



# Universidad Nacional de Córdoba

FACULTAD DE CIENCIAS EXACTAS, FÍSICAS Y NATURALES

## CÓDIGO INTERMEDIO

*Trabajo integrador final*

*Practica y construccion de compiladores*

Autores:

Julián González



## **Resumen**

Este documento explicara los codigos intermedios usados por los compiladores GCC y CLANG/LLVM, como asi comparar sus principales características viendo ventajas y desventajas entre ellos.

# Índice general

<b>1. Introducción</b>	<b>1</b>
<b>2. Código intermedio de GCC</b>	<b>2</b>
I.    Analizador léxico y sintáctico . . . . .	2
II.   Analizador semántico . . . . .	3
III. <i>Generic</i> . . . . .	3
IV. <i>Gimple</i> . . . . .	3
V. <i>Register Transfer Language</i> . . . . .	3
VI. <i>Assembler</i> . . . . .	4
<b>3. Código intermedio de CLANG/LLVM</b>	<b>5</b>
I.    Analizador léxico y sintáctico . . . . .	5
II.   Analizador semántico . . . . .	6
III.  LLVM IR . . . . .	6
IV.  Selección de instrucciones . . . . .	6
V.    Asignación de registros . . . . .	6
VI.  Programación de la instrucción . . . . .	6
VII. <i>Assembler</i> . . . . .	6
<b>4. Comparación entre GCC y CLANG/LLVM</b>	<b>7</b>
I.    Analizador léxico y sintáctico . . . . .	7
II.   Gimple vs LLVM IR . . . . .	8
III.  RTL vs MachineInstr . . . . .	9
IV.  Assembler . . . . .	9
<b>5. Conclusión</b>	<b>11</b>

# Índice de figuras

2.1. Proceso de compilación de GCC. . . . .	2
3.1. Proceso de compilación de CLANG/LLVM. . . . .	5

# Listings

2.1. Comando de compilación para obtener codigo-ejemplo.c.005t.original [1] para GCC. . . . .	2
2.2. Comando de compilación para obtener codigo-ejemplo.c.006t.gimple [1] para GCC. . . . .	3
2.3. Comando de compilación del archivo codigo-ejemplo.c [1] para GCC. . . . .	3
2.4. Comando de compilación para obtener codigo-ejemplo.c.330r.final [1] para GCC. . . . .	4
2.5. Comando de compilación para obtener codigo-ejemplo.s [1] para GCC. . . . .	4
3.1. Comando de compilación para obtener <i>tokens</i> [1] para CLANG/LLVM. . . . .	6
3.2. Comando de compilación para obtener <i>ast</i> [1] para CLANG/LLVM. . . . .	6
3.3. Comando de compilación para obtener <i>llvm-ir.ll</i> [1] para CLANG/LLVM. . . . .	6
3.4. Comando de compilación para obtener <i>llvm-ir-optimizedd.ll</i> [1] para CLANG/LLVM. . . . .	6
3.5. Comando de compilación para obtener <i>machineinstrs</i> [1] para CLANG/LLVM. . . . .	6
3.6. Comando de compilación para obtener <i>assembly.s</i> [1] para CLANG/LLVM. . . . .	6
4.1. Fragmento del código fuente del archivo codigo-ejemplo.c. [1] . . . . .	7
4.2. Fragmento del árbol de GCC del archivo codigo-ejemplo.c.005t.original. [1] . . . . .	7
4.3. Fragmento del árbol de CLANG/LLVM del archivo <i>ast</i> . [1] . . . . .	7
4.4. Fragmento del código fuente del archivo codigo-ejemplo.c. [1] . . . . .	8
4.5. Fragmento del <i>Gimple</i> de GCC del archivo codigo-ejemplo.c.006t.gimple. [1] . . . . .	8
4.6. Fragmento del LLVM IR de CLANG/LLVM del archivo <i>llvm-ir.ll</i> . [1] . . . . .	8
4.7. Fragmento del código fuente del archivo codigo-ejemplo.c. [1] . . . . .	9
4.8. Fragmento de RTL de GCC del archivo codigo-ejemplo.c.330r.final. [1] . . . . .	9
4.9. Fragmento de MachineInstr de CLANG/LLVM del archivo <i>machineinstrs</i> . [1] . . . . .	9
4.10. Fragmento del código fuente del archivo codigo-ejemplo.c. [1] . . . . .	9
4.11. Fragmento del código <i>assembler</i> de GCC del archivo codigo-ejemplo.s. [1] . . . . .	9
4.12. Fragmento del código <i>assembler</i> de CLANG/LLVM del archivo <i>assembly.s</i> . [1] . . . . .	10

