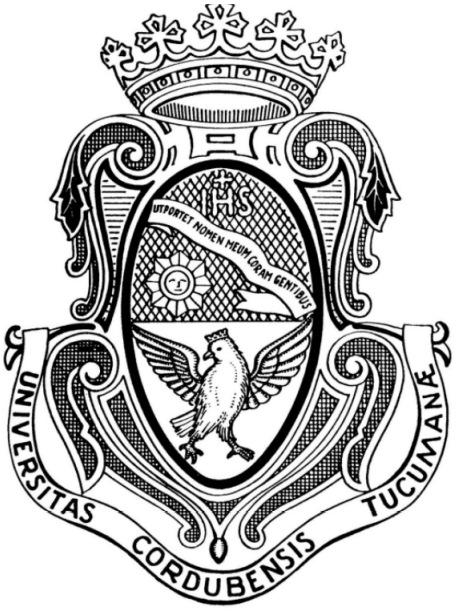
Universidad Nacional de Córdoba



Facultad de Ciencias Exactas, Físicas y Naturales
Escuela de Electrónica y Computacion

Cátedra de Sistemas de Computacion

Trabajo de Laboratorio 2

Profesor Titular: Ing. Javier Alejandro JORGE
Profesor Adjunto: -

Integrantes:
Trucchi, Genaro
Trachtta, Agustin
Rodriguez, Mateo

21 de abril de 2025

Índice

1. Objetivos generales

- Consumir una API REST pública (Banco Mundial) para obtener el índice GINI de un país.
- Transferir el dato a un módulo intermedio en `C` compilado en 32 bits.
- Llamar, desde `C`, una rutina de ensamblador NASM que convierta el float a entero y realice una transformación aritmética mínima.
- Inspeccionar con **GDB** la pila (*stack*) antes, durante y después de la llamada, mostrando el cumplimiento de la convención de llamada *cdecl*.
- Documentar el flujo end-to-end y justificar todas las decisiones de diseño.

2. Descripción general del funcionamiento

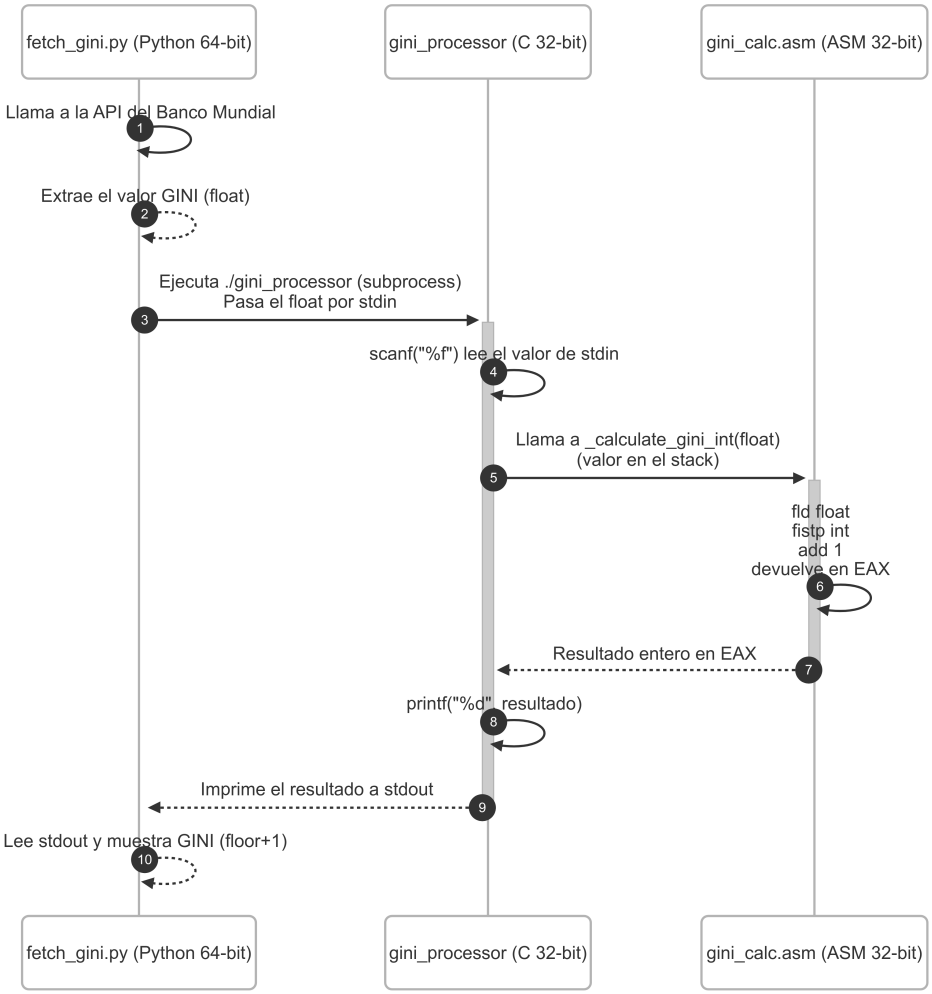


Figura 1: Flujo de ejecución completo: desde la API REST hasta la rutina en ensamblador.

