

# Laboratorio 01: RISC-V y Venus



Fecha límite de entrega: lunes 12 de mayo.

Este laboratorio te invita a explorar de manera práctica el lenguaje ensamblador RISC-V utilizando el simulador Venus: aprenderás a traducir fragmentos de C a instrucciones de bajo nivel, depurar paso a paso con herramientas como el depurador y Memcheck, gestionar memoria y practicar operaciones con arreglos y recursión, todo ello sin depender de saltos o bifurcaciones en algunos ejercicios clave; al finalizar, habrás reforzado tu comprensión de cómo la CPU interpreta ceros y unos como datos, instrucciones o direcciones de memoria, y estarás listo para enfrentarte a programas RISC-V más complejos.

## Introducción al lenguaje ensamblador

En este punto de la carrera, seguramente han trabajado con programas en C (con la extensión de archivo .c), utilizando el programa **gcc** para compilarlos a código máquina, y luego ejecutándolos directamente en su computador. Ahora, en esta materia estamos cambiando nuestro enfoque al emplear lenguaje ensamblador RISC-V, que es un lenguaje de nivel más bajo mucho más cercano al código máquina. No podemos ejecutar código RISC-V directamente en tu computadora porque probablemente está diseñada para ejecutar código máquina de otros lenguajes ensamblador, como x86 o ARM.

En este laboratorio, trabajaremos con varios archivos de ensamblador RISC-V, cada uno con extensión .s. Para ejecutarlos, usaremos **Venus**, un ensamblador y simulador educativo para RISC-V.

## Venus: Primeros pasos

Para comenzar con Venus, por favor revisa la documentación en la [referencia de Venus](#) con respecto a la pestaña "Editor" y la pestaña "Simulator". Recomendamos que leas esta página completa en algún momento, pero estas secciones deberían ser suficientes para empezar.

## Ejercicio 1: Conceptos básicos de Venus

Los archivos creados y editados en Venus se perderán cada vez que cierres la pestaña. A medida que vayas trabajando cada ejercicio recuerda copiar y pegar en un archivo local en tu computador.

Navega dentro de Venus a la pestaña **Editor**. Dentro del editor puede insertar el siguiente programa (**ej1\_hola.s**) en lenguaje ensamblador RISC-V:

```

1 .text
2
3 addi a0 x0 1
4 addi a1 x0 1234
5
6 # Esta llamada imprime el entero almacenado en a1
7 ecall
8
9 # Esta parte del código, permite finalizar el programa.
10 addi a0 x0 17
11 addi a1 x0 0
12 ecall

```

1. Deberías ver el contenido de `ej1_hola.s` en el editor. Este editor se comporta como la mayoría de los otros editores de texto, aunque sin muchas de las características más avanzadas.
2. Para ensamblar el programa abierto en el editor, **haz clic** en la pestaña "Simulator" y luego en "Assemble & Simulate from Editor".
3. En el futuro, si ya tienes un programa abierto en el simulador, haz clic en "Re-assemble from Editor" en su lugar. Ten en cuenta que esto sobrescribirá todo lo que tengas en la pestaña del simulador, como una sesión de depuración existente.
4. Para ejecutar el programa, **haz clic** en "Run".
  - Puedes observar que en el cuadro de la parte inferior se despliega la salida de la ejecución del programa.
  - También notarás otros botones como "Step", "Prev" y "Reset":
    - Step: ejecuta la siguiente instrucción del programa, permitiéndote observar el efecto de cada instrucción paso a paso.
    - Prev: retrocede a la instrucción ejecutada anteriormente, útil para analizar cómo cambian los registros y la memoria con cada paso.
    - Reset: reinicia el programa desde el principio, limpiando el estado de los registros y la memoria para comenzar una nueva ejecución.
  - Puedes utilizar estos botones para depurar y entender mejor el comportamiento de tus programas en RISC-V.
5. Regresa a la pestaña del editor y edita `ej1_hola.s` para que la salida imprima `2025`.

▼ *Pista: ¿Qué hace ecall?*

El valor en `a1` se imprime cuando se ejecuta `ecall`. Sin embargo, el funcionamiento interno de `ecall` está fuera del alcance de este Lab. Pero ustedes tienen todo el Internet a su disposición, ¿cierto?

6. **Realiza** una captura de pantalla del programa modificado y de la salida generada. Debes crear un reporte de laboratorio con demostración de las actividades realizadas. Asegurate de documentar adecuadamente el proceso para generar posteriormente el informe.

7. **Guarda** los cambios que acabas de hacer copiando el contenido del editor en tu archivo local.