# Capítulo 1

## Introducción

El código intermedio es un código interno usado por el compilador para representar el código fuente. El código intermedio está diseñado para llevar a cabo el procesamiento del código fuente, como es la optimización y la traducción a código máquina.

Una de las características más esenciales del código intermedio es ser independiente del *hardware*. Por lo tanto, permite la portabilidad entre distintos sistemas.

Otra propiedad importante de todo código intermedio es su fácil generación a partir del código fuente, como así también su fácil traducción al código máquina para la arquitectura deseada.

No existe un único código intermedio, sino que hay distintos tipos y categorías, variando de compilador en compilador. Aunque un mismo compilador puede utilizar varios tipos de código intermedio en el proceso [2].

A continuación, se presentan los códigos intermedios utilizados por los compiladores GCC y CLANG/LLVM, especificando las características de cada uno y comparando sus prestaciones posteriormente.

## Capítulo 2

# Código intermedio de GCC

A continuación se exponen los distintos códigos intermedios que GCC utiliza en la compilación. Los distintos códigos intermedios están relacionados en la forma que la salida de cada uno es la entrada del siguiente, avanzando desde una representación general de alto nivel hacia una específica de bajo nivel.

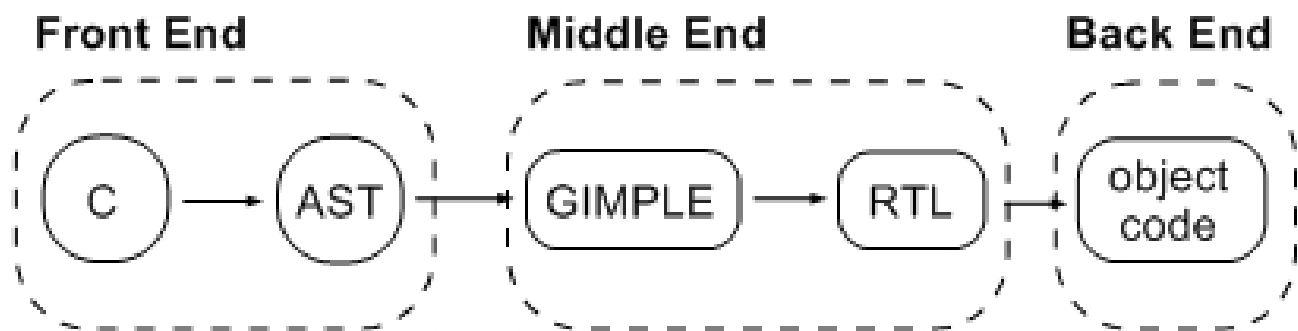


Figura 2.1: Proceso de compilación de GCC.

## I Analizador léxico y sintáctico

El analizador léxico lee la secuencia de caracteres desde la salida del preprocesador y agrupa los caracteres en secuencias llamadas lexemas. Por cada lexema, el analizador genera un token con la manera:

*[token name, attribute value]*

Donde *token name* son símbolos abstractos usados durante el análisis sintáctico, y *attribute value* es un puntero a una entrada en la tabla de símbolos. GCC no permite obtener los *tokens* válidos en forma de texto. Es un archivo interno. El analizador sintáctico chequea si la gramática del lenguaje acepta la secuencia de *tokens* generados por el analizador léxico, si no reporta errores de sintaxis. Además, el analizador sintáctico utiliza el primer componente de cada token para crear una representación en manera de árbol que muestre la estructura de los *tokens*, es decir, un árbol sintáctico [3]. Con el *flag* `-fdump-tree-original-raw` se obtiene la representación textual del árbol abstracto sintáctico.

```
1 $ gcc -fdump-tree-original-raw codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.1: Comando de compilación para obtener `codigo-ejemplo.c.005t.original` [1] para GCC.

## II Analizador semántico

El analizador semántico utiliza el árbol sintáctico y la información de la tabla de símbolos para revisar la consistencia semántica del programa fuente con respecto a la definición del lenguaje.

## III *Generic*

*Generic* es un código intermedio independiente del lenguaje con estructura de árbol que es generado por el *front end*. *Generic* es capaz de representar todos los lenguajes admitidos por GCC. *Generic* se produce eliminando construcciones específicas del lenguaje del árbol de parseo. GCC pasa del árbol abstracto sintáctico a la representación en *Gimple* en lenguajes como C.

## IV *Gimple*

*Gimple* es un código intermedio de tres direcciones resultante de desglosar *Generic* en tuplas de no más de tres operandos, a través de la herramienta interna de GCC llamada *Gimplifier*. *Gimple* introduce variables temporales para poder computar expresiones complejas y permite supervisar el flujo de control a nivel inferior con sentencias secuenciales y saltos incondicionales. *Gimple* es el código intermedio principal de GCC (los lenguajes C y C++ se convierten a *Gimple* sin pasar por *Generic*), además de ser conveniente para optimizar.

Existen tres tipos de *Gimple*:

- *Gimple* de alto nivel que es lo que se obtiene después de desglosar el *Generic*.
- *Gimple* de bajo nivel que se obtiene al linealizar todas las estructuras de flujo de control de del *Gimple* de alto nivel, incluidas las funciones anidadas, el manejo de excepciones y los bucles.
- *Gimple* SSA es el *Gimple* de bajo nivel reescrito en la forma SSA [4].

Con el *flag* `-fdump-tree-gimple` se obtiene la representación en la forma de *Gimple*.

```
1 $ gcc -fdump-tree-gimple codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.2: Comando de compilación para obtener `codigo-ejemplo.c.006t.gimple` [1] para GCC.

Además, con el *flag* `-fdump-tree-all-graph` GCC genera muchos archivos con la extensión `.cfg` los cuales pueden visualizarse con una herramienta online Graphviz [5]. Esta herramienta permite ver la evolución del código en las distintas pasadas de una manera mucho más conveniente para el usuario.

```
1 $ gcc -fdump-tree-all-graph codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.3: Comando de compilación del archivo `codigo-ejemplo.c` [1] para GCC.

## V *Register Transfer Language*

*Register Transfer Language* es un código intermedio de bajo nivel semejante al lenguaje ensamblador. La mayor parte del trabajo del compilador se realiza en *Register Transfer Language*. Tiene una forma interna, formada por estructuras que apuntan a otras estructuras, y una forma textual que se utiliza en la descripción de la máquina y en los volcados de depuración impresos. El formulario textual usa paréntesis anidados para indicar los punteros en el formulario interno. Con el *flag* `-fdump-rtl-final` se obtiene la representación en la forma de *Register Transfer Language* ya optimizado por el compilador.



```
1 $ gcc -fdump-rtl-final codigo-ejemplo.c -o codigo-ejemplo
```

Listing 2.4: Comando de compilación para obtener codigo-ejemplo.c.330r.final [1] para GCC.

## VI *Assembler*

Por último, es posible obtener la salida en *Assembler* con el *flag* -S.

```
1 $ gcc -S codigo-ejemplo.c -o codigo-ejemplo.s
```

Listing 2.5: Comando de compilación para obtener codigo-ejemplo.s [1] para GCC.

## Capítulo 3

# Código intermedio de CLANG/LLVM

Clang es el *frontend* de LLVM para la familia de los lenguajes de C. Clang consiste en un preprocesador C, un analizador léxico, un analizador sintáctico, un analizador semántico y un generador IR. El Optimizador analiza la IR y la traduce a una forma más eficiente. *opt* es la herramienta de optimización de LLVM. El *Backend* genera código máquina al mapear la IR al conjunto de instrucciones del hardware de destino. La herramienta backend de LLVM es *llc*. Genera código máquina a partir del IR de LLVM en tres fases: selección de instrucciones, asignación de registros y programación de la instrucción.

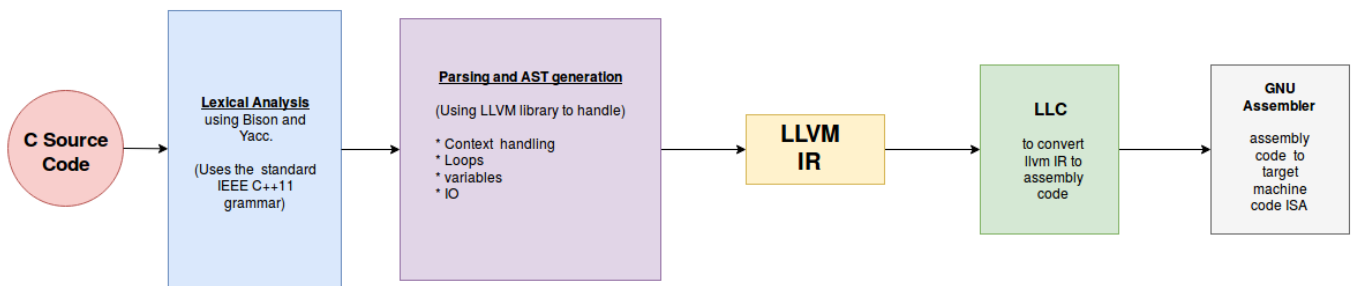


Figura 3.1: Proceso de compilación de CLANG/LLVM.

### I Analizador léxico y sintáctico

El analizador léxico lee la salida del preprocesador y realiza su tokenización. El analizador sintáctico chequea si la gramática del lenguaje acepta la secuencia de *tokens* generados por el analizador léxico y reporta errores de sintaxis. El analizador sintáctico produce el *error recovery*, si es posible. La tokenización se organiza de la siguiente manera:

- La definición del tipo de token en cuestión.
- El *token*.
- *Flag* que describe un adicional del *token*.
- La ubicación del token en el código fuente.

Después de analizar la gramática de la secuencia de *tokens*, emite un árbol abstracto sintáctico. Los nodos de un árbol abstracto sintáctico representan declaraciones, sentencias y tipos [6].

```
1 $ clang -fsyntax-only -Xclang -dump-tokens codigo-ejemplo.c 2>&1 | tee tokens
```

Listing 3.1: Comando de compilación para obtener *tokens* [1] para CLANG/LLVM.

```
1 $ clang -fsyntax-only -Xclang -ast-dump codigo-ejemplo.c -fno-color-diagnostics > ast
```

Listing 3.2: Comando de compilación para obtener *ast* [1] para CLANG/LLVM.

## II Analizador semántico

El analizador semántico recorre el árbol abstracto sintáctico, determinando si las sentencias del código tienen un significado válido. Esta fase comprueba los errores de tipo.

## III LLVM IR

El generador IR traduce el árbol abstracto sintáctico a código intermedio [7].

```
1 $ clang -S -emit-llvm codigo-ejemplo.c -o llvm-ir.ll
```

Listing 3.3: Comando de compilación para obtener *llvm-ir.ll* [1] para CLANG/LLVM.

En esta fase también trabaja el optimizador que mejora la eficiencia del código basándose en su comprensión del comportamiento en tiempo de ejecución del programa.

```
1 $ opt -O2 -S llvm-ir.ll -o llvm-ir-optimized.ll
```

Listing 3.4: Comando de compilación para obtener *llvm-ir-optimizedd.ll* [1] para CLANG/LLVM.

## IV Selección de instrucciones

La selección de instrucciones es el mapeo de instrucciones IR al conjunto de instrucciones de la máquina de destino. Este paso utiliza un espacio de nombres infinito de registros virtuales.

## V Asignación de registros

La asignación de registros es el mapeo de registros virtuales a registros reales en la arquitectura de destino.

## VI Programación de la instrucción

La programación de la instrucción es la reordenación de operaciones para reflejar las restricciones de funcionamiento de la máquina de destino [8].

```
1 $ llc llvm-ir.ll -print-machineinstrs 2>&1 tee machineinstrs
```

Listing 3.5: Comando de compilación para obtener *machineinstrs* [1] para CLANG/LLVM.

## VII Assembler

Por último, es posible obtener la salida en lenguaje *assembler* del código fuente.

```
1 $ llc llvm-ir.ll -o assembly.s
```

Listing 3.6: Comando de compilación para obtener *assembly.s* [1] para CLANG/LLVM.

## Capítulo 4

# Comparación entre GCC y CLANG/LLVM

A partir de un mismo código fuente en lenguaje C, se procederá a la compilación empleando tanto GCC como CLANG/LLVM. Luego, se llevará a cabo una comparación y breve descripción de lo observado. Se presentan los códigos relevantes que coinciden con las etapas nombradas anteriormente.

### I Analizador léxico y sintáctico

GCC no permite obtener la salida del analizador léxico, por lo que se analizará el árbol abstracto sintáctico.

GCC describe el árbol relacionando sus nodos a partir de punteros con la concatenando "@z un número. Entonces, es posible armar el árbol siguiendo estos punteros. En este fragmento se ve como la declaración de la variable `.a`, que es del tipo entera con un valor inicial de "20", es traducida al árbol.

```
1 int a = 20;
```

Listing 4.1: Fragmento del código fuente del archivo `codigo-ejemplo.c`. [1]

Se observa en @5 la palabra clave `var_decl`, la cual indica la declaración de una variable. En este caso, el campo `name` apunta a @9, que es un nodo identificador con el nombre de la variable `.a`. También, se puede obtener el tipo de la variable refiriéndose a @10, que muestra `integer_type`, y lo mismo para `init` se refiere al valor inicial apuntado por @12 que muestra que es el valor 20.

```
1 @5 var_decl name: @9 type: @10 scope: @11 srcp: codigo-ejemplo.c:5
  ↳ init: @12
2 @9 identifier_node strg: a lngt: 1
3 @10 integer_type name: @24 size: @13 align: 32 prec: 32 sign:
  ↳ signed min: @25 max: @26
4 @11 function_decl name: @27 type: @28 srcp: codigo-ejemplo.c:3
5 @12 integer_cst type: @10 int: 20
```

Listing 4.2: Fragmento del árbol de GCC del archivo `codigo-ejemplo.c.005t.original`. [1]

El árbol construido por CLANG/LLVM se muestra de una forma más gráfica con barras e indentación. Se observa una `FunctionDecl`, la cual hace referencia al `main()` y adentro de esta una `DeclStmt`, que se encuentra en la línea 5 del código fuente. El nodo `VarDecl` define la existencia una variable llamada `a` de tipo `int`, con un valor de 20.

```
1 |-FunctionDecl 0x207e190 <line:3:1, line:17:1> line:3:5 main 'int ()'
  | '-CompoundStmt 0x207ea60 <col:12, line:17:1>
  | | -DeclStmt 0x207e2d0 <line:5:5, col:15>
  | | | '-VarDecl 0x207e248 <col:5, col:13> col:9 used a 'int' cinit
  | | | | '-IntegerLiteral 0x207e2b0 <col:13> 'int' 20
```

Listing 4.3: Fragmento del árbol de CLANG/LLVM del archivo `ast`. [1]

## II Gimple vs LLVM IR

En esta etapa se comparará como el bucle del código fuente se desarrolla en el código intermedio de ambos compiladores.

```

1  for (int i = 0; i < c; i++)
2  {
3      arr[i] = suma(a, b);
4  }
```

Listing 4.4: Fragmento del código fuente del archivo codigo-ejemplo.c. [1]

GCC emplea *Gimple* para esta etapa que tiene una familiaridad con el lenguaje C, en el cual se ve que se crean etiquetas para realizar el control del salto y el bucle se transforma en sentencias *if else*. También, se generan variables temporales para alojar resultados, como por ejemplo "\_1" para el resultado que devuelve la función suma.

```

1  {
2      int i;
3      i = 0;
4      goto <D.1955>;
5      <D.1954>:
6      _1 = suma (a, b);
7      arr[i] = _1;
8      i = i + 1;
9      <D.1955>:
10     if (i < c) goto <D.1954>; else goto <D.1952>;
11     <D.1952>:
12 }
```

Listing 4.5: Fragmento del *Gimple* de GCC del archivo codigo-ejemplo.c.006t.gimple. [1]

Por otro lado, se tiene LLVM IR, el cual ya tiene un parecido con el lenguaje *assembler* más que con el lenguaje C. Se utilizan variables temporales para los resultados de las instrucciones sin repetir concatenando "%z" un número. La etiqueta 10 hace referencia al control del salto que en la última instrucción decide si saltar a la etiqueta 14 o 24. En la etiqueta 14 se cargan las variables %2 y %3 que hacen referencia a las variables a y b para realizar la llamada a la función suma. Después, se guarda el resultado en el *array*.

```

1 10:                                     ; preds = %21, %0
2  %11 = load i32, i32* %6, align 4
3  %12 = load i32, i32* %4, align 4
4  %13 = icmp slt i32 %11, %12
5  br i1 %13, label %14, label %24
6
7 14:                                     ; preds = %10
8  %15 = load i32, i32* %2, align 4
9  %16 = load i32, i32* %3, align 4
10 %17 = call i32 @suma(i32 %15, i32 %16)
11 %18 = load i32, i32* %6, align 4
12 %19 = sext i32 %18 to i64
13 %20 = getelementptr inbounds [10 x i32], [10 x i32]* %5, i64 0, i64 %19
14 store i32 %17, i32* %20, align 4
15 br label %21
16
17 21:                                     ; preds = %14
18 %22 = load i32, i32* %6, align 4
19 %23 = add nsw i32 %22, 1
20 store i32 %23, i32* %6, align 4
21 br label %10
```

Listing 4.6: Fragmento del LLVM IR de CLANG/LLVM del archivo llvm-ir.ll. [1]

### III RTL vs MachineInstr

En esta etapa ambos códigos son poco legibles para humanos por lo que su descripción no es sencilla. Se muestra como la declaración de las variables a y b son expresadas por cada compilador.

```
1 int a = 20;
2 int b = 30;
```

Listing 4.7: Fragmento del código fuente del archivo codigo-ejemplo.c. [1]

```
1 (insn 6 3 7 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
2 (const_int -60 [0xffffffffffffffc4])) [2 a+0 S4 A32])
3 (const_int 20 [0x14])) "codigo-ejemplo.c":5:9 75 {*movsi_internal}
4 (nil))
5 (insn 7 6 8 2 (set (mem/c:SI (plus:DI (reg/f:DI 6 bp)
6 (const_int -56 [0xffffffffffffffc8])) [2 b+0 S4 A64])
7 (const_int 30 [0x1e])) "codigo-ejemplo.c":6:9 75 {*movsi_internal}
8 (nil))
```

Listing 4.8: Fragmento de RTL de GCC del archivo codigo-ejemplo.c.330r.final. [1]

```
1 MOV32mi %stack.1, 1, $noreg, 0, $noreg, 20 :: (store 4 into %ir.2)
2 MOV32mi %stack.2, 1, $noreg, 0, $noreg, 30 :: (store 4 into %ir.3)
```

Listing 4.9: Fragmento de MachineInstr de CLANG/LLVM del archivo machineinstrs. [1]

### IV Assembler

Por último, la salida en código *assembler* de ambos compiladores es prácticamente la misma. A continuación, se expone la función suma expresada en código *assembler* destacando la similitud entre ambos compiladores aunque los códigos intermedios sean notoriamente diferentes en el proceso.

```
1 int suma (int a, int b) {
2     return a + b;
3 }
```

Listing 4.10: Fragmento del código fuente del archivo codigo-ejemplo.c. [1]

```
1 .globl suma
2 .type suma, @function
3 suma:
4 .LFB1:
5 .cfi_startproc
6 endbr64
7 pushq %rbp
8 .cfi_def_cfa_offset 16
9 .cfi_offset 6, -16
10 movq %rsp, %rbp
11 .cfi_def_cfa_register 6
12 movl %edi, -4(%rbp)
13 movl %esi, -8(%rbp)
14 movl -4(%rbp), %edx
15 movl -8(%rbp), %eax
16 addl %edx, %eax
17 popq %rbp
18 .cfi_def_cfa 7, 8
19 ret
```

```
20 .cfi_endproc
21 .LFE1:
22 .size suma, .-suma
```

Listing 4.11: Fragmento del código *assembler* de GCC del archivo codigo-ejemplo.s. [1]

```
1 .globl suma # -- Begin function suma
2 .p2align 4, 0x90
3 .type suma,@function
4 suma: # @suma
5 .cfi_startproc
6 # %bb.0:
7 pushq %rbp
8 .cfi_def_cfa_offset 16
9 .cfi_offset %rbp, -16
10 movq %rsp, %rbp
11 .cfi_def_cfa_register %rbp
12 movl %edi, -8(%rbp)
13 movl %esi, -4(%rbp)
14 movl -8(%rbp), %eax
15 addl -4(%rbp), %eax
16 popq %rbp
17 .cfi_def_cfa %rsp, 8
18 retq
19 .Lfunc_end1:
20 .size suma, .Lfunc_end1-suma
21 .cfi_endproc
22 # -- End function
```

Listing 4.12: Fragmento del código *assembler* de CLANG/LLVM del archivo assembly.s. [1]

## Capítulo 5

# Conclusión

Es posible observar que la compilación empleando GCC como CLANG/LLVM respeta las fases estándar de todo compilador moderno. Las diferencias que se encuentran residen en los formatos de código interno que utilizan, en especial, código intermedio de *middle-end*: GIMPLE, para GCC, y LLVM IR, para CLANG/LLVM. A su vez, podemos observar que el código en *assembler* final es el mismo en esencia, independiente del compilador usado. Dichas diferencias residen en el nombre de variables y otros *tags* así como las direcciones de cada elemento (distintas por obvias razones). CLANG/LLVM puede resultar más amigable para el estudiante que desea comprender el mundo de los compiladores gracias a sus herramientas y su versatilidad.



# Bibliografía

- [1] J. Gonzalez, “Repositorio de trabajo.” [Online]. Available: <https://github.com/jukkanghost2/compiladores-optativa>
- [2] “Crafting interpreters.” [Online]. Available: <http://www.craftinginterpreters.com/contents.html>
- [3] “Gcc ast.” [Online]. Available: <http://icps.u-strasbg.fr/~pop/gcc-ast.html>
- [4] “Gcc intermediate.” [Online]. Available: <https://blog.aeste.my/2015/07/27/exploring-the-intermediate-representation-of-gcc/>
- [5] “Graphviz online.” [Online]. Available: <https://dreampuf.github.io/GraphvizOnline>
- [6] “Clang ast.” [Online]. Available: <https://clang.llvm.org/docs/IntroductionToTheClangAST.html>
- [7] “Llvm ir.” [Online]. Available: <https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/a-quick-primer/index.html>
- [8] “Llvm docs.” [Online]. Available: <https://llvm.org/docs/CodeGenerator.html>