Estructura condicional

Organización de Computadoras 2019

Universidad Nacional de Quilmes

1. Introducción y motivación

En la tarea de programación puede surgir la necesidad de bifurcar el flujo del programa, es decir, ejecutar una acción u otra, dependiendo de una condición.

Los programas que construimos hasta ahora se ejecutan de manera secuencial -a excepción de la invocación a las rutinas- donde cada instrucción es leída de la memoria y como ya sabemos la posición de memoria a leer es la que almacena el registro PC. También se sabe que el valor de PC se incrementa en función del tamaño de cada instrucción, por lo que si se quiere evitar que la próxima instrucción a ejecutar sea siempre la misma, se necesita un mecanismo que permita modificar el valor del PC de manera condicional. Este mecanismo está dado por las instrucciones conocidas como **Saltos**.

Una condición que debe poder expresarse en cualquier lenguaje es la **verificación por igualdad** entre dos valores de dos variables o entre una variable y una constante. Por ejemplo, suponer la siguiente condición, expresada en un lenguaje de pseudo-código:

```
Si (R0 = 1) entonces
    R6 <-- 0x000A
```

Lo anterior puede expresarse en términos de un lenguaje ensamblador como sigue:

```
       {Comparar el contenido de R0 con 1}
       {si es diferente saltar abajo}
       MOV R6, 0x000A
abajo: fin
```

Para esto, la arquitectura `Q` propone una nueva versión `Q4` que incluye **instrucciones de salto** y una instrucción de comparación. Los **Saltos** son instrucciones que permiten modificar el valor del PC logrando que el programa continúe su ejecución en una instrucción distinta a la siguiente posición de memoria. La instrucción donde se desea continuar la ejecución, se debe señalar con una etiqueta.

Los saltos disponibles se clasifican en condicionales e incondicionales.

- **Saltos Incondicionales:** la actualización del PC se lleva a cabo siempre que se ejecute el salto
- **Saltos Condicionales:** la actualización del PC se lleva a cabo si se cumple una determinada condición

Instrucción	Descripción
JE	Igual / Cero
JNE	No igual
JLE	Menor o igual
JG	Mayor
JL	Menor
JGE	Mayor o igual
JLEU	Menor o igual sin signo
JGU	Mayor sin signo
JCS	Carry / Menor sin signo
JNEG	Negativo
JVS	Overflow

Cuadro 1: Saltos condicionales

La instrucción de salto incondicional es **JMP**; funciona similar a un **CALL**; tiene la forma: `JMP etiqueta` y al ejecutarse carga el PC con la dirección de memoria donde está `etiqueta`. A diferencia del **CALL** no modifica la pila.

Las instrucciones de Salto Condicional son **JE, JNE, JLE, JG, JL, JGE, JLEU, JGU, JCS, JNEG** y **JVS**. Cada una de estas evalúa una condición distinta (ver el cuadro 1) y tienen la siguiente sintaxis: `Jxx etiqueta`. La intención es modificar el PC si la condición del salto se cumple, de lo contrario no tiene efecto alguno.

El problema que se necesitaba resolver se puede detallar un poco mas aplicando el salto `JE` y el salto `JMP`, como sigue:

```
                    {Comparar el contenido de R0 con 1}
                    JE saltarParaAsginarA
                    JMP salir
saltarParaAsignarA: MOV R6, 0x000A
             salir: RET
```

Por último se necesita una instrucción que permita comparar dos valores de 16 bits para concluir, o no, si son iguales, y es es posible reformular este problema en términos de una resta:

$$A = B \iff A - B = 0$$

y entonces el problema se reduce a indicar si el resultado es cero.

El **CMP** es la instrucción que se incluye entre las instrucciones de dos operandos cuyo objetivo es comparar dos operandos por medio de una resta, antes de usar un salto condicional. Dicha resta, se lleva a cabo sólo a los fines de comparar, su resultado se descarta y esto implica que no se modifica el destino (a diferencia de las demás instrucciones de dos operandos).