## Traducción de C a RISC-V

En este ejemplo, vamos a desglosar el proceso de traducir un programa en C a un programa RISC-V. El siguiente programa imprimirá el *n*-ésimo número de Fibonacci. ¡Aunque esta sección es un poco larga, por favor léela completa!

```
1 #include <stdio.h>
2
3 int n = 12;
4
5 // Función para encontrar el n-ésimo número de Fibonacci
6 int main(void) {
7     int curr_fib = 0, next_fib = 1;
8     int new_fib;
9     for (int i = n; i > 0; i--) {
10         new_fib = curr_fib + next_fib;
11         curr_fib = next_fib;
12         next_fib = new_fib;
13     }
14     printf("%d\n", curr_fib);
15     return 0;
16 }
```

Vamos a desglosar cómo vamos a traducir esto paso a paso. Abre el editor de Venus. Primero, necesitamos definir la variable global *n*. En RISC-V, las variables globales se declaran bajo la directiva *.data*. Esto representa el segmento de datos. Se verá así:

```
1 .data
2 n: .word 12
```

- *n* es el nombre de la variable
- *.word* significa que el tamaño de los datos es de una palabra
- *12* es el valor asignado a *n*

Pasemos a inicializar *curr\_fib* y *next\_fib*

```
1 .text
2 main:
3     add t0, x0, x0 # curr_fib = 0
4     addi t1, x0, 1 # next_fib = 1
```

Aquí hemos agregado la directiva *.text*. Todo lo que esté bajo esta directiva es nuestro código.

Recuerda que `x0` siempre contiene el valor 0.

No necesitamos hacer nada para declarar `new_fib` (no declaramos variables en RISC-V).

A continuación, pasemos al bucle. Empezaremos configurando las variables del bucle. El siguiente código asignará a `i` el valor de `n`:

```
1 la t3, n # cargar la dirección de la etiqueta n
2 lw t3, 0(t3) # obtener el valor que se almacena en la dirección indicada por la
  etiqueta n
```

Puedes pensar en el código anterior como si hiciera algo como lo siguiente:

```
1 t3 = &n;
2 t3 = *t3;
```

Aquí tenemos una nueva instrucción: `la`. Esta instrucción carga la dirección de una etiqueta. La primera línea esencialmente configura a `t3` para que sea un puntero a `n`. Luego usamos `lw` para desreferenciar `t3`, lo cual asignará a `t3` el valor almacenado en `n`.

Ahora, probablemente estés pensando: "¿Por qué no podemos asignar directamente `n` a `t3`?" En la sección `.text`, no hay forma de acceder directamente a `n`. (Piensa en ello. No podemos decir `add t3, n, x0`. Los argumentos de `add` deben ser registros y `n` no es un registro.) La única manera de acceder a `n` es obteniendo primero la dirección de `n`. Una vez obtenida la dirección de `n`, necesitamos desreferenciarla, lo cual se puede hacer con `lw`. `lw` accederá a la memoria en la dirección que especifiques y cargará el valor almacenado en esa dirección. En este caso, especificamos la dirección de `n` y le añadimos un desplazamiento de `0`.

Ahora entremos en el bucle. Primero, crearemos la estructura externa a continuación:

```
1 fib:
2     beq t3, x0, finalizar # salir del bucle una vez que hayamos completado n
  iteraciones
3     ...
4     addi t3, t3, -1 # contador de decremento
5     j fib # bucle
6 finalizar:
```

La primera línea (`fib:`) es una etiqueta que usaremos para saltar de nuevo al comienzo del bucle.

La siguiente línea (`beq t3, x0, finalizar`) especifica nuestra condición de terminación. Aquí, saltaremos a otra etiqueta, `finalizar`, una vez que `t3` (que representa `i`) alcance `0`.

La siguiente línea (`addi t3, t3, -1`) decrementa `i` al final del cuerpo del bucle. Es importante hacer esto al final porque `i` se usa dentro del cuerpo del bucle. Si lo actualizáramos justo después de `beq`, entonces no tendría el valor correcto dentro del cuerpo del bucle.

La siguiente instrucción salta de nuevo al inicio del bucle.

Ahora, agreguemos el cuerpo del bucle.

```
1 fib:
2     beq t3, x0, finalizar # salir del bucle una vez que hayamos completado n
    iteraciones
3     add t2, t1, t0 # new_fib = curr_fib + next_fib;
4     mv t0, t1 # curr_fib = next_fib;
5     mv t1, t2 # next_fib = new_fib;
6     addi t3, t3, -1 # contador de decremento
7     j fib # bucle
8 finalizar:
```

Nada especial aquí. Las líneas correspondientes en C están escritas en los comentarios.

¡Imprimamos el n-ésimo número de Fibonacci!

```
1 finalizar:
2     addi a0, x0, 1 # argumento a ecall para ejecutar imprimir entero
3     addi a1, t0, 0 # argumento a ecall, el valor a imprimir
4     ecall # llamada a ecall -> imprimir entero
```

Imprimir es una llamada al sistema. Aprenderás más sobre esto durante el semestre, pero una llamada al sistema es esencialmente una forma de que tu programa interactúe con el sistema operativo. Para hacer una llamada al sistema en RISC-V, usamos una instrucción especial llamada **ecall**. Para imprimir un entero, necesitamos pasar dos argumentos a **ecall**. El primer argumento especifica lo que queremos que **ecall** haga (en este caso, imprimir un entero). Para especificar que queremos imprimir un entero, pasamos un **1**. El segundo argumento es el entero que queremos imprimir.

En C, estamos acostumbrados a funciones que se ven como **ecall(1, t0)**. En RISC-V, no podemos pasar argumentos de esta manera. Para pasar un argumento, debemos colocarlo en un registro de argumentos (**a0-a7**). Cuando la función se ejecute, buscará los argumentos en estos registros. (Si aún no has visto esto en clase, pronto lo verás). El primer argumento debe colocarse en **a0**, el segundo en **a1**, etc.

