Estructura de Computadores
Tema 2. Representación de programas a nivel máquina
Bloque II: Control

Saltos, ajustes y copias condicionales
Traducción de sentencias condicionales y bucles

Dpto. Arquitectura y Tecnología de Computadores, Universidad de Granada

Índice

1. Partiendo de un código real

Vamos a motivar nuestra discusión analizando el código en un caso real: la plataforma educativa SWAD. El núcleo de swad, llamado **swad-core**, es un programa compilado de unos 2,1 MiB que se ejecuta en un servidor GNU/Linux (este programa es llamado por el servidor web Apache) cada vez que un usuario pulsa en un enlace, icono o botón de la plataforma en su navegador (Fig. 1). Se puede acceder al código fuente de swad-core en GitHub: https://github.com/acanas/swad-core. En octubre de 2017, swad-core era un programa de 234 345 líneas escrito en C y compuesto por 90 módulos. Uno de estos módulos es **swad_user**, constituido por los archivos fuente `swad_user.c` y `swad_user.h`, y que engloba diversas funcionalidades relacionadas con el listado y gestión de usuarios. Dentro de `swad_user.c` hay 218 funciones y nos vamos a centrar en una de ellas, la función **`Usr_ICanChangeOtherUsrData`**, llamada cuando queremos comprobar si es posible modificar los datos de otro usuario. Llamamos a esta función, por ejemplo, cuando queremos administrar un usuario dentro de una asignatura (Fig. 1).

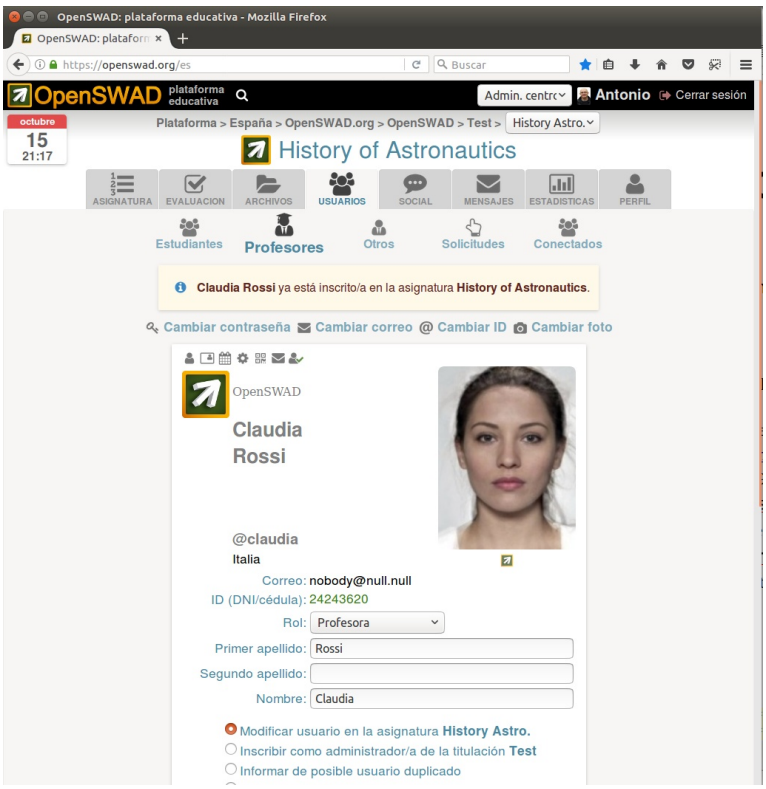


Figura 1. Captura de pantalla de la plataforma educativa SWAD. La función Usr_ICanChangeOtherUsrData es llamada para comprobar si el usuario identificado puede editar, entre otros campos, el nombre y apellidos del usuario mostrado. La función Enr_PutActionsRegRemOneUsr es llamada para mostrar las posibles acciones a realizar (lista inferior).

1.1 Código fuente en C

Este es el código fuente de la función **`Usr_ICanChangeOtherUsrData`**, correspondiente a la versión 17.25.4, del 24 de octubre de 2017:

```
/*****************************************************************************/
/*************** Check if I can change another user's data *******************/
/*****************************************************************************/

bool Usr_ICanChangeOtherUsrData (const struct UsrData *UsrDat)
  {
   if (UsrDat->UsrCod == Gbl.Usrs.Me.UsrDat.UsrCod) // It's me
      return true;

   switch (Gbl.Usrs.Me.Role.Logged)
     {
      case Rol_TCH:
         /* Check 1: I can change data of users who do not exist in database */
         if (UsrDat->UsrCod <= 0)    // User does not exist (creating a new user)
            return true;

         /* Check 2: I change data of users without password */
         if (!UsrDat->Password[0])   // User has no password (never logged)
            return true;

         return false;
      case Rol_DEG_ADM:
      case Rol_CTR_ADM:
      case Rol_INS_ADM:
      case Rol_SYS_ADM:
         return Usr_ICanEditOtherUsr (UsrDat);
      default:
         return false;
     }
  }
```

1.2 Código en ensamblador

Podemos compilar el archivo `swad_user.c` con un nivel de optimización 2 mediante la orden:

```
gcc -O2 -S swad_user.c -o swad_user.s
```

para obtener el archivo en ensamblador de x86-64 `swad_user.s`.

El fragmento del código en ensamblador correspondiente a la función `Usr_ICanChangeOtherUsrData` es el siguiente:

```
        .globl  Usr_ICanChangeOtherUsrData
        .type   Usr_ICanChangeOtherUsrData, @function
Usr_ICanChangeOtherUsrData:
.LFB96:
        .cfi_startproc
        movq    (%rdi), %rax
        cmpq    Gbl+84936(%rip), %rax
        je      .L556
        movl    Gbl+97440(%rip), %edx
        cmpl    $5, %edx
        je      .L552
        jb      .L555
        cmpl    $9, %edx
        ja      .L555
        jmp     Usr_ICanEditOtherUsr
        .p2align 4,,10
        .p2align 3
```

```
.L555:
        xorl    %eax, %eax
        ret
        .p2align 4,,10
        .p2align 3
.L552:
        testq   %rax, %rax
        jle     .L556
        cmpb    $0, 345(%rdi)
        sete    %al
        ret
        .p2align 4,,10
        .p2align 3
.L556:
        movl    $1, %eax
        ret
        .cfi_endproc
```

1.3 Análisis de la traducción de C a ensamblador

Vamos a analizar la traducción que realiza el compilador centrándonos en las instrucciones máquina, sin tener en cuenta las directivas. El lector interesado en esas directivas puede consultar los siguientes documentos:

- *CFI directives in assembly files*
  https://www.imperialviolet.org/2017/01/18/cfi.html
- *GNU as. .p2align directive*
  https://sourceware.org/binutils/docs/as/P2align.html

Analicemos la traducción del compilador por partes:

1

C

```
bool Usr_ICanChangeOtherUsrData (const struct UsrData *UsrDat)
  {
   if (UsrDat->UsrCod == Gbl.Usrs.Me.UsrDat.UsrCod) // It's me
      return true;
```

Ensamblador

```
        movq    (%rdi), %rax
        cmpq    Gbl+84936(%rip), %rax
        je      .L556

...

.L556:
        movl    $1, %eax
        ret
```

- El primer parámetro de una función se pasa a esta siguiendo la convención de llamada x86-64 en Linux a través del registro `rdi`. En este caso se trata de `UsrDat`, un puntero a una estructura `UsrData` que contiene los datos del usuario que al que vamos a administrar (la profesora Claudia Rossi en la figura).
- La función en ensamblador comienza con la instrucción `movq (%rdi),%rax`, que copia en `rax` el valor del código de usuario `UsrDat->UsrCod`. El desplazamiento en ese direccionamiento es 0 porque el campo `UsrCod` es el primero de la estructura. La copia es de 64 bits porque `UsrCod` es de tipo `long`, ocupando 8 bytes:

```
struct UsrData
  {
```

```
    long UsrCod;
    ...
}
```

- La instrucción `cmpq Gbl+84936(%rip),%rax` compara el valor de `UsrDat->UsrCod` (ahora en `rax`) con el valor de `Gbl.Usrs.Me.UsrDat.UsrCod` que está en la posición de memoria apuntada por `Gbl+84936(%rip)`. 84936 es el desplazamiento del campo `Usrs.Me.UsrDat.UsrCod` dentro de la estructura global `Gbl` que contiene el estado del programa. En este caso se usa direccionamiento relativo al contador de programa `rip`. La instrucción `cmp` (*compare*, comparar) resta el contenido de la posición de memoria `Gbl+84936(%rip)` del contenido de `rax` , es decir `UsrDat->UsrCod – Gbl.Usrs.Me.UsrDat.UsrCod`. El resultado de la resta no se almacena en ningún sitio, pero los indicadores de estado en el registro RFLAGS se activan de acuerdo con el resultado.
- La instrucción `je .L556` (*jump if equal*, saltar si igual) saltará a la etiqueta `.L556` si el indicador de estado ZF está activado (si vale 1), y esto ocurre cuando el resultado de la resta es 0. A – B = 0 equivale a decir que A = B.
- En la dirección de la etiqueta `.L556`, la instrucción `movl $1,%eax` copia un 1 (*true*) en el registro `eax`, que según la convención de llamadas es el lugar donde se devuelve el resultado. Realmente en este caso lo importante es devolver 1 en `al`, ya que el tipo bool tiene un tamaño de un byte.
- Por último, `ret` devuelve el control al lugar en el que se hizo la llamada a esta función.

2

C

```
   switch (Gbl.Usrs.Me.Role.Logged)
     {
      case Rol_TCH:
        /* Check 1: I can change data of users who do not exist in database */
         if (UsrDat->UsrCod <= 0)    // User does not exist (creating a new user)
            return true;

         /* Check 2: I change data of users without password */
         if (!UsrDat->Password[0])   // User has no password (never logged)
            return true;

         return false;
```

Ensamblador

```
        movl    Gbl+97440(%rip), %edx
        cmpl    $5, %edx
        je      .L552
...

.L552:
        testq   %rax, %rax
        jle     .L556
        cmpb    $0, 345(%rdi)
        sete    %al
        ret
...

.L556:
        movl    $1, %eax
        ret
```

- La instrucción `movl Gbl+97440(%rip),%edx` copia el rol del usuario identificado, almacenado en `Gbl.Usrs.Me.Role.Logged`, en el registro `edx`.

- La instrucción `cmpl $5,%edx` resta `edx` menos 5 para comparar el rol del usuario con `Rol_TCH` (dato enumerado con el valor 5, que indica que el usuario está identificado con el rol de profesor). Si el rol es profesor, es decir, si `edx` vale 5, el indicador de cero ZF se pondrá a 1 como resultado de la resta.
- La instrucción `je .L552` (*jump if equal*, saltar si igual) examina ese indicador ZF; si vale 1 (`Gbl.Usrs.Me.Role.Logged` vale 5) el control se transferirá a la instrucción en la etiqueta `.L552`; si vale 0 el programa continuará por la instrucción siguiente (el resto de los casos del switch que veremos más abajo).
- En `.L552` la instrucción `testq %rax,%rax` realiza la operación *and* bit a bit del contenido del registro `rax` consigo mismo con el fin de activar los indicadores de estado de cara a un salto condicional posterior. Recordemos que `rax` contiene el valor del código de usuario `UsrDat->UsrCod`.
- La instrucción `jle .L556` (*jump if less or equal*, saltar si menor o igual) examina ciertos indicadores de estado (después veremos cuáles) y salta a la instrucción en la dirección etiquetada como `.L556` si esos bits de estado indican “menor o igual”, es decir, salta si `rax` (que contiene el código de usuario `UsrDat->UsrCod`) es menor o igual que 0. En caso contrario, el flujo del programa continúa por la siguiente instrucción.
- En `.L556` la instrucción `movl $1,%eax` copia un 1 (*true*) en el registro `eax`, que como hemos visto antes es el lugar donde se devuelve el resultado.
- `ret` devuelve el control al lugar en el que se hizo la llamada a esta función.
- Si `jle .L556` no salta, es decir, si `UsrDat->UsrCod` > 0, la siguiente instrucción que se ejecuta es `cmpb $0, 345(%rdi)`, que compara el byte en `UsrDat->Password[0]` con 0 reflejando la comparación en los indicadores de estado. Si `UsrDat->Password[0]` es cero significa que la cadena `Password` está vacía, que el usuario aún no tiene contraseña; esto es equivalente a decir que no ha entrado aún en la plataforma. Recordemos que `rdi` contiene el puntero `UsrDat` a una estructura `UsrData`. Con el desplazamiento 345 direccionamos el primer byte del campo `Password` (345 = 8 + 44 + 256 + 4 + 16 + 17):

```
struct UsrData
  {
   long UsrCod;                                                        //   8 bytes
   char EncryptedUsrCod [Cry_BYTES_ENCRYPTED_STR_SHA256_BASE64 + 1];   //  44 bytes
   char UsrIDNickOrEmail[Cns_MAX_BYTES_EMAIL_ADDRESS + 1];             // 256 bytes
                              //  4 bytes de relleno para que Ids comience alineada a 8
   struct
     {
      struct ListIDs *List;   //  8 bytes
      unsigned Num;           //  4 bytes
     } Ids;                                                            //  16 bytes
   char Nickname        [Nck_MAX_BYTES_NICKNAME_WITHOUT_ARROBA + 1];   //  17 bytes
   char Password        [Pwd_BYTES_ENCRYPTED_PASSWORD + 1];
   ...
```

```
}
```

- La instrucción `sete %al` pone el registro `al` a 1 (*true*) "si igual", es decir, si el resultado de la última instrucción aritmético-lógica activó ZF. En nuestro caso `al` (el lugar donde se devuelve el resultado booleano de la función) valdrá:
  - 1 (*true*) si `UsrDat->Password[0]` es cero, cadena vacía.
  - 0 (*false*) si `UsrDat->Password[0]` es distinto de cero, cadena no vacía.

  Los 56 bits más significativos del registro `rax` no se modifican, pero como el tipo `bool` es de tamaño byte sólo importa lo que devolvamos en `al`.
- `ret` devuelve el control al lugar en el que se hizo la llamada a esta función.