De esta manera, la rutina que se estaba trabajando puede reformularse:

```
rutina:  CMP R0, 0x0001
```

```
                    JE saltarParaAsginarA
                    JMP salir
saltarParaAsignarA: MOV R6, 0x000A
             salir: RET
```

2. Funcionamiento de los saltos

La instrucción de salto incondicional **JMP** funciona similar a un **CALL** dado que tiene la forma `JMP etiqueta` y al ejecutarse carga el PC con la dirección de memoria donde está `etiqueta`. A diferencia del **CALL** no modifica la pila.

Por otro lado, los saltos pueden modificar el registro `PC` de dos maneras:

- **Salto Absoluto:** el nuevo valor del PC se expresa en términos de una dirección de memoria
- **Salto Relativo:** el nuevo valor del PC se expresa en términos de un desplazamiento con respecto a la siguiente instrucción

2.1. Ensamblado de saltos y CMP

El **JMP** es un salto absoluto, con lo cuál, el valor del **operando origen** debe ser la dirección de memoria donde se desea saltar (siguiente instrucción a ejecutar). Esta dirección se expresa como una etiqueta, por esta razón el **JMP** al igual que el **CALL** ocupará dos celdas, una que contiene el código de operación junto con el relleno y el modo de direccionamiento (en caso de una etiqueta: inmediato) y una segunda celda que contenga la dirección de memoria a la que se quiere saltar.

Operación	Cod Op	Efecto
JMP	1010	PC ← Origen

Los saltos restantes, es decir los condicionales, son saltos relativos. Este tipo de saltos no reemplazan el valor del PC con la dirección de memoria de la siguiente instrucción (a donde se quiere bifurcar), sino que se calcula dónde se tiene que saltar utilizando un desplazamiento. Es decir que al valor del PC se suma el valor del desplazamiento, expresado como un número en $CA2(8)$. Se utiliza **CA2** porque al disponer de números negativos es posible realizar saltos tanto hacia una instrucción posterior como una instrucción anterior.

`1111`	CodOp (4)	Desplazamiento(8)

Los primeros cuatro bits del formato de instrucción se corresponden con la cadena 1111_2, funcionando como prefijo del `CopOp`. Si **al evaluar la condición de salto** (ver Cuadro 2) esta se cumple, se le suma al PC el valor del **desplazamiento**, representado en $CA2(8)$. En caso contrario la instrucción no altera PC. Entonces el ciclo de ejecución para los flags condicionales puede detallarse como se indica en la siguiente figura:

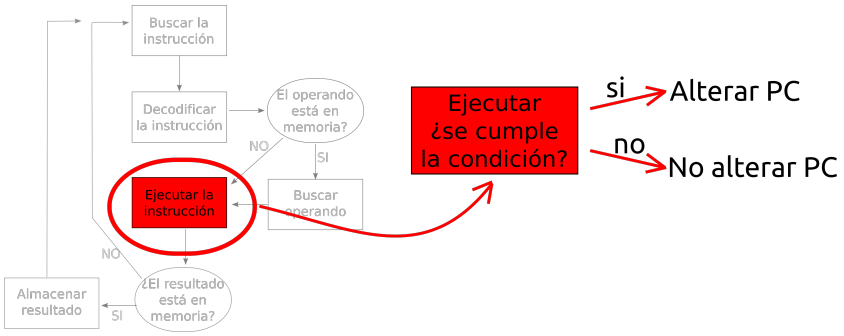

Figura 1: Ciclo de ejecución de instrucción

Codop	Op.	Descripción
0001	JE	Igual / Cero
1001	JNE	No igual
0010	JLE	Menor o igual
1010	JG	Mayor
0011	JL	Menor
1011	JGE	Mayor o igual
0100	JLEU	Menor o igual sin signo
1100	JGU	Mayor sin signo
0101	JCS	Carry / Menor sin signo
0110	JNEG	Negativo
0111	JVS	Overflow