Para configurar los argumentos, colocamos un **1** en **a0** y colocamos el entero que queríamos imprimir en **a1**.

A continuación, ¡terminemos nuestro programa! Esto también requiere **ecall**.

```
1 addi a0, x0, 10 # argumento para ecall para la función de terminar
2 ecalls # llamada a ecalls -> terminar
```

En este caso, **ecall** solo necesita un argumento. Configurar **a0** a **10** especifica que queremos terminar el programa.

¡Y ahí lo tienes! ¡Aquí está nuestro programa completo!

```
1 .data
2 n: .word 12
3
4 .text
5 main:
6     add t0, x0, x0 # curr_fib = 0
7     addi t1, x0, 1 # next_fib = 1
8     la t3, n # cargar la dirección de la etiqueta n
9     lw t3, 0(t3) # obtener el valor que se almacena en la dirección indicada por la
    etiqueta n
10 fib:
11     beq t3, x0, finalizar # salir del bucle una vez que hayamos completado n
    iteraciones
12     add t2, t1, t0 # new_fib = curr_fib + next_fib;
13     mv t0, t1 # curr_fib = next_fib;
14     mv t1, t2 # next_fib = new_fib;
15     addi t3, t3, -1 # contador de decremento
16     j fib # bucle
17 finalizar:
18     addi a0, x0, 1 # argumento a ecall para ejecutar imprimir entero
19     addi a1, t0, 0 # argumento a ecall, el valor a imprimir
20     ecall # llamada a ecall -> imprimir entero
21     addi a0, x0, 10 # argumento para ecall para la función de terminar
22     ecall # llamada a ecall -> terminar
```



Debes guardar tu código en un archivo **fib.s**.

## Ejercicio 2: Uso del depurador de Venus

### 1. En el editor

- Copia **fib.s** en el editor de Venus.
- Haz clic en la pestaña "Simulator" y en el botón "Assemble & Simulate from Editor" (o en "Re-assemble from Editor").
- La instrucción actual se resalta en color azul claro. La instrucción actual es la instrucción que aún no se ha ejecutado, pero está a punto de ejecutarse.

Este ejercicio te pedirá que escribas tus respuestas en un archivo llamado **ej2\_respuestas.txt**. Los números de las preguntas pueden ser diferentes de los números de los pasos, ¡por favor ten cuidado!

### 1. Copia **fib.s** en el depurador de Venus.

- Pregunta 1: ¿Cuál es el código máquina de la instrucción resaltada? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo **0x**.
- Pregunta 2: ¿Cuál es el código máquina de la instrucción en la dirección **0x34**? La respuesta

debe ser un número hexadecimal de 32 bits, con el prefijo 0x.

2. **Haz clic** en el botón "Step" para avanzar a la siguiente instrucción. La segunda instrucción ahora debería estar resaltada.
3. **Haz clic** en el botón "Prev" para deshacer la última instrucción ejecutada. Ten en cuenta que deshacer puede o no deshacer operaciones realizadas por **ecall**, como salir del programa o imprimir en la consola.
4. En el lado derecho de la pantalla, **haz clic** en la pestaña "Registers" para ver los valores de los 32 registros. Esta pestaña puede que ya esté seleccionada. Asegúrate de que estás viendo los registros enteros, no los registros de punto flotante.
  - Pregunta 3: ¿Cuál es el valor del registro **sp**? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo 0x.
5. **Continúa avanzando paso a paso** hasta que cambie el valor en **t1**.
  - Pregunta 4: ¿Cuál es el nuevo valor del registro **t1**? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo 0x.
  - Pregunta 5: ¿Cuál es el código máquina de la *instrucción actual*? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo 0x.
6. **Continúa dando pasos** hasta que estés en la dirección **0x10**. En este punto, el valor de **t3** ha sido actualizado.
  - Pregunta 6: ¿Cuál es el valor del registro **t3**? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo 0x.
7. Si miramos la instrucción actual, estamos cargando desde el registro **t3**. Usa la pestaña "Memory" (al lado de la pestaña "Registers") e ingresa la respuesta de la pregunta 6 (el valor de **t3**) en el cuadro "Address". Puede que necesites desplazarte hacia abajo en la pestaña de memoria antes de que sea visible. Presiona "Go" para ir a esa dirección de memoria.
  - Pregunta 7: ¿Cuál es el byte al que apunta **t3**? La respuesta debe ser un número hexadecimal de 8 bits (1 byte), con el prefijo 0x.
8. **Pon un punto de interrupción** en la dirección **0x28** haciendo clic en la fila de esa dirección. La fila debería volverse de color rojo claro y debería aparecer un símbolo de punto de interrupción.
9. **Continúa** hasta el punto de interrupción presionando "Run".
  - Pregunta 8: ¿Cuál es el valor del registro **t0**? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo 0x.
10. **Continúa** 6 veces más.
  - Pregunta 9: ¿Cuál es el nuevo valor del registro **t0**? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo 0x.
11. A veces, leer valores hexadecimales no es muy útil. **Configura** la opción de visualización a "Decimal" usando el menú desplegable en la parte inferior de la pestaña de registros. Esto también se puede hacer en la pestaña de memoria.
  - Pregunta 10: ¿Cuál es el valor del registro **t0** en decimal? La respuesta debe ser un número decimal sin prefijo.

