

Práctica 01

DOCENTE	CARRERA	CURSO
MSc. Vicente Enrique Machaca Arceda	Escuela Profesional de Ingeniería de Software	Compiladores

PRÁCTICA	TEMA	FECHA DE ENTREGA
01	Compiladores	02 de setiembre 2021

1. Competencias del curso

- Conocer las bases de la teoría de la computación para la implementación de un compilador.
- Implementar un compilador para un lenguaje de baja complejidad.

2. Competencias de la práctica

- Comprender las bases teóricas de un compilador.

3. Datos de los estudiantes

- Integrantes:
 - Camila Euridice Huamani Tito.
 - Ronald Fabricio Centeno Cardenas.

4. Ejercicios

1. Redacta el siguiente código, genera el código ensamblador y explica en que parte (del código ensamblador) se definen las variables c y m. **(2 puntos)**.

```
int main(){  
char* c = "abcdef";  
int m = 11148;  
return 0;  
}
```

Solución:

```

pushq %rbp
.seh_pushreg %rbp
movq %rsp, %rbp
.seh_setframe %rbp, 0
subq $48, %rsp
.seh_stackalloc 48
.seh_endprologue
call __main
leaq .LC0(%rip), %rax
movq %rax, -8(%rbp)
movl $11148, -12(%rbp)
movl $0, %eax
addq $48, %rsp
popq %rbp
ret
.seh_endproc
.ident "GCC: (GNU) 11.2.0"

```

Explicación:

Según la imagen dentro del recuadro se encuentra donde están siendo almacenadas las variables *c* y *m*, respectivamente, en diferentes espacios de memoria en el array creado con anterioridad.

2. Redacta el siguiente código, genera el código ensamblador y explica en qué parte (del código ensamblador) se define la división entre 8. (2 puntos).

```

int main(){
char* c = "abcdef";
int m = 11148;
int x = m/8;
return 0;
}

```

Solución:

```

.seh_endprologue
call __main
leaq .LC0(%rip), %rax
movq %rax, -8(%rbp)
movl $11148, -12(%rbp)
movl -12(%rbp), %eax
leal 7(%rax), %edx
testl %eax, %eax
cmovs %edx, %eax
sarl $3, %eax
movl %eax, -16(%rbp)
movl $0, %eax
addq $48, %rsp
popq %rbp
ret

```

Explicación:

En la parte enmarcada se define la división entre 8, donde separa espacio y realiza los procedimientos para después volverlo a guardar en la variable (en este caso en la variable x).

3. Redacta el siguiente código, genera el código ensamblador y explica en qué parte (del código ensamblador) se define la división entre 4. (2 puntos).

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

Solución:

```
call    __main  
leaq    .LC0(%rip), %rax  
movq    %rax, -8(%rbp)  
movl    $11148, -12(%rbp)  
movl    -12(%rbp), %eax  
leal    7(%rax), %edx  
testl   %eax, %eax  
cmovs   %edx, %eax  
sarl    $3, %eax  
movl    %eax, -16(%rbp)  
movl    -12(%rbp), %eax  
leal    3(%rax), %edx  
testl   %eax, %eax  
cmovs   %edx, %eax  
sarl    $2, %eax  
movl    %eax, -20(%rbp)  
movl    -12(%rbp), %eax  
movl    %eax, %edx  
shrl    $31, %edx  
addl    %edx, %eax  
sarl    %eax  
movl    %eax, -24(%rbp)  
movl    $0, %eax  
addq    $64, %rsp  
popq    %rbp  
ret
```

Explicación:

Al obtener el resultado del anterior ejercicio es rápido encontrar la división entre 4 el cual primero llama a la variable guardada m y despues realizar los procedimientos de la división y almacenar en la variable y, el cual se encuentra en la ultima linea dentro del recuadro rojo.

4. Redacta el siguiente código, genera el código ensamblador y explica en que parte (del código ensamblador) se define la división entre 2. (2 puntos).

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

Solucin:

```
call    __main
leaq    .LC0(%rip), %rax
movq    %rax, -8(%rbp)
movl    $11148, -12(%rbp)
movl    -12(%rbp), %eax
leal    7(%rax), %edx
testl   %eax, %eax
cmovs   %edx, %eax
sarl    $3, %eax
movl    %eax, -16(%rbp)
movl    -12(%rbp), %eax
leal    3(%rax), %edx
testl   %eax, %eax
cmovs   %edx, %eax
sarl    $2, %eax
movl    %eax, -20(%rbp)
movl    -12(%rbp), %eax
movl    %eax, %edx
shrl    $31, %edx
addl    %edx, %eax
sarl    %eax
movl    %eax, -24(%rbp)
movl    $0, %eax
addq    $64, %rsp
popq    %rbp
ret
```

Explicación:

Al igual que el anterior ejercicio, identificar la operación es rápida, en este caso llama a la variable `m` y realiza su operación para después guardarla en la última línea el cual es

`movl %eax, -24(%rbp)`

5. Redacta el siguiente código, genera el código ensamblador y explica: (4 puntos):

- En que parte del código ensamblador se define la función `div4`.
- En que parte del código ensamblador se invoca a la función `div4`.
- En que parte del código ensamblador dentro de la función `div4` se procesa la división.