3

C

```
      case Rol_DEG_ADM:
      case Rol_CTR_ADM:
      case Rol_INS_ADM:
      case Rol_SYS_ADM:
         return Usr_ICanEditOtherUsr (UsrDat);
      default:
        return false;
```

Ensamblador

```
        jb      .L555
        cmpl    $9, %edx
        ja      .L555
        jmp     Usr_ICanEditOtherUsr
...

.L555:
        xorl    %eax, %eax
        ret
```

- Para entender este código hay que tener en cuenta los valores de los siguientes roles:
  - `Rol_DEG_ADM` (administrador de titulación, valor 6)
  - `Rol_CTR_ADM` (administrador de centro, valor 7)
  - `Rol_INS_ADM` (administrador de institución, valor 8)
  - `Rol_SYS_ADM` (administrador del sistema, valor 9)
- La instrucción `jb .L555` (*jump if less*, saltar si menor) salta a `.L555` si el resultado de la comparación anterior `cmpl $5,%edx` indica que el contenido de `edx` (el rol de usuario `Gbl.Usrs.Me.Role.Logged`) es menor que 5. La comparación si igual a 5 ya se ha realizado antes, de modo que si la instrucción no salta sabemos que el contenido de `edx` no sólo es mayor o igual que 5, sino que es mayor estricto que 5.
- La instrucción `cmpl $9,%edx` compara el registro `edx` con 9.
- `ja .L555` salta a `.L555` si `eax` es mayor estricto que 9. Teniendo en cuenta las comparaciones anteriores, se salta a `.L555` cuando el valor de `Gbl.Usrs.Me.Role.Logged` se encuentra fuera del intervalo [6,9].
- En `.L555` la instrucción `xorl %eax, %eax` pone a cero el registro `eax` para devolver *false*.
- `ret` devuelve el control al lugar en el que se hizo la llamada a esta función.
- En el caso de que el valor de `Gbl.Usrs.Me.Role.Logged` se encuentre dentro del intervalo [6,9], se salta a la función `Usr_IcanEditOtherUsr` mediante la instrucción `jmp Usr_IcanEditOtherUsr`. Podemos pensar que el compilador debería haber llamado a `Usr_IcanEditOtherUsr` con la instrucción `call Usr_IcanEditOtherUsr` y después retornar con `ret` en lugar de saltar con

`jmp`. Usar `jmp` es una optimización que puede realizar el compilador en este caso porque ambas funciones, la que llama (`Usr_IcanChangeOtherUsrData`) y la que es llamada (`Usr_IcanEditOtherUsr`) devuelven un dato booleano. Cuando `Usr_IcanEditOtherUsr` retorne con `ret` no lo hará a nuestra función `Usr_IcanChangeOtherUsrData`, sino a la función que la llamó, ya que en la cabecera de la pila sigue el valor del contador de programa `rip` introducido por la llamada `call Usr_IcanChangeOtherUsrData`.

2. Instrucciones de comparación

En el ejemplo anterior podemos comprobar que en lenguaje máquina no existen estructuras del tipo if-then-else. En lugar de ello, se usan combinaciones de comparaciones y saltos para implementar cualquier estructura de control. Primero se ejecuta una instrucción del tipo `cmp`, `test` u otras instrucciones aritmético-lógicas para modificar los bits de estado del procesador. Después, una instrucción de salto o ajuste comprueba esos bits de estado para modificar el flujo de ejecución de instrucciones o para ajustar variables booleanas.

Estas son las instrucciones de comparación del ejemplo:

```
cmpq      Gbl+84936(%rip), %rax
cmpl      $5, %edx
cmpl      $9, %edx
testq     %rax, %rax
cmpb      $0, 345(%rdi)
```

La instrucción **`cmp`** realiza la resta de dos números, sin almacenar el resultado, pero afectando a los indicadores de estado.

La instrucción **`test`** realiza la operación *and* bit a bit entre los dos operandos, sin almacenar el resultado, pero afectando a los indicadores de estado.

3. Instrucciones de salto y de ajuste condicionales

Si el flujo de ejecución de las instrucciones debe alterarse saltando a la instrucción en otra dirección de memoria distinta de la siguiente a la actual, debe ejecutarse una instrucción de salto incondicional **`jmp`**, que siempre ejecuta el salto:

Condición	Instrucción de salto incondicional
Saltar siempre	`jmp`

Aparte de la instrucción de salto incondicional, en el ejemplo han aparecido varias instrucciones de salto y de ajuste condicionales usadas tras hacer comparaciones:

```
je   .L556
je   .L552
jb   .L555
ja   .L555
jle  .L556
sete %al
```

Hay más instrucciones de salto (*jump*) y ajuste (*set*) condicionales. Entre ellas podemos incluir las siguientes instrucciones que saltan a otro punto del programa o ajustan una variable booleana en función de una comparación previa entre dos números:

Instrucciones de salto condicional en comparaciones		
Condición	Números sin signo	Números con signo
=	`je`	
≠	`jne`	
<	`jb, jnae`	`jl, jnge`
≥	`jae, jnb`	`jge, jnl`
>	`ja, jnbe`	`jg, jnle`
≤	`jbe, jna`	`jle, jng`

Instrucciones de ajuste condicional en comparaciones		
Condición	Números sin signo	Números con signo
=	`sete`	
≠	`setne`	
<	`setb, setnae`	`setl, setnge`
≥	`setae, setnb`	`setge, setnl`
>	`seta, setnbe`	`setg, setnle`
≤	`setbe, setna`	`setle, setng`

Llegados a este punto, nos planteamos las siguientes preguntas:

- ¿Por qué hay grupos de instrucciones distintas para comparaciones entre números sin signo y con signo?
- ¿Qué comprobaciones “mágicas” tiene que realizar el procesador para saber si se cumple o no la condición en cada caso?

Para responderlas necesitamos estudiar:

- Los bits de estado afectados por instrucciones aritméticas y lógicas.
- Las operaciones de suma y resta para números sin signo y con signo.

4. Bits de estado afectados por instrucciones aritméticas y lógicas

Como en otras familias de procesadores, los x86 disponen de un registro de indicadores que guarda varios bits de estado. Este registro es conocido como FLAGS en 16 bits, EFLAGS en 32 bits y RFLAGS en 64 bits. Las instrucciones de ajuste y salto condicionales examinan varios bits de este registro de indicadores, puestos a 0 o a 1 por instrucciones aritméticas o lógicas previas, entre ellas las de comparación `cmp` y `test`. Estos bits de estado son:

- **CF** (Carry Flag): 1 si el resultado está fuera de rango en operaciones sin signo. En la suma es una copia del acarreo producido en el bit más significativo. En la resta es la negación del acarreo producido en el bit más significativo. Las instrucciones `inc` y `dec` no afectan a CF.
- **PF** (Parity Flag): 1 si el número de unos del byte menos significativo del resultado es par.
- **ZF** (Zero Flag): 1 si el resultado es 0.
- **SF** (Sign flag): 1 si el resultado es negativo en operaciones con signo. Es una copia del bit más significativo del resultado.
- **OF** (Overflow flag): 1 si el resultado está fuera de rango en operaciones con signo.

La instrucción `mov` nunca afecta a los indicadores de estado.

5. Operaciones de suma y resta sin signo y con signo

Las operaciones de suma y resta operan de forma exactamente igual sobre enteros sin signo y con signo. El procesador no puede distinguir números enteros sin signo y con signo. Es el programador el responsable de usar el tipo de datos y las instrucciones máquina correctas.

5.1 Números sin signo

Cuando hablamos de números sin signo (*unsigned* en inglés) queremos decir que interpretaremos cualquier representación binaria de *n* bits como un número positivo.

Para $n = 8$ bits tendremos 2^8 combinaciones binarias en el rango [0, 255]:

```
1111 1111 (255)
   ...
1000 0000 (128)
0111 1111 (127)
   ...
0000 0000 (  0)
```

5.1.1 Desbordamiento en números sin signo

En números sin signo el desbordamiento en la suma o resta se indica con el indicador de acarreo CF, no con el overflow OF, como ponen de manifiesto estos ejemplos:

Ejemplo 1 con $n = 8$ bits:

```
  0110 0000 (2^6 + 2^5 =  64 + 32 =  96)
+ 0110 0000 (2^6 + 2^5 =  64 + 32 =  96)
  ---------
0|1100 0000 (2^7 + 2^6 = 128 + 64 = 192)
```

El bit de acarreo CF = 0 indica en este caso que no hay acarreo, 192 es el resultado correcto.
Los bits de signo y overflow (SF = 1, OF = 1) no tienen relevancia en operaciones sin signo.

Ejemplo 2 con $n = 8$ bits:

```
  1100 0000 (2^7 + 2^6 = 128 + 64 = 192)
+ 1100 0000 (2^7 + 2^6 = 128 + 64 = 192)
  ---------
1|1000 0000 (2^7 = 128)
```

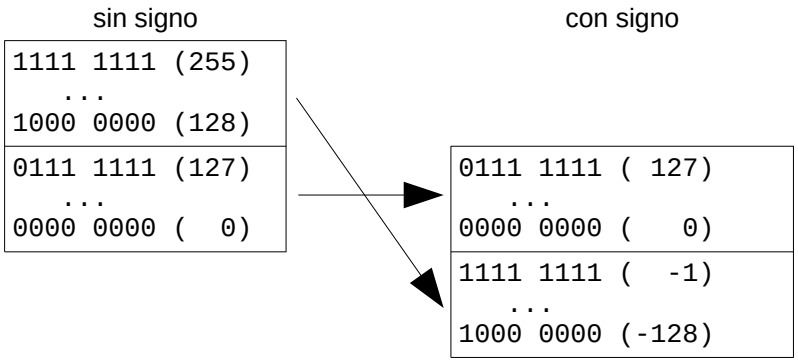
El bit de acarreo CF = 1 indica en este caso que ha habido acarreo, 128 no es el resultado correcto.
Los bits de signo y overflow (SF = 1, OF = 0) no tienen relevancia en operaciones sin signo.

5.2 Números con signo

Cuando hablamos de números con signo (*signed* en inglés) queremos decir que interpretaremos las representaciones binarias como números tanto negativos como positivos. Para los números negativos la mayoría de los computadores actuales utilizan la representación en complemento a 2:

- Si $x \geq 0$: representación normal de x en el rango $[0, 2^{n-1}-1]$. El bit de signo s en la posición $n-1$ es 0.
- Si $x < 0$: representación de x en complemento a 2. El bit de signo s en la posición $n-1$ es 1.

Con 8 bits tendremos 2^8 combinaciones binarias, representando números en el rango [–128, 255]:



5.2.1 Definición de complemento a 2

Como muestra el esquema anterior, el número –1 se representa mediante la combinación binaria `111...111` (255 si lo consideráramos sin signo), el número –2 se representa mediante la combinación binaria `111...110` (254 si lo consideráramos sin signo), etc:

número x		representación f(x)
–1	→	`1111 1111 (256 - 1 = 255)`
–2	→	`1111 1110 (256 - 2 = 254)`
–3	→	`1111 1101 (256 - 3 = 253)`
		...
–126	→	`1000 0010 (256 - 126 = 130)`
–127	→	`1000 0001 (256 - 127 = 129)`
–128	→	`1000 0000 (256 - 128 = 128)`

¿Cuál es la expresión general para esta representación de los números negativos?

En el ejemplo, $2^8 + x = 256 + x$

En general, $2^n + x$

Teniendo en cuenta que $|x| = -x$ para $x < 0$, podemos escribir alternativamente:

$$2^n + x = 2^n - (-x) = 2^n - |x|$$

Llamamos a esta representación de los negativos “representación en complemento a 2” o C_2:

$$x < 0,\ C_2(|x|) = 2^n - |x|$$

Ejemplo: ¿Cómo representar $x = -5$ con $n = 8$ bits?

$C_2(|-5|) = 2^8 - |-5| = 2^8 - 5 = 256 - 5$

```
 1 0000 0000 (256)
-  0000 0101 (  5)
 -----------
   1111 1011 (251 sin signo o -5 con signo)
```

Cuando realizamos operaciones de suma o resta con registros o posiciones de memoria de n bits, realmente usamos aritmética modular, ya que nos quedamos con los n bits menos significativos del

resultado. Así, podemos definir más formalmente la función complemento a dos para cualquier x, positivo o negativo, como:

$$\mathbf{C_2\,(x) = (2^n - x)\ mod\ 2^n}$$

- Si $x \geq 0 \rightarrow C_2\,(x) = (2^n - x) \bmod 2^n = -x$
  - Ejemplo con $n = 8$ bits:
    - $C_2\,(5) = (256 - 5) \bmod 256 = 251 \bmod 256 = 1111\ 1011_2 = -5$
- Si $x < 0 \rightarrow C_2\,(x) = (2^n - x) \bmod 2^n = (2^n - (-|x|)) \bmod 2^n = (2^n + |x|) \bmod 2^n = |x| = -x$
  - Ejemplo con $n = 8$ bits:
    - $C_2\,(-5) = (256 + 5) \bmod 256 = 261 \bmod 256 = 1\ 0000\ 0101_2 \bmod 256 = 0000\ 0101_2 = 5$

Por tanto, en aritmética modular, la función complemento a dos realiza la negación del número, lo cambia de signo.

$C_2\,(x) = -x$. En ensamblador de x86 esto lo hace la instrucción máquina `neg`.

Como siempre trabajaremos en aritmética modular, podemos prescindir de escribir " mod 2^n" constantemente, escribiendo habitualmente $C_2\,(x) = 2^n - x$.

5.2.2 Cómo calcular rápidamente el complemento a dos

Veamos un método más práctico de hacer la resta $2^n - x$. Para ello, sumemos 1 y restemos 1 a la expresión $2^n - x$:

$$C_2\,(x) = 2^n - x = 2^n - x + 1 - 1 = ((2^n - 1) - x) + 1 = (111...111 - x) + 1 = C_1\,(x) + 1$$

El complemento a 1 de un número se puede calcular muy rápidamente en binario cambiando ceros por unos y unos por ceros, ya que para cada uno de los n bits $1 - 0 = 1$, y $1 - 1 = 0$. En ensamblador de x86 el complemento a uno se puede calcular mediante la instrucción máquina `not`.

Por tanto, como $C_2\,(x) = C_1\,(x) + 1$, calcular el complemento a 2 de un número consiste en:

1. Hacer el complemento a 1 (cambiar ceros por unos y unos por ceros).
2. Sumar 1 al resultado.

Ejemplo: ¿Cómo representar $x = -5$ con $n = 8$ bits?

$$-5 = C_2\,(5) = C_1\,(5) + 1$$

`5 = 0000 0101`

cambiamos ceros por unos y unos por ceros

`C₁ (5) = 1111 1010`

sumamos 1

`-5 = C₂ (5) = C₁ (5) + 1 = 1111 1011`

Podemos fijarnos en que sumar 1 consiste en buscar el primer cero comenzando por la derecha cambiando todos los unos por ceros y ese primer cero por un uno. Usando esta idea, calcular el complemento a 2 de un número consiste en:

1. Buscar el primer 1 comenzando por la derecha, copiando todos los dígitos binarios hasta ese primer 1, incluido.
2. Complementar el resto de bits hasta la izquierda (cambiar ceros por unos y unos por ceros).

