



Ingeniería Inversa

Adan Avilés

Septiembre 2021

Índice

1. División del código en basic blocks	3
2. Realizar el diagrama de flujo de los basic blocks.	5
3. ¿Existe alguna estructura de control? Indica qué basic blocks intervienen en ella.	5
4. Convertir el código completo de la función en código C.	7
5. Compilar el código generado e indicar el código resultante tras su ejecución. Compilar en 32bits agregando la opción <code>-m32r</code> .	9
6. Modificar el código fuente en C, para que genere un nuevo código a partir de otra cadena dada.	9

1. División del código en basic blocks

Utilizando lo aprendido en teoría, sabemos que el algoritmo para dividir el lenguaje ensamblador en basic blocks se basa en los siguientes aspectos:

- Identificación de las instrucciones líder.
 - La primera instrucción del código.
 - La instrucción de destino de una instrucción de salto.
 - La instrucción que sigue inmediatamente a una instrucción de salto.
- A partir de una instrucción líder, todas las instrucciones siguientes hasta la próxima instrucción líder (sin incluirla) o bien hasta el final del código constituyen un bloque básico asociado a la instrucción líder. Por tanto, cada bloque básico solo puede tener una instrucción líder.

Por tanto, el primer basic block empezará en la primera instrucción del código (bloque de preparación), y acabará en la primera instrucción de salto:

Bloque 1. B_1

```
0x0000054d <+0>: lea ecx,[esp+0x4]
0x00000551 <+4>: and esp,0xffffffff0
0x00000554 <+7>: push DWORD PTR [ecx-0x4]
0x00000557 <+10>: push ebp
0x00000558 <+11>: mov ebp,esp
0x0000055a <+13>: push ebx
0x0000055b <+14>: push ecx
0x0000055c <+15>: sub esp,0x10
```

El siguiente bloque empieza en la instrucción siguiente a la instrucción del salto anterior, terminando también en otra respectiva instrucción de salto:

Bloque 2. B_2

```
0x0000055f <+18>: call 0x450 <_x86.get_pc_thunk.bx>
0x00000564 <+23>: add ebx,0x1a9c
0x0000056a <+29>: mov DWORD PTR [ebp-0x10],0x0
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax
0x0000057a <+45>: sub esp,0xc
```

Podemos tener en cuenta que la función *call* es considerada una función de salto, pero en nuestro caso, realiza un salto a una zona del código no disponible y vuelve al mismo sitio, por tanto no se considera que sea una instrucción de salto para la construcción de los basic blocks.

Bloque 3. B_3

```
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
0x00000585 <+56>: add esp,0x10
```

Análogamente, tenemos otro *call* a la función *<strlen@plt>*, de la que no disponemos, por tanto no se considerará tampoco una instrucción de salto para construir los basic blocks.

Bloque 4. B_4

```
0x00000588 <+59>: mov DWORD PTR [ebp-0x18],eax
0x0000058b <+62>: mov DWORD PTR [ebp-0xc],0x0
0x00000592 <+69>: jmp 0x5ad <main+96>
```

Este bloque acaba con la función *jmp* a la posición +96 del código, para después en +102 devolvernos a +71, por lo tanto podremos obtener los basic block:

Bloque 5. B_5

```
0x00000594 <+71>: mov edx,DWORD PTR [ebp-0xc]
0x00000597 <+74>: mov eax,DWORD PTR [ebp-0x14]
0x0000059a <+77>: add eax,edx
0x0000059c <+79>: movzx eax,BYTE PTR [eax]
0x0000059f <+82>: movsx eax,al
0x000005a2 <+85>: imul eax,DWORD PTR [ebp-0x18]
0x000005a6 <+89>: add DWORD PTR [ebp-0x10],eax
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1
```

El bloque siguiente empieza ahí dado que es el destino de una instrucción de salto.