Cuadro 2: Codigo de operación y condiciones de cada salto condicional

Por último la instrucción `CMP` se ensambla siguiendo el formato de instrucción descripto a continuación:

Operación	Cod Op	Efecto
CMP	0110	Dest - Origen

2.2. Cómo se implementa una comparación

Como se dijo en la sección 1, a la hora de resolver el desafío de comparar dos valores y concluir, o no, si son iguales, es posible reformular el problema en términos de una resta:$A = B \iff A - B = 0$ y entonces el problema se reduce a indicar si el resultado es cero.

Con la misma idea de restar los valores, se los puede comparar por mayor (o menor) como sigue:

$$A < B \iff A - B < 0$$

y este otro problema se traslada a determinar si el resultado es negativo. En un sistema en complemento a 2 esto se responde con una señal (el valor de un bit que sale de un circuito) que se encienda cuando el primer bit del resultado de operar $A - B$ es 1. Sin embargo, el primer bit no indica nada desde el punto

de vista de binario sin signo, y por lo tanto debemos pensar en otra forma de determinar si el minuendo (A) es menor al sustraendo (B). Para indicar esto se dispone del bit de acarreo (en inglés *carry*) o prestamo si es una resta (en inglés *borrow*) que la ALU genera como parte del resultado de la resta.

Entonces, cuando la ALU ejecuta una instrucción aritmética, además de realizar la operación y generar un resultado, calcula un conjunto de indicadores que **caracterizan ese resultado**. Estos indicadores se denominan *flags* o banderas y son bits que se almacenan conjuntamente en un registro de la CPU.

En el caso de la arquitectura Q se tienen 4 flags y cada uno indica una situación distinta:

Flag Z (Zero)

Toma el valor 1 cuando el resultado de una operación es una cadena completa de ceros. Es útil para la comparación por igual (A=B) mencionada arriba.

```
                  111
Ejemplo en una resta:   -  111   Z=1
                          ---
                          000

                   111
Ejemplo en una suma:   +  001   Z=1
                          ---
                          000
```

Flag N (Negative)

Toma el valor 1 cuando el primer bit del resultado es 1. Es útil para concluir si un valor es menor a otro en el contexto de CA2, como se describió antes.

```
                  100
Ejemplo en una resta:   -  001   N=0
                          ---
                          011

                   101
Ejemplo en una suma:   +  001   N=1
                          ---
                          110
```

Flag C (Carry)

Toma el valor 1 cuando la resta o la suma tuvieron acarreo o borrow. Es útil para concluir si un valor es menor a otro en el contexto de BSS, como se describió antes.

```
                  100
Ejemplo en una resta:   -  001   C=0
                          ---
                          011

                   111
Ejemplo en una suma:   +  001   C=1
                          ---
                          000
```

Es importante ver que el Carry indica una situación de error en el contexto del sistema BSS (ver figura 2) pero no implica un error en el caso de un sistema CA2 (ver figura 3)

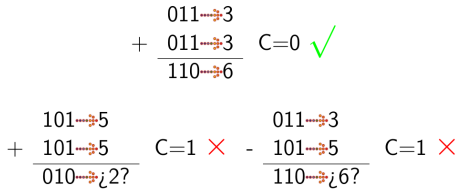

Figura 2: Significado del flag Carry en un contexto BSS

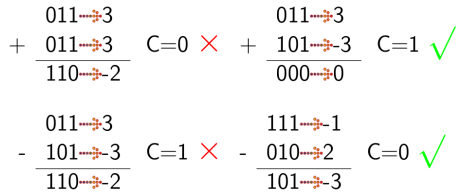

Figura 3: Significado del flag Carry en un contexto CA2

Flag V (Overflow)