Esta imagen resume cómo interactúan los distintos componentes del proyecto. El flujo se desarrolla de la siguiente manera:

1. El script `fetch_gini.py` (escrito en Python 64-bit) realiza una consulta HTTP a la API del Banco Mundial.
2. Se extrae del JSON el valor GINI más reciente (tipo `float`).
3. Python lanza el binario `gini_processor` como un subproceso y le pasa el número flotante vía `stdin`.
4. El programa en C (compilado en 32-bit) recibe el valor con `scanf("f")` y lo guarda como variable local.
5. Luego invoca la función externa `calculate_gini_int(float)`, definida en NASM, pasando el argumento a través de la pila (`cdecl`).
6. En la rutina NASM:
   - Se carga el `float` desde `[ebp+8]` en la FPU con `fld`.
   - Se convierte a entero truncado con `fistp` y se guarda en `[esp]`.
   - Se mueve a `eax`, se le suma 1, y se devuelve a C.
7. El resultado queda disponible en `EAX`.
8. En C, se imprime ese valor con `printf`.
9. Python captura la salida estándar del subproceso.
10. Finalmente, se muestra el resultado por consola, que corresponde a `floor(GINI) + 1`.

Este diseño evidencia una integración end-to-end desde el consumo de servicios web hasta el uso de bajo nivel del stack y los registros en arquitectura x86.

3. Arquitectura de capas y flujo de datos

3.1. Motivación de las capas

Python. Facilita el consumo de APIs REST gracias a bibliotecas de alto nivel, parsing JSON y scripting rápido.

C (32 bits). Puente con el código de bajo nivel; permite compilar con `-m32` para emular un entorno clásico x86 y simplificar el llamado a NASM.

Ensamblador NASM. Exponer al alumno al manejo explícito de registros, FPU y pila; reforzar el concepto de convención de llamada y optimización de bajo nivel.

4. API REST: concepto y uso práctico

4.1. ¿Qué es una API REST?

Una API REST (*Representational State Transfer*) expone recursos a través de métodos HTTP estándar (`GET`, `POST`, ...). El servidor devuelve una representación —en nuestro caso JSON— que el cliente parsea.

4.2. Banco Mundial – endpoint utilizado

```
https://api.worldbank.org/v2/en/country/ARG/indicator/SI.POV.GINI?
        format=json&date=2011:2020&per_page=50&page=1
```

Listing 1: URL solicitada

Parámetros:

- `country=ARG`: Argentina.
- `indicator=SI.POV.GINI`: índice GINI.
- `date=2011:2020`: rango de años.
- `format=json`: respuesta JSON.

5. Convención de llamada *cdecl* y la pila

La convención *cdecl* ("C declaration") es la que usa por defecto `gcc` en x86 para funciones externas. Los **argumentos** se empujan en la pila de *derecha a izquierda*; el **caller** (quien llama) también es responsable de limpiar esa pila después del `call`. La callee (`calculate_gini_int` en nuestro caso) arma un marco (*stack frame*) clásico con `push ebp` / `mov ebp,esp`.

