Universidad La Salle

Ingeniería de Software

Compiladores

Práctica 5

Profesor: Vicente Machaca Arceda

Integrantes:
Fran Guido Felix Gomez
Patrick Leopoldo Paredes Neira

Arequipa - 2022

1 *Ejercicio 1:*

Redacta el siguiente código, genera el código ensamblador y explica en qué parte (del código ensamblador) se definen las variables c y m.

1.1 Código

```
#include <iostream>
int main() {
  char *c ="abcdef";
  int m = 11148;

  return 0;
}
```

1.2 Código Ensamblador

```
        .LC0:
        .string "abcdef"
main:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], OFFSET FLAT:.LC0
        mov     DWORD PTR [rbp-12], 11148
        mov     eax, 0
        pop     rbp
        ret
__static_initialization_and_destruction_0(int, int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L5
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L5
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L5:
        nop
        leave
        ret
_GLOBAL__sub_I_main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

1.3 Explicación de la definición de las variables c y m

Primeramente identificamos las etiquetas que son: LC0, main, staticinitializationanddestruction0(int, int), L5 y finalmente GLOBALsubImain. Observamos que el código que se encuentra debajo de la etiqueta LCO, simplemente da información de la cadena.

En donde encontramos la definición de las variables c y m es en el código que perteneces a la etiqueta *main* ya que esta representa las instrucciones.

En la etiqueta main encontramos las instrucciones mov QWORD PTR[rbp-8], OFFSET FLAG: LC0 que es aqui en donde va a definir la variable c y va a ser almacenado en qword ptr en ese espacio de memoria, en la siguiente instrucción ocurre lo mismo se va a definir el entero m 11148 en ese espacio de memoria. finalmente colocamos un cero en el registro eax, ya que es importante para realizar el comando return 0 en c++, finalmente sacamos el registro rbp con la instrucción pop y hacemos un ret.

```
main:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], OFFSET FLAT:.LC0
        mov     DWORD PTR [rbp-12], 11148
        mov     eax, 0
        pop     rbp
        ret
```

2 *Ejercicio 2:*

Redacta el siguiente código, genera el código ensamblador y explica en qué parte (del código ensamblador) se define la división entre 8.

2.1 Código

```
#include <iostream>
int main() {
  char *c = (char *)"abcdef";
  int m = 11148;
  int x = m / 8;

  return 0;
}
```

2.2 Código Ensamblador

```
        .LC0:
        .string "abcdef"
main:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], OFFSET FLAT:.LC0
        mov     DWORD PTR [rbp-12], 11148
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+7]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 3
        mov     DWORD PTR [rbp-16], eax
        mov     eax, 0
        pop     rbp
        ret
__static_initialization_and_destruction_0(int, int):
        push    rbp
```

```
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L5
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L5
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L5:
        nop
        leave
        ret
_GLOBAL__sub_I_main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

2.3 Explicación del algoritmo

Nos situamos en la primera parte del código .LCO: que guarda un string como se lee, en la sexta línea vemos la primera definición de c que llama a .LCO luego en la siguiente línea vemos la definición del entero al verse el entero al final. Pasamos a la división como ésta necesita de m para poder dividirse entre 8 tiene que ser llamada por la siguiente definición que es x, en la linea 9 lea pasa a leer lo que almacena x es decir la divicion de m/8 luego vemos que se realiza unos test en la linea 10 para ver la división por último en la linea 13 le da un espacio de memoria a esta división, entonces la declaración de esta divicion va desde la línea 8 a 13.

```
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+7]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 3
        mov     DWORD PTR [rbp-16], eax
```

3 *Ejercicio 3:*

Redacta el siguiente código, genera el código ensamblador y explica en qué parte (del código ensamblador) se define la división entre 4.

3.1 Código

```
#include <iostream>
int main(){
char* c =(char *) "abcdef";
int m = 11148;
int x = m/8;
int y = m/4;
int z = m/2;
return 0;
}
```

```
        .LC0:
        .string "abcdef"
main:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], OFFSET FLAT:.LC0
        mov     DWORD PTR [rbp-12], 11148
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+7]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 3
        mov     DWORD PTR [rbp-16], eax
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        mov     DWORD PTR [rbp-20], eax
        mov     eax, DWORD PTR [rbp-12]
        mov     edx, eax
        shr     edx, 31
        add     eax, edx
        sar     eax
        mov     DWORD PTR [rbp-24], eax
        mov     eax, 0
        pop     rbp
        ret
__static_initialization_and_destruction_0(int, int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L5
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L5
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L5:
        nop
        leave
        ret
_GLOBAL__sub_I_main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

3.2 Explicación de la parte en la que se define la división entre 4

En la etiqueta main, podemos observar que existe un comando en assembler que se utiliza para hacer divisiones de 2, 4, 8, etc. El comando se llama sar y que es un desplazamiento aritmético a la derecha, este comando nos será de gran ayuda para encontrar la división entre 4 primero antes de llegar a sar entran muchas instrucciones como lea, que nos permite tener un indice respectivo y colocar allí la información del registro, después usamos test para hacer una evaluación entre los registros y luego aparece el movs que es como el mov que se realiza si la diferencia entre los dos registros es negativa. Finalmente llegamos al sar que es una división pero nos dice que lo que se encuentra en el registro eax tiene que ser divido entre 4 porque serian 2 desplazamientos aritméticos que realizaría, finalmente almacenamos mediante un mov el resultado en el registro eax.

```
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        mov     DWORD PTR [rbp-20], eax
```

4 *Ejercicio 4:*

4.1 Código

```
#include <iostream>
int main(){
char* c =(char *) "abcdef";
int m = 11148;
int x = m/8;
int y = m/4;
int z = m/2;
return 0;
}
```

4.2 Código Ensamblador

```
.LC0:
        .string "abcdef"
main:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], OFFSET FLAT:.LC0
        mov     DWORD PTR [rbp-12], 11148
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+7]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 3
        mov     DWORD PTR [rbp-16], eax
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        mov     DWORD PTR [rbp-20], eax
        mov     eax, DWORD PTR [rbp-12]
        mov     edx, eax
        shr     edx, 31
        add     eax, edx
        sar     eax
        mov     DWORD PTR [rbp-24], eax
        mov     eax, 0
```

```
        pop     rbp
        ret
__static_initialization_and_destruction_0(int, int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L5
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L5
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L5:
        nop
        leave
        ret
_GLOBAL__sub_I_main:
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

4.3 Explicación del algoritmo

Nuevamente nos situamos en las primeras lineas de código y vemos características similares como en los dos ejercicios anteriores, al encontrarse 3 divisiones lo primero que hacemos es ubicar en que parte se encuentran estas divisiones y encontramos un mov después de la definición de c lo cual nos indica que existe otra definición que en este caso sería la de m, luego se ve la definición de la primera división, porque dentro encontramos algunas variables que nos indican que lo que se está desarrollando es un procedimiento aritmético como el caso de test y sar con esto ya tendríamos ubicadas las dos primeras divisiones y nos damos cuenta que desde la linea 20 a la 25 se expresa la división entre 2.

```
        mov     eax, DWORD PTR [rbp-12]
        mov     edx, eax
        shr     edx, 31
        add     eax, edx
        sar     eax
        mov     DWORD PTR [rbp-24], eax
```

5 *Ejercicio 5:*

Redacta el siguiente código, genera el código ensamblador y explica:

5.1 Código

```
#include <iostream>
int div4(int x){
return x/4;
}

int main(){
char* c = "abcdef";
```

```
int m = 11148;
int x = m/8;
int y = m/4;
int z = m/2;

int rpt = div4(5);

return 0;
}
```

5.2 Código Ensamblador

```
        div4(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        pop     rbp
        ret
.LC0:
        .string "abcdef"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-8], OFFSET FLAT:.LC0
        mov     DWORD PTR [rbp-12], 11148
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+7]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 3
        mov     DWORD PTR [rbp-16], eax
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        mov     DWORD PTR [rbp-20], eax
        mov     eax, DWORD PTR [rbp-12]
        mov     edx, eax
        shr     edx, 31
        add     eax, edx
        sar     eax
        mov     DWORD PTR [rbp-24], eax
        mov     edi, 5
        call    div4(int)
        mov     DWORD PTR [rbp-28], eax
        mov     eax, 0
        leave
        ret
__static_initialization_and_destruction_0(int, int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
```

```
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L7
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L7
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L7:
        nop
        leave
        ret
_GLOBAL__sub_I_div4(int):
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

5.3 En qué parte del código ensamblador se define la función div4.

Se define luego de la etiqueta div4(int), se hace un push añadiendo el registro luego se hace un mov que coloca el registro rsp en rbp, luego separamos memoria para poder utilizar después, mediante la instrucción lea podemos obtener el indice que necesitamos para identificar este registro anterior, después hacemos comparaciones y de ser el caso se realizan movs para los registros, si todo esto se puede realizar continuamos recien con la división entre 4 que nos proporciona el comando sar, que es una división entre 4 al registro eax y finalmente hacemos un pop al otro registro rbp y utilizamos ret para retornar el valor del registro.

```
div4(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        pop     rbp
        ret
```

5.4 En qué parte del código ensamblador se invoca a la función div4.

Se ubica en la etiqueta main, en la instrucción call div4(int) aquí se realiza la llamada a la función anteriormente creada.

```
        call    div4(int)
```

5.5 En qué parte del código ensamblador dentro de la función div4 se procesa la división.

En el código que se encuentra debajo de la etiqueta div4(int), en la instrucción sar eax, 2 ; realiza la división entre 4 al registro eax, después es sacado el registro rbp para finalmente retornar su valor del registro.

```
        sar     eax, 2
        pop     rbp
        ret
```

6 *Ejercicio 6:*

6.1 Código

```
#include <iostream>
int diiv(int x, int y){
return x/y;
}

int div4(int x){
return x/4;
}

int main(){
char* c = "abcdef";
int m = 11148;
int x = m/8;
int y = m/4;
int z = m/2;

int rpt = diiv(5,4);
int rpt2 = div4(5);

return 0;
}
```

6.2 Código Ensamblador

```
        diiv(int, int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     eax, DWORD PTR [rbp-4]
        cdq
        idiv    DWORD PTR [rbp-8]
        pop     rbp
        ret
div4(int):
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     eax, DWORD PTR [rbp-4]
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        pop     rbp
        ret
.LC0:
        .string "abcdef"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-8], OFFSET FLAT:.LC0
        mov     DWORD PTR [rbp-12], 11148
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+7]
```

```
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 3
        mov     DWORD PTR [rbp-16], eax
        mov     eax, DWORD PTR [rbp-12]
        lea     edx, [rax+3]
        test    eax, eax
        cmovs   eax, edx
        sar     eax, 2
        mov     DWORD PTR [rbp-20], eax
        mov     eax, DWORD PTR [rbp-12]
        mov     edx, eax
        shr     edx, 31
        add     eax, edx
        sar     eax
        mov     DWORD PTR [rbp-24], eax
        mov     esi, 4
        mov     edi, 5
        call    diiv(int, int)
        mov     DWORD PTR [rbp-28], eax
        mov     edi, 5
        call    div4(int)
        mov     DWORD PTR [rbp-32], eax
        mov     eax, 0
        leave
        ret
__static_initialization_and_destruction_0(int, int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        cmp     DWORD PTR [rbp-4], 1
        jne     .L9
        cmp     DWORD PTR [rbp-8], 65535
        jne     .L9
        mov     edi, OFFSET FLAT:_ZStL8__ioinit
        call    std::ios_base::Init::Init() [complete object constructor]
        mov     edx, OFFSET FLAT:__dso_handle
        mov     esi, OFFSET FLAT:_ZStL8__ioinit
        mov     edi, OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L9:
        nop
        leave
        ret
_GLOBAL__sub_I_diiv(int, int):
        push    rbp
        mov     rbp, rsp
        mov     esi, 65535
        mov     edi, 1
        call    __static_initialization_and_destruction_0(int, int)
        pop     rbp
        ret
```

6.3 En qué parte del código ensamblador se define la función diiv.

La primera función está definida en las primeras lineas de código en la etiqueta diiv(int, int) vemos que se definen x y y después de push asignandoles espacios de memoria para luego retornar el resultado en las lineas 6,7,8 en la que mov define el return luego idiv que realiza la división de los dos valores que serán asignados.

```
diiv(int, int):
```

```
push    rbp
mov     rbp, rsp
mov     DWORD PTR [rbp-4], edi
mov     DWORD PTR [rbp-8], esi
mov     eax, DWORD PTR [rbp-4]
cdq
idiv    DWORD PTR [rbp-8]
pop     rbp
ret
```

6.4 En qué parte del código ensamblador se invoca la función diiv.

La división es invocada en la etiqueta main en la que se realizan las operaciones para esto tenemos que ver las definiciones anteriores y dónde acaban nos encotramos con la definición de c, x, y, z, luego vemos la etiqueta call la cual esta llamando a diiv(int,int) y es esta la que invoca la división.

```
mov     esi, 4
mov     edi, 5
call    diiv(int, int)
mov     DWORD PTR [rbp-28], eax
```

6.5 En qué parte del código ensamblador dentro de la función diiv se procesa la división.

La división se procesa justo después de las declaraciones de int x e int y vemos un mov que define el return e invoca a x luego idiv que indica que tiene que ser dividido por el valor dado.

```
mov     eax, DWORD PTR [rbp-4]
cdq
idiv    DWORD PTR [rbp-8]
```

7 *Ejercicio 7:*

De las preguntas anteriores, se ha generado código por cada función, ambas dividen entre 4, pero difieren un poco en su implementación. Investigue a qué se debe dicha diferencia y comente cuáles podrían ser las consecuencias.

7.1 En qué parte del código ensamblador dentro de la función diiv se procesa la división.

Las divisiones al no siempre dar un resultado entero o al ser entre 0 tienden a requerir de más excepciones es por esto que el código llega a crecer al requerir de estas.