

Instrucciones

Arquitectura de Computadores – IIC2343

Todo computador debe tener instrucciones para ejecutar las operaciones aritméticas básicas

P.ej.:

add a,b,c —*suma las variables b y c y pone la suma en a*

La notación es rígida:

- cada instrucción aritmética ejecuta solo una operación
... y siempre debe tener exactamente tres variables

P.ej., para poner en **a** la suma de **b**, **c**, **d** y **e**:

```
add a,b,c  
add a,a,d  
add a,a,e
```

→ para sumar cuatro variables se necesitan tres instrucciones, cada una en una línea por sí sola

¿Cómo traduciría el compilador el siguiente segmento de un programa en C o Java a instrucciones del lenguaje assembly?

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

Respuesta:

```
add a,b,c  
sub d,a,e
```

¿Y qué haría el compilador en el caso de la siguiente sentencia?

$$f = (g+h) - (i+j)$$

Respuesta:

```
add t0,g,h  
add t1,i,j  
sub f,t0,t1
```

t0 y **t1** son variables auxiliares para almacenar temporalmente los resultados de **g+h** e **i+j** antes de hacer la resta

Los tres operandos de las instrucciones aritméticas deben corresponder a localidades especiales construidas directamente en el hardware
—**los registros**:

- normalmente, un número limitado de registros, p.ej., 32
- normalmente, de 32 bits de largo c/u

Para los nombres de los registros, usaremos el signo **\$** seguido de dos caracteres:

- **\$s0, \$s1, ...** son los registros correspondientes a las variables en programas en C, Java, Python, etc.
- **\$t0, \$t1, ...** son registros para almacenar resultados intermedios necesarios para compilar los programas a instrucciones del lenguaje assembly

P.ej., el código compilado para $f = (g + h) - (i + j)$ sería entonces

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Los lenguajes de programación también tienen arreglos (y otras estructuras), que pueden contener muchos más datos que la cantidad de registros del computador

Estas estructuras se almacenan en la memoria del computador:

- que tiene capacidad para almacenar miles de millones de datos

... → el lenguaje assembly debe tener instrucciones para *transferir datos entre la memoria y los registros*:

- la instrucción debe proporcionar la dirección de memoria de la palabra

La memoria es como un arreglo unidimensional muy grande, en que la *dirección* (de memoria) actúa como índice

lw (load) copia datos desde la memoria a un registro

$g = h + A[8]$ —(una) sentencia en C, Java, etc.

Si la dirección de partida, o base, de **A** está en el registro **\$s3** (el *registro base*)

... y las variables **g** y **h** están asociadas a los registros **\$s1** y **\$s2**

- el compilador asocia variables con registros y coloca los arreglos en memoria

lw \$t0, 8(\$s3) —dos instrucciones en assembly: primero
add \$s1, \$s2, \$t0 —... copia $A[8]$ al registro $\$t0$; luego,
—... suma y pone el resultado en g

La memoria está organizada en *bytes* (de 8 bits c/u), cada uno con una dirección: 0, 1, 2,

Pero los computadores modernos operan sobre *palabras (words)* de 4 bytes (consecutivos, 32 bits),

... por lo que las direcciones de las palabras son 0, 4, 8, ...

- ... de modo que el ejemplo anterior requiere un ajuste

Veámoslo con la siguiente instrucción

sw (store) copia datos desde un registro a la memoria

$A[12] = h + A[8]$ —(una) sentencia en C, Java, etc.

lw \$t0, 32(\$s3) —tres instrucciones en assembly: copia
add \$t0, \$s2, \$t0 —... $A[8]$ al registro \$t0; suma; y
sw \$t0, 48(\$s3) —... copia el registro \$t0 a $A[12]$

En muchos programas, aparecen operandos constantes, o *inmediatos*, en las operaciones:

- una posibilidad es, primero, cargar la constante desde la memoria a un registro para, luego, poder usarla
- ... pero esto es muy lento

Nuestro lenguaje assembly tiene versiones de las instrucciones aritméticas en que uno de los operandos es una constante

P.ej., la instrucción *add-immediate*, or **addi**:

`addi $s3, $s3, 4` — $\$s3 = \$s3 + 4$

Es útil poder operar sobre conjuntos de bits dentro de una palabra o sobre bits individuales —operaciones lógicas:

típicamente, AND, OR, NOT y los *shifts*, *left* y *right*

Los *shifts* mueven todos los bits de una palabra a la izquierda o a la derecha, llenando los bits que quedan “vacíos” con 0s:

P.ej., si **\$s0** contiene

0000 0000 0000 0000 0000 0000 0000 1001₂ (= 9₁₀)

... y ejecutamos la instrucción

sll \$t2, \$s0, 4 —*shift left logical*

... para hacer un *shift left* de 4 posiciones dejando el resultado en **\$t2**, entonces en **\$t2** queda

0000 0000 0000 0000 0000 0000 1001 0000₂ (= 144₁₀)

and y **or** son operaciones bit a bit entre los contenidos de dos registros

P.ej., si **\$t1** y **\$t2** contienen

0000 0000 0000 0000 0011 1100 0000 0000₂

... y 0000 0000 0000 0000 0000 1101 1100 0000₂,

... respectivamente, entonces la ejecución de

and \$t0, \$t1, \$t2

... deja en **\$t0** el valor

0000 0000 0000 0000 0000 1100 0000 0000₂

Todo lenguaje de programación,

... incluyendo los lenguajes de máquina y los lenguajes *assembly* correspondientes,

... deben tener instrucciones para controlar la ejecución condicional de otras instrucciones:

- dependiendo de los datos de entrada y de los valores creados durante la ejecución del programa,
- ... se ejecutan unas instrucciones u otras

P.ej.,

beq *registro1, registro2, L1*

... significa “vaya a la instrucción etiquetada *L1* si el valor en *registro1 es igual* al valor en *registro2*” —*branch if equal*

bne *registro1, registro2, L1*

... significa “vaya a la instrucción etiquetada *L1* si el valor en *registro1 no es igual* al valor en *registro2*” —*branch if not equal*

Compilemos la siguiente sentencia a nuestro assembly:

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

Si las variables **f**, **g**, **h**, **i** y **j** corresponden a los registros **\$s0** a **\$s4**:

```
bne      $s3, $s4, Else
add      $s0, $s1, $s2
j Exit
Else: sub      $s0, $s1, $s2
Exit:
```

j Exit es un *branch incondicional* a la instrucción etiquetada **Exit**; la instrucción **j** se llama *jump*

El *assembler* (ensamblador) alivia al compilador y al programador de tener que calcular direcciones de memoria para los *branches*

Las decisiones también son importantes para repetir la ejecución de una computación —*loops*

Las mismas instrucciones condicionales sirven para estos casos

Compilemos el siguiente loop:

```
while (x[i] == k):  
    i = i+1
```

Supongamos que **i** y **k** corresponden a los registros **\$s3** y **\$s5**, y la base de **x** está en **\$s6**; y queremos guardar **x[i]** en **\$t0**

Primero, necesitamos la dirección de **x[i]** —sumamos **i** a la base de **x**, por lo que antes hay que multiplicar **i*4**, usando *shift left logical*

Loop:	sll	\$t1, \$s3, 2	—multiplicación i*4
	add	\$t1, \$t1, \$s6	—suma i + x[0]
	lw	\$t0, 0(\$t1)	—carga x[i] en \$t0
	bne	\$t0, \$s5, Exit	—el test del loop
	addi	\$s3, \$s3, 1	—incremento de i
	j	Loop	—volvemos al principio

Exit: