Máscaras y repeticiones

Organización de computadoras 2019

Universidad Nacional de Quilmes

1. Máscaras

En muchas situaciones, dada una cadena de bits, es necesario trabajar solo con determinadas posiciones y que todas las restantes mantengan su valor original o bien tengan un valor que se desee.
Por ejemplo podría ser necesario lograr los siguientes efectos:

- Invertir ciertas posiciones dejando el resto intacto
- Invertir solo el último bit
- Conocer el valor del 3er bit de una cadena y que el resto de la caderna sea cero
- etc

Conceptualmente se trata de mantener solo los bits que son de interés, para ocultar los demás detrás de valores determinados. En la figura 1 se ilustra una cadena de 8 bits y una máscara de 4 *ventanas* en los bits menos significativos, los bits mas significativos se encuentran *bloqueados*. En la figura 2 se muestra el resultado de tapar una parte de la cadena para dejar los bits que si eran de interés.

Podemos pensar que la cadena de 8 bits codifica dos datos independientes, los 4 bits mas significativos son un desplazamiento en X y los 4 menos significativos un desplazamiento en Y.

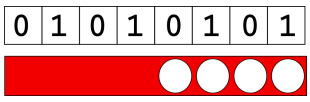

Figura 1: Cadena y máscara

Figura 2: Cadena y máscara

La tarea de aplicar una máscara es en realidad una operación lógica entre dos cadenas: la cadena con la información a analizar y otra cadena del mismo tamaño

	0101			0101
AND	0011	$\Rightarrow$	AND	0011
	????			0001
	0101			0101
XOR	0011	$\Rightarrow$	XOR	0011
	????			0010

Cuadro 1: Ejemplo de operaciones bit a bit

con una estructura determinada (bloqueos y ventanas), que denominaremos **máscara**. Las operaciones lógicas entre cadenas son en realidad operaciones que se llevan a cabo bit a bit. Por ejemplo ver el cuadro 1.

Por ejemplo, suponiendo que hay una operación lógica binaria **BIN** (de la cual no importa su efecto), se conoce una cadena **X** de 4 bits y una máscara **M** también de 4 bits, la operación sería la siguiente:

$$\begin{array}{rl} & X_3X_2X_1X_0 \\ \text{BIN} & M_3M_2M_1M_0 \\ \hline & R_3R_2R_1R_0 \end{array}$$

Donde, cada bit R_i es el resultado de aplicar X_i BIN M_i, por ejemplo X_2 BIN $M_2 = R_2$

En los siguientes apartados se analizará cada operación lógica en relación a los bloqueos y ventanas que necesita.

Conjunción

De la tabla de verdad de la conjunción se deduce que:

- Al utilizar en la máscara valor 1 en determinados bits, el resultado de la operación en esas mismas posiciones coincidirá con el valor de la cadena de la cual se desea extraer información. **Es decir que, en la conjunción, un bit en 1 en la máscara determina una ventana**

$$X \wedge 1 = X$$

- En cambio, donde la máscara tenga valor 0, el resultado tendrá valor 0. **Es decir que, en la conjunción, un bit en 0 en la máscara determina un bloqueo**.

$$X \wedge 0 = 0$$

De esta manera, siguiendo el ejemplo de la cadena anterior pero llevándolo a cadenas de 16 bits, la cadena de la mascara que se necesita es `0000000011111111`, ya que la aplicación bit a bit de la conjunción es como sigue (notar que en color rojo se marcan los bloqueos):

$$\begin{array}{rl} & 0101010101010101 \\ \wedge & 0000000011111111 \\ \hline & 0000000001010101 \end{array}$$

Esta cadena de resultado representa el desplazamiento en Y que ahora puede usarse de manera independiente. Veamos entonces como sería el uso de máscaras en una rutina, mediante la instrucción `AND`, que calcula la conjunción bit a bit.

```
MOV R3, 0x5555 ---> 0101010101010101
MOV R4, R3
AND R3, 0x00FF ---> 0000000011111111 (En R3 queda despY)
AND R4, 0xFF00 ---> 1111111100000000 (En R4 queda despX)
CALL dibujarLinea --> Requiere en R3 y R4 sus parametros
```

Disyunción

De la tabla de verdad de la disyunción se deduce que:

- Al utilizar en la máscara valor 1 en determinados bits, el resultado de la operación en esas mismas posiciones será 1 independientemente del valor de la cadena de la cual se quiere extraer información. **Es decir que, en la disyunción, un bit en 1 en la máscara determina un bloqueo**.

$$X \vee 1 = 1$$

- En cambio, donde la máscara tenga valor 0, el resultado coincidirá con el valor que tenga la cadena. **Es decir que, en la disyunción, un bit en 0 de la máscara determina una ventana**.

$$X \vee 0 = 1$$

Con esta operación de disyunción es posible enmascarar la cadena del ejemplo para analizar los 8 bits menos significativos, pero usando una máscara diferente: `11110000`:

$$\begin{array}{rr} & 0101010101010101 \\ \vee & 1111111100000000 \\ \hline & 1111111101010101 \end{array}$$

Esta cadena de resultado representa el desplazamiento en Y que ahora puede usarse de manera independiente. Veamos entonces como sería el uso de máscaras en una rutina, mediante la instrucción `OR`, que calcula la disynción bit a bit.

```
MOV R3, 0x5555 ---> 0101010101010101
OR  R3, 0xFF00 ---> 1111111100000000 (En R4 queda despY)
```

Or exclusivo

De la tabla de verdad del or exclusivo se deduce que:

- Al utilizar en la máscara valor 1 en determinados bits el valor del resultado de la operación en esas mismas posiciones será el contrario al original.

$$X \oplus 1 = \overline{X}$$

- En cambio, donde la máscara tenga valor 0, el resultado coincidirá con el valor que tenga la cadena, lo que denominamos una ventana

$$X \oplus 0 = X$$

Ejemplo: $\begin{array}{rr} & 0110 \\ \text{XOR} & 0101 \\ \hline & 0011 \end{array}$

1.1. Rutinas con máscaras

Como se dijo anteriormente, una rutina aplica máscaras para destacar determinados bits dentro de una cadena y a partir de eso llevar a cabo ciertas tareas, que pueden estar condicionadas a los valores de dichos bits. Por ejemplo, considerar la documentación de la rutina `esPar`:

Requiere	Un valor BSS(16) en el registro R6
Modifica	??
Retorna	el código 000F en R5 si R6 es par y un el código 000A en caso contrario

Una posible solución para esta rutina es:

```
esPar: MOV R4, R6
       AND R4, 0x0001 --> descartar bits restantes
       CMP R4, 0x0001
       JNE siEs
       MOV R5, 0x000A --> valor para indicar que es impar
       JMP fin
siEs:  MOV R5, 0x000F --> valor para indicar que es par
 fin:  RET
```

Controlando las rutinas

Para corroborar que las rutinas cumplen con el objetivo esperado, se deben utilizar otras rutinas de *test* o pruebas para hacer control de calidad sobre las primeras.

En cada caso se deben proveer valores que cumplan lo requerido por la rutina (campo **requiere**) y luego de invocarla se debe comparar el resultado con el valor esperado, poniendo en R0 el valor `F000` en caso de exito o el valor `FFFF` en caso de fallo.

Volviendo al ejemplo de la rutina `esPar`, el siguiente test controla que al darle el valor 8 (que es par) la rutina `esPar` retorna el código esperado (`000F`).

```
testEsPar: MOV R6, 0x0008
         CALL esPar
         CMP R5, 0x000F
         JE funcionabien
         MOV R0, 0xFFFF
         JMP fin
funcionabien: MOV R0, 0xF000
fin: RET
```

Por otro lado, el siguiente test controla el caso impar, dándole a la rutina el valor `AAA1` y esperando encontrar el código `000A`.

```
testEsImpar: MOV R6, 0xAAA1
         CALL esPar
         CMP R5, 0x000A
         JE funcionabien
         MOV R0, 0xFFFF
         JMP fin
funcionabien: MOV R0, 0xF000
fin: RET
```

```
        INICIALIZACION // Inicializar contadores o acumuladores
 ciclo: CMP X, Y // X e Y pueden ser registros o celdas
        JE finCiclo //Puede ser cualquier salto condicional
        INSTRUCCIONES //Deberian modificar a X o Y
        JMP ciclo //Nos permite volver a la condicion del ciclo
finCiclo: // Finaliza repeticion, continua el programa
```

Cuadro 2: Esquema de una iteración

2. Repeticiones

Mediante el uso de los saltos podemos mover el PC hacia atrás y lograr que una secuencia de instrucciones se ejecuten mas de una vez. Es importante que tengamos alguna forma de parar está repetición, ya que de lo contrario el programa entraría en un ciclo infinito, repitiendo y repitiendo dicha secuencia de instrucciones y no terminando nunca. En general esto lo hacemos con saltos condicionales, revisando alguna condición que puede cambiar cada vez que se ejecuta la secuencia de instrucciones. Esto nos lleva al concepto de realizar iteracciones.

Existen muchos problemas que requieren iteraciones (repeticiones) para ser resueltos, es decir, requieren realizar la misma operación varias veces hasta llegar al resultado final. Por ejemplo, supongamos que queremos realizar una multiplicación sin utilizar la instruccion `MUL`: Calcular A*B se podría realizar sumando B veces el valor A (o viceversa). Pensándolo como un programa podría ser:

1. Inicializar un acumulador R en cero.
2. Verificar si B es cero.
3. Si B es cero, salir.
4. Asignar $R \leftarrow R + A$ (Recordemos que la primera vez R es cero)
5. Asignar $B \leftarrow B - 1$ (Al ir restando 1 a B nos acercamos al final del problema)
6. Volver al paso 2.

Esto se puede resolver con lo que conocemos de la arquitectura Q, ya que con saltos condicionales podemos realizar el paso 3.
Un posible esquema para una repetición se muestra en el cuadro 2. Es importante notar que pueden hacerse repeticiones de otras formas, este es un esquema a modo de ejemplo.