Ejemplo: ¿Cómo representar $x = -68$ con $n = 8$ bits?

```
68  = 0110 1000
```

copiamos todos los bits comenzando por la derecha hasta encontrar el primer 1

```
= .... 1000
```

cambiamos ceros por unos y unos por ceros en el resto de los bits

```
-68  = 1001 1000
```

5.2.3 ¿Por qué se usa la representación en complemento a 2 para los números negativos?

Desde hace varias décadas, prácticamente cualquier procesador usa el complemento a 2 porque ello permite usar un único circuito aritmético para realizar tanto la suma sin signo como la suma con signo. Veamos por qué.

¿Cómo realizamos la suma $x + y$ para $y < 0$? Si $y < 0$, lo representamos en complemento a dos:

$$C_2(|y|) = (2^n - |y|) \bmod 2^n$$

Para sumar $x + y$, con $y < 0$, realizamos realmente la suma $x + (2^n - |y|) \bmod 2^n$.

Teniendo en cuenta que si $y < 0$, $|y| = -y$:

$$x + (2^n - |y|) \bmod 2^n = x + 2^n - (-y) \bmod 2^n = (x + 2^n + y) \bmod 2^n = x + y$$

La clave está en que sumar 2^n no afecta al resultado al limitar la suma a n bits, es decir, al usar aritmética modular. Así que para calcular $x + (-5)$ lo hacemos sumando $(x + 2^n + (-5)) \bmod 2^n$ obteniendo el resultado correcto.

Ejemplo:

$x = 7$
$y = -5$
$n = 8$ bits

```
  0000 0111 ( 7)
+ 1111 1011 (-5)
  ---------
1|0000 0010 (7 + 2⁸ – 5 = 2⁸ + (7 – 5) = 2⁸ + 2 = 256 + 2 = 258)
  0000 0010 (258 mod 2⁸ = 2)
```

También podemos ver que realizar la resta x – y puede hacerse con el mismo sumador binario, calculando previamente el complemento a dos de y y luego sumando:

$$x - y = x + (-y) = x + C_2(y) = x + (2^n - y) \bmod 2^n = x - y$$

Más abajo mostraremos un posible circuito sumador / restador que permite realizar la suma y la resta tanto de números sin signo como de números con signo.

5.2.4 Desbordamiento en números con signo

En números con signo el desbordamiento en la suma o resta se indica con el indicador de *overflow* OF, no con el acarreo CF, como ponen de manifiesto estos ejemplos:

Ejemplo 1 con n = 8 bits:

```
  0110 0000 (2⁶ + 2⁵ = 64 + 32 = 96)
+ 0110 0000 (2⁶ + 2⁵ = 64 + 32 = 96)
  ---------
0|1100 0000 ( -64); C₂(1100 0000) = 0011 1111 + 1 = 0100 0000 (2⁶ =  64)
```

El bit de acarreo (CF = 0) no tiene relevancia en operaciones con signo.

El bit de signo SF = 1 indica en este caso que el resultado es negativo. El bit de overflow OF = 1 indica que hay desbordamiento, -64 no es el resultado correcto.

Ejemplo 2 con n = 8 bits:

```
  1100 0000 ( -64); C₂(1100 0000) = 0011 1111 + 1 = 0100 0000 (2⁶ =  64)
+ 1100 0000 ( -64); C₂(1100 0000) = 0011 1111 + 1 = 0100 0000 (2⁶ =  64)
  ---------
1|1000 0000 (-128); C₂(1000 0000) = 0111 1111 + 1 = 1000 0000 (2⁷ = 128)
```

El bit de acarreo (CF = 1) no tiene relevancia en operaciones con signo.

El bit de signo SF = 1 indica en este caso que el resultado negativo. El bit de overflow OF = 0 indica que no hay desbordamiento, -128 es el resultado correcto.

5.2.5 ¿Cómo se calcula el overflow OF en la suma?

Sean dos operandos x e y. La suma con signo $x + y$ da como resultado r. Llamemos s_x al bit de signo de x, s_y al bit de signo de y, y s_r al bit de r.

Las 8 combinaciones posibles de esos 3 bits de signo son:

Caso	s_x	s_y	s_r	OF	Acarreo c_n (bit n–1 a n) y c_{n-1} (bit n–2 a n–1)	Comentarios
1	0	0	0	0	$c_n = 0$, $c_{n-1} = 0$	Sumamos dos positivos. El resultado es positivo → no hay desbordamiento.
2	0	0	1	**1**	$\mathbf{c_n = 0, c_{n-1} = 1}$	Sumamos dos positivos. El resultado (incorrecto) es negativo → desbordamiento.
3	0	1	0	0	$c_n = 1$, $c_{n-1} = 1$	Sumamos un número positivo y uno negativo. El resultado siempre es menor que el número positivo, y mayor o igual que el número negativo → no hay desbordamiento.
4	0	1	1	0	$c_n = 0$, $c_{n-1} = 0$	
5	1	0	0	0	$c_n = 1$, $c_{n-1} = 1$	
6	1	0	1	0	$c_n = 0$, $c_{n-1} = 0$	
7	1	1	0	**1**	$\mathbf{c_n = 1, c_{n-1} = 0}$	Sumamos dos negativos y el resultado (incorrecto) es positivo → desbordamiento.
8	1	1	1	0	$c_n = 1$, $c_{n-1} = 1$	Sumamos dos negativos y el resultado es negativo → no hay desbordamiento.

Como vemos, el overflow OF puede calcularse a partir de los bits de signo como $\bar{s}_x \cdot \bar{s}_y \cdot s_r + s_x \cdot s_y \cdot \bar{s}_r$, o a partir de los acarreos como $c_n \wedge c_{n-1}$.

Aclaremos los casos 1 y 2:

```
  0xxx xxxx
+ 0yyy yyyy
  ---------
  ?rrr rrrr
```

El máximo número positivo con signo que podemos representar con n bits es $2^{n-1} - 1$. Para $n = 8$ bits el máximo positivo es $2^7 - 1 = 127$. En los casos 1 y 2 tenemos $x > 0$ e $y > 0$, luego $0 \le x \le 2^{n-1} - 1$, $0 \le y \le 2^{n-1} - 1$. Los límites para la suma $x + y$ son:

$$0 \le x + y \le (2^{n-1} - 1) + (2^{n-1} - 1) = 2 \cdot 2^{n-1} - 2 = 2^n - 2 \text{ (254 para } n = 8 \text{ bits)}$$

El rango $[0, 2^n - 2]$ lo podemos dividir en dos partes:

- $0 \le x + y \le 2^{n-1} - 1$ ([0, 127] para $n = 8$ bits), el bit de signo s_r es 0, no hay desbordamiento (caso 1).
  - s_r es 0 porque el acarreo c_{n-1} entre el bit 6 y el 7 es 0. El acarreo entre el bit 7 y el 8 (c_n o CF) es 0.
- $2^{n-1} \le x + y \le 2^n - 2$ ([128, 254] para $n = 8$ bits), el bit de signo s_r es 1, hay desbordamiento (caso 2).
  - s_r es 1 porque el acarreo c_{n-1} entre el bit 6 y el 7 es 1. El acarreo entre el bit 7 y el 8 (c_n o CF) es 0.

Aclaremos los casos 6 y 7:

```
  1xxx xxxx
+ 1yyy yyyy
  ---------
  ?rrr rrrr
```

El mínimo número negativo que podemos representar con n bits es -2^{n-1}. Para $n = 8$ bits el mínimo negativo es $-2^7 = -128$. En los casos 6 y 7 tenemos $x < 0$ e $y < 0$, luego $-2^{n-1} \le x \le -1$, $-2^{n-1} \le y \le -1$. Los límites para la suma $x + y$ son:

$(-2^{n-1}) + (-2^{n-1}) \le x + y \le (-1) + (-1)$, $2 \cdot (-2^{n-1}) \le x + y \le -2$, $-2^n \le x + y \le -2$. Para 8 bits, $-256 \le x + y \le -2$.

El rango $[-2^n, -2]$ lo podemos dividir en dos partes:

- $-2^n \le x + y \le -2^{n-1} - 1$ ([–256, –129] para $n = 8$ bits), el bit de signo s_r es 0, hay desbordamiento (caso 7).
  - s_r es 0 porque el acarreo c_{n-1} entre el bit 6 y el 7 es 0. El acarreo entre el bit 7 y el 8 (c_n o CF) es 1.

- $-2^{n-1} \leq x + y \leq -2$ ([–128, –2] para n = 8 bits), el bit de signo s_r es 1, no hay desbordamiento (caso 8).
  - s_r es 1 porque el acarreo c_{n-1} entre el bit 6 y el 7 es 1. El acarreo entre el bit 7 y el 8 (c_n o CF) es 1.

5.2.6 ¿Cómo se calcula el overflow OF en la resta?

Podemos transformar la resta $x - y$ en una suma $x + (-y)$. La tabla quedaría así:

$x - y$				$x + (-y)$				OF	Acarreo c_n (bit n–1 a n) y c_{n-1} (bit n–2 a n–1)	Comentarios
Caso	s_x	s_y	s_r	Caso tabla anterior	s_x	s_{-y}	s_r			
1	0	0	0	3	0	1	0	0	$c_n = 1$, $c_{n-1} = 1$	Restamos positivo menos positivo, lo que es igual a sumar positivo y negativo. El resultado siempre es menor que el n.º positivo, y mayor o igual que el n.º negativo → no hay desbordamiento.
2	0	0	1	4	0	1	1	0	$c_n = 0$, $c_{n-1} = 0$	
3	0	1	0	1	0	0	0	0	$c_n = 0$, $c_{n-1} = 0$	Restamos positivo menos negativo, lo que es igual a sumar dos positivos. El resultado es positivo → no hay desbordamiento.
4	0	1	1	2	0	0	1	**1**	**$c_n = 0$, $c_{n-1} = 1$**	Restamos positivo menos negativo, lo que es igual a sumar dos positivos. El resultado (incorrecto) es negativo → desbordamiento.
5	1	0	0	7	1	1	0	**1**	**$c_n = 1$, $c_{n-1} = 0$**	Restamos negativo menos positivo, lo que es igual a sumar dos negativos. El resultado (incorrecto) es positivo → desbordamiento.
6	1	0	1	8	1	1	1	0	$c_n = 1$, $c_{n-1} = 1$	Restamos negativo menos positivo, lo que es igual a sumar dos negativos. El resultado es negativo → no hay desbordamiento.
7	1	1	0	5	1	0	0	0	$c_n = 1$, $c_{n-1} = 1$	Restamos negativo menos negativo, lo que es igual a sumar negativo y positivo. El resultado siempre es menor que el nº positivo, y mayor o igual que el nº negativo → no hay desbordamiento.
8	1	1	1	6	1	0	1	0	$c_n = 0$, $c_{n-1} = 0$	

Ahora el overflow OF puede calcularse a partir de los bits de signo como $\bar{s}_x \cdot s_y \cdot s_r + s_x \cdot \bar{s}_y \cdot \bar{s}_r$, o a partir de los acarreos (igual que en la suma) como $c_n \wedge c_{n-1}$.

En los siguientes enlaces podemos encontrar más información sobre acarreo y overflow:

- Ian! D. Allen, *The CARRY flag and OVERFLOW flag in binary arithmetic.* http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt
- *Overflow Detection for Adders.* University of Maryland. https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Comb/overflow.html
- *Overflow Detection.* CIS-77 Introduction to Computer Systems, Bristol Community College, Computer Information Systems Department

http://www.c-jump.com/CIS77/CPU/Overflow/lecture.html

- *EX1C: Hierarchical design of circuits and standard combinational blocks. Chapter 1: Combinational Circuits. PROJECT 4: An 8-bit adder/subtractor using the 2's complement (2C) conventions.* Digital Circuits and Systems (CSD). Universidad Politécnica de Cataluña (UPC) http://digsys.upc.es/ed/CSD/terms/00_old/1415Q2/EX/EX1C.html#P4

5.3 Circuito para realizar la suma y la resta

El circuito de la Fig. 2 representa una posible implementación de un sumador / restador de *n* bits construido a partir de *n* sumadores de un bit. Aparte de las entradas *x* e *y*, de *n* bits cada una, podemos observar la entrada sub / $\overline{\text{add}}$:

- Cuando sub / $\overline{\text{add}}$ vale 0 el circuito es un sumador que calcula $r = x + y$.
- Cuando sub / $\overline{\text{add}}$ vale 1 el circuito se convierte en un restador, ya que suma a *x* el complemento a 2 de *y* (sumando el complemento a 1 de *y* más un acarreo inicial $c_0 = 1$) para realizar la operación $r = x - y$.

El circuito incluye dos puertas XOR para calcular el acarreo CF y el overflow OF, con el siguiente funcionamiento:

- La puerta XOR superior calcular CF realizando la operación XOR entre c_n y la entrada sub / $\overline{\text{add}}$, ya que:
  - Para la suma (sub / $\overline{\text{add}}$ = 0), el acarreo de salida c_n del circuito es el acarreo (*carry*) correcto de la suma de números sin signo $x + y$, es decir CF = c_n.
  - Para la resta (sub / $\overline{\text{add}}$ = 1), el circuito calcula $x - y$ como la suma $x + C_2(y) = x + C_1(y) + 1$, y c_n no coincide con el débito que se obtendría al restar sin signo $x - y$, sino que es su negación, por tanto tenemos que negar c_n, es decir CF = $\overline{c_n}$.

  La siguiente tabla de verdad resume el cálculo de CF:

	sub/$\overline{\text{add}}$	c_n	CF
add	0	0	0
	0	1	1
sub	1	0	1
	1	1	0

  Por tanto, CF = c_n ^ sub/$\overline{\text{add}}$.

  Nota: Como acabamos de ver, los procesadores de la familia x86 invierten el acarreo de salida c_n en la resta para obtener el bit que se almacena en el registro de indicadores de estado. Existen procesadores, por ejemplo los de la familia ARM, que no lo hacen, de modo que en ellos se almacena directamente el acarreo de salida c_n en el registro de estado incluso en la resta.

- La puerta XOR inferior tiene dos entradas: el acarreo de entrada c_{n-1} y el de salida c_n del sumador más significativo (el correspondiente a los bits n–1). Cuando c_{n-1} es distinto de c_n, se ha producido un desbordamiento para números con signo, representado por el indicador de estado OF. Por tanto, OF = $c_n \wedge c_{n-1}$.

	sub/$\overline{add}$	c_n	CF
add	0	0	0
	0	1	1
sub	1	0	1
	1	1	0

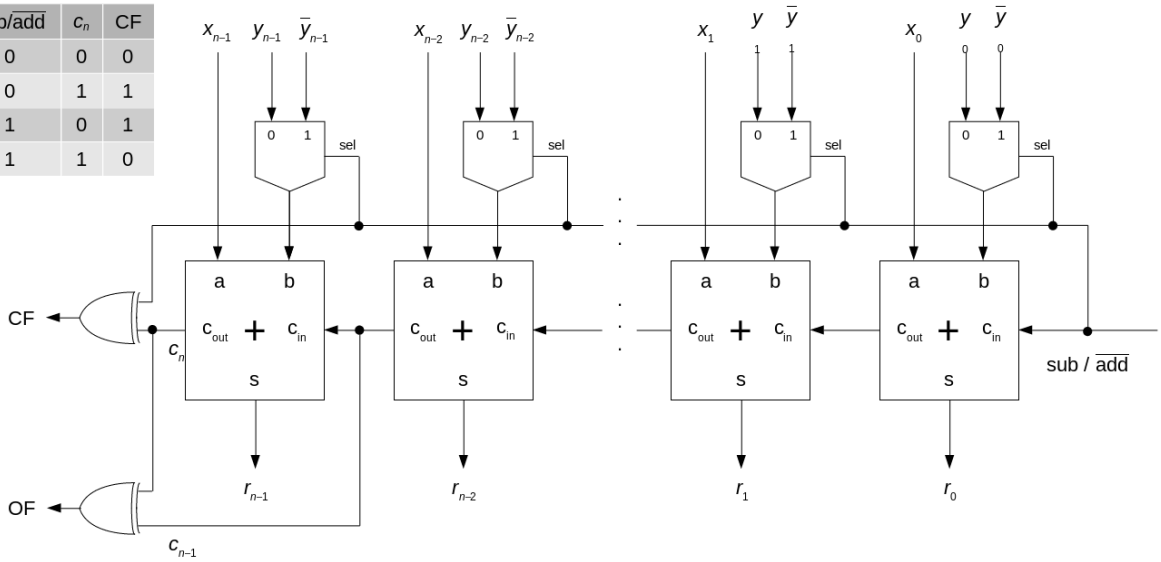


Figura 2. Sumador / restador de n bits con generación de acarreo y overflow, construido a partir de n sumadores de 1 bit. x e y son los dos operandos de n bits y r el resultado de n bits. Si la señal de entrada sub / $\overline{add}$ = 1 el circuito realiza la resta sumando el complemento a dos de y. Si sub / $\overline{add}$ = 0 el circuito realiza la suma. CF es el indicador de acarreo (desbordamiento para números sin signo) y OF el indicador de overflow (desbordamiento para números con signo).

En los siguientes enlaces podemos encontrar más información sobre este circuito aritmético:

- *Design and Implementation of Various Arithmetic Circuits.* Electronics & Communications → Hybrid Electronics Lab → List Of Experiments. Virtual Labs Project. https://he-coep.vlabs.ac.in/Experiment1/Theory.html?domain=ElectronicsandCommunications&lab=Hybrid%20Electronics%20Lab
- Leopoldo Silva Bijit. *Arithmetic Circuits. 5.2 Networks for Binary Addition.* Sistemas Digitales. Departamento de Electrónica. Universidad Técnica Federico Santa María. http://www2.elo.utfsm.cl/~lsb/elo211/aplicaciones/katz/chapter5/chapter05.doc2.html

6. Volviendo a los saltos y ajustes

Ahora que sabemos que en la suma y resta de números sin signo la información de desbordamiento la proporciona el bit de acarreo y para números con signo la proporciona el bit de overflow, supongamos que ejecutamos una secuencia de instrucciones del tipo:

```
cmp B, A
jcond etiqueta   o bien   setcond variable
```

Vamos a deducir qué bits ha de comprobar el procesador para cada una de las instrucciones de salto o de ajuste condicionales. En la siguiente tabla se indican las condiciones que han de comprobarse:

<table>
<tr><td rowspan="2">C
O
N
D</td><td colspan="3">Números sin signo</td><td colspan="4">Números con signo</td></tr>
<tr><td>Instrucción</td><td colspan="2">Condición a comprobar</td><td>Instrucción</td><td colspan="3">Condición a comprobar</td></tr>
<tr><td>=</td><td>je
sete</td><td colspan="2">A=B ↔ A–B=0 ↔ ZF=1</td><td>je
sete</td><td colspan="3">A=B ↔ A–B=0 ↔ ZF=1</td></tr>
<tr><td>≠</td><td>jne
setne</td><td colspan="2">A≠B ↔ A–B≠0 ↔ ZF=0</td><td>jne
setne</td><td colspan="3">A≠B ↔ A–B≠0 ↔ ZF=0</td></tr>
<tr><td><</td><td>jb=jnae
setb=setnae</td><td colspan="2">A<B ↔ A–B → acarreo ↔ CF=1</td><td>jl=jnge
setl=setnge</td><td>A<B ↔</td><td>SF=1 y OF=0
SF=0 y OF=1 (– – + = +)</td><td>↔ SF≠OF</td></tr>
<tr><td>≥</td><td>jae=jnb
setae=setnb</td><td colspan="2">A≥B ↔ A–B → no acarreo ↔ CF=0</td><td>jge=jnl
setge=setnl</td><td>A≥B ↔</td><td>SF=0 y OF=0
SF=1 y OF=1 (+ – – = –)</td><td>↔ SF=OF</td></tr>
<tr><td rowspan="3">></td><td rowspan="3">ja=jnbe
seta=setnbe</td><td rowspan="3">A>B ↔</td><td>A≥B ↔ A–B → no acarreo ↔ CF=0</td><td rowspan="3">jg=jnle
setg=setnle</td><td rowspan="3">A>B ↔</td><td>A≥B ↔</td><td>SF=0 y OF=0
SF=1 y OF=1 (+ – – = –)</td><td>↔ SF=OF</td></tr>
<tr><td>y</td><td colspan="3">y</td></tr>
<tr><td>A≠B ↔ A–B≠0 ↔ ZF=0</td><td colspan="3">A≠B ↔ A–B≠0 ↔ ZF=0</td></tr>
<tr><td rowspan="3">≤</td><td rowspan="3">jbe=jna
setbe=setna</td><td rowspan="3">A≤B ↔</td><td>A<B ↔ A–B → acarreo ↔ CF=1</td><td rowspan="3">jle=jng
setle=setng</td><td rowspan="3">A≤B ↔</td><td>A<B ↔</td><td>SF=1 o OF=0
SF=0 o OF=1 (– – + = +)</td><td>↔ SF≠OF</td></tr>
<tr><td>o</td><td colspan="3">o</td></tr>
<tr><td>A=B ↔ A–B=0 ↔ ZF=1</td><td colspan="3">A=B ↔ A–B=0 ↔ ZF=1</td></tr>
</table>

Más abreviadamente:

COND	Números sin signo		Números con signo	
	Instrucción	Condición a comprobar	Instrucción	Condición a comprobar
=	je=jz sete=setz	**ZF**	je=jz sete=setz	**ZF**
≠	jne=jnz setne=setnz	**~ZF**	jne=jnz setne=setnz	**~ZF**
<	jb=jnae=jc setb=setnae=setc	**CF**	jl=jnge setl=setnge	**SF^OF**
≥	jae=jnb=jnc setae=setnb=setnc	**~CF**	jge=jnl setge=setnl	**~(SF^OF)**
>	ja=jnbe seta=setnbe	**~CF · ~ZF** **~(CF | ZF)**	jg=jnle setg=setnle	**~(SF^OF) · ~ZF** **~((SF^OF) | ZF)**
≤	jbe=jna setbe=setna	**CF | ZF**	jle=jng setle=setng	**(SF^OF) | ZF**

6.1 Otras instrucciones de salto y ajuste

Aparte de las instrucciones de salto y ajuste condicionales útiles en comparaciones que acabamos de estudiar, existen otras instrucciones de salto y ajuste que examinan un determinado bit o el contenido de un registro. Son las siguientes:

Condición	Instrucción	Descripción	Indicadores
Overflow	`jo` `seto`	jump/set if overflow	**OF**
No overflow	`jno` `setno`	jump/set if not overflow	**~OF**
< 0	`js` `sets`	jump/set if sign	**SF**
≥ 0	`jns` `setns`	jump/set if not sign	**~SF**
Paridad par	`jp=jpe` `setp=setpe`	jump/set if parity / jump/set if parity even	**PF** (8 bits menos signif. del resultado contienen un nº par de unos)
Paridad impar	`jnp=jpo` `setnp=setpo`	jump/set if not parity / jump/set if parity odd	**~PF** (8 bits menos signif. del resultado contienen un nº impar de unos)
Registro *CX = 0	`jcxz,jecxz,jrcxz`	jump if `cx`/`ecx`/`rcx` is zero	**~CX**, **~ECX**, **~RCX**

7. Instrucciones de copia (mov) condicional

Con el Pentium Pro (1995) se introdujeron las instrucciones `cmovcc` (*conditional move*), que realizan una copia sólo cuando se cumple una determinada condición, permitiendo asignar o devolver uno de entre dos valores según esa condición. En la siguiente tabla se muestran todas las instrucciones `cmovcc` para realizar la copia condicional tras una comparación entre dos valores:

COND	Números sin signo		Números con signo	
	Instrucción	Condición a comprobar	Instrucción	Condición a comprobar
=	`cmove=cmovz`	**ZF**	`cmove=cmovz`	**ZF**
≠	`cmovne=cmovnz`	**~ZF**	`cmovne=cmovnz`	**~ZF**
<	`cmovb=cmovnae=cmovc`	**CF**	`cmovl=cmovnge`	**SF^OF**
≥	`cmovae=cmovnb=cmovnc`	**~CF**	`cmovge=cmovnl`	**~(SF^OF)**
>	`cmova=cmovnbe`	**~CF · ~ZF**	`cmovg=cmovnle`	**~(SF^OF) · ~ZF**

		~(CF | ZF)		**~((SF^OF) | ZF)**
≤	`cmovbe=cmovna`	**CF | ZF**	`cmovle=cmovng`	**(SF^OF) | ZF**

Igual que sucede con las instrucciones de ajuste y de salto, existen otras instrucciones de copia condicional no relacionadas directamente con comparaciones:

Condición	Instrucción	Indicadores
Overflow	`cmovo`	**OF**
No overflow	`cmovno`	**~OF**
< 0	`cmovs`	**SF**
≥ 0	`cmovns`	**~SF**
Paridad par	`cmovp=cmovpe`	**PF**
Paridad impar	`cmovnp=cmovpo`	**~PF**

7.1 Un ejemplo real

Para entender el funcionamiento de las instrucciones de movimiento condicional volvemos al ejemplo de la plataforma SWAD. En este caso vamos a centrarnos en el siguiente fragmento de código de la función `Enr_PutActionsRegRemOneUsr`, dentro del módulo `swad_enrolment`, que muestra la lista de acciones de la parte inferior de la Fig. 1.

7.1.1 Código fuente en C

El fragmento de código C que vamos a examinar es el siguiente:

```
bool Enr_PutActionsRegRemOneUsr (bool ItsMe)
  {
   ...

   /***** Register user in course / Modify user's data *****/
   if (Gbl.CurrentCrs.Crs.CrsCod > 0 &&
       Gbl.Usrs.Me.Role.Logged >= Rol_STD)
     {
      sprintf (Gbl.Alert.Txt,UsrBelongsToCrs ? (ItsMe ? Txt_Modify_me_in_the_course_X :
                                                        Txt_Modify_user_in_the_course_X) :
                                               (ItsMe ? Txt_Register_me_in_X :
                                                        Txt_Register_USER_in_the_course_X),
               Gbl.CurrentCrs.Crs.ShrtName);
      ...
```

7.1.2 Código en ensamblador

Compilamos `swad_enrolment.c` mediante la orden:

```
gcc -O2 -S swad_enrolment.c -o swad_enrolment.s
```

para obtener un archivo en ensamblador de x86-64 `swad_enrolment.s`. Este es el fragmento correspondiente al código anterior:

```
       ...

       cmpq    $0, Gbl+142616(%rip)
```

```
        jle     .L596
        cmpl    $2, Gbl+97440(%rip)
        jbe     .L596
        testb   %r13b, %r13b
        jne     .L615
        testb   %r12b, %r12b
        movq    Txt_Register_me_in_X(%rip), %rcx
        cmove   Txt_Register_USER_in_the_course_X(%rip), %rcx
.L551:
        movl    $Gbl+142904, %r8d
        movl    $16384, %edx
        movl    $1, %esi
        movl    $Gbl+1988, %edi
        xorl    %eax, %eax
        ...
        call    __sprintf_chk

        ...

.L615:
        testb   %r12b, %r12b
        movq    Txt_Modify_me_in_the_course_X(%rip), %rcx
        cmove   Txt_Modify_user_in_the_course_X(%rip), %rcx
        jmp     .L551

        ...
```

7.1.3 Análisis de la traducción de C a ensamblador

Vamos a estudiar detalladamente la correspondencia entre el código C y el código ensamblador:

1

C	`if (Gbl.CurrentCrs.Crs.CrsCod > 0 && Gbl.Usrs.Me.Role.Logged >= Rol_STD)`

Ensamblador	`cmpq $0, Gbl+142616(%rip)`
	`jle .L596`
	`cmpl $2, Gbl+97440(%rip)`
	`jbe .L596`

- El código de la asignatura actual se encuentra almacenado en el campo `Gbl.CurrentCrs.Crs.CrsCod`, de tipo `long` (entero con signo de 64 bits). La instrucción `cmpq $0,Gbl+142616(%rip)` compara este código con 0.
- La instrucción `jle .L596` salta fuera del `if` si el código `CrsCod` (con signo) es menor o igual que 0. No es necesario comprobar la segunda parte de la condición ya que si no se cumple la primera parte, el *and* de las dos tampoco.
- El rol del usuario identificado está almacenado en `Gbl.Usrs.Me.Role.Logged`, de tipo enumerado (entero de 32 bits sin signo). La instrucción `cmpl $2,Gbl+97440(%rip)` compara `Gbl.Usrs.Me.Role.Logged` con 2. El valor enumerado `Rol_STD` (estudiante) es 3.
- La instrucción `jbe .L596` salta fuera del `if` si `Logged` (valor sin signo) es menor o igual que 2, es decir, si es menor que 3.
- El código continúa por la siguiente instrucción cuando se cumplen simultáneamente las dos condiciones contrarias a las que provocan el salto, es decir, cuando el código de la asignatura es mayor que 0 y el rol del usuario es mayor o igual que `Rol_STD`.

2

C	`UsrBelongsToCrs ?`
Ensamblador	`testb %r13b, %r13b` `jne .L615`

- La variable local `UsrBelongsToCrs` indica si el usuario pertenece a la asignatura actual. Es de tipo `bool`, con un tamaño de 1 byte, y está almacenada en el registro de 8 bits `%r13b`. La instrucción `testb %r13b,%r13b` realiza la operación *and* bit a bit del registro consigo mismo, para activar los indicadores de estado.
- `jne .L615` (saltar si igual / si cero) salta a la instrucción que comienza en la etiqueta `.L615` si el contenido de `%r13b` es distinto de 0, es decir, si `UsrBelongsToCrs` es verdadero.

3

C	`(ItsMe ? Txt_Modify_me_in_the_course_X :` `Txt_Modify_user_in_the_course_X)`

Ensamblador	`.L615:`
	`testb %r12b, %r12b`
	`movq Txt_Modify_me_in_the_course_X(%rip), %rcx`
	`cmove Txt_Modify_user_in_the_course_X(%rip), %rcx`
	`jmp .L551`

Este grupo de instrucciones se ejecuta cuando el usuario mostrado pertenece a la asignatura actual.

- La variable local `ItsMe` ("Soy yo") indica si el usuario mostrado coincide con el usuario identificado. Es de tipo `bool`, con un tamaño de 1 byte, y está almacenada en el registro de 8 bits `%r12b`. La instrucción `testb %r12b,%r12b` realiza la operación *and* bit a bit del registro consigo mismo, para activar los indicadores de estado.
- La instrucción `movq Txt_Modify_me_in_the_course_X(%rip),%rcx` copia la dirección de la cadena `Txt_Modify_me_in_the_course_X` en el registro `%rcx`, como preparación para llamar a `sprintf`.
- La instrucción `cmove Txt_Modify_user_in_the_course_X(%rip),%rcx` copia condicionalmente si "igual / cero" la dirección de la cadena `Txt_Modify_user_in_the_course_X` en el registro `%rcx`, como preparación para llamar a `fprintf`, sólo si el contenido del registro `%r12b` es 0, es decir, si `ItsMe` es falso. De este modo, en `%rcx` queda un puntero a un mensaje u otro en función de si el usuario mostrado es el mismo que el identificado o no.
- `jmp .L551` vuelve al código común tras las dos comparaciones, que llama a `sprintf`.

4

C	`(ItsMe ? Txt_Register_me_in_X :` `Txt_Register_USER_in_the_course_X)`

Ensamblador	`testb %r12b, %r12b`
	`movq Txt_Register_me_in_X(%rip), %rcx`
	`cmove Txt_Register_USER_in_the_course_X(%rip), %rcx`

Este grupo de instrucciones es similar al anterior, para el caso en que el usuario mostrado no pertenece a la asignatura actual.

5

C	`sprintf (Gbl.Alert.Txt,...,` `Gbl.CurrentCrs.Crs.ShrtName);`

Ensamblador	`.L551:`
	`movl $Gbl+142904, %r8d`
	`movl $16384, %edx`
	`movl $1, %esi`
	`movl $Gbl+1988, %edi`
	`xorl %eax, %eax`
	`...`
	`call __sprintf_chk`

Este fragmento de código llama a la función `__sprintf_chk:`

```
int __sprintf_chk(char *str,int flag,size_t strlen,const char
*format,...);
```

Los parámetros han de pasarse a través de registros:

Parámetro formal	Parámetro en este código	Registro
`char *str`	`Gbl.Alert.Txt ($Gbl+1988)`	`rdi`
`int flag`	`1` (Nivel de medidas de seguridad en la forma de verificar la pila, los valores de los parámetros, etc.)	`esi`
`size_t strlen`	`16384`	`rdx`
`const char *format`	Cadena con formato apuntada por `rcx` en instrucciones anteriores	`rcx`
5º parámetro	`Gbl.CurrentCrs.Crs.ShrtName ($Gbl+142904)`	`r8d`

La instrucción `xor %eax,%eax` pone a 0 `rax` de acuerdo con la especificación ABI x86_64 System V, que indica que en las funciones con un número de parámetros variable el registro `al` debe contener el número de registros vectoriales usados.

8. Otras instrucciones condicionales

Aparte de las instrucciones que hemos visto, existen otras instrucciones relacionadas con las sentencias que cambian el flujo de ejecución de un programa: las instrucciones de bucle y las de comprobación de bits individuales.

8.1 Instrucciones de bucle

La instrucción **`loop`**, con una sintaxis similar a `jmp`, permite implementar bucles usando el registro `cx`, `ecx` o `rcx` como contador. El funcionamiento de la instrucción es el siguiente:

rcx = rcx – 1
if rcx ≠ 0, saltar a etiqueta destino

La estructura de control sería la siguiente:

```
          mov n, %rcx
bucle:    ...cuerpo del bucle...
          loop bucle
```

Las instrucciones `loope=loopz` y `loopne=loopnz` saltan en función del valor de rcx y del bit de estado ZF. En la siguiente tabla se resume el funcionamiento de las tres instrucciones:

Instrucción	Funcionamiento
`loop`	Decrementa el contador. Salta si el contador es ≠ 0.
`loope=loopz`	Decrementa el contador. Salta si el contador es ≠ 0 y ZF = 1.
`loopne=loopnz`	Decrementa el contador. Salta si el contador es ≠ 0 y ZF = 0.

Estas instrucciones son lentas y los compiladores actuales optan usar instrucciones más simples.

8.2 Instrucciones de comprobación de bits individuales

La instrucción **`bt`** (*bit test*) copia el bit *n* del operando en el indicador de acarreo CF. Puede usarse junto con las instrucciones de salto y ajuste en función de CF (`jc`, `jnc`, `setc`, `setnc`). La sintaxis de la instrucción es

`bt` *n*`,`*x*

donde *n* es un registro o una constante que contiene el número de bit a comprobar y *x* es el registro o la posición de memoria a comprobar. Por ejemplo `bt $4,%eax` comprueba el bit 4 del registro `eax`.

Además de la instrucción `bt` existen otras instrucciones que además de leer el valor de un bit, lo modifican:

Instrucción	Funcionamiento
`btc`	Comprueba un bit y lo complementa.
`btr`	Comprueba un bit y lo pone a cero (*reset*).
`bts`	Comprueba un bit y lo pone a uno (*set*).

9. Traducción de sentencias condicionales

En los dos ejemplos de código real anteriores hemos visto la traducción a ensamblador de varias sentencias condicionales llevada a cabo por el compilador. En esta sección vamos a resumir la implementación genérica de todas las sentencias condicionales del lenguaje C. Para cada sentencia mostraremos el código general en C y su traducción típica a ensamblador y pondremos un ejemplo real compilado con la versión 5.4.0 de `gcc` y el nivel de optimización `-O1`.

9.1 Sentencia if then

Las sentencias `if` que no incluyen `else` se suelen implementar usando la siguiente estructura:

Sentencia en C	Estructura en ensamblador

`if (`*`cond`*`)`
`    `*`bloque-then`
	`        `*`comprobar-cond`
	`        `*`jno-cond`* `endif`
	`        `*`bloque-then`
	`endif:`

Ejemplo:

Código C	Código ensamblador

`static void if_then (int x,int y)`
`{`
`    if (x < y)`
`        puts ("then");`
 	
`    puts ("the");`
`    puts ("end");`
`}`
	`        .rodata`
	`.LC0:   .string "then"`
	`.LC1:   .string "the"`
	`.LC2:   .string "end"`
	`        .text`
	`if_then:`
	`        subq    $8, %rsp`
	
	`        cmpl    %esi, %edi      # x : y ?`
	`        jge     .L2             # x >= y`
	
	`        movl    $.LC0, %edi     # then`
	`        call    puts`
	`.L2:`
	`        movl    $.LC1, %edi     # the`
	`        call    puts`
	`        movl    $.LC2, %edi     # end`
	`        call    puts`
	
	`        addq    $8, %rsp`
	`        ret`

9.2 Sentencia if then else

Las sentencias `if` que incluyen `else` se suelen implementar usando la siguiente estructura:

Sentencia en C	Estructura en ensamblador

`if (`*`cond`*`)`
`    `*`bloque-then`
`else`
`    `*`bloque-else`
	`        `*`comprobar-cond`
	`        `*`jno-cond`* `else`
	`        `*`bloque-then`
	`        jmp endif`
	`else:`
	`        `*`bloque-else`
	`endif:`

Ejemplo:

Código C	Código ensamblador

`static void if_then_else (int x,int y)`
`{`
`    if (x < y)`
`        puts ("then");`
`    else`
	`        .rodata`
	`.LC0:   .string "then"`
	`.LC1:   .string "else"`
	`.LC2:   .string "the"`
	`.LC3:   .string "end"`

```
        puts ("else");

    puts ("the");
    puts ("end");
}
```

```
        .text
if_then_else:
        subq    $8, %rsp

        cmpl    %esi, %edi      # x : y ?
        jge     .L2             # x >= y

        movl    $.LC0, %edi     # then
        call    puts
        jmp     .L3
.L2:
        movl    $.LC1, %edi     # else
        call    puts
.L3:
        movl    $.LC2, %edi     # the
        call    puts
        movl    $.LC3, %edi     # end
        call    puts

        addq    $8, %rsp
        ret
```

9.3 Condiciones compuestas con AND

Cuando se evalúa una expresión condicional compuesta con AND el compilador usa un atajo: si la primera parte de la expresión es falsa, podemos ahorrarnos la comprobación de la segunda expresión saltando directamente al bloque *else*.

Sentencia en C	Estructura en ensamblador

`if (cond1 && cond2)`
`    bloque-then`
`else`
`    bloque-else`
	`    comprobar-cond1`
	`    jno-cond1 else`
	`    comprobar-cond2`
	`    jno-cond2 else`
	`    bloque-then`
	`    jmp endif`
	`else:`
	`    bloque-else`
	`endif:`

Ejemplo:

Código C	Código ensamblador

```
static void if_and_then_else (int x,int y,int z)
{
    if (x < y && y < z)
        puts ("then");
    else
        puts ("else");

    puts ("the");
    puts ("end");
}
```

```
        .rodata
.LC0:   .string "then"
.LC1:   .string "else"
.LC2:   .string "the"
.LC3:   .string "end"
        .text
if_and_then_else:
        subq    $8, %rsp

        cmpl    %esi, %edi      # x : y ?
        jge     .L2             # x >= y
        cmpl    %edx, %esi      # y : z ?
        jge     .L2             # y >= z

        movl    $.LC0, %edi     # then
        call    puts
        jmp     .L3
.L2:
        movl    $.LC1, %edi     # else
```

	`call puts`
	`.L3:`
	`movl $.LC2, %edi # the`
	`call puts`
	`movl $.LC3, %edi # end`
	`call puts`
	
	`addq $8, %rsp`
	`ret`

9.4 Condiciones compuestas con OR

Cuando se evalúa una expresión condicional compuesta con OR el compilador usa un atajo: si la primera parte de la expresión es verdadera, podemos ahorrarnos la comprobación de la segunda expresión saltando directamente al bloque *then*.

Sentencia en C	Estructura en ensamblador

`if (cond1 || cond2)`
`  bloque-then`
`else`
`  bloque-else`
	`  comprobar-cond1`
	`  jcond1 then`
	`  comprobar-cond2`
	`  jno-cond2 else`
	`then:`
	`  bloque-then`
	`  jmp endif`
	`else:`
	`  bloque-else`
	`endif:`

Ejemplo:

Código C	Código ensamblador

`static void if_or_then_else (int x,int y,int z)`
`{`
`  if (x < y || y < z)`
`  puts ("then");`
`  else`
`  puts ("else");`
 	
`  puts ("the");`
`  puts ("end");`
`}`
	`  .rodata`
	`.LC0: .string "then"`
	`.LC1: .string "else"`
	`.LC2: .string "the"`
	`.LC3: .string "end"`
	`  .text`
	`if_or_then_else:`
	`  subq $8, %rsp`
	
	`  cmpl %esi, %edi  # x : y ?`
	`  jl .L5  # x < y`
	`  cmpl %edx, %esi  # y : z ?`
	`  jge .L2  # y >= z`
	`.L5:`
	`  movl $.LC0, %edi  # then`
	`  call puts`
	`  jmp .L4`
	`.L2:`
	`  movl $.LC1, %edi  # else`
	`  call puts`
	`.L4:`
	`  movl $.LC2, %edi  # the`
	`  call puts`
	`  movl $.LC3, %edi  # end`
	`  call puts`
	
	`  addq $8, %rsp`
	`  ret`

9.5 Operador condicional

El operador condicional `?` se suele usar para elegir uno de entre dos valores dependiendo de una condición. Se suele implementar usando la siguiente estructura, que aprovecha las instrucciones de copia condicional `cmovcc`:

Sentencia en C	Estructura en ensamblador
cond `?` *expresión-then* `:` *expresión-else*	`t =` *expresión-then* `result =` *expresión-else* *if (cond) result = t*

Ejemplo:

Código C	Código ensamblador

`int conditional_operator (int x,int y)`
`{`
`  return (x < y) ? y - x : x - y;`
`}`
	`  .globl conditional_operator`
	`conditional_operator:`
	`  movl  %esi, %edx  # y`
	`  subl  %edi, %edx  # t = y – x`
	
	`  movl  %edi, %eax  # x`
	`  subl  %esi, %eax  # result = x – y`
	
	`  cmpl  %esi, %edi  # x : y ?`
	`  cmovl  %edx, %eax  # if (x < y) result = t`
	
	`  ret`

El hecho de que los valores de ambas ramas (si la condición se cumple y si no se cumple) tengan que calcularse antes de realizar la copia condicional impide que esta estructura de control se use para traducir cualquier expresión que use el operador condicional. En los siguientes casos el compilador optará por traducir el código fuente a una estructura de control clásica del tipo `if … else`: en la que sólo se calcula el valor de la rama correcta:

- **Cálculos costosos:** `val = Test(x) ? Hard1(x) : Hard2(x);`
  - Sólo tiene sentido emplear la copia condicional cuando los cálculos de ambas ramas son sencillos. En este ejemplo puede resultar innecesariamente costoso llamar a las dos funciones.
- **Cálculos arriesgados:** `val = p ? *p : 0;`
  - Puede tener efectos no deseables. En el ejemplo no se debería acceder a la posición apuntada por el puntero `p` cuando su valor es `NULL` (0).
- **Cálculos con efectos colaterales:** `val = x > 0 ? x*=7 : x+=3;`
  - En el ejemplo la variable `x` podría terminar con un valor incorrecto si se calcularan ambas ramas.