12. **Haz clic** en la instrucción en la dirección **0x28** nuevamente para quitar el punto de interrupción.
13. **Haz clic** en "Run" para terminar la ejecución del programa, ya que ya no hay más puntos de interrupción.
  - Pregunta 11: ¿Cuál es la salida del programa? La respuesta debe ser un número decimal sin prefijo.

## Ejercicio: La misma secuencia, dos interpretaciones

En este ejercicio vas a reforzar la idea de que, a nivel de máquina, los datos son simplemente ceros y unos, y que **la interpretación** (como carácter o como número) depende únicamente de la syscall que uses. Se divide en **dos partes** muy similares, para que compares los resultados.

### Parte A: Imprimir letras ('A'–'Z')

#### Objetivo

Crear un programa RISC-V que recorra los códigos ASCII de 65 a 90 e imprima cada valor como un **carácter** mayúsculo.

#### Tareas

1. Escribe el archivo **ej1\_alpha\_loop.s** con el siguiente esqueleto:

```
.data
space: .asciiz " "      # separador en blanco

.text
.globl _start
_start:
    # 1) Inicializar ASCII 'A'
    li    t0, 65        # t0 = 65 ('A')

loop:
    # 2) Imprimir carácter
    li    a0, 11         # ecall print_char ①
    mv    a1, t0         # carácter a imprimir
    ecall                          # invoca print_char

    # 3) Imprimir espacio
    li    a0, 4          # ecall print_str
    la    a1, space
    ecall                          # invoca print_str

    # 4) Incrementar y comprobar fin de bucle
    addi  t0, t0, 1      # siguiente código ASCII
    li    t1, 91         # 'Z' + 1 = 91
```



```

blt  t0, t1, loop    # mientras t0 < 91, repetir bucle

# 5) Salida limpia
li   a0, 10          # ecall exit
ecall                          # invoca exit

```

① Observa con atención el valor asignado a **a0**.

2. Ensambla y ejecuta en Venus.
3. Anota la **salida exacta**.

## Parte B: Imprimir códigos numéricos (65–90) con espacio

### Objetivo

Reutilizar la misma lógica (el registro **t0** recorre 65–90) pero, en lugar de **print\_char**, usar **print\_int** para **interpretar** cada valor como un **entero** y luego separar por espacios.

### Tareas

1. Copia **ej1\_alpha\_loop.s** a un nuevo archivo llamado **ej1\_alpha\_codes.s** y reemplaza el bucle de impresión con este:

```

.data
space: .asciiz " "      # separador en blanco

.text
.globl _start
_start:
    # 1) Inicializar valor 65
    li  t0, 65          # t0 = 65

loop:
    # 2) Imprimir entero
    li  a0, 1            # ecall print_int ①
    mv  a1, t0            # entero a imprimir
    ecall                  # invoca print_int

    # 3) Imprimir espacio
    li  a0, 4            # ecall print_str
    la  a1, space
    ecall                  # invoca print_str

    # 4) Incrementar y comprobar fin de bucle
    addi t0, t0, 1        # siguiente valor
    li  t1, 91            # 90 + 1 = 91
    blt  t0, t1, loop    # mientras t0 < 91, repetir bucle

    # 5) Salida limpia

```

```
li    a0, 10          # ecall exit
ecall                # invoca exit
```

① Observa con atención el valor asignado a **a0**.

2. Ensambla y ejecuta en Venus.
3. Anota la **salida exacta**.

## Reflexión

- ¿Qué observa en la **misma secuencia** de valores?
- ¿Cómo cambia el resultado cuando usas **print\_char** vs. **print\_int**?
- ¿Qué nos enseña esto sobre la **abstracción** de registros y llamadas al sistema en RISC-V?

Con este par de ejercicios, comprenderás que los bits no “son” letras ni números hasta que el software (syscall) decide **cómo** interpretarlos.

## ¿Qué es **utils.s**?

### Archivo **utils.s**

Este módulo define tres funciones fundamentales (**malloc**, **free** y **exit**) que envuelven llamados al sistema (**ecall**) del entorno Venus.

Estas funciones actúan como *wrappers* para facilitar operaciones de manejo de memoria dinámica y finalización del programa:

```
.globl malloc, free, exit
```

Esto permite que Venus reconozca y exponga estas funciones como símbolos globales reutilizables en otros archivos.

### **malloc**

```
malloc:
    mv a1 a0          # Pasa el tamaño a a1
    li a0 0x3CC       # Identificador del entorno Venus
    addi a6 x0 1       # Código de operación 1 = malloc
    ecall             # Solicita la asignación de memoria
    jr ra             # Retorna al llamador con puntero en a0
```

Permite solicitar memoria dinámica. El tamaño en bytes debe pasarse en **a0**. El puntero al bloque asignado será devuelto en **a0**.

## free

```
free:
    mv a1 a0          # Dirección a liberar
    li a0 0x3CC
    addi a6 x0 4      # Código de operación 4 = free
    ecall
    jr ra
```

Libera un bloque de memoria previamente reservado con **malloc**. La dirección a liberar debe pasarse en **a0**.