Toma el valor 1 cuando el resultado no tiene el signo esperado, es decir en los siguientes casos

(a) Cuando al sumar dos positivos se obtiene un negativo

$$\begin{array}{r} \text{positivo} \\ + \ \underline{\text{positivo}} \\ \text{negativo} \end{array} \times \qquad \begin{array}{r} 010 \Rightarrow 2 \\ + \ \underline{010 \Rightarrow 2} \\ 100 \Rightarrow -4 \end{array}, \ V{=}1$$

(b) Cuando al sumar dos negativos se obtiene un positivo

$$\begin{array}{r} \text{negativo} \\ + \ \underline{\text{negativo}} \\ \text{positivo} \end{array} \times \qquad \begin{array}{r} 100 \Rightarrow -4 \\ + \ \underline{100 \Rightarrow -4} \\ 000 \Rightarrow 0 \end{array}, \ V{=}1$$

(c) Cuando al restarle algo positivo a algo negativo se obtiene algo positivo

$$\begin{array}{r} \text{negativo} \\ - \ \underline{\text{positivo}} \\ \text{positivo} \end{array} \times \qquad \begin{array}{r} 110 \Rightarrow -2 \\ - \ \underline{011 \Rightarrow 3} \\ 011 \Rightarrow 3 \end{array}, \ V{=}1$$

(d) Cuando al restarle algo negativo a algo positivo se obtiene algo negativo

$$\begin{array}{r} \text{positivo} \\ - \ \underline{\text{negativo}} \\ \text{negativo} \end{array} \times \qquad \begin{array}{r} 001 \Rightarrow 1 \\ - \ \underline{100 \Rightarrow -4} \\ 101 \Rightarrow -3 \end{array}, \ V{=}1$$

2.3. Saltos condicionales

A partir de los *flags* descriptos antes es posible describir las condiciones de los saltos condicionales, como se indica en la Tabla 3.

Op.	Descripción	Condición
JE	Igual / Cero	Z
JNE	No igual	$\overline{Z}$
JLE	Menor o igual	$Z + (N \oplus V)$
JG	Mayor	$\overline{Z + (N \oplus V)}$
JL	Menor	$N \oplus V)$
JGE	Mayor o igual	$\overline{(N \oplus V)}$
JLEU	Menor o igual sin signo	$C + Z$
JGU	Mayor sin signo	$\overline{(C + Z)}$
JCS	Carry / Menor sin signo	C
JNEG	Negativo	N
JVS	Overflow	V

Cuadro 3: Detalle de la condición de cada salto condicional

3. Ejemplos

Rutina con salto absoluto

```
rutina: MOV R0, 0x0000
        MOV R1, 0xAAAA
        JMP salir
        MOV R1, 0xBBBB
 salir: RET
```

Desde el punto de vista práctico esta rutina no tiene mucha utilidad, dado que realiza un salto absoluto y siempre va a evitar la ejecución de la instrucción `MOV R1, 0xBBBB`. Lo que se intenta mostrar, es que la instrucción `JMP`, modificará el PC asignando el valor de la dirección de memoria donde se encuentre ensamblada la instrucción `RET`

Rutina con salto condicional

```
    rutina: CMP R0, 0x0000
            JL menoracero
            CMP R0, 0x000A
            JG mayoradiez
            MOV R3, 0x0001
            JMP fin
menoracero: MOV R3, 0x0000
            JMP fin
mayoradiez: MOV R3, 0x0002
       fin: RET
```

Se enumera a continuación la secuencia de pasos y la explicación de cada uno de éstos:

