
a = b + c;
d = a - e;
```

El código ensamblador sería:

```
ADD X0, X1, X2
SUB X3, X0, X4
```

#2 Ejemplo de suma y resta

Supongamos que las variables `f,g,h,i,j` están almacenadas en los registros `X0,X1,X2,X3,X4` respectivamente.

```
f = (g + h) - (i + j);
```

El código ensamblador sería:

```
ADD X0, X1, X2 // X0 = g + h
ADD X3, X3, X4 // X3 = i + j
SUB X0, X0, X3 // X0 = (g + h) - (i + j)
```

Operandos de memoria

La memoria podría pensarse como un arreglo unidimensional grande, donde la dirección actúa como el índice del arreglo. LEGv8 utiliza direccionamiento por bytes, donde cada palabra representa 8 bytes.

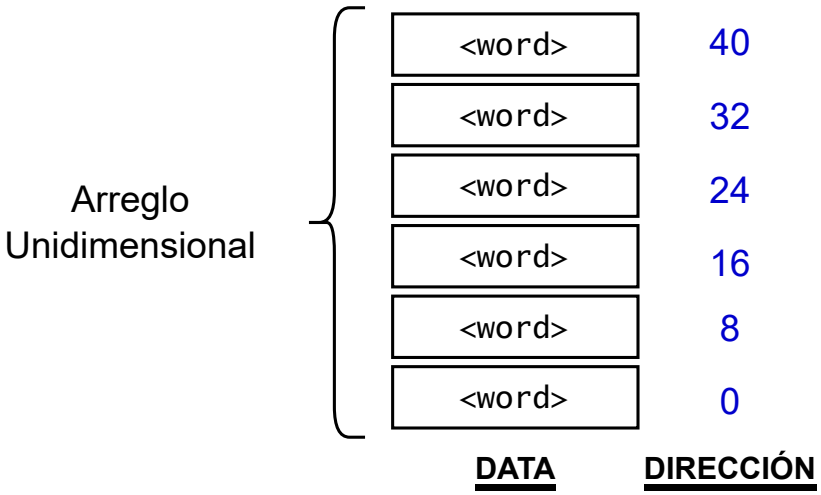

Figura: Dirección de memoria

Operandos de memoria - LDUR

La intrucción de transferencia de datos `LDUR` copia información desde la memoria hacia un registro que se denomina *carga (LOAD)*. El formato de la instrucción consta de el registro en el que se cargará, seguido del registro junto con la constante usada para acceder a la memoria. Donde la suma de los valores del registro y la constante se usan para acceder a la memoria.

Acceso a direcciones de memoria

Como LEGv8 utiliza direccionamiento por bytes, cuando un array que se almacena en memoria continuamente quiere ser accedido, se debe tener en cuenta que la dirección de memoria se incrementa en 8 bytes por cada elemento del array. Por ejemplo, si se busca acceder al elemento 7 de un array, el desplazamiento que se le debe agregar al registro base del array es $8 \times 7 = 56$ y por lo tanto para guardar lo que esta en $A[7]$, suponiendo que la dirección base del array es $X0$, la instrucción sería `LDUR X1, [X0, #56]`.

Ejemplo de acceso a memoria - LDUR

Supongamos que la dirección base del array A es $X22$ y la variable h esta almacenada en el registro $X21$.

```
h = A[8];
```

El código ensamblador sería:

```
LDUR X21, [X22, #64]
```

Operandos de memoria - STUR

La instrucción complementaria a *"load"* se denomina *"store"*, esta instrucción copia datos desde un registro hacia la memoria. El formato de la instrucción es: el registro que se almacenará en memoria, seguido del registro base y la constante que se usará para el desplazamiento.

Ejemplo de acceso a memoria - STUR

Supongamos que la variable h esta almacenada en el registro $X21$ y la dirección base del array A es $X22$.