## exit

```
exit:
    mv a1 a0          # Código de salida
    li a0 17          # ecall 17 = terminar ejecución
    ecall
```

Finaliza el programa con el código de salida dado en **a0**. Útil para reportar errores o resultados al entorno.

### utils.s

```
1 .globl malloc, free, exit
2
3 .text
4 malloc:
5     mv a1 a0
6     li a0 0x3CC
7     addi a6 x0 1
8     ecall
9     jr ra
10
11 free:
12     mv a1 a0
13     li a0 0x3CC
14     addi a6 x0 4
15     ecall
16     jr ra
17
18 exit:
19     mv a1 a0
20     li a0 17
21     ecall
```

## Consideraciones

- Estas funciones dependen del entorno Venus, específicamente de su soporte extendido de `ecall` personalizado (`a0 = 0x3CC`).
- Son especialmente útiles cuando se desea manejar memoria sin recurrir a una simulación manual con `.space` o arreglos estáticos.

Puedes cargar este archivo en Venus usando la función **Upload** en la terminal, para que otros archivos puedan invocar `malloc`, `free` y `exit` sin redefinirlos.

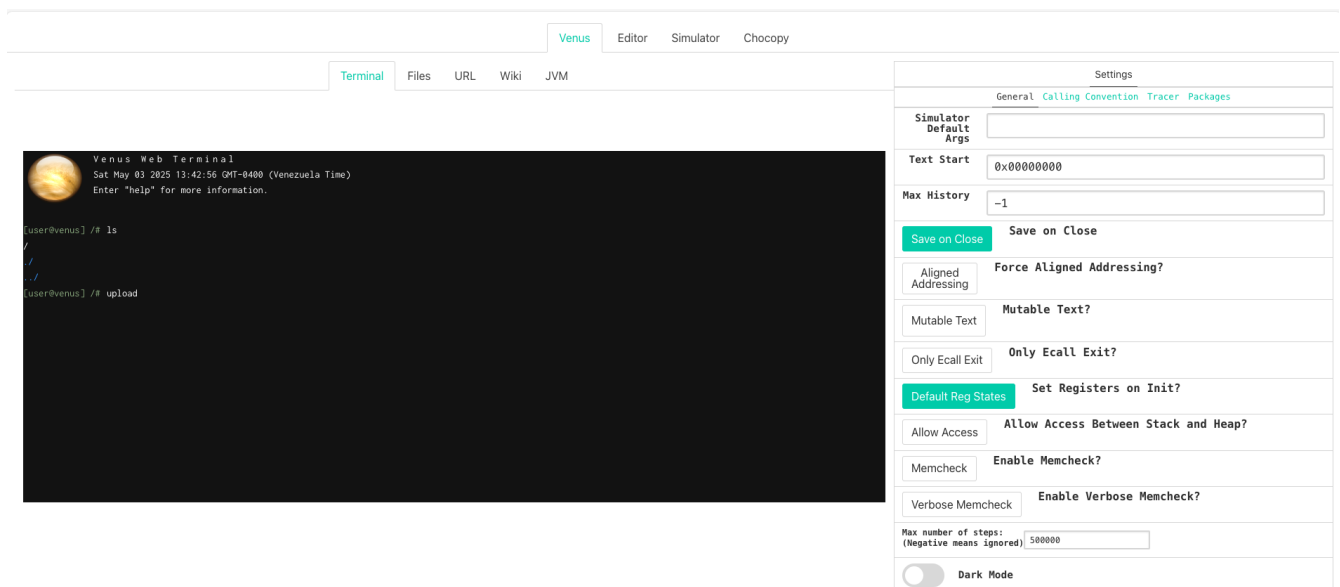
## Cómo integrar `utils.s` en tu flujo de trabajo con Venus

Para aprovechar las funciones helper en tus ejercicios (por ejemplo, en `ej1_hola.s` o `ej1_alpha_loop.s`), es necesario que Venus conozca y ensamble también `utils.s`. Sigue estos pasos:

1. **Recarga tu navegador** para partir de un estado limpio: presiona F5 o el botón de recarga en tu navegador y abre de nuevo <https://venus.cs61c.org>.
2. En la interfaz de Venus, ve a la pestaña **Venus** y selecciona la subpestaña **Terminal**.
3. En el prompt del terminal de Venus, escribe:

```
[user@venus] /# ls
[user@venus] /# upload
```