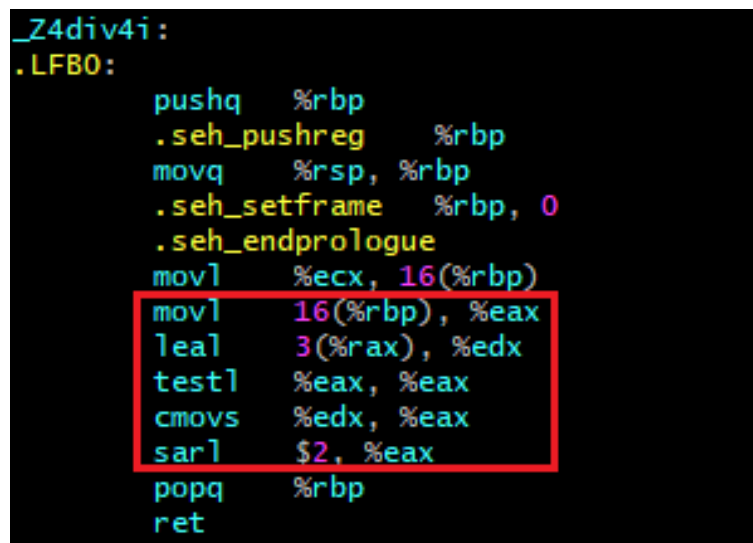
```
int div4(int x){
    return x/4;
}

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;
}
```

Solución:



```
_Z4div4i:
.LFB0:
    pushq   %rbp
    .seh_pushreg %rbp
    movq    %rsp, %rbp
    .seh_setframe %rbp, 0
    .seh_endprologue
    movl    %ecx, 16(%rbp)
    movl    16(%rbp), %eax
    leal    3(%rax), %edx
    testl   %eax, %eax
    cmovs   %edx, %eax
    sarl    $2, %eax
    popq    %rbp
    ret
```

```

shr1    $31, %edx
addl    %edx, %eax
sarl    %eax
movl    %eax, -24(%rbp)
movl    $5, %ecx
call    _Z4div4i
movl    %eax, -28(%rbp)
movl    $0, %eax
addq    $64, %rsp
popq    %rbp
ret

```

Explicación:

Para el primer y último punto de este ejercicio se puede observar en la primera imagen, ya que en esta se define la función *div4* y la operación de dividir entre 4, que se encuentra enmarcada en el recuadro, y para el segundo punto está la segunda imagen, la cual guarda el número en este caso 5 y después con la palabra **call** invoca a la función *div4* para realizar el procedimiento y guardar el resultado en la variable *rpt*.

6. Redacta el siguiente código, genera el código ensamblador y explica: **(4 puntos)**:

- En que parte del código ensamblador se define la función *div*.
- En que parte del código ensamblador se invoca a la función *div*.
- En que parte del código ensamblador dentro de la función *div* se procesa la división.

```

int div(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 = div(5,4);
    int rpt2 = div4(5);

```

```

    return 0;
}

```

Solución y explicación:

- Al definir la función *div* en ensamblador como podemos ver en la primera imagen se utiliza la reserva de `.globl` y `.def` junto a **Z3divii**, de esta forma se da el flujo de la definición de la función *div* e indicando el salto que debe realizar por medio `.seh_proc Z3divii`, para que se realice el procedimiento requerido.

```

1      .file "ejer6.cpp"
2      .text
3      .globl _Z3divii
4      .def _Z3divii; .scl 2; .type 32; .endef
5      .seh_proc _Z3divii
6      _Z3divii:

```

- Para invocar la función en lenguaje ensamblador verificamos la palabra **call**, donde nos indica el llamado al proceso que la función con los parametros antes propuestos por medio de **movl**.

```

78      movl    %eax, -24(%rbp)
79      movl    $4, %edx
80      movl    $5, %ecx
81      call    _Z3divii

```

- Para procesar la división dentro de la función *div*, observamos que el lenguaje ensamblador dentro del proceso que debe correr dentro de **Z3divii**, guarda las variables *X* y *Y* para luego procesar la operación indicada en la función.

```

12      .seh_endprologue
13      movl    %ecx, 16(%rbp)
14      movl    %edx, 24(%rbp)
15      movl    16(%rbp), %eax
16      cltd
17      idivl   24(%rbp)
18      popq    %rbp
19      ret

```

- De las preguntas anteriores, se ha generado código por cada función, ambas dividen entre 4, pero difieren un poco en su implementación. Investigue a que se debe dicha diferencia y comente cuales podran ser las consecuencias. (4 puntos)

Explicación:

La principal diferencia entre la implementación de estas dos funciones es primero el número de parametros que utilizan, siendo en el primer caso la función *div*, aquí se da la utilización de dos variables de tipo **int** como parametros externos, que luego al dividirse retornan un resultado que depende del tipo de función recreada.

En el segundo caso la función *div4* mantiene el uso de un parametro externo de tipo *int* que luego será dividido con el número 4, retornando un valor del mismo tipo de dato.

La consecuecna más proxima al usar dos tipos de datos externos es que el resultado no muchas veces sera el deseado, ya que a falta de limites que propongan condicionales para salvaguardar el valor retornado, este podria simplemente no ser compilado, a diferencia de una función que solo utilice un parametro para funcionar, ella por si sola compilará el resultado en base al proceso indicado para dicha variable ingresada como parametro en la función.

En resumen, la utilización de dos parametros no controlados podria simplemente no ser adecuada para el proceso de utilización de una función, mientras que la forma estática del proceso graduado por una constante, siempre devolvera un valor del mismo tipo declarado en la función, resguardando de esta forma el retorno del valor deseado pero quitando dinamismo a la operación.

5. Enlace del repositorio

Enlace repositorio GitHub:

<https://github.com/rcentenoc/PRACTICA01.git>