```
A[12] = h + A[8];
```

El código ensamblador sería:

```
LDUR X23, [X22, #64] // X23 = A[8]
ADD X21, X21, X23 // X21 = h + A[8]
STUR X21, [X22, #96] // A[12] = h + A[8]
```

Operaciones lógicas

Las operaciones lógicas son instrucciones que se realizan bit a bit. Son utilizadas para examinar y modificar los bits de una palabra.

Operación Lógica	Instrucción	Sintaxis
Shift Left	`LSL`	`LSL X0, X1, #3`
Shift Right	`LSR`	`LSR X0, X1, #3`
Bit-by-bit AND	`AND`	`AND X0, X1, X2`
Bit-by-bit OR	`ORR`	`ORR X0, X1, X2`
Bit-by-bit NOT	`EOR`	`EOR X0, X1, X2`

Operaciones lógicas - Shift Left

La operación `LSL` realiza un desplazamiento a la izquierda de los bits de un registro. El número de bits que se desplazan se especifica en el tercer operando. Esta operación es equivalente a multiplicar por 2^n, donde n es el número de bits que se desplazan. Por ejemplo, si se desea multiplicar por 8 el contenido de $X1$ y almacenarlo en $X0$, la instrucción sería `LSL X0, X1, #3`.

Operaciones lógicas - Shift Right

La operación `LSR` realiza un desplazamiento a la derecha de los bits de un registro. El número de bits que se desplazan se especifica en el tercer operando. Esta operación es equivalente a dividir por 2^n, donde n es el número de bits que se desplazan. Por ejemplo, si se desea dividir por 4 el contenido de $X1$ y almacenarlo en $X0$, la instrucción sería `LSR X0, X1, #2`.

Operaciones Lógicas - AND

La operación `AND` realiza una operación bit a bit de la operación AND entre registros, basicamente toma bit a bit y realiza la operación AND, dejando un 1 si ambos bits son 1 y un 0 en caso contrario.
Por ejemplo el registro $X11$ contiene:

```
00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
```

y el registro $X10$ contiene:

```
00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
```

La instrucción `AND X9, X10, X11` resultará en:

```
00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000
```

Operaciones Lógicas - ORR

La operación `ORR` realiza una operación bit a bit de la operación OR entre registros, basicamente toma bit a bit y realiza la operación OR, dejando un 1 si al menos uno de los bits es 1 y un 0 en caso contrario.
Por ejemplo el registro $X11$ contiene:

```
00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
```

y el registro $X10$ contiene:

```
00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
```

La instrucción `ORR X9, X10, X11` resultará en:

```
00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000
```

Operaciones Lógicas - EOR y NOT

- La operación `EOR` realiza una operación bit a bit de la operación XOR entre registros, basicamente toma bit a bit y realiza la operación XOR, dejando un 1 si los bits son diferentes y un 0 si son iguales.
- La operación `NOT` realiza una operación bit a bit de la operación NOT entre registros, basicamente toma bit a bit y realiza la operación NOT, dejando un 1 si el bit es 0 y un 0 si el bit es 1.

Condicionales

El lenguaje ensamblador de LEGv8 posee dos instrucciones de este tipo, estas son `CBZ` y `CBNZ`.

- `CBZ registro, L1` Esta instrucción significa *ir a la instrucción etiquetada L1 si el valor en el registro es igual a cero*. CBZ significa *comparar y saltar si es cero*.
- `CBNZ registro, L1` Esta instrucción significa *ir a la instrucción etiquetada L1 si el valor en el registro es diferente de cero*. CBNZ significa *comparar y saltar si no es cero*.

Ejemplo de condicionales

En el siguiente segmento de código, f, g, h, i y j son variables. Si las cinco variables f a j corresponden a los cinco registros X19 a X23, ¿cuál es el código LEGv8 compilado para esta instrucción if en C?

```
if (i == j) f = g + h; else f = g - h;
```

La primera expresión compara la igualdad entre dos variables en registros. Dado que las instrucciones anteriores solo pueden probar si un registro es cero, el primer paso es restar j de i para probar si la diferencia es cero.

Parecería que luego querríamos ramificar si la diferencia es cero (CBZ). En general, el código será más eficiente si probamos la condición opuesta para saltar sobre el código que se ramifica si la diferencia no es igual a cero (CBNZ). De esta forma usariamos las dos instrucciones, usando el registro X9 para almacenar el resultado de restar j de i:

```
SUB X9,X22,X23  // X9 = i - j
CBNZ X9, Else   // ir a Else si i != j (X9 != 0)
```

Luego, si i es igual a j, queremos calcular f = g + h, y saltar sobre el cálculo de f = g - h. Por lo tanto, el código sería:

```
ADD X19, X20, X21 // f = g + h
B Exit // saltar sobre Else, ir a Exit
Else: SUB X19, X20, X21 // f = g - h
Exit:
```

El código completo sería:

```
SUB X9,X22,X23  // X9 = i - j
CBNZ X9, Else   // ir a Else si i != j (X9 != 0)
ADD X19, X20, X21 // f = g + h
B Exit // saltar sobre Else, ir a Exit
Else: SUB X19, X20, X21 // f = g - h
Exit:
```

En pocas palabras, el código compara i y j, y si son iguales, calcula f = g + h; de lo contrario, calcula f = g - h.

Condicionales para bucles

Se asume que i y k se corresponden con los registros X22 y X24, y la base del arreglo save está en X25. ¿Cuál es el código ensamblador LEGv8 correspondiente a este código C?

```
while (save[i] == k) i += 1;
```

El primer paso es cargar `save[i]` en un registro temporal. Antes de poder cargar `save[i]` en un registro temporal, necesitamos su dirección. Antes de poder sumar i a la base del arreglo save para formar la dirección, debemos multiplicar el índice i por 8. Podemos usar un desplazamiento a la izquierda (LSL), ya que desplazar a la izquierda por 3 bits multiplica por 2^3. Necesitamos agregarle la etiqueta Loop para poder regresar a esa instrucción al final del ciclo:

```
Loop: LSL X10,X22,#3  // X10 = i * 8
      ADD X10,X10,X25  // X10 = &save[i]
```

Ahora podemos usar esa dirección para cargar save[i] en un registro temporal:

```
LDUR X9,[X10,#0]  // X9 = save[i]
```

La siguiente instrucción resta k de save[i] y coloca la diferencia en X11 para configurar la prueba del ciclo. Si X11 no es 0, entonces son desiguales (`save[i] != k`):

```
SUB X11,X9,X24  // X11 = save[i] - k
```

La siguiente instrucción realiza la prueba del ciclo, saliendo si `save[i] != k`:

```
CBNZ X11,Exit  // salir si save[i] != k
```

Luego se suma 1 a i y se repite el ciclo:

```
ADD X22,X22,#1  // i += 1
B Loop  // repetir el ciclo
Exit:
```

El código completo sería:

```
Loop: LSL X10,X22,#3  // X10 = i * 8
      ADD X10,X10,X25  // X10 = &save[i]
      LDUR X9,[X10,#0]  // X9 = save[i]
      SUB X11,X9,X24  // X11 = save[i] - k
      CBNZ X11,Exit  // salir si save[i] != k
      ADD X22,X22,#1  // i += 1
      B Loop  // repetir el ciclo
Exit:
```

Ejercicio 7 - Práctico 6

Ejercicio 7:

Dadas las siguientes sentencias en assembler LEGv8:

```
ADDI X9, X6, #8
ADD  X10, X6, XZR
STUR X10, [X9, #0]
LDUR X9, [X9, #0]
ADD  X0, X9, X10
```

7.1) Asumiendo que los registros `X0`, `X6` contienen las variables `f` y `A` (dirección base del arreglo), escribir la secuencia mínima de código "C" que representa.

7.2) Asumiendo que los registros `X0`, `X6` contienen los valores `0xA`, `0x100`, y que la memoria contiene los valores de la tabla, encuentre el valor del registro `X0` al finalizar el código assembler.

Dirección	Valor
0x100	0x64
0x108	0xC8
0x110	0x12C

Figura: Ejercicio 7 - Práctico 6

Interpretación de las instrucciones:

```
ADDI X9, X6, #8 // X9 = &A[0] + 8 = A[1]
ADD X10, X6, XZR // X10 = &A[0]
STUR X10, [X9, #0] // A[1] = &A[0]
LDUR X9, [X9, #0] // X9 = A[1]
ADD X0, X9, X10 // f = A[0] + A[0]
```

La secuencia mínima de C es:

```
f = A[0] + A[0];
```

La dirección de memoria de A[0] es $0x100$ y el contenido de A[0] es $0x64$, entonces el programa definirá a X0 como $0xC8$.

Ejercicio 8.1 - Práctico 6

Dado el contenido de los siguientes registros:

a) X9 = 0x55555555, y X10=0x12345678

¿Cuál es el valor del registro X11 luego de la ejecución del siguiente código assembler en LEGv8?

