



# ***Universidad Nacional de Córdoba***

*Facultad de Ciencias Exactas, Físicas y Naturales*

*Sistemas de computación*

**TP2**

Grupo:

*Epsilon*

Profesores:

*Jorge, Javier Alejandro*

*Lamberti, Germán Andrés*

*Solinas, Miguel Ángel*

Alumnos:

*Campos, Mariano*

*Erlicher, Ezequiel*

*González, Damián Marcelo*

## Primera Iteración

Link en el repo: <https://github.com/EzeErlicher/SDC-TP2-Epsilon/tree/1.0.0>

Para esta instancia se desarrolló la capa de alto nivel en Ruby, la cual permite obtener los datos del índice GINI para Argentina, a través de la API del Worldbank. Se realiza la búsqueda del dato más actualizado.

<http://api.worldbank.org/v2/en/country/AR/indicator/SI.POV.GINI?format=json>.

```
1  require 'net/http'
2  require 'json'
3
4  require_relative 'native_binding'
5
6  URL = 'http://api.worldbank.org/v2/en/country/AR/indicator/SI.POV.GINI?format=json'.freeze
7
8  # @return [Hash]
9  def get_gini_rates_for_ar
10   JSON.parse(Net::HTTP.get(URI(URL)))
11 end
12
13 # It's ensured that at least one year from the pile of data has a GINI index.
14 # @return [Array(String, Float)] year, and GINI index respectively
15 def get_latest_useful_value(json)
16   # seek first entry with useful data
17   json.last.each do |entry|
18     if(gini_index = entry['value'])
19       return [entry['date'], gini_index]
20     end
21   end
22 end
23
24 =begin
25 RUNTIME
26 =end
27
28 puts("Looking for latest GINI index entry for Argentina...")
29 year, gini_index = get_latest_useful_value(get_gini_rates_for_ar())
30 puts("Found! #{year}: #{gini_index}")
31 # handle data to next lower layer
32 puts("Passing #{gini_index} float value to lower level computing, expecting #{gini_index.to_i + 1}...")
33 modified_gini_index = NativeBinding.float_to_int(gini_index)
34 puts("Result: #{modified_gini_index}.")
```

Figura 1: Archivo *higher\_layer.rb*; runtime

### Explicación:

- **require 'net/http'** y **require 'json'**: Requieren libraries core de ruby, que luego se utilizan para realizar una solicitud HTTP y analizar la respuesta JSON.
- **require\_relative 'native\_binding'**: Carga el módulo Ruby que define el método `NativeBinding.float_to_int()`.
- **get\_gini\_rates\_for\_ar()**: Envía una solicitud HTTP GET a la URL y parsea la respuesta JSON hacia una estructura hash/array de Ruby.

- **get\_latest\_useful\_value(json)**: Busca dentro del JSON obtenido en anterior function call, el valor mas actualizado de GINI, correspondiente a Argentina, y lo devuelve.
- **modified\_gini\_index = NativeBinding.float\_to\_int(gini\_index)**: Llama a la función de la capa inferior en C float\_to\_int(float value) para cambiar el número flotante a entero y sumarle 1.

```

1  require 'ffi'
2
3  module NativeBinding
4
5      SHARED_LIB_PATH = "#{File.dirname(__FILE__)}/libmidlayer.so".freeze
6
7      extend FFI::Library
8      ffi_lib SHARED_LIB_PATH
9
10     attach_function :float_to_int, [:float], :int
11 end
12

```

Figura 2: Módulo NativeBinding en native\_binding.rb

#### Explicación:

- Permite a Ruby llamar a la función nativa de C directamente mediante **NativeBinding.float\_to\_int()**.
  - Importa la librería **ffi** para interactuar con código nativo (C).
  - Se carga la librería compartida llamada **libmidlayer.so**.
  - Importa la función C llamada **float\_to\_int()**.

```

1  #include <stdio.h>
2
3
4  int float_to_int(float value){
5
6      int casted_value = (int)value;
7
8      casted_value++;
9
10     return casted_value;
11
12 };

```

Figura 3: Código mid\_layer.c

En la siguiente captura, se muestra la compilación de la librería compartida y la ejecución del archivo **higher\_layer.rb**, mostrando la correcta vinculación con la función en C.

```
mariano-campos@mariano-campos-HP-Laptop-14-dk1xxx:~/Escritorio/SdC/Lab2/tem/SDC-TP2-Epsilon-1.0.0$ gcc -fPIC -shared mid_layer.c -o libmidlayer.so
mariano-campos@mariano-campos-HP-Laptop-14-dk1xxx:~/Escritorio/SdC/Lab2/tem/SDC-TP2-Epsilon-1.0.0$ ruby higher_layer.rb
Looking for latest GINI index entry for Argentina...
Found! 2022: 40.7
Passing 40.7 float value to lower level computing, expecting 41...
Result: 41.
mariano-campos@mariano-campos-HP-Laptop-14-dk1xxx:~/Escritorio/SdC/Lab2/tem/SDC-TP2-Epsilon-1.0.0$
```

*Figura 4: Ejecución en la consola*

## Segunda iteración

Link en el repo: <https://github.com/EzeErlicher/SDC-TP2-Epsilon/tree/master>

**Aclaración:** es esta segunda iteración a la cual se le hizo un merge a master

Para esta segunda iteración, las operaciones que realizaba la función en C ahora son llevada a cabo por un archivo en Assembly llamado **float\_to\_int.asm**. La función en C ahora simplemente se encarga de invocar a la rutina en assembly y devolver el resultado a la capa de alto nivel (Ruby).

```
1  #include <stdio.h>
2
3
4  // Nombre de la rutina en assembly
5  extern int float_to_int(float value);
6
7
8
9  int call_assembly_routine(float value){
10     int result = float_to_int(value);
11     return result;
12 }
```

*Figura 5: Nuevo mid\_layer.c*

En la siguiente imagen se muestra el programa implementado en assembler, la funcionalidad del código se explica más adelante en el debugging con gdb:

```

1  global float_to_int
2
3  section .text
4  ; int float_to_int(float f);
5  ; Recibe el float en xmm0 (por convención del ABI)
6  ; Retorna el entero en eax
7  float_to_int:
8      ; Convertir float a entero truncado (xmm0 -> eax)
9      cvttss2si eax, xmm0
10
11     ; Sumar 1 al entero
12     add eax, 1
13
14     ; Retornar (resultado está en eax)
15     ret
16
17
18

```

*Figura 6: Archivo en Assembly*

Finalmente, en la siguiente imagen, se puede ver que se obtiene el mismo resultado de la primera iteración

```

ezerlich@ezerlich-Lenovo-V330-15IKB:~/Documents/TPS-SDC-2025/SDC-TP2-Epsilon$ git branch -a
1.0.0
* 2.0.0
master
remotes/origin/1.0.0
remotes/origin/2.0.0
remotes/origin/HEAD -> origin/master
remotes/origin/master
ezerlich@ezerlich-Lenovo-V330-15IKB:~/Documents/TPS-SDC-2025/SDC-TP2-Epsilon$ ruby higher_layer.rb
Looking for latest GINI index entry for Argentina...
Found! 2022: 40.7
Passing 40.7 float value to lower level computing, expecting 41...
Result: 41.

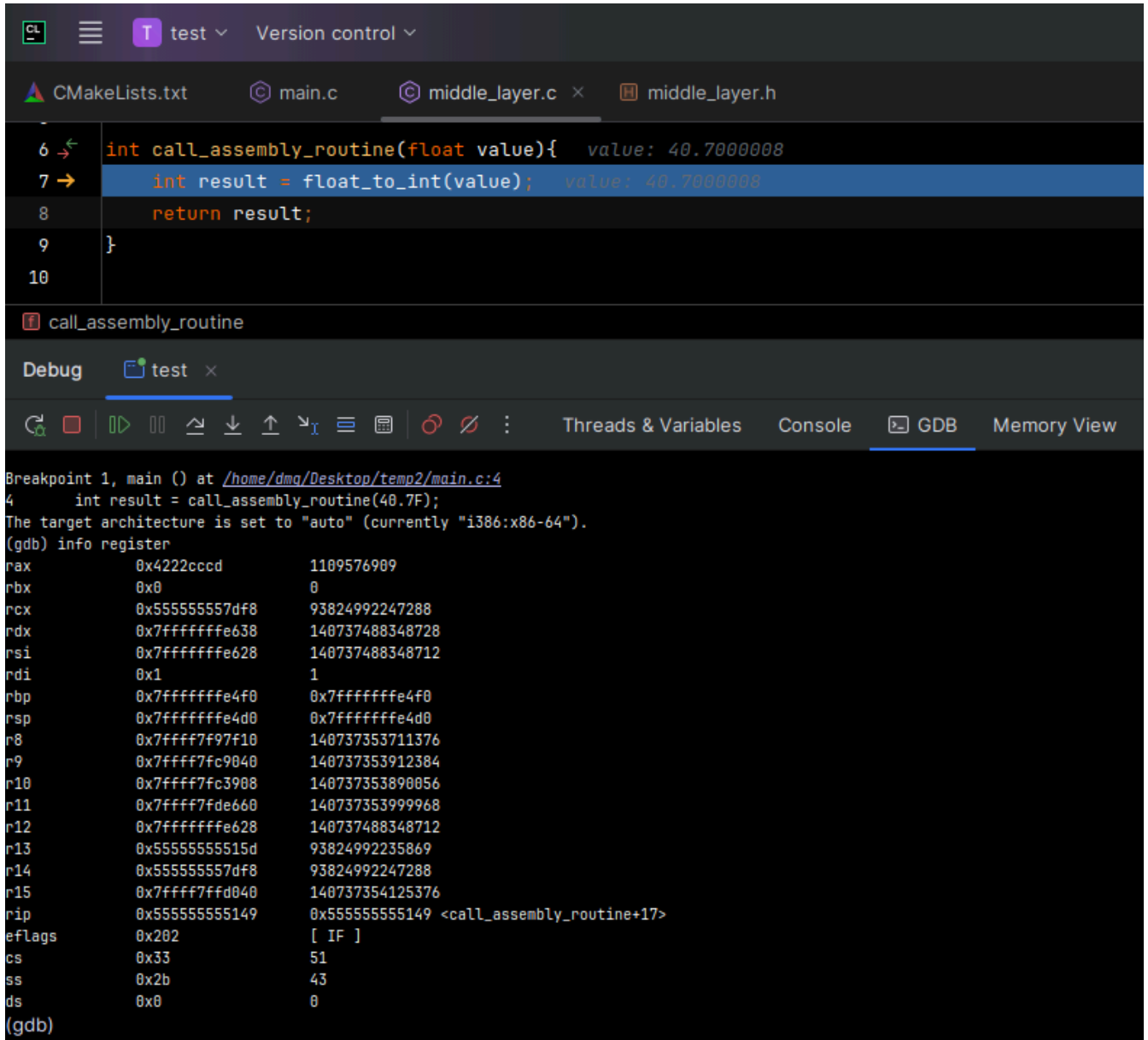
```

*Figura 7: Ejecución del stack Ruby-C-Assembly*

## Debugging a el llamado a Assembly

En la segunda iteración además se realiza un debugging del stack frame, donde se muestra el estado de los registros del stack: RBP,RSP. Adicionalmente se pueden observar otros registro de trabajos útiles como el puntero de instrucciones y registros de propósito general. Se construyó un software wrapper minimalista en C, para facilitar el debugging e inspección de la memoria, al ejecutarse el código correspondiente a Assembly. Las imágenes a continuación son extraídas de la IDE CLion, esta usa de fondo gdb.

En la siguiente imagen se puede observar los registros previo al llamado de la función:



```

6 → int call_assembly_routine(float value){ value: 40.7000008
7 →   int result = float_to_int(value); value: 40.7000008
8     return result;
9 }
10

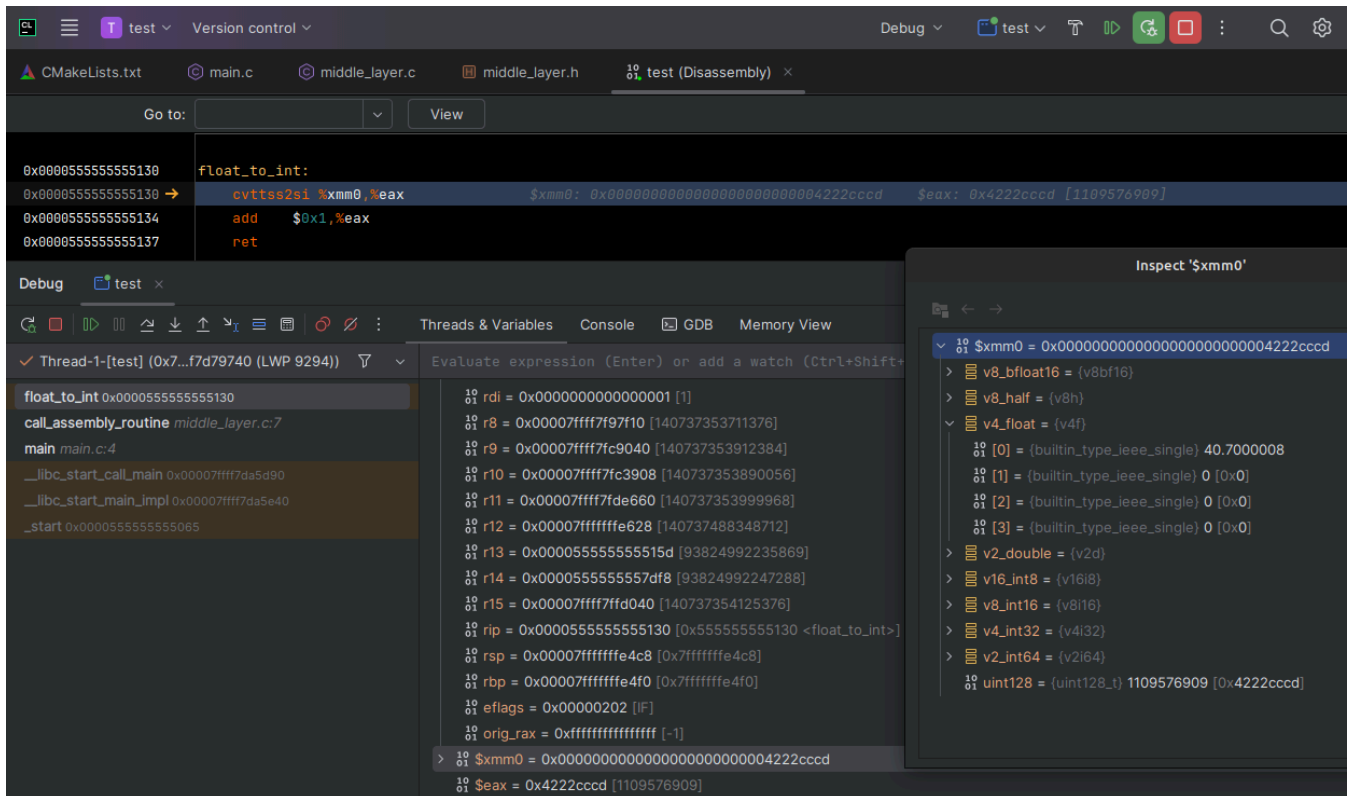
call_assembly_routine

Debug test x

Breakpoint 1, main () at /home/dmq/Desktop/temp2/main.c:4
4   int result = call_assembly_routine(40.7F);
The target architecture is set to "auto" (currently "i386:x86-64").
(gdb) info register
rax             0x4222cccd      1109576909
rbx             0x0
rcx             0x55555557df8    93824992247288
rdx             0x7fffffff638    140737488348728
rsi             0x7fffffff628    140737488348712
rdi             0x1
rbp             0x7fffffff4f0    0x7fffffff4f0
rsp             0x7fffffff4d0    0x7fffffff4d0
r8              0x7ffff7f97f10    140737353711376
r9              0x7ffff7fc9040    140737353912384
r10             0x7ffff7fc3908    140737353890056
r11             0x7ffff7fde660    140737353999968
r12             0x7fffffff628    140737488348712
r13             0x5555555515d    93824992235869
r14             0x555555557df8    93824992247288
r15             0x7ffff7ffd040    140737354125376
rip             0x55555555149    0x55555555149 <call_assembly_routine+17>
eflags          0x202          [ IF ]
cs              0x33          51
ss              0x2b          43
ds              0x0           0
(gdb)

```

En la próxima imagen podemos observar el cambio en registro RSP, donde se guarda la dirección de retorno al programa de C. Se muestran otros registros de propósito general además del “xmm0” de 128b donde se almacena el parámetro 40.7 correspondiente al coeficiente de gini obtenido de la API rest:

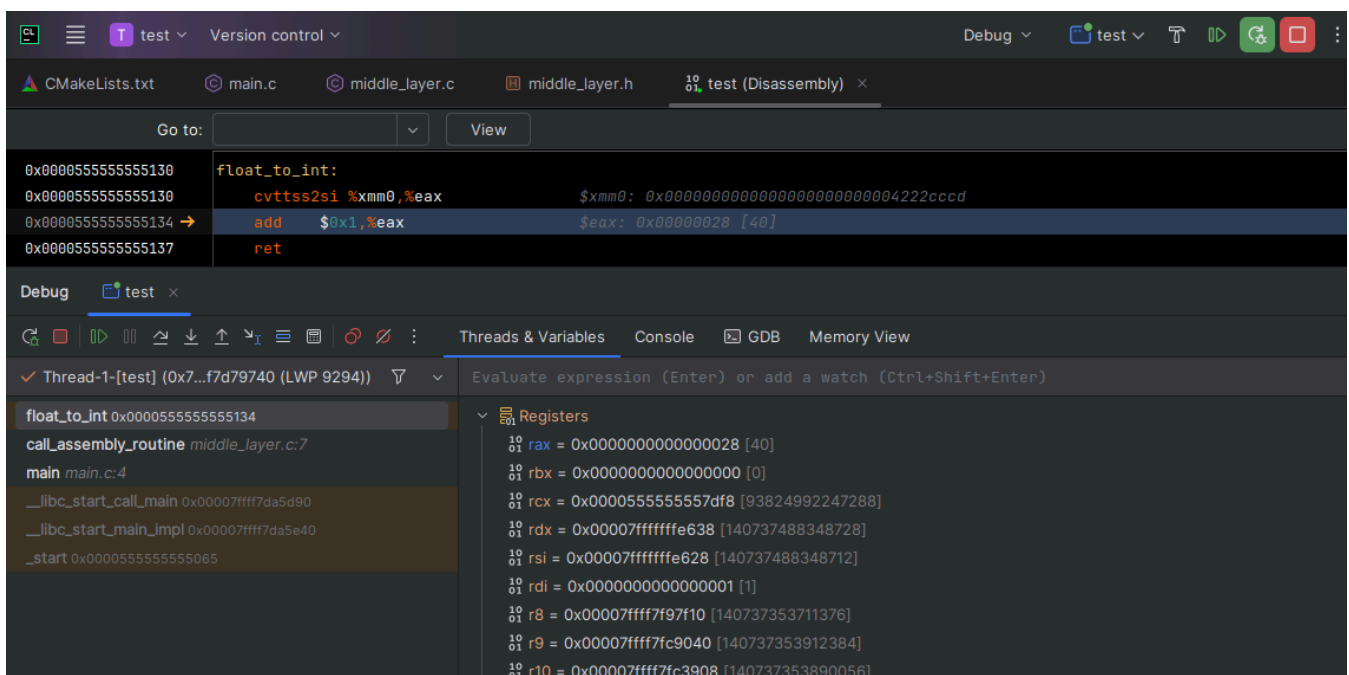


Disassembly window showing the `float_to_int` function. The instruction `cvtts2si %xmm0,%eax` is highlighted in blue. The register values are:

- `$xmm0`: 0x00000000000000004222cccd
- `$eax`: 0x4222cccd [1109576909]

The `Inspect '$xmm0'` window shows the contents of the `$xmm0` register, including `v8_bfloat16`, `v8_half`, `v4_float`, `v2_double`, `v16_int8`, `v8_int16`, `v4_int32`, `v2_int64`, and `uint128`.

Posterior a la ejecución de `cvtts2si`, como se puede observar en la sig. imagen en azul, se modifican los registros `rax`, `rip`, y `eax`. `rax` es una extensión de `eax`, por lo que en este caso, al tratarse de un valor pequeño de manipular (de 2 dígitos decimales), tienen *exactamente* el mismo valor (no hay overflow). Como se puede apreciar, el opcode `cvtts2si` copia el valor del registro de la FPU `xmm0` al registro de propósito general `eax`. Obviamente se actualiza el instruction pointer `rip`.



Disassembly window showing the `float_to_int` function. The instruction `cvtts2si %xmm0,%eax` is highlighted in blue. The register values are:

- `$rax`: 0x00000000000000004222cccd
- `$rbx`: 0x0000000000000000
- `$rcx`: 0x0000000000000000
- `$rdx`: 0x0000000000000000
- `$rsi`: 0x0000000000000000
- `$rdi`: 0x0000000000000000
- `$r8`: 0x0000000000000000
- `$r9`: 0x0000000000000000
- `$r10`: 0x0000000000000000

The `Registers` window shows the contents of the registers, including `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, and `r10`.

```

_start 0x0000555555555065
10 r11 = 0x00007ffff7fde660 [140737353999968]
01
10 r12 = 0x00007fffffe628 [140737488348712]
01
10 r13 = 0x000055555555515d [93824992235869]
01
10 r14 = 0x00005555555557df8 [93824992247288]
01
10 r15 = 0x00007ffff7ffd040 [140737354125376]
01
10 rip = 0x0000555555555134 [0x555555555134 <float_to_int+4>]
01
10 rsp = 0x00007fffffe4c8 [0x7fffffe4c8]
01
10 rbp = 0x00007fffffe4f0 [0x7fffffe4f0]
01
10 eflags = 0x00000202 [IF]
01
10 orig_rax = 0xffffffffffff [-1]
01
10 $eax = 0x00000028 [40]
> 10 $xmm0 = 0x000000000000000000000000422cccd
01

```

Posterior a ejecutar *add*, como se puede observar en la sig. imagen en azul, se modifican los registros *rax*, *rip*, y *eax*. Como se puede apreciar, el opcode *add* suma la constante 1 (0x01) al registro *eax*, y guarda el resultado en el mismo registro. Obviamente se actualiza el instruction pointer *rip*.

The screenshot shows the Visual Studio Code interface with the following components:

- Top Bar:** Includes icons for Explorer, Search, Run and Debug, and a "test" button. The "Version control" dropdown is visible.
- File Explorer:** Shows the project structure with files like CMakeLists.txt, main.c, middle\_layer.c, middle\_layer.h, and the active file "test (Disassembly)".
- Assembly View:** Displays the disassembly of the "test" function. The instructions shown are:
 

```

float_to_int:
0000000055555555130  float_to_int:
0000000055555555130  cvttsd2si %xmm0,%eax          $xmm0: 0x0000000000000000000000004222cccd
0000000055555555134  add $0x1,%eax                 $eax: 0x00000029 [41]
0000000055555555137  ret
      
```
- Debug Console:** Shows the "Thread-1-[test] (0x7...f7d79740 (LWP 9294))" and the "Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)" prompt.
- Registers Panel:** Displays the state of registers:
  - `rax`: 0x0000000000000029 [41]
  - `rbx`: 0x0000000000000000 [0]
  - `rcx`: 0x00005555555555df8 [93824992247288]
  - `rdx`: 0x00007ffffff638 [140737488348728]
  - `rsi`: 0x00007ffffff628 [140737488348712]
  - `rdi`: 0x0000000000000001 [1]
  - `r8`: 0x00007ffff7f97f10 [140737353711376]
  - `r9`: 0x00007ffff7fc9040 [140737353912384]
  - `r10`: 0x00007ffff7fc3908 [140737353890056]

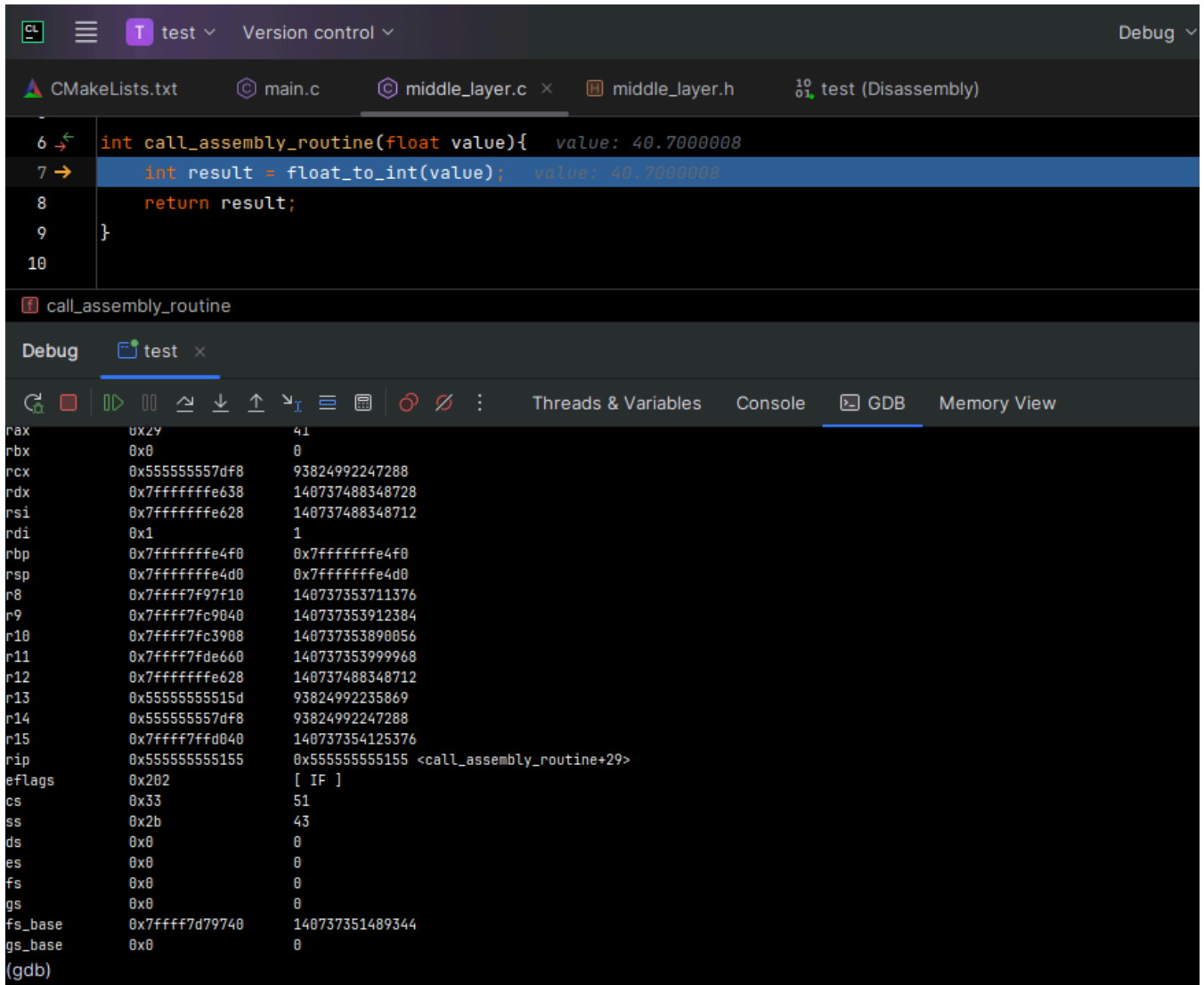
```

_start 0x0000555555555065
r11 = 0x00007ffff7fe660 [140737353999968]
r12 = 0x00007ffffffe628 [140737488348712]
r13 = 0x000055555555515d [93824992235869]
r14 = 0x0000555555557df8 [93824992247288]
r15 = 0x00007ffff7fd040 [140737354125376]
rip = 0x0000555555555137 [0x555555555137 <float_to_int+7>]
rsp = 0x00007ffffffe4c8 [0x7ffffffe4c8]
rbp = 0x00007ffffffe4f0 [0x7ffffffe4f0]
eflags = 0x00000202 [IF]
orig_rax = 0xffffffffffff [-1]
Seax = 0x00000029 [41]
xmm0 = 0x00000000000000000000000000422cccd

```



Por último, al salir de la función escrita en Assembly, se observa la modificación del registro *rsp* (que apunta al tope del stack frame *mas nuevo*). Como era de esperarse el valor se incrementa (recuérdese que la pila crece hacia direcciones de memoria menores, por lo tanto cuando decrece el tamaño del stack, *rsp* se incrementa).



The screenshot shows a debugger interface with the following components:

- Source View:** Displays the C code for `call_assembly_routine`. The function takes a `float value` and returns an `int result` obtained from `float_to_int(value)`.
- Disassembly View:** Shows the assembly code for `call_assembly_routine`. The instruction `int result = float_to_int(value);` is highlighted.
- Register Window:** Displays the state of various registers. The `rsp` register is highlighted, showing its value as `0x7fffffff4d0`.

Register	Value
rax	0x29
rbx	0x0
rcx	0x555555557df8
rdx	0x7fffffff638
rsi	0x7fffffff628
rdi	0x1
rbp	0x7fffffff4f0
rsp	0x7fffffff4d0
r8	0x7ffff797f10
r9	0x7ffff7fc9040
r10	0x7ffff7fc3908
r11	0x7ffff7fde660
r12	0x7fffffff628
r13	0x5555555515d
r14	0x555555557df8
r15	0x7ffff7ffd040
rip	0x55555555155 <call_assembly_routine+29>
eflags	0x202 [ IF ]
cs	0x33
ss	0x2b
ds	0x0
es	0x0
fs	0x0
gs	0x0
fs_base	0x7ffff7d79740
gs_base	0x0

Como nota podemos decir que los registros RBP y RSP no sufren grandes modificaciones porque la función de assembler no utiliza variables locales que se puedan almacenar en la pila para el tratamiento de los datos, por ser una función sencilla.

## Conclusión

Esta experiencia nos resultó muy interesante, dado que pudimos ver claramente como tener la capacidad de ir hacia el código máquina, o al contrario, hacia una capa de alto nivel, donde crear cosas sea más rápido, nos otorga la habilidad de resolver problemas, como cuellos de botella en la performance de nuestros programas, o proveer soluciones de acuerdo a tal o cual arquitectura, a nivel hardware.