- **CMP R0, 0x0000**, compara el contenido del registro R0 con el valor `0x0000`. Como es sabido, la instrucción `CMP` realiza una resta, lo que implica el cálculo de los flags, lo que deja el contexto preparado para ejecutar estructuras condicionales (Saltos)
- **JL menoracero**, especifica un salto condicional que cambiará el valor del PC sólo cuando se cumple que el contenido de RO sea menor que 0(cero). Esto se evalúa mediante los flags
- **CMP R0, 0x000A** si el flujo de ejecución procesa esta instrucción es porque la condición de la instrucción anterior (`JL menoracero`) no se cumplió. Nuevamente se realiza una comparación. Es similar al primer punto. Ahora, los flags dependerán de la resta entre el contenido de R0 y 0x000A
- **JG mayoradiez**, evalúa los flags, pero en este caso, chequea si RO (su contenido) es mayor estricto que 10
- **MOV R3, 0x0001**, asigna a R3 el valor 1
- **JMP fin**, realiza un salto absoluto equivalente al ejemplo 1
- **menoracero: MOV R3, 0x0000**, se puede ver que delante de la instrucción se encuentra la etiqueta "menoracero". La misma se utilizá para determinar qué instrucción se desea ejecutar en el caso que se cumpla la condición al momento del salto
- **JMP fin**, otro salto absoluto. Se utiliza para evitar la ejecución de la asignación siguiente y que en R3 quede el valor 0. De no existir esta instrucción se ejecutaría la siguiente y se "pisaría.[el] valor
- **mayoradiez: MOV R3, 0x0002**, se sigue la misma lógica que con la instrucción `menoracero: MOV R3, 0x0000`
- **fin: RET**, fin de ejecución de la rutina

Importante: notar que la instrucción que sigue a la de salto no es la que se ejecuta cuando se cumple la condición del mismo, sino la negativa. Por ello, al pensar un programa, generalmente es conveniente hacerlo utilizando las instrucciones de salto contrarias a las que intuitivamente pensamos. Por ejemplo, si queremos hacer una rutina que asigne el valor 0x0001 al registro R0 cuando R1 es mayor que 0x0004 haremos:

```
rutina: CMP R1, 0x0004
        JLE fin
        MOV R0, 0x0000
   fin: RET
```

Se utilizó JLE (menor o igual) cuando el enunciado mencionaba que se modifique R0 si R1 es mayor.

4. Verificar las rutinas

Para corroborar que las rutinas cumplen con la funcionalidad esperada se deben implementar otras rutinas o programas de *test* (pruebas/validación) para hacer un control de calidad sobre las primeras. Con este objetivo, se utiliza un registro que funciona como flag o indicador: si el registro vale 0 entonces la rutina pasó correctamente la validación. Si el registro vale 1 entonces el resultado obtenido no es el que se esperaba.

En cada caso se deben proveer valores que cumplan lo requerido por la rutina (aquello que se indica en el campo **requiere** de la documentación) y luego de invocarla se debe comparar el resultado obtenido con aquel valor que se esperaba según su documentación. Si coincide se carga el registro flag con el valor `F000`, y en caso contrario con el valor `FFFF`.

Esquema general: a continuación se describe el esquema general de una rutina de test

```
Pasaje de parámetros ⇒    testRutina:   MOV Rx, 0x...
Invocación ⇒                            CALL rutinaAValidar
Comparar con dato esperado ⇒            CMP Ry, 0x...
salto al caso de exito ⇒                JE funcionaBien
registrar fallo ⇒                       MOV R0, 0xFFFF
                                        JMP fin
registrar exito ⇒         funcionaBien: MOV R0, 0xF000
                                  fin:  RET
```

Ejemplo: considerando la documentación de la rutina `avg`:

```
;REQUIERE dos valores de 16 bits en R6 y R7
:MODIFICA --
;RETORNA en R6 el promedio ENTERO entre R6 y R7
```

Un test válido podría ser:

```
testAvg: MOV R6, 0x0008
         MOV R7, 0x000A
         CALL avg
         CMP R6, 0x0009
         JE funcionabien
         MOV R0, 0xFFFF
         JMP fin
funcionabien: MOV R0, 0xF000
fin: RET
```