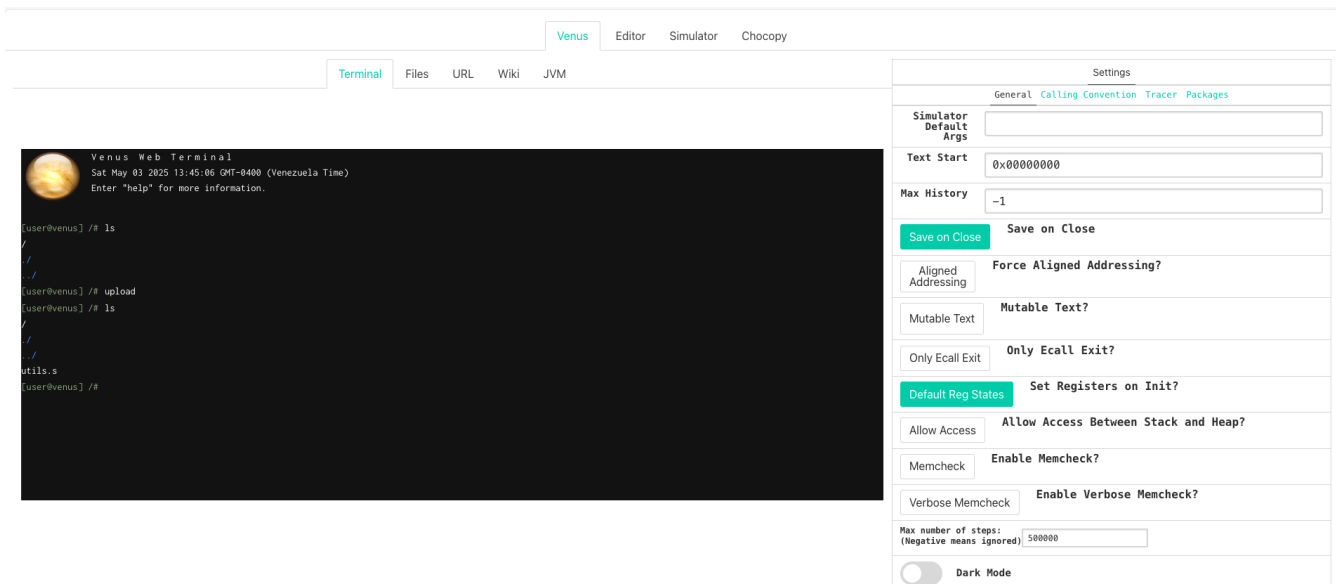
y presiona **Enter**.



1. Se abrirá un cuadro de diálogo para seleccionar archivos desde tu máquina local. Elige el archivo `utils.s` y confirma.
2. De vuelta en el terminal, escribe:

```
[user@venus] /# ls
```

y presiona **Enter**. Deberías ver **utils.s** listado entre los archivos del directorio.



1. A partir de este momento, Venus podrá ensamblar **utils.s** junto con tus programas.
2. Verificar archivos: Puedes abrir el archivo recién cargado desde la pestaña **Files**:
  - En la pestaña **Files** verás **utils.s** (y potencialmente otros archivos **.s** que hayas cargado usando **upload**. Esto lo exploraremos en otra sección), y si lo deseas puedes abrirlo desde esta pestaña.
  - También puedes editarlo directamente con:

```
[user@venus] /# edit utils.s
```

3. Uso de las helpers definidas en **utils.s**
  - Ahora en tu programa puedes hacer uso de las funciones definidas en **utils.s**.

## Venus: Memcheck

Cuando programamos en lenguajes como C, **valgrind** es la herramienta preferida para depurar errores de acceso a memoria (como **Segmentation fault (core dumped)**). Para Venus, tenemos una función llamada "memcheck" que realiza algo similar. Los mensajes de error de memcheck están diseñados para imitar los mensajes de error de **valgrind**. Nota: esta función se desarrolló en el 2022, ¡así que puede llegar a haber algún error!

Memcheck viene en dos modos:

- Modo normal (o simplemente "memcheck"): Este modo mostrará cualquier lectura o escritura inválida en la memoria. Si hay memoria no liberada cuando el programa finaliza, también imprimirá el número de bytes de memoria no liberada.
- Modo detallado (o "memcheck verbose"): Además del modo normal, este modo también imprime cada lectura/escritura de memoria, junto con una lista de los bloques que no se liberaron cuando el programa finaliza.

Puedes habilitar estos modos en la pestaña de Venus. Si están seleccionadas tanto 'Enable Memcheck?' como 'Enable Memcheck Verbose?', memcheck se ejecutará en modo detallado.

Debes reabrir el archivo que estás depurando después de habilitar o deshabilitar memcheck.

## Ejercicio 3: Uso de Memcheck

Al igual que en el ejercicio anterior, este ejercicio te pedirá que escribas tus respuestas en `ej3_respuestas.txt`. Los números de las preguntas pueden ser diferentes de los números de los pasos, ¡por favor ten cuidado!

1. **Abre** `ej3_memcheck.s` y `utils.s` en el terminal de Venus y lee todo el programa para tener una idea de lo que hace.
  - Realiza este paso usando el procedimiento detallado en la sección `¿Qué es utils.s?`, haciendo upload de ambos archivos.
2. **Ejecuta** el programa. ¡Oh, no, el programa arrojó un error!
  - Pregunta 1: ¿Qué dirección intentó acceder el programa, pero causó el error? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo `0x`.
  - Pregunta 2: ¿Cuántos bytes intentaba acceder el programa? La respuesta debe ser un número decimal sin prefijo.
3. Esto parece un error de memoria, así que probemos con memcheck. **Habilita** memcheck (modo normal) y **vuelve a abrir** `ej3_memcheck.s`.
4. **Ejecuta** el programa. ¡Mira, hay un error de memcheck con más detalles! **Lee** cuidadosamente el mensaje de error.
  - Pregunta 3: ¿Qué dirección intentó acceder el programa, pero causó el error? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo `0x`.
  - Pregunta 4: ¿Cuántos bytes fueron asignados en el bloque relacionado con el error? La respuesta debe ser un número sin unidades.
  - Pregunta 5: ¿Qué línea del archivo fuente causó este error? La respuesta debe ser un número.
5. **Compara** tu respuesta de la pregunta 2 con la de la pregunta 4. Ten en cuenta que memcheck puede cambiar la dirección de memoria que devuelve `malloc`.
6. Intentemos depurar este error. **Recuerda** que `t1` contiene el contador del bucle.
  - Pregunta 6: ¿Cuál es el valor de `t1` según el mensaje de error de memcheck? La respuesta debe ser un número decimal.
7. **Corrige** este error en el código fuente y **guarda** el archivo.
8. **Ejecuta** el programa de nuevo. El proceso se completó sin errores de acceso inválido. Sin embargo, el programa indica que hay algo de memoria no liberada.
  - Pregunta 7: ¿Cuántos bytes no fueron liberados cuando el programa terminó? La respuesta debe ser un número decimal sin unidades.
9. **Ejecuta nuevamente** el programa con memcheck en modo detallado. Recuerda volver a abrir

el archivo.

- Pregunta 8: ¿Cuál es la dirección del bloque que no se liberó? La respuesta debe ser un número hexadecimal de 32 bits, con el prefijo `0x`.

10. **Corrige** este error llamando a `free`.

11. **Deshabilita** memcheck para los próximos dos ejercicios.

▼ *Pista:*

Coloca un puntero al comienzo del arreglo en `a0`, luego llama a `free` usando `jal`.

## Ejercicio 4: Práctica de arreglos

Considera la función de valores discretos `f` definida en los enteros del conjunto  $\{-3, -2, -1, 0, 1, 2, 3\}$ . Aquí está la definición de la función:

```
1 f(-3) = 6
2 f(-2) = 61
3 f(-1) = 17
4 f(0) = -38
5 f(1) = 19
6 f(2) = 42
7 f(3) = 5
```

Implementa la función en `ej4_discreta_fn.s` en RISC-V, con la condición de que tu código **NO** pueda usar ninguna instrucción de salto o bifurcación. Asegúrate de que tu código esté guardado localmente. Hemos proporcionado algunas pistas en caso de que te quedes atascado.

Todos los valores de salida se almacenan en el arreglo de salida que se pasa a `f` a través del registro `a1`. **NO** necesitas crear el arreglo por tu cuenta. Puedes indexar ese arreglo para obtener la salida correspondiente a la entrada.

**Asegúrate de escribir solo en los registros `t` y `a`. Si usas otros registros, pueden ocurrir cosas extrañas (pronto aprenderás por qué).**

▼ *Pista 1:*

Puedes acceder a los valores del arreglo usando `lw`.

▼ *Pista 2:*

`lw` requiere que el desplazamiento sea un valor inmediato. Cuando calculemos el desplazamiento para este problema, se almacenará en un registro. Como no podemos usar un registro como desplazamiento, podemos sumar el valor almacenado en el registro a la dirección base para calcular la dirección del elemento en el índice que nos interesa. Luego podemos realizar un `lw` con un desplazamiento de `0`.

1. Necesitamos multiplicar el índice por el tamaño de los elementos del arreglo.
2. Luego sumamos este desplazamiento a la dirección base del arreglo para obtener la dirección del elemento que deseamos leer.

### 3. Leemos el elemento.

```
1 slli t2, t0, 2 # paso 1 (ver arriba)
2 add t2, t2, t1 # paso 2 (ver arriba)
3 lw t3, 0(t2) # paso 3 (ver arriba)
```

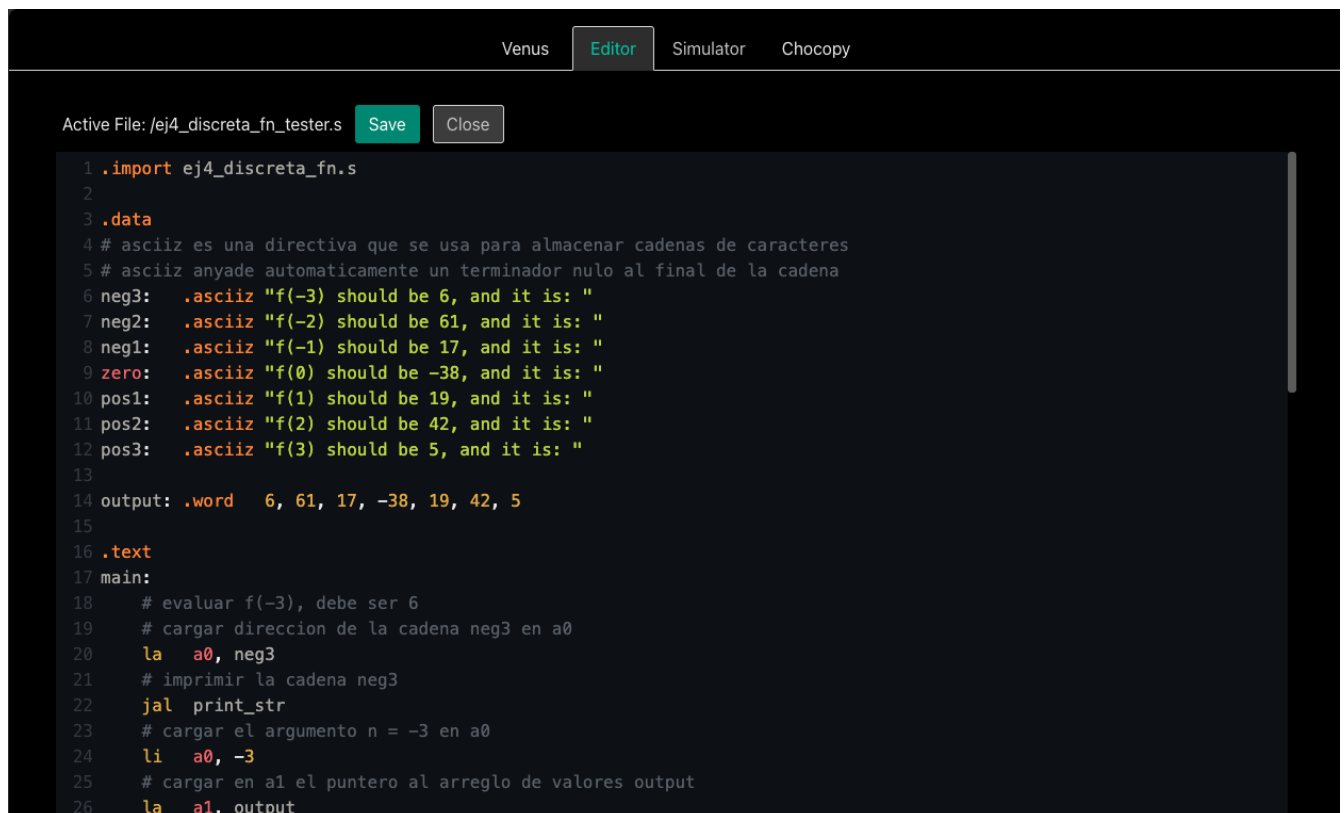
#### ▼ Pista 3:

$f(-3)$  debería almacenarse en el desplazamiento 0,  $f(-2)$  debería almacenarse en el desplazamiento 1, y así sucesivamente.

## Pruebas

Para probar tu función, crea el archivo `ej4_discreta_fn_tester.s` y ejecútalo en el simulador. Debe poder ejecutarse todo este programa usando tu función. Asegúrate de que pase todas las pruebas.

Fíjate que aparece incluida en la línea 1 la función que desarrollaste en el paso anterior:



```
Active File: /ej4_discreta_fn_tester.s [Save] [Close]

1 .import ej4_discreta_fn.s
2
3 .data
4 # asciiz es una directiva que se usa para almacenar cadenas de caracteres
5 # asciiz anyade automaticamente un terminador nulo al final de la cadena
6 neg3: .asciiz "f(-3) should be 6, and it is: "
7 neg2: .asciiz "f(-2) should be 61, and it is: "
8 neg1: .asciiz "f(-1) should be 17, and it is: "
9 zero: .asciiz "f(0) should be -38, and it is: "
10 pos1: .asciiz "f(1) should be 19, and it is: "
11 pos2: .asciiz "f(2) should be 42, and it is: "
12 pos3: .asciiz "f(3) should be 5, and it is: "
13
14 output: .word 6, 61, 17, -38, 19, 42, 5
15
16 .text
17 main:
18 # evaluar f(-3), debe ser 6
19 # cargar direccion de la cadena neg3 en a0
20 la a0, neg3
21 # imprimir la cadena neg3
22 jal print_str
23 # cargar el argumento n = -3 en a0
24 li a0, -3
25 # cargar en a1 el puntero al arreglo de valores output
26 la a1, output
```

## Ejercicio 5: Factorial

Asegúrate de que memcheck esté deshabilitado para este ejercicio.

En este ejercicio, implementarás la función factorial en RISC-V. Esta función recibe un parámetro entero  $n$  y devuelve  $n!$ . El esqueleto de esta función (archivo `ej5_factorial.s`) es el siguiente:

```
.globl factorial
```



```

.data
n: .word 8

.text
# No te preocupes por entender el código en la clase principal.
# Pronto aprenderás más sobre las llamadas a funciones en la clase.
main:
    la t0, n
    lw a0, 0(t0)
    jal ra, factorial

    addi a1, a0, 0
    addi a0, x0, 1
    ecall # Imprimir Resultado

    addi a1, x0, '\n'
    addi a0, x0, 11
    ecall # Imprimir newline

    addi a0, x0, 10
    ecall # Exit

# factorial toma un argumento:
# a0 contiene el número del cual queremos calcular el factorial
# El valor de retorno debe almacenarse en a0
factorial:
    # TU CÓDIGO AQUÍ

    # Así es como se regresa desde una función.
    # Aprenderás más sobre esto más adelante.
    # Esta debería ser la última línea de tu programa.
    jr ra

```

El argumento que se pasa a la función se encuentra en la etiqueta `n`. Puedes modificar `n` para probar diferentes factoriales. Para implementar, deberás agregar instrucciones debajo de la etiqueta `factorial`. Existe una solución recursiva, pero recomendamