ÍNDICE

IV
CLASE 4

Estos son los apuntes no oficiales del curso CC3301-1 cursado en Primavera 2022 bajo las cátedras del docente Luis Mateu B. El material usado es directamente obtenido de cátedras, en adición a los materiales del curso. [1].

1 Introducción

Para esta clase hay un material audio/visual preparado en un formato especial. Es muy similar a una clase con pizarra. Las instrucciones para bajar la clase y el programa que se necesita para ver la clase están acá, en la sección clases disponibles con el relator, operaciones con bits. En los apuntes corresponde al capítulo de operaciones con bits. [2]

Es importante que revisen este material para la clase auxiliar del viernes 18, porque será sobre operaciones con bits.

2 Operaciones con bits

2.1 Tabla de operadores lógicos

Símbolo	Operación	Significado	Ejemplo (1100 op 1010)
&	and	**y** bit a bit	1000
|	or	**o** bit a bit	1110
∧	xor	**o exclusivo** bit a bit	0110

2.2 Tabla de la negación bit a bit

Símbolo	Operación	Significado	Ejemplo (10110111)
∼	not	negación bit a bit	01001000

2.3 Tabla de desplazamientos

Símbolo	Operación	Significado	Ejemplo (10111011 op 2)	Equivalencia
<<	shift left	desp. de bits a la izquierda	11101100	$x<<y \;\; x \cdot 2^y$
>>	unsigned shift right	desp. de bits a la derecha	00101110	$x>>y \;\; \lfloor \frac{x}{2^y} \rfloor$
>>	signed shift right	desp. de bits a la derecha	11101110	$x>>y \;\; \lfloor \frac{x}{2^y} \rfloor$

2.4 Desplazamiento a la derecha de números negativos

Note que si desplazamos -5 en un bit, el resultado es -3 y no -2, pues $\lfloor \frac{-5}{2^1} \rfloor = \lfloor -2{,}5 \rfloor = -3$.

En matemáticas $\lfloor -2{,}5 \rfloor = \text{máx}\,\{x \mid x \in \mathbb{Z} \wedge x \leq -2{,}5\}$

2.4.1 *Ejercicio*

Tome un tiempo para verificar que $\underbrace{11111011}_{-5} >> 1 \equiv \underbrace{11111101}_{-3}$

3 Casos de borde de los desplazamientos

3.1 Desplazamientos mayores al tamaño de un entero

Se debe tener cuidado con los desplazamientos, por ejemplo:

Consideremos la siguiente asignación de variables.

```
int a = 1; // a de 32 bits
int b = a << 32;
```

Frente a este desplazamiento de **b** nos preguntamos. ¿Cuánto vale b?.

Se espera que se naturalmente nuestra respuesta sea 0, pero sucede que este valor de b calculado de esta manera no está especificado en C. En procesadores intel o amd b es 1.

La regla general es que los resultados de **x** << **n** y x >> n no están especificados si **n** es mayor o igual al tamaño de **x** en bits o **n** < 0. Tampoco están definidos para valores negativos de **n**.

3.2 Por qué desplazamientos no especificados

3.2.1 *La culpa del legado*

¿Qué acaba de ocurrir? Descompongamos **n** en sus 32 bits:

$$n = n_{31}n_{30}...n_5n_4...n_0$$

Resulta que en procesadores *intel* cuando se desplaza un entero **x** de 32 bits en **n** solo se consideran los 5 bits menos significativos de **n**. Como 32 se representa en binario como 32 = 00...10...0, sus 5 bits menos significativos son ceros y, en consecuencia, se termina desplazando en cero bits, por eso el resultado es el mismo x.

Esto no es un error, sino que es culpa del legado. En los años 70, el presupuesto en transistores para los primeros procesadores de *intel* eran apenas 8000 transistores, de usar más transistores el chip resultaría muy caro, debido a ello se debía optar por una política de economizar de transistores como fuera posible. Por ejemplo: Al desplazar, ignorando los bits más significativos de n, puesto que desplazar en el tamaño de x o más no tenía utilidad práctica.

El problema es que una vez tomada esa decisión, intel tendría que calcular para siempre los desplazamientos de esa manera. Después de todo, intel debe su éxito debido a la compatibilidad que han tenido sus procesadores nuevos con sus procesadores antiguos. Cambiar el cómo se hacen los desplazamientos hubiese roto unos cuantos programas.

En otros procesadores el resultado sí es el esperado (ejemplo a << 32 sí es 0). Si C especificara que el resultado es 0 los desplazamientos en C serían caros en procesadores intel, pues habría que consultar si n ≥ |x| (montón de ciclos y el desplazamiento sería mayor a un ciclo del reloj). Por eso se prefirió no especificarlo en C, de esta forma los desplazamientos son eficientes en todos los procesadores a pesar de que puedan dar resultados distintos en casos de borde.

4 Ejemplo: conjuntos de frutas con mapas de bits

A continuación veremos un primer ejemplo de aplicación de las operaciones con bits. Consiste en programar las operaciones sobre conjuntos representados por medio de mapas de bits. Las típicas operaciones sobre conjuntos son: Agregar un elemento, determinar pertenencia, unir conjuntos, etc.

4.1 El tipo Fruta

Considere la siguiente definición de tipo en C.

```
typedef enum fruta {
    manzana,pera,guinda,durazno,damasco}
    Fruta;
Fruta f = durazno; // f ≡ 3
f = f + 1; // f ≡ 4 ≡ damasco
```

Código 1: Ejemplo de definición de tipo en C

Sin entrar en detalles de sintaxis de esta declaración, aseguremos que la enumeración de frutas es manzana (constante 0), pera (constante 1), guinda (constante 2), durazno y damasco. El nuevo tipo se llama Fruta (tipo entero)

Si se declara una fruta `f` con valor inicial durazno, esto es equivalente a que el valor inicial del entero `f` es 3, porque durazno es la constante 3. Es perfectamente legítimo escribir `f=f+1`, con lo que `f` será 4 (guarda un damasco). Por supuesto, este ejemplo no tiene mucho sentido, pero se hace énfasis en que el lenguaje lo permite y el resultado es siempre el mismo.

Visitemos un ejemplo con sentido. Se define el nuevo tipo conjunto, será un conjunto de frutas y se representa mediante un entero de tipo int. Se declara el conjunto `c`, la variable `c` es simplemente un int destinado a guardar conjuntos de frutas. Se codecaptionompone el entero c en todos sus bits. La idea es que el bit menos significativos representa la presencia (1) o ausencia (0) de una manzana en el conjunto.

4.2 El tipo conjuntos de frutas

```
typedef int Conjunto // de frutas
Conjunto c;
    c
```

4.3 Mapas de bits

Se hace la codecaptionomposición de `c` en bits ($c = ...c_5c_4c_3c_2c_1c_0$). Basándonos en lo anterior, c_1 representa presencia de la pera, por ejemplo.

Ejercicio ¿Qué frutas están presentes en el conjunto ...010010?. [1]

Esta representación para un conjunto se llama **mapa de bits** y es muy eficiente para representar conjuntos de toda clase

4.4 Implementación eficiente de operaciones sobre conjuntos

Calculemos 1 desplazado en f bits. ¿Qué representa?. Por ejemplo $1 << 3$ (3 representa al durazno).

Entonces 1 desplazado en 3 es ...001000. Lo que es equivalente al **singleton durazno** $\equiv$ {durazno}.

Entonces 1 desplazado en f bits corresponde a contruir un conjunto con solo `f` como fruta. Ponga atención a la siguiente función. La gracia de representar esta manera los conjuntos de fruta es que el costo de singleton es solo un ciclo del reloj (lo que toma el desplazamiento)

```
Conjunto singleton(Fruta f) {
    return 1 << f;
}
```

[1] La respuesta es la pera y el damasco

4.4.1 *Más operaciones (Unión, Agregar, Pertenece)*

```
Conjunto Union(Conjunto r, Conjunto s) {
    return r | s;
}
```

```
Conjunto agregar(Fruta f, Conjunto s) {
    return Union(s, singleton(f));
}
```

```
int pertenece(Frtuta f, Conjunto s) {
    return s & singleton(f);
}
```

4.4.2 *Complemento de un conjunto*

```
Conjunto complemento(Conjunto s) {
    return  s
}
```

5 Mostrar un conjunto y ejemplo de uso

5.1 Mostrar un conjunto

```
void monstar(Conjunto s) {
    for(Fruta f = manzana; f <= damasco; f++) {
        if(pertenece(f,s)) {
            switch(f)   {
            case manzana: printf("manzana");break;
            case pera: printf("pera");break;
            case guinda: printf("guinda");break;
            case durazno: printf("durazno");break;
            case damasco: printf("damasco");break;
            }
        }
    }
    printf("\n")
}
```

5.2 Ejemplo de uso de conjuntos

```
int main() {
    Conjunto s1 = agregar(manzana,
                  agregar(guinda,
                    singleton(damasco)));
    Conjunto s2 = complemento(s1);
    mostrar(s1);
    mostrar(s2);
    return 0;
}
```

5.3 Ejercicios

- Programe una función que entregue la intersección de 2 conjuntos (en no más de 1 ciclo del reloj)
- Programe una función que entregue la diferencia (de 2 conjuntos) (en no más de 2 ciclos del reloj)

6 Ejemplo: extracción de bits usando máscaras

A menudo, en aplicaciones que manipulan grandes volúmenes de información, se requiere ser muy cuidadosos con el uso de la memoria, por eso se colocan múltiples enteros pequeños en un solo entero de 32 bits. Por ejemplo:

Considere un entero **x** codecaptionompuesto en todos sus bits, desde el más significativo (x_{31}) hasta el menos significativo (x_0). **x** guarda un número pequeño de tan solo k bits, desde el bit x_i hasta el bit x_{i-k+1}. El problema consiste en extraer ese número pequeño, es decir, al llamar la función extraer(x,i,k), se debe entregar como resultado un entero que contiene x_i hasta x_{i-k+1} en los bits menos significativos y 0 en los más significativos.

Esto se logra por etapas.

1) Primero tenemos todos los bits de x

2) Lo transformamos borrando los bits que están antes de x_i

3) Desplazamos el resultado a la derecha en $i - k + 1$ bits

6.1 Máscaras de bits

Volvamos a la máscara de bits. Las máscaras de bits son enteros que se usan para borrar o encender bits en otro entero. El principio es que

- x & m: borra los bits de x que en m sean 0 (porque al hacer un *and* con cualquier bit con 0, se obtiene 0)
- x | m: enciende los bits de (porque al hacer un *or* de cualquier bit con 1, el resultado siempre es 1)

En el problema que queremos resolver se necesita borrar los 31-i primeros bits, necesitamos una máscara con los 31-i primeros bits en 0.

En general para construir una máscara con los j primeros bits en 0, se inicia con -1, que tiene todos sus bits en 1, luego se convierte al tipo sin signo.

```
((unsigned)-1) >> j
```

Luego se desplaza a la derecha en j bits, esto rellena por la izquierda con j bits en 0, es la máscara que necesitamos.

Para construir una máscara con los últimos j bits en 0, partimos con un -1 y lo desplazamos a la izquierda en j bits, que rellena por la derecha con j bits en 0. Esa es la máscara.

```
((unsigned)-1) << j
```

6.2 La función extraer

6.2.1 *Solución 1*

Esta es la primera solución a la función extraer: Entrega un entero sin signo, recibe como parámetros un entero x sin signo, un entero i con la posición del bit más significativo del número que se desea extraer, y el entero k que corresponde a su tamaño en bits.

El resultado de la función es x & la máscara que tiene el $31 - i$ primeros bits en cero, es decir, corresponde al primer caso de máscara. Con esto borramos los primeros $k - i$ bits, faltando una etapa.

Desplazamos en $i - k + 1$ bits, con lo que x_{i-k+1} quedó en el bit menos significativo.

6.2.2 *Solución 2*

Sin usar máscaras. Consiste en desplazar x a la izquierda en $31 - i$ bits, eliminando por completo los $31 - i$ bits de la izquierda, y luego desplazando en 31-i bits a la derecha, esto rellena con $31 - i$ bits en 0. Pero además debemos desplazar a la izquierda otros $i - k + 1$ bits. Simplificando nos queda $32 - k$bits. Es la solución más directa, pero en general para resolver los problemas típicos de manejos de bits, usted necesitará usar máscaras y desplazamientos.

REFERENCIAS

[1] Mateu, Luis: *Programación de Software de Sistemas - Novedades.* `https://www.u-cursos.cl/ingenieria/2022/1/CC3301/1/novedades/`, 2022.

[2] Mateu, Luis: *Programación de Software de Sistemas - Apuntes.* `https://wiki.dcc.uchile.cl/cc3301/temario/`, 2022.