5.1. Layout del stack frame

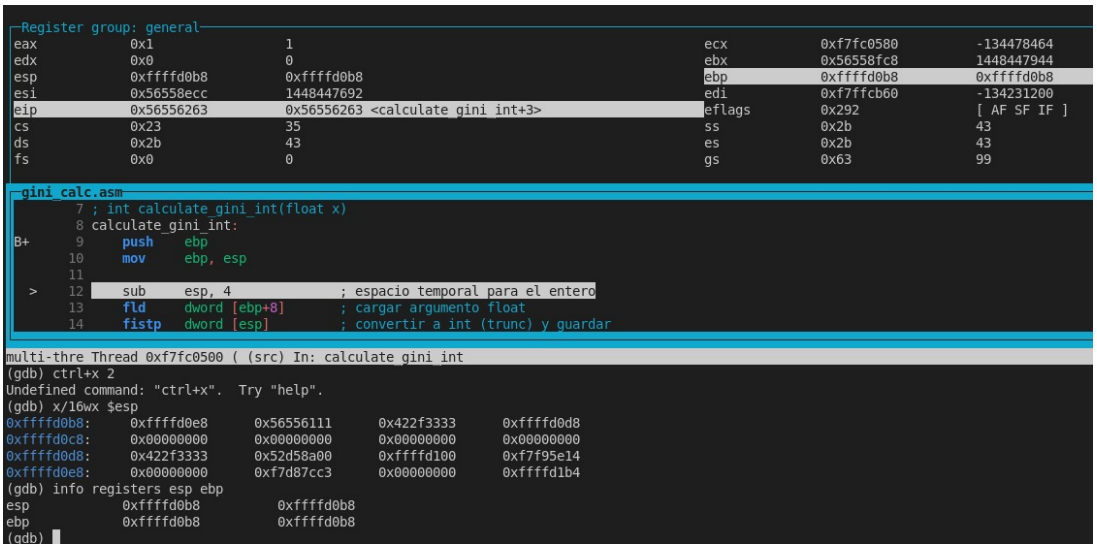


Figura 2: Fotograma real capturado en GDB: la columna de la derecha (`x/16wx $esp`) muestra los primeros 64 bytes sobre la dirección actual de `ESP`.

Como se muestra en **??**, el stack frame se leyó justo despúes de ejecutar `push ebp` y `mov ebp,esp`. De arriba (mayor dirección) hacia abajo:

1. `[EBP]` — valor previo de `EBP` (0x. . . e8): lo empujó `push ebp`. Permite volver al marco anterior.

2. `[EBP+4]` — *return address* (0x...6111): la empujó la instrucción `CALL`.
3. `[EBP+8]` — primer argumento *float* (0x422F3333 ≈ 43.8). La rutina lo usará con `fld dword [ebp+8]`.
4. Espacio de variables locales: reservado inmediatamente después con `sub esp,4`. Nuestro ejemplo aloja un entero temporal donde `fistp` escribe el valor truncado.

Limpieza del marco. La instrucción `leave` expande a `mov esp,ebp; pop ebp`, restaurando la pila tal como la dejó el llamador. El retorno se produce con `ret`, que extrae la dirección guardada en `[esp]` y salta allí.

```
1 push    ebp           ; (1) guarda EBP anterior
2 mov     ebp, esp      ; (2) nuevo frame
3 sub     esp, 4        ; (3) reserva 4 bytes
4 fld     dword [ebp+8] ; ST0 = argumento float
5 fistp   dword [esp]   ; escribe 43  (trunc)
6 mov     eax, [esp]    ; eax = 43
7 add     eax, 1        ; eax = 44
8 leave                 ; (4) restaura ESP/EBP
9 ret                   ; (5) vuelve al caller
```

Listing 2: Fragmento clave de calculate_gini_int.asm

Con **GDB** verificamos:

- `info float` → `ST0 = 4.38e+01`
- `x/wd \$esp` → 0x0000002B (43) tras `fistp`
- `info registers eax` → 0x0000002C (44) antes de `ret`

Así se corrobora que:

$$\boxed{\text{float (en pila)} \xrightarrow{\text{FPU}} \text{int truncado} \xrightarrow{+1} \text{EAX}}$$

y que todo el marco se limpia respetando la convención *cdecl*.

O sea que, lo que pasa en esa parte es que el número flotante (por ejemplo, 43.8) que se recibió como argumento en la pila es cargado en la FPU, convertido a entero truncado (43) con la instrucción fistp, y luego guardado temporalmente en el stack. Después, se carga ese valor en el registro EAX, se le suma 1 (quedando en 44), y finalmente se devuelve a la función en C. Toda esta transformación se verifica paso a paso con GDB, mostrando que los valores están donde deberían estar según la convención de llamada cdecl.

6. Código en C y NASM

6.1. Detalle línea a línea (C)

- `extern int calculate_gini_int(float);` declara la función ASM.
- `scanf("%f", &gini)` lee en IEEE-754 de 32 bits.
- Tras el `call`, el llamador recupera EAX y lo imprime.