9.6 Sentencia switch

La sentencia `switch` puede implementarse con un árbol de comparaciones en cascada o con una tabla de saltos. El compilador decide cuál es la implementación óptima en cada situación.

9.6.1 Implementación en árbol

El siguiente caso es un ejemplo de implementación con un árbol de comparaciones:

Código C

```
static void switch_tree (int x)
{
        switch (x)
        {
        case 100:
                puts ("100");
                break;
        case 200:
                puts ("200");
                break;
        case 300:
                puts ("300");
                break;
        case 400:
                puts ("400");
                break;
        case 500:
                puts ("500");
                break;
        case 600:
                puts ("600");
                break;
        case 700:
                puts ("700");
                break;
        default:
                puts ("default");
                break;
        }
}
```

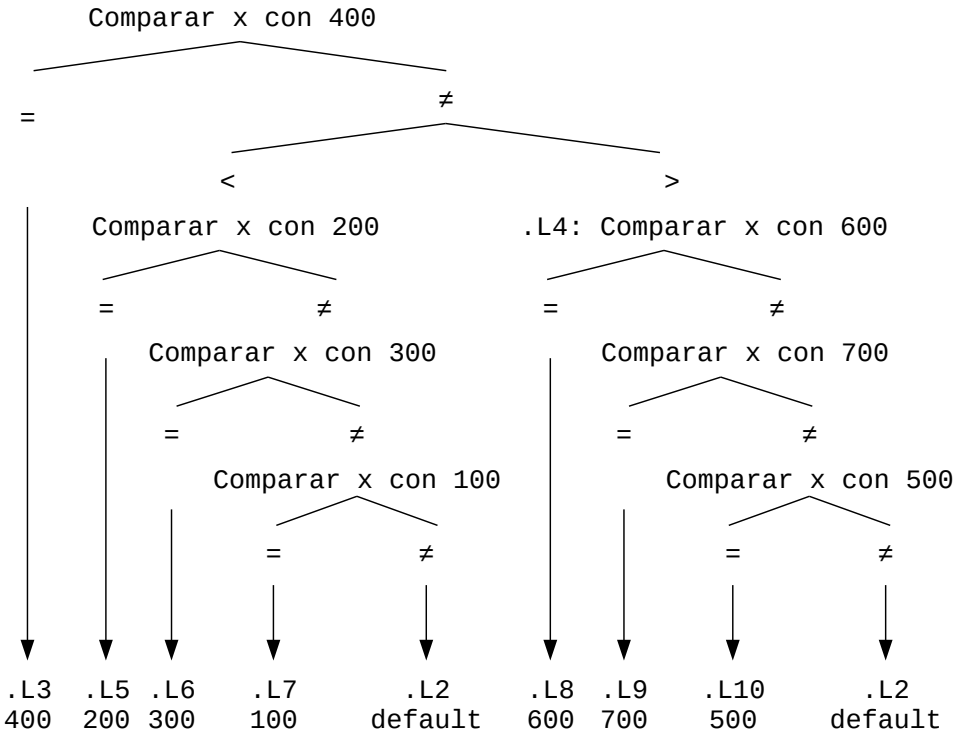
Código ensamblador

```
        .rodata
.LC0:   .string "100"
.LC1:   .string "200"
.LC2:   .string "300"
.LC3:   .string "400"
.LC4:   .string "500"
.LC5:   .string "600"
.LC6:   .string "700"
.LC7:   .string "default"
        .text
switch_tree:
        subq    $8, %rsp

        cmpl    $400, %edi
        je      .L3
        cmpl    $400, %edi      # Eliminada con -O2
        jg      .L4
        cmpl    $200, %edi
        je      .L5
        cmpl    $300, %edi
        je      .L6
        cmpl    $100, %edi
        jne     .L2
        jmp     .L7
.L4:
        cmpl    $600, %edi
        je      .L8
        cmpl    $700, %edi
        je      .L9
        cmpl    $500, %edi
        jne     .L2
        jmp     .L10
.L7:
        movl    $.LC0, %edi     # 100
        call    puts
        jmp     .L1
.L5:
        movl    $.LC1, %edi     # 200
        call    puts
        jmp     .L1
.L6:
        movl    $.LC2, %edi     # 300
        call    puts
        jmp     .L1
.L3:
        movl    $.LC3, %edi     # 400
        call    puts
        jmp     .L1
.L10:
        movl    $.LC4, %edi     # 500
        call    puts
        jmp     .L1
.L8:
        movl    $.LC5, %edi     # 600
        call    puts
        jmp     .L1
```

	.L9:
	movl $.LC6, %edi # 700
	call puts
	jmp .L1
	.L2:
	movl $.LC7, %edi # 800
	call puts
	.L1:
	addq $8, %rsp
	ret

El uso de una estructura de control en forma de árbol frente a una cadena `if...else if...else if...` permite minimizar el número máximo de saltos. A continuación se muestra el árbol de comparaciones para el ejemplo anterior:



9.6.2 Implementación con una tabla de saltos

Cuando el número de casos es suficientemente grande y los valores de los casos no dejan muchos huecos entre ellos, el compilador opta por utilizar una tabla de saltos, mucho más eficiente en términos de tiempo promedio de ejecución que el árbol de comparaciones. La tabla de saltos aprovecha el modo de direccionamiento indirecto de la instrucción `jmp` que permite saltar a la dirección almacenada en una posición de memoria en lugar de saltar directamente a una etiqueta. Antes de llegar a esa instrucción de salto, el código debe descartar los casos fuera de la tabla, que corresponden al caso `default`. El siguiente ejemplo muestra una implementación con una tabla de saltos:

Código C	Código ensamblador
`static void switch_table (int x)`	`.rodata`

```
{
        switch (x)
        {
        case 0:
                puts ("0");
                /* no break */
        case 1:
                puts ("1");
                break;
        case 2:
        case 3:
                puts ("2, 3");
                break;
        case 4:
                puts ("4");
                break;
        default:
                puts ("x<0 || x>4");
                break;
        }
}
```

```
.LC0:   .string "0"
.LC1:   .string "1"
.LC2:   .string "2, 3"
.LC3:   .string "4"
.LC4:   .string "x<0 || x>4"

        .text
switch_table:
        subq    $8, %rsp

        cmpl    $4, %edi        # x : 4 ?
        ja      .L2             # x > 4 || x < 0
        movl    %edi, %edi      # rdi = 0:edi
        jmp     *.L4(,%rdi,8)

        .rodata
.L4:
        .quad   .L3
        .quad   .L5
        .quad   .L6
        .quad   .L6
        .quad   .L7

        .text
.L3:                            # 0
        movl    $.LC0, %edi
        call    puts
.L5:                            # 1
        movl    $.LC1, %edi
        call    puts
        jmp     .L1
.L6:                            # 2, 3
        movl    $.LC2, %edi
        call    puts
        jmp     .L1
.L7:                            # 4
        movl    $.LC3, %edi
        call    puts
        jmp     .L1
.L2:                            # x > 4 || x < 0
        movl    $.LC4, %edi
        call    puts
.L1:
        addq    $8, %rsp
        ret
```

En este caso el compilador usa un truco en la comparación inicial: emplea la instrucción `ja` (saltar si mayor para comparaciones de números sin signo) aunque la variable `x` sea `signed int`, permitiendo comprobar los casos $x > 4$ y $x < 0$ con una única instrucción. Observe que cuando $x < 0$, la instrucción `ja` lo considera un número sin signo con un valor grande, mucho mayor que 4.

10. Traducción de bucles

En esta sección veremos cómo traduce el compilador las tres estructuras del lenguaje C para hacer bucles, comenzando por la más sencilla, el bucle `do while`, continuando por el bucle `while` y terminando con el bucle `for`. Vamos a realizar conversiones de los bucles hasta llegar a un código en C basado en la sentencia `goto`, de traducción inmediata a ensamblador.

10.1 Bucle do while

Los bucles `do...while` pueden traducirse una versión usando etiquetas y la sentencia `goto`, de acuerdo a la siguiente estructura:

Código C	Versión usando goto

`do`
`  Body`
`while (Test);`
	`loop:`
	`  Body`
	`  if (Test)`
	`  goto loop;`

La versión con `goto` se puede traducir fácilmente a ensamblador. Podemos verlo en el siguiente ejemplo, que calcula el número de bits a 1 en un entero:

Código C	Versión usando goto

`int pcount_do (unsigned x)`
`{`
`  int result = 0;`
 	
`  do`
`  {`
`  result += x & 1;`
`  x >>= 1;`
`  } while (x);`
 	
`  return result;`
`}`
	`int pcount_do (unsigned x)`
	`{`
	`  int result = 0;`
	
	`loop:`
	`  result += x & 1;`
	`  x >>= 1;`
	`  if (x)`
	`  goto loop;`
	
	`  return result;`
	`}`

Código ensamblador

```
pcount_do:
        movl    $0, %eax        # result = 0
.L2:
        movl    %edi, %edx      # x
        andl    $1, %edx        # x & 1
        addl    %edx, %eax      # result += x & 1
        shrl    %edi            # x >>= 1
        jne     .L2             # if (x != 0) goto .L2
        ret
```

Observe que el parámetro `x` se ha declarado como `unsigned` y no como `int`. Esto es necesario para que el compilador traduzca el operador de desplazamiento a la derecha `>>` por la instrucción `shr` (desplazamiento lógico a la derecha) en lugar de `sar` (desplazamiento aritmético a la derecha), introduciendo ceros por la izquierda en lugar de repetir el bit de signo. La instrucción `sar` provocaría un bucle infinito cuando el bit más significativo de `x` fuera 1.

10.2 Bucle while

Los bucles `while` pueden traducirse una versión que use `do...while`, y a su vez esta a una que use etiquetas y la sentencia `goto`, de acuerdo a la siguiente estructura:

Código C	Versión usando do while	Versión usando goto
`while (Test)` `  Body`	`if (Test)` `  do`	`  if (!Test)` `  goto done;`

	Body while (**Test**);	loop: **Body** if (**Test**) goto loop; done:

La versión con `goto` se puede traducir fácilmente a ensamblador. Podemos verlo en el siguiente ejemplo, de nuevo referida al cálculo del número de bits a 1 en un entero:

Código C	Versión usando do while	Versión usando goto

`int pcount_while (unsigned x)`
`{`
`  int result = 0;`
 	
`  while (x)`
`  {`
`  result += x & 1;`
`  x >>= 1;`
`  }`
 	
`  return result;`
`}`
	`int pcount_while (unsigned x)`
	`{`
	`  int result = 0;`
	
	`  if (x)`
	`  do`
	`  {`
	`  result += x & 1;`
	`  x >>= 1;`
	`  }`
	`  while (x);`
	
	`  return result;`
	`}`
		`int pcount_while (unsigned x)`
		`{`
		`  int result = 0;`
		
		`  if (!x)`
		`  goto done;`
		`loop:`
		`  result += x & 1;`
		`  x >>= 1;`
		`  if (x)`
		`  goto loop;`
		`done:`
		`  return result;`
		`}`

Código ensamblador		

`pcount_while:`
`  movl  $0, %eax  # result = 0`
`  testl  %edi, %edi  # ¿x == 0?`
`  je  .L2`
`.L3:`
`  movl  %edi, %edx  # x`
`  andl  $1, %edx  # x & 1`
`  addl  %edx, %eax  # result += x & 1`
`  shrl  %edi  # x >>= 1`
`  jne  .L3  # if (x != 0) goto .L3`
`.L2:`
`  ret`		

10.3 Bucle for

Los bucles `for` pueden traducirse una versión que use while, esta a su vez a una basada en `do...while`, y por último esta a una que use etiquetas y la sentencia `goto`, de acuerdo a la siguiente estructura:

Código C	Versión usando while	Versión usando do while	Versión usando goto

`for (Init; Test; Update)`
`  Body`
	`Init;`
	`while (Test)`
	`  {`
	`  Body`
	`  Update;`
	`  }`
		`Init;`
		`if (Test)`
		`  do`
		`  {`
		`  Body`
		`  Update;`
		`  }`
		`  while (Test);`
			`  Init;`
			`  if (!Test)`
			`  goto done;`
			`loop:`
			`  Body`
			`  Update;`
			`  if (Test)`
			`  goto loop;`
			`done:`

La versión con `goto` se puede traducir fácilmente a ensamblador. Podemos verlo en el siguiente ejemplo, de nuevo referida al cálculo del número de bits a 1 en un entero:

Código C

```
#define WSIZE (8 * sizeof (int))

int pcount_for (unsigned x)
{
    int i;
    int result = 0;
    unsigned mask;

    for (i = 0; i < WSIZE; i++)
    {
        mask = 1 << i;
        result += (x & mask) != 0;
    }

    return result;
}
```

Versión usando while

```
#define WSIZE (8 * sizeof (int))

int pcount_for (unsigned x)
{
    int i;
    int result = 0;
    unsigned mask;

    i = 0;
    while (i < WSIZE)
    {
        mask = 1 << i;
        result += (x & mask) != 0;
        i++;
    }

    return result;
}
```

Versión usando do while

```
#define WSIZE (8 * sizeof (int))

int pcount_for (unsigned x)
{
    int i;
    int result = 0;
    unsigned mask;

    i = 0;
    if (i < WSIZE)
        do
        {
            mask = 1 << i;
            result += (x & mask) != 0;
            i++;
        }
        while (i < WSIZE);

    return result;
}
```

Versión usando goto

```
#define WSIZE (8 * sizeof (int))

int pcount_for (unsigned x)
{
    int i;
    int result = 0;
    unsigned mask;

    i = 0;
    if (!(i < WSIZE))
        goto done;
loop:
    mask = 1 << i;
    result += (x & mask) != 0;
    i++;
    if (i < WSIZE)
        goto loop;
done:
    return result;
}
```

Código ensamblador

```
pcount_for:
        movl    $0, %eax        # result = 0
        movl    $0, %ecx        # i = 0
        movl    $1, %esi        # 1
.L2:
        movl    %esi, %edx      # 1
        sall    %cl, %edx       # mask = 1 << i
        testl   %edi, %edx      # x & mask
        setne   %dl             # (x & mask) != 0
        movzbl  %dl, %edx
        addl    %edx, %eax      # result += (x & mask) != 0
        addl    $1, %ecx        # i++
        cmpl    $32, %ecx       # ¿i == 32?
        jne     .L2             # if (i == 32) goto .L2
        ret
```

Observe que la comparación previa al bucle, `if (i < WSIZE)`, no es necesario traducirla porque siempre es cierta, ya que `i` vale 0 y `WSIZE` es una constante y vale 32.