```
LSL X11, X9, #4
ORR X11, X11, X10
```

El contenido de X9 puede ser representado como:

0101 0101 0101 0101 0101 0101 0101 0101

Despues de hacerle un desplazamiento a la izquierda de 4 bits, el contenido de X11 sería:

0101 0101 0101 0101 0101 0101 0101 0000

El contenido de X10 puede ser representado como:

0001 0010 0011 0100 0101 0110 0111 1000

Por lo tanto, el contenido de X11 sería:

0101 0111 0111 0101 0101 0111 0111 1000

Ejercicio 1 - Práctico 7

Para estos dos programas con entrada y salida en X0, decir que función realizan:

```
        SUBIS X0, X0, #0
        B.LT else
        B done
else: SUB X0, XZR, X0
done:
```

Respuesta: El programa calcula el valor absoluto de X0.

```
        MOV X9, X0
        MOV X0, XZR
loop:   ADD X0, X0, X9
        SUBI X9, X9, #1
        CBNZ X9, loop
done:
```

Respuesta: El programa calcula el valor de la suma de los números de 0 a X0.

MOVZ y MOVK

Las instrucciones `MOVZ` y `MOVK` son instrucciones de carga inmediata. La instrucción `MOVZ` carga un valor inmediato en un registro dejando los demas bits en cero, mientras que la instrucción `MOVK` carga un valor inmediato en un registro, pero solo en los bits especificados por la máscara.

Instrucción	Sintaxis	Significado
`MOVZ`	`MOVZ X0, #0x1234, LSL #0`	`X0 = 0x1234`
`MOVK`	`MOVK X0, #0x5678, LSL #16`	`X0 = 0x12345678`

Comparaciones

El conjunto completo de comparaciones para números incluye menor que ($<$), menor o igual que ($\leq$), mayor que ($>$), mayor o igual que ($\geq$), igual ($=$) y diferente ($\neq$). La comparación de patrones de bits también debe considerar la diferencia entre números con signo y sin signo. En los números con signo, un patrón de bits con un 1 en el bit más significativo representa un número negativo y, por supuesto, es menor que cualquier número positivo, que debe tener un 0 en el bit más significativo.

FLAGS

En LEGv8, existen cuatro bits de condición, que registran el resultado de una instrucción, llamados FLAGS:

- **Negativo (N):** el resultado que estableció el código de condición tenía un 1 en el bit más significativo (indicando un número negativo).
- **Cero (Z):** el resultado que estableció el código de condición fue 0.
- **Overflow (V):** el resultado que estableció el código de condición causó un overflow (el resultado era demasiado grande para ser representado).
- **Carry (C):** el resultado que estableció el código de condición produjo un carry del bit más significativo o un borrow en el bit más significativo.

Salto condicional

Las instrucciones de salto condicional utilizan combinaciones de estos códigos de condición para realizar las comparaciones deseadas y los saltos. En la arquitectura LEGv8, esta instrucción de salto condicional se llama B.cond. La parte cond de la instrucción se puede usar con cualquiera de las instrucciones de comparación con signo:

Instrucción	Sintaxis	Significado
B.EQ	B.EQ L1	ir a L1 si Z = 1
B.NE	B.NE L1	ir a L1 si Z = 0
B.GE	B.GE L1	ir a L1 si N = V
B.LT	B.LT L1	ir a L1 si N $\neq$ V
B.GT	B.GT L1	ir a L1 si Z = 0 y N = V
B.LE	B.LE L1	ir a L1 si Z = 1 o N $\neq$ V

Ejercicio 2 - Práctico 7

Dado el siguiente programa LEGv8, dar el valor final de X10, dado que inicialmente X10=0x0000000000000001.

```
        SUBIS XZR, X9, #0
        B.GE else
        B done
else:   ORRI X10, XZR, #2
done:
```

1.1: Dado que inicialmente X9=0x0000000000101000.

1.1: Dado que inicialmente X9=0x0000000000101000.

- `SUBIS XZR, X9, #0` *XZR = 0x0000000000000000*
- `B.GE else` *No se cumple la condición, se salta a la siguiente instrucción*
- `ORRI X10, XZR, #2` *X10 = 0x0000000000000002*
- `done:` *Fin del programa*
- **Resultado:** *X10 = 0x000 0000 0000 0002*

El programa asigna el valor 2 a X10.