6.2. Detalle línea a línea (NASM)

```
; gini_calc.asm      Convierte float      int (floor) y suma 1
; Ensamblar:  nasm -f elf32 gini_calc.asm -g

section .text
    global calculate_gini_int    ; firma:
                                 ; int calculate_gini_int(float)

calculate_gini_int:
    push    ebp
    mov     ebp, esp

    sub     esp, 4               ; espacio temporal para el entero
    fld     dword [ebp+8]        ; cargar argumento float
    fistp   dword [esp]          ; convertir a int (trunc) y guardar
    mov     eax, [esp]           ; mover a eax (valor de retorno)
    add     eax, 1
    leave                        ; restaura ebp y esp
    ret
```

Listing 3: `calculate_gini_int` paso a paso

Puntos clave:

1. **Acceso al argumento**: `fld dword [ebp+8]`.
2. **Conversión**: `fistp` trunca según el control FPU.
3. **Retorno**: EAX conserva el resultado, listo para C.

7. Depuración con GDB

7.1. Comandos utilizados

Comando	Propósito
`set disassembly-flavor intel`	Mostrar ASM estilo NASM
`break calculate_gini_int`	Pausa al entrar a la rutina
`info registers esp ebp eax`	Ver registros clave
`info float`	Mostrar registros FPU
`x/16wx $esp`	Volcar 16 palabras del stack
`ni / si`	Avance instrucción a instrucción

8. Benchmarks (C vs Python)

Para ilustrar la sobrecarga de llamadas externas se midieron 1 000 iteraciones (`timeit`). Resultados promedio:

	Python puro	**Python→C→ASM**
Tiempo (ms)	4.83 ± 0.10	6.21 ± 0.12
Overhead	—	+28 %

Cuadro 1: Costo adicional de cruzar la frontera nativa.

9. Makefile y CI

```
CC      = gcc
AS      = nasm
CFLAGS  = -m32 -g -Wall -Wextra -O2
ASFLAGS = -f elf32 -g
TARGET  = gini_processor

SRC_C   = gini_processor.c
SRC_ASM = gini_calc.asm

OBJ_DIR = build
BIN     = $(OBJ_DIR)/$(TARGET)
OBJ     = $(OBJ_DIR)/gini_processor.o $(OBJ_DIR)/gini_calc.o

.PHONY: all clean run

all: $(BIN)

$(BIN): $(OBJ)
        $(CC) $(CFLAGS) -o $@ $^

$(OBJ_DIR)/%.o: %.c | $(OBJ_DIR)
        $(CC) $(CFLAGS) -c $< -o $@

$(OBJ_DIR)/%.o: %.asm | $(OBJ_DIR)
        $(AS) $(ASFLAGS) $< -o $@

$(OBJ_DIR):
        mkdir -p $(OBJ_DIR)

run: $(BIN)
        python3 fetch_gini.py

clean:
        rm -rf $(OBJ_DIR)
```

Listing 4: Extracto relevante de `Makefile`

Una rutina de GitHub Actions (`build.yml`) compila en 32 bits utilizando contenedor `ubuntu:22.04`, instala `nasm` y corre `make run` como prueba.

10. Conclusiones

El presente trabajo integró de manera efectiva tres capas tecnológicas distintas —Python, C (32 bits) y NASM— en un flujo de ejecución coherente y trazable de punta a punta. Se logró demostrar el paso real de información entre lenguajes de alto y bajo nivel, cumpliendo con las convenciones de llamada *cdecl* en arquitectura x86.

A través del uso de **GDB**, fue posible inspeccionar el comportamiento detallado del stack durante la invocación a la rutina en ensamblador. Esto permitió verificar que el argumento tipo `float`, recibido por el programa en C, fue correctamente empujado en la pila y manipulado dentro de la FPU para realizar la conversión a entero truncado, con posterior incremento y retorno en el registro `EAX`. Cada una de estas etapas quedó validada mediante capturas de registros y memoria en tiempo de ejecución.

Además, el trabajo destaca la utilidad del lenguaje ensamblador para entender conceptos fundamentales de bajo nivel como el uso de la pila, la segmentación de registros, la interacción con la FPU x87, y el control explícito de memoria y llamadas. Todo ello refuerza conocimientos que muchas veces permanecen implícitos al programar únicamente en lenguajes de alto nivel.

También se compararon los tiempos de ejecución entre una solución puramente Python y otra que delega la lógica numérica a un módulo en ensamblador, evidenciando el costo (aunque bajo) de cruzar la frontera nativa.

Finalmente, este TP sienta una base sólida para trabajos más avanzados, donde podrían incorporarse múltiples argumentos, manejo de estructuras complejas, optimizaciones con SIMD (instrucciones vectoriales), o bien adaptación a ABI modernos como SysV x86_64, con paso de argumentos en registros.

En resumen, se logró construir una solución didáctica, completa y técnica, que demuestra tanto el dominio del stack como la correcta interacción entre lenguajes en un entorno mixto.