11. Cuestionario y problemas

Para afianzar los conceptos, le proponemos un cuestionario con 36 preguntas de tipo test y 8 problemas.

11.1 Cuestionario de autoevaluación

Le sugerimos las siguientes 36 preguntas de autoevaluación de elección única relacionadas con el tema tratado y agrupadas por su temática. Cada pregunta tiene 4 opciones, una de las cuales es correcta.

11.1.1 Preguntas sobre representación de enteros e indicadores de estado

1. El número –12 se almacenará en complemento a 2 en el registro `eax` como:
   a) 0xFFFFFF0C
   b) 0xFF0C
   c) 0xFFFFFFF4
   d) 0xFFF4
2. La instrucción `not`:
   a) realiza el complemento a dos
   b) realiza el complemento a uno (cambiar unos por ceros y ceros por unos)
   c) realiza la operación no-or (or negada)
   d) realiza un salto condicional si negativo
3. La instrucción `negl`:
   a) realiza el complemento a dos
   b) realiza el complemento a uno (cambiar unos por ceros y ceros por unos)
   c) realiza la operación no-or (or negada)
   d) realiza un salto condicional si negativo
4. En una resta de dos números en complemento a dos, se produce desbordamiento cuando…
   a) los dos operandos son negativos y el resultado es positivo.
   b) los dos operandos son positivos y el resultado es negativo.
   c) Las dos respuestas a y b son correctas.
   d) Ninguna de las anteriores es correcta.
5. ¿Cómo actúa el indicador SF del registro de indicadores de estado?
   a) Se pone a 1 cuando el resultado es negativo.
   b) Se pone a 0 cuando el resultado es negativo.
   c) Se pone a 1 cuando el resultado de una operación es 0.
   d) Se pone a 1 cuando el resultado es positivo.
6. La diferencia entre el indicador de acarreo y el de overflow es que...
   a) uno se activa cuando se opera con números con signo y otro cuando son sin signo
   b) el flag de acarreo indica que ha habido acarreo en una operación con números enteros (ints), el de overflow indica que ha habido desbordamiento en una operación con números en punto flotante (p.f.)

   c) ambos se recalculan tras cada operación aritmético-lógica con ints, correspondiendo al programador consultar uno u otro según piense que sus datos son con o sin signo
   d) el de acarreo indica que el resultado es demasiado grande (para p.f.) o positivo (si se trata de ints) para poder almacenarse, el de overflow indica que es demasiado pequeño (p.f.) o negativo (ints)
7. Un overflow nunca puede ocurrir cuando:
   a) se suman dos números positivos
   b) se suman dos números negativos
   c) se suma un número positivo a un número negativo
   d) se resta un número positivo de un número negativo

11.1.2 Preguntas sobre instrucciones de comparación

8. La instrucción IA32 `test` sirve para...
   a) Testear el código de condición indicado, y poner un byte a 1 si se cumple
   b) Mover el operando fuente al destino, pero sólo si se cumple la condición indicada
   c) Realizar la operación resta (a – b) pero no guardar el resultado, sino simplemente ajustar los indicadores de estado
   d) Realizar la operación and lógico bit-a-bit (a & b) pero no guardar el resultado, sino simplemente ajustar los indicadores de estado
9. La instrucción `test` hace...
   a) lo mismo que `sub`
   b) lo mismo que `sub`, pero no guarda el resultado, sólo ajusta los indicadores de estado
   c) lo mismo que `and`
   d) lo mismo que `and`, pero no guarda el resultado, sólo ajusta los indicadores de estado
10. ¿Cuál es la diferencia entre las instrucciones `subl` y `cmpl`?
   a) `subl` realiza una resta y `cmpl` realiza una suma
   b) `subl` no afecta a los indicadores de estado y `cmpl` sí
   c) `subl` tiene en cuenta el acarreo de entrada y `cmpl` no
   d) `subl` almacena el resultado sobrescribiendo uno de los operandos y `cmpl` no
11. La diferencia entre las instrucciones `test` y `cmp` consiste en que
   a) `test` realiza una operación and lógico, mientras que `cmp` realiza una resta
   b) `test` modifica sólo los flags lógicos (ZF, SF) mientras que `cmp` modifica los aritmético-lógicos (ZF, SF, CF, OF)
   c) ambas respuestas son correctas
   d) ambas respuestas son incorrectas
12. ¿Cuál de los siguientes grupos de instrucciones sólo modifican los indicadores de estado sin almacenar el resultado de la operación?
   a) `and`, `or`, `xor`
   b) `adc`, `sbb`
   c) `cmp`, `test`
   d) `imul`, `idiv`
13. Para comprobar si el entero almacenado en `eax` es cero (y posiblemente saltar a continuación usando `jz`/`jnz`), gcc genera el código
   a)
   ```
   cmp %eax, $0
   ```
   b)
   ```
   test %eax
   ```
   c)
   ```
   cmp %eax
   ```

d) `test %eax, %eax`

11.1.3 Preguntas sobre instrucciones condicionales

14. La instrucción `seta %al` (`seta` significa "*set if above*"):
    a) Pone `al` a 1 si CF=0 y ZF=0
    b) Pone `al` a 1 si CF=0 o ZF=0
    c) Pone `al` a 1 si CF=1 y ZF=0
    d) Pone `al` a 1 si CF=1 o ZF=1
15. La instrucción `setg %al`:
    a) No cambia el contenido de `al`
    b) Pone siempre `al` a 1.
    c) Pone `al` a 1 en algunos casos.
    d) Complementa `al` si el resultado de la comparación anterior es A > B.
16. Las instrucciones de salto...
    a) son uno de los tipos de instrucciones máquina con menor frecuencia dinámica de uso.
    b) complican el diseño eficiente de los procesadores segmentados.
    c) siempre utilizan direccionamiento absoluto.
    d) Todas las afirmaciones anteriores son ciertas.
17. ¿Cuál de las siguientes instrucciones no modifica necesariamente la secuencia de ejecución del programa?
    a) `jmp dir`
    b) `jne dir`
    c) `call dir`
    d) `ret`
18. Después de ejecutar una instrucción de suma sobre dos números con signo de la que sabemos que no provocará overflow (los dos números son pequeños en valor absoluto), queremos comprobar si el resultado de la suma es menor que 0. ¿Qué indicador de estado necesita comprobar la instrucción de salto condicional equivalente a… ?
    `if (resultado<0) goto label;`
    a) CF
    b) OF
    c) SF
    d) ZF
19. Una instrucción de "salto si igual" tiene que comprobar el valor de:
    a) el bit de acarreo
    b) el bit de cero
    c) el bit de signo
    d) los bits de signo y desbordamiento
20. Una instrucción de "salto si menor", para números sin signo, tiene que comprobar el valor de:
    a) el bit de acarreo
    b) el bit de cero
    c) el bit de signo
    d) los bits de signo y desbordamiento
21. Una instrucción de salto si menor, para números positivos sin signo, tiene que comprobar el valor de:

a) el bit de acarreo
b) el bit de cero
c) el bit de signo
d) los bits de signo y desbordamiento

22. ¿Qué combinación de indicadores de estado corresponde al código de condición `b` (*below*)?
    a) CF
    b) OF
    c) CF ^ OF
    d) OF ^ SF
23. Las instrucciones `jb` y `jnae` provocan un salto si...
    a) SF == 1
    b) CF == 1
    c) ZF == 0
    d) ZF != SF
24. Tras comparar números con signo, para saltar si menor se utilizan:
    a) El acarreo
    b) El acarreo y el cero
    c) El overflow
    d) El overflow y el signo
25. La instrucción `jge` / `jnl` provoca un salto si…
    a) SF = 1
    b) CF = 1
    c) SF = 0
    d) OF = SF
26. ¿Cuál de las siguientes parejas de mnemotécnicos de ensamblador IA32 corresponden a la misma instrucción máquina?
    a) `cmp` (comparar), `sub` (restar)
    b) `jc` (saltar si acarreo), `jl` (saltar si menor, para números con signo)
    c) `jz` (saltar si cero), `je` (saltar si igual)
    d) `sar` (desplazamiento aritmético a la derecha), `shr` (desplazamiento lógico a la derecha)
27. La instrucción `cmovb %edx,%eax`
    a) copia en `eax` el contenido de `edx` si el indicador de acarreo es 1
    b) copia el byte bajo de `edx` en el byte bajo de `eax`
    c) copia en `eax` el byte de memoria apuntado por la dirección contenida en `edx`
    d) copia en `eax` el contenido de `edx` si `eax` es menor que `edx`

11.1.4 Preguntas sobre traducción de sentencias condicionales y bucles

28. Para traducir una construcción *if-then-else* de lenguaje C a lenguaje ensamblador, gcc utiliza generalmente
    a) un salto condicional, según la condición expresada en el código C
    b) un salto condicional, según la condición opuesta a la del código C, y otro salto incondicional
    c) dos saltos condicionales (uno para la parte *if* y otro para la parte *else*)
    d) dos saltos condicionales y dos saltos incondicionales

29. Si A y B son dos enteros almacenados respectivamente en `eax` y `ebx`, ¿cuál de las siguientes implementaciones de `if (!A && !B) {...then part...}` es incorrecta?

```
a)          or      %ebx, %eax
            jne     not_true
            ...then part...
   not_true:
            ...
b)          cmp     $0, %eax
            jne     not_true
            cmp     $0, %ebx
            jne     not_true
            ...then part...
     not_true:
            ...
c)          test    %ebx, %eax
            jne     not_true
            ...then part...
     not_true:
            ...
d)          test    %eax, %eax
            jne     not_true
            test    %ebx, %ebx
            jne     not_true
            ...then part...
     not_true:
            ...
```

30. ¿Qué salida produce el siguiente código? Asuma representación de datos de arquitectura IA32.

```
unsigned int x = 0xDEADBEEF;
unsigned short y = 0xFFFF;
signed int z = -1;
if (x > (signed short) y)
    printf("Hello");
if (x > z)
    printf("World");
```

(Aviso: en comparaciones con un dato `unsigned` se pasa el otro dato a `unsigned`)

a) No imprime nada
b) Imprime “Hello”
c) Imprime “World”
d) Imprime “HelloWorld”

31. Si el registro `eax` contiene `X`, la secuencia de instrucciones siguiente:

```
cmpl    $6, %eax
jae     Destino
```

salta a la etiqueta `Destino` sólo si:

a) `X >  6`
b) `X <= 6`
c) `X <  0 || X >= 6`

d) `X >= 0 && X <= 6`

32. Al ejecutar el fragmento de código:

```
leal    -48(%eax), %edx
cmpl    $9, %edx
ja      .L2
```

se salta a `.L2` si el contenido del registro `eax`:

a) está dentro del intervalo [48,57]
b) es mayor o igual que 48
c) es mayor o igual que 57
d) está fuera del intervalo [48,57]

33. De entre las siguientes construcciones de flujo de control en lenguaje C, la que se traduce más directamente a lenguaje ensamblador es...
    a) El bucle *for*
    b) El bucle *while*
    c) El bucle *do-while*
    d) La selección *switch-case*
34. Para traducir una construcción *do-while* de lenguaje C a lenguaje ensamblador, gcc utiliza generalmente
    a) un salto condicional hacia adelante, según la misma condición que en lenguaje C
    b) un salto condicional hacia atrás, según la misma condición que en lenguaje C
    c) un salto condicional hacia adelante, según la condición opuesta a la de lenguaje C
    d) un salto condicional hacia atrás, según la condición opuesta a la de lenguaje C
35. Una sentencia en C del tipo "`while (test) body;`" puede transformarse en código "goto" como:

a)
```
if (!test) goto done; loop: body; if (test) goto loop;
done:
```
b)
```
loop: body; if (test) goto loop;
```
c)
```
if (test) goto true; goto done; true: body; done:
```
d)
```
loop: if (test) goto done; body; goto loop; done:
```

36. Una sentencia en C del tipo "`while (test) body;`" puede transformarse en código "goto" como:

a)
```
    if (!test) goto done;
loop:
    body;
    if (test) goto loop;
done:
```
b)
```
loop:
    body;
    if (test) goto loop;
```
c)
```
    if (test) goto true;
    goto done;
true:
    body;
done:
```
d)
```
loop:
    if (test) goto done;
    body;
    goto loop;
```

```
done:
```

11.2 Problemas

Le proponemos los siguientes 8 problemas:

1. Bucles while (examen del 28 de enero de 2013; 0,5 puntos). Una función, `fun_a`, tiene la siguiente estructura general:

```
int fun_a (unsigned x) {
    int val = 0;

    while ( ________ ) {
        ________;
    }
    return ________;
}
```

El compilador GCC genera el siguiente código ensamblador:

```
      # x en %ebp+8
      movl        8(%ebp), %edx
      movl        $0, %eax
      testl %edx, %edx
      je          .L7
.L10:
      xorl        %edx, %eax
      shrl        %edx        # Desplazar a derecha 1
      jne         .L10
.L7:
      andl        $1, %eax
```

Analice el funcionamiento de este código y responda a las siguientes preguntas:

A. Use la versión en código ensamblador para rellenar las partes que faltan del código C.

B. Describa en castellano qué calcula esta función.

Solución:

A.

```
int fun_a (unsigned x) {
    int val = 0;

    while (x) {
        val ^= x;
        x >>= 1;
    }
    return val & 0x1;
```

```
}
```

B. Calcula la paridad impar de *x* (devuelve 1 si hay un número impar de unos, 0 si par)

2. Bucles for (examen del 9 de septiembre de 2013; 0,5 puntos). Una función, `fun_c`, tiene la siguiente estructura general:

```
long fun_c(unsigned long x) {
    long val = 0;
    int i;

    for ( ________ ; ________ ; ________ ; ) {
        ________;
    }
    ________;
    return ________;
}
```

El compilador C GCC genera el siguiente código ensamblador:

```
      # x en %rdi
fun_c:
      movl        $0, %ecx
      movl        $0, %edx
      movabsq     $72340172838076673, %rsi      # Mover un valor de 64b
.L2:
      movq        %rdi, %rax
      andq        %rsi, %rax
      addq        %rax, %rcx
      shrq        %rdi                          # Desplazar a derecha 1
      addl        $1, %edx
      cmpl        $8, %edx
      jne         .L2
      movq        %rcx, %rax
      sarq        $32, %rax
      addq        %rcx, %rax
      movq        %rax, %rdx
      sarq        $16, %rdx
      addq        %rax, %rdx
      movq        %rdx, %rax
      sarq        $8, %rax
      addq        %rdx, %rax
      andl        $255, %eax
      ret
```

Analice el funcionamiento de este código y responder a las siguientes preguntas. Resultará útil convertir la constante decimal de la línea 4 (`movabsq`) a hexadecimal.

A. Use la versión ensamblador para rellenar las partes que faltan del código C.

B. Describa en castellano qué calcula esta función.

Solución:

A.

```
long fun_c(unsigned long x) {
    long val = 0;
    int i;

    for (i = 0; i < 8; i++) {
        val += x & 0x0101010101010101;
        x >>= 1;
    }
    val += (val >> 32);
    val += (val >> 16);
    val += (val >> 8);
    return val & 0xFF;
}
```

B. Calcula el n.º de bits a 1 (*popcount*) de `x`.

Va haciendo 8 sumas parciales en paralelo en los 8 bytes de `val`. Luego se suman las 4 sumas superiores (7-4) con las 4 inferiores (3-0), se siguen acumulando 2 superiores (3-2) con 2 inferiores (1-0), y por último, la suma del byte (1) con la del byte inferior (0). Se retorna sólo ese byte, en donde están sumados todos los bits. La ventaja es que hace sólo 8 iteraciones, 3 sumas y una máscara en lugar de 64 iteraciones.

3. Ensamblador (examen del 14 de febrero de 2014; 0,5 puntos). Dado el siguiente fragmento de código escrito en C, escriba el código de la función equivalente utilizando el ensamblador de IA32:

```
int max (int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Para el desarrollo de este problema ha de seguirse la convención de paso de parámetros _cdecl.

Solución:

Alternativa 1

```
max:
      pushl %ebp
      movl  %esp, %ebp
      movl  8(%ebp), %eax
      cmpl  12(%ebp), %eax
      jle   .else
```

Alternativa 2

```
max:
      pushl %ebp
      movl  %esp, %ebp
      movl  8(%ebp), %edx
      movl  12(%ebp), %eax
      cmpl  %edx, %eax
```

```
      movl  8(%ebp), %eax
      jmp   .cont
.else:
      movl  12(%ebp), %eax
.cont:
      popl  %ebp
      ret
```

```
      jge   .cont
.else:      #innecesario
      movl  %edx, %eax
.cont:
      popl  %ebp
      ret
```

4. Ensamblador IA32 (examen del 3 de septiembre de 2014; 0,5 puntos). Traduzca a ensamblador de IA32 la siguiente función escrita en C:

```
int hex2bin (int c) {
    if (c >= '0' && c <= '9')
        return c - '0';
    if (c >= 'A' && c <= 'F')
        return c + 10 - 'A';
    if (c >= 'a' && c <= 'f')
        return c + 10 - 'a';
    return -1; // Error code
}
```

Códigos de los caracteres:

- '0' → 48
- '9' → 57
- 'A' → 65
- 'Z' → 70
- 'a' → 97
- 'z' → 102

Solución:

Otras soluciones son posibles, siempre que produzcan el resultado correcto y no añadan complejidad innecesaria. Se muestra una solución sin optimizar (a la izquierda) y otra optimizada (a la derecha).

Solución sin optimizar

```
hex2bin:
      pushl %ebp
      movl  %esp, %ebp

      cmpl  $48, 8(%ebp)
      jl    .L2
      cmpl  $57, 8(%ebp)
      jg    .L2
      movl  8(%ebp), %eax
      subl  $48, %eax
      jmp   .L3
.L2:
      cmpl  $65, 8(%ebp)
      jl    .L4
      cmpl  $70, 8(%ebp)
      jg    .L4
      movl  8(%ebp), %eax
      subl  $55, %eax
      jmp   .L3
.L4:
      cmpl  $97, 8(%ebp)
      jl    .L5
      cmpl  $102, 8(%ebp)
      jg    .L5
      movl  8(%ebp), %eax
      subl  $87, %eax
```

Solución optimizada

```
hex2bin:



      movl  4(%esp), %eax
      leal  -48(%eax), %edx
      cmpl  $9, %edx
      ja    .L2
      movl  %edx, %eax
      ret

.L2:
      leal  -65(%eax), %edx
      cmpl  $5, %edx
      ja    .L4
      subl  $55, %eax
      ret


.L4:
      leal  -97(%eax), %edx
      subl  $87, %eax
      cmpl  $5, %edx
      movl  $-1, %edx
      cmova %edx, %eax
      ret
```

```
      jmp   .L3
.L5:
      movl  $-1, %eax
.L3:
      popl  %ebp
      ret
```

5. Ensamblador IA32 (examen del 2 de febrero de 2015; 0,75 puntos). Traduzca a ensamblador de IA32 la siguiente función escrita en C:

```
int max (int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Utilizando la función en ensamblador anterior, implemente el código de la siguiente función:

```
void maxV (int *v1, int *v2, int *v3, int N)
{
    int i;
    for (i = 0; i < N; i++)
        v3[i] = max (v1[i], v2[i]);
}
```

Para el desarrollo de este problema a ha de seguirse la convención de paso de parámetros _cdecl.

Solución:

Otras soluciones son posibles, siempre que produzcan el resultado correcto y no añadan complejidad innecesaria. Se muestra una solución sin optimizar (a la izquierda) y otra optimizada (a a la derecha).

Solución sin optimizar

```
max:
      push  %ebp
      mov   %esp, %ebp  # Ajuste marco
      mov   8(%ebp), %eax     # EAX=a
      mov   12(%ebp), %edx    # EDX=b
      cmp   %edx, %eax  # a:b?
      jg    .finmax     # a>b  - nada
      mov   %edx, %eax  # a<=b – ret b
.finmax:
      pop   %ebp        # Destruir marco
      ret

maxV:
      push  %ebp
      mov   %esp, %ebp  # Ajuste marco
      push  %esi  # Usar todos los
      push  %edi  # salva-invocado
      push  %ebx  # porque hay un call
```

Solución optimizada

```
max:
      push  %ebp
      mov   %esp, %ebp  # Ajuste marco
      mov   8(%ebp), %eax     # EAX=a
      mov   12(%ebp), %edx    # EDX=b
      cmp   %edx, %eax  # a:b?
      cmovle %edx, %eax # a<=b – ret b

      pop   %ebp        # Destruir marco
      ret

maxV:
      push  %ebp
      mov   %esp, %ebp  # Ajuste marco
      push  %esi  # Usar todos los
      push  %edi  # salva-invocado
      push  %ebx  # porque hay un call
```

```
                                        sub   $8, %esp    # pila para max
      mov   8(%ebp), %esi     # ESI = v1  mov   8(%ebp), %esi     # ESI = v1
      mov   12(%ebp), %edi    # EDI = v2  mov   12(%ebp), %edi    # EDI = v2
      cmp   $0, 20(%ebp)      # N:0?      cmp   $0, 20(%ebp)      # N:0?
      jle   .finmaxV    # N<=0 - acabar   jle   .finmaxV    # N<=0 - acabar
      mov   $0, %ebx    # for i=0; EBX=i  mov   $0, %ebx    # for i=0; EBX=i
.bucle:                                 .bucle:
      push  (%edi,%ebx,4) # arg 2º v2[i]  mov   (%edi,%ebx,4), %eax # v2[i]
      push  (%esi,%ebx,4) # arg 1º v1[i]  mov   %eax, 4(%esp)     # arg 2º
                                        mov   (%esi,%ebx,4), %eax # v1[i]
                                        mov   %eax, (%esp)      # arg 1º
      call  max                         call  max
      add   $8, %esp    # Recuperar pila
      mov   16(%ebp), %edx    # EDX = v3  mov   16(%ebp), %edx    # EDX = v3
      mov   %eax, (%edx,%ebx,4) # v3[i]   mov   %eax, (%edx,%ebx,4) # v3[i]
      inc   %ebx              # i++       inc   %ebx              # i++
      cmp   20(%ebp), %ebx    # i:N?      cmp   20(%ebp), %ebx    # i:N?
      jne   .bucle      # i!=N - repetir  jne   .bucle      # i!=N - repetir
.finmaxV:                               .finmaxV:
                                        add   $8, %esp    # Recuperar pila
      pop   %ebx  # Salva-invocados     pop   %ebx  # Salva-invocados
      pop   %edi                        pop   %edi
      pop   %esi                        pop   %esi
      pop   %ebp  # Destruir marco       pop   %ebp  # Destruir marco
      ret                               ret
```

6. Ensamblador IA32 (examen del 8 de febrero de 2016; 0,2 puntos). Al ejecutar el fragmento de código:

```
leal  -48(%eax), %edx
cmpl  $9, %edx
ja    .L2
```

¿Para qué valores de `%eax` se salta a `.L2`?

Solución:

El enunciado no requería comentar el código, el comentario se ofrece sencillamente como explicación.

```
leal  -48(%eax), %edx   # edx = eax-48
cmpl  $9, %edx          # > 9?
ja    .L2               # above (sin signo)
```

EAX podría contener un código ASCII, en cuyo caso LEA lleva el char ‘0’ a valor 0, ninguno de los caracteres desde ‘0’ a ‘9’ cumplen la condición (‘9’ se queda justo en el valor 9), y el resto de caracteres la cumplen.

Respuesta: se salta a .L2 para valores de EAX = [0..47] U [58..0xffffffff]

Alternativa: *NO* se salta para valores de EAX = [48..57]

7. Ensamblador a C (examen del 7 de septiembre de 2016; 0,6 puntos). Dada la siguiente función en ensamblador x86-64:

```
loop:
# a la entrada: a en %rdi, n en %esi
      movl  $0, %r8d
      movl  $0, %ecx
      testl %esi, %esi
      jle   .L3
.L6:
      movl  (%rdi,%rcx,4), %edx
      leal  3(%rdx), %eax
      testl %edx, %edx
      cmovns %edx, %eax
      sarl  $2, %eax
      addl  %eax, %r8d
      addq  $1, %rcx
      cmpl  %ecx, %esi
      jg    .L6
.L3:
      movl  %r8d, %eax
      ret
```

Rellenar los huecos en el código C correspondiente.
Se debe usar sintaxis de indexación en arrays para acceder a los elementos de a.
Naturalmente, no se deben usar nombres de registros x86-64 en código C.
Si se duda sobre la equivalencia aritmética de la operación de desplazamiento, expresarla también como desplazamiento en lenguaje C.

```
int loop(int a[], int n)
{
    int i, sum;
    sum = _____;
    for (i = _________; _________; _________) {
        sum += _______________;
    }
    return ____________;
}
```

Solución:

```
int loop(int a[], int n)
{
    int i, sum;
    sum = 0;
    for (i = 0; i<n; i++) {
        sum += a[i]/4;
        //alter.sum += (a[i]>0? a[i] : a[i]+3) >> 2;
    }
```

```
    return sum;
}
```

8. Ensamblador (examen del 23 de enero de 2017; 0,7 puntos). Dado el siguiente programa escrito en C:

```
#include <stdio.h>

unsigned int x;
unsigned short y;
signed int z;

void main (void) {
      if (x > (signed short) y)
            printf ("Hello ");
      if (x > z)
            printf ("world");
}
```

a) Escriba la función **main()** en ensamblador de IA32. Para ello, tenga en cuenta estas aclaraciones:

1. En comparaciones con tamaños diferentes, se pasa el dato de menor tamaño al tamaño del mayor.
2. En comparaciones que implican **unsigned** y **signed**, se pasa el **signed** a **unsigned**.
3. Si hay que realizar un cambio de tamaño y de "signedness" (de **signed** a **unsigned** o viceversa), primero se realiza el cambio de tamaño manteniendo el signo y luego el de "signedness".

Ayuda:

- La instrucción **movzwl src,dst** pasa de 16 a 32 bits añadiendo ceros por la izquierda
- La instrucción **movswl src,dst** pasa de 16 a 32 bits extendiendo el signo
- Las instrucciones se salto condicional para **unsigned** usan los sufijos *a* (above) y *b* (below)

b) Suponiendo que las variables *x*, *y* y *z* estuvieran inicializadas así:

```
unsigned int x = 0xFFFFFFFF;
unsigned short y = 0xDEAD;
signed int z = -1;
```

¿qué imprimiría el programa?

Solución:

a) Se puntúa 0,6 p por el siguiente programa (0,05 p por instrucción)

```
movswl      y, %eax          # Cambio a int y luego a unsigned
cmpl        x, %eax          # Compara y con x
jae/jnb     .L2              # Comparación unsigned
pushl       $hello
call        printf
addl        $4, %esp
```

```
.L2:
   movl        z, %eax           # Cambio a unsigned
   cmpl        %eax, x           # Compara x con z
   jbe/jna     .L1               # Comparación unsigned
   pushl       $world
   call        printf
   addl        $4, %esp
.L1:
```

b) Imprimiría “Hello ” (se puntúa 0,1 p si la respuesta es correcta)
   1º comparación: `0xFFFF FFFF >  0xFFFF DEAD`
   2º comparación: `0xFFFF FFFF == 0xFFFF FFFF`

12. Bibliografía

Randal E. Bryant, David R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Third edition. Pearson Global Edition, 2016.
http://csapp.cs.cmu.edu/

Intel. *Intel® 64 and IA-32 Architectures Software Developer Manuals.* 2017.
https://software.intel.com/en-us/articles/intel-sdm

Agner Fog. *Optimizing subroutines in assembly language. An optimization guide for x86 platforms.* Technical University of Denmark. 1996 - 2017.
http://agner.org/optimize/optimizing_assembly.pdf

Henry S. Warren. *Hacker's Delight.* Second Edition, Addison-Wesley, 2012.
http://www.hackersdelight.org/