Bloque 6. B_6

```
0x000005ad <+96>: mov eax,DWORD PTR [ebp-0xc]
0x000005b0 <+99>: cmp eax,DWORD PTR [ebp-0x18]
0x000005b3 <+102>: jl 0x594 <main+71>
```

Como la línea +104 está después de una instrucción de salto, constituye un bloque por sí sola:

Bloque 7. B_7

```
0x000005b5 <+104>: sub esp,0x8
```

Siguiendo las indicaciones, la primera instrucción después de una instrucción de salto, es una instrucción líder. Además, como hemos comentado anteriormente, al volver al mismo sitio de ejecución, la función *call* de +117 no se considera como salto en la construcción de los basic blocks.

Bloque 8. B_8

```
0x000005b8 <+107>: push DWORD PTR [ebp-0x10]
0x000005bb <+110>: lea eax,[ebx-0x1992] ; “[+] Código generado: %i\n”
0x000005c1 <+116>: push eax
0x000005c2 <+117>: call 0x3d0 <printf@plt>
0x000005c7 <+122>: add esp,0x10
```

Para terminar, presentamos el bloque de restitución del código.

Bloque 9. B_9

```
0x000005ca <+125>: mov eax,0x0
0x000005cf <+130>: lea esp,[ebp-0x8]
0x000005d2 <+133>: pop ecx
0x000005d3 <+134>: pop ebx
0x000005d4 <+135>: pop ebp
0x000005d5 <+136>: lea esp,[ecx-0x4]
0x000005d8 <+139>: ret
```

2. Realizar el diagrama de flujo de los basic blocks.

Sabemos que la instrucción *jl* del B_6 es una instrucción de salto condicional, desde ese bloque podemos entonces volver a +71 (en el bloque 5) o seguir hacia el bloque 7. Por tanto el diagrama sería:

3. ¿Existe alguna estructura de control? Indica qué basic blocks intervienen en ella.

Se observa que los bloques B_4 , B_5 y B_6 forman una estructura de control basada en el condicional del bloque 6.

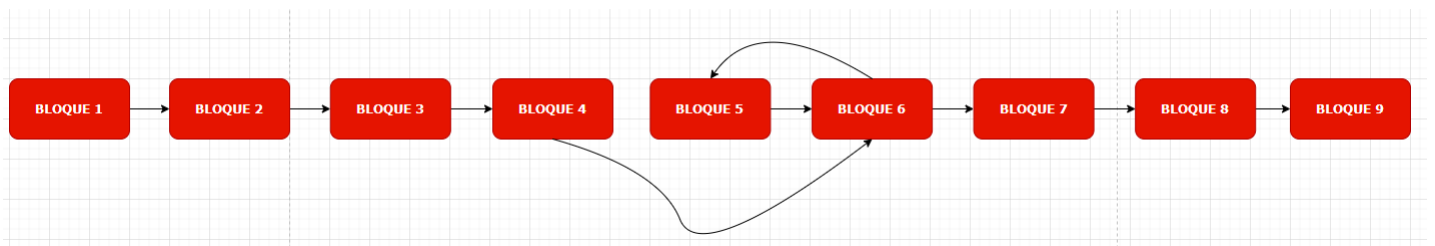


Figura 1: Diagrama de flujo de los bloques

Esta condición, en caso de cumplirse, realiza un salto a +71, que es el inicio del bloque 5 y tras recorrerse los bloques 5 y 6, se vuelve a comprobar la condición. Si esta vez no se cumple, proseguirá por el bloque 7 (o volverá a entrar en el bucle.).

Esta estructura, puede ser tanto de un bucle *FOR* como de un bucle *WHILE*, aparentemente. Pues en primer lugar se produce el salto y establece las condiciones (salto del bloque 4 al bloque 6, donde esta la condición) y luego se tienen en cuenta las condiciones.

4. Convertir el código completo de la función en código C.

El código en C será:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int valor = 0;
    char* cadena = "3jd9cjfk98hnd";
    int longitud = strlen(cadena);

    for (int i = 0; i < longitud; i++) {
        valor += cadena[i] * longitud;
    }

    printf("[+] Codigo generado: %i\n", valor);
}
```

Vamos a explicar a continuación cómo hemos obtenido este código: (NOTA: Pese a que en el código original de las líneas anteriores aparezca `*` al lado de `char`, es por un fallo del editor \LaTeX , nos referimos a `*`)

En primer lugar, vemos que se usan las funciones *printf* y *strlen*, que muestran el resultado por pantalla y calculan la longitud de una cadena, respectivamente, por tanto es necesario incluir *stdio.h* y *string.h*.

A continuación, empezamos la construcción de la función `main()`, teniendo en cuenta los parámetros *argc* y *argv*. Podemos ver que en la posición +29 se prepara el valor `0x0`, que es el 0, en `[ebp-0x10]`, que se usa después en +89 y +107. En +89, se usa la instrucción *add*, por tanto se está haciendo una suma al valor ya existente en la dirección indicada, y como +107 es la última antes de la función *printf* (tras cargar la cadena de texto “[+] Codigo generado: %i\n”).

Vemos ahora que la variable que contiene el valor inicial 0, contiene también el valor final que aparece por pantalla: podemos iniciar una variable `int` con valor 0.

```
int valor = 0;
```

Seguidamente, observamos que se carga la cadena de texto "3jd9cjfk98hnd".^{en} [ebp-0x14], y se usa como parámetro en la función *strlen*, como podemos ver en

```
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
0x00000577 <+42>: mov DWORD PTR [ebp-0x14],eax
0x0000057d <+48>: push DWORD PTR [ebp-0x14]
0x00000580 <+51>: call 0x3e0 <strlen@plt>
```

Además, el resultado de *strlen* está almacenado en [ebp-0x18], por tanto es necesario crear una variable para almacenarlo.

```
char* cadena = "3jd9cjfk98hnd";
int longitud = strlen(cadena);
```

Entramos ahora con las condiciones del bucle. Tras almacenar el valor 0 en [ebp-0xc] se produce un salto a +96, que es el comienzo de la comprobación del bucle. Esta comprobación, se hace entre [ebp-0x18] y [ebp-0xc], la longitud de la cadena de texto y la variable de control, respectivamente. Podemos deducir que la condición para que el bucle avance es que la variable de control sea menor en valor a la de la longitud de texto. Esto lo podemos observar en +102

```
0x000005b3 <+102>: jl 0x594 <main+71>
```

Pues *jl* se refiere a jump if less.

Como último paso, vemos que aumenta en uno el valor almacenado en [ebp-0xc].

```
0x000005a9 <+92>: add DWORD PTR [ebp-0xc],0x1
```

Lo cual nos da pie el código:

```
for (int i = 0; i < longitud; i++)
```

El grosso de la operación viene a continuación. Entre +71 y +82 se obtiene un carácter en específico de nuestra cadena. Este valor, se multiplica por la longitud de la cadena (que estaba almacenada en [ebp-0x18]) siendo almacenado el resultado en el registro *eax*, y sumándose a la variable de resultado [ebp-0x10] (como se puede ver en +85 y +89).

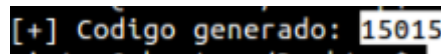
```
valor += cadena[i] * longitud;
```

Por último, finalizado el bucle se carga "[+] Código generado:% i\n" se usa con la variable almacenada en [ebp-0x10] para mostrar su valor por pantalla, siendo parámetros de *printf*. De donde obtenemos:

```
printf("[+] Código generado: % i\n", valor)
```


5. Compilar el código generado e indicar el código resultante tras su ejecución. Compilar en 32bits agregando la opción `-m32r`.

Si compilamos como se nos dice, obtenemos:



```
[+] Codigo generado: 15015
```

Figura 2: Resultado del código

6. Modificar el código fuente en C, para que genere un nuevo código a partir de otra cadena dada.

Se nos solicita que modifiquemos el valor de la cadena en `+36`, es decir, de:

```
0x00000571 <+36>: lea eax,[ebx-0x19a0] ; "3jd9cjfk98hnd"
```

Por tanto solo tendremos que cambiar `"3jd9cjfk98hnd"` por `"Congratulations!"`, obteniendo el código:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int valor = 0;
    char* cadena = "Congratulations!";
    int longitud = strlen(cadena);

    for (int i = 0; i < longitud; i++) {
        valor += cadena[i] * longitud;
    }

    printf("[+] Codigo generado: %i\n", valor);
}
```

que da como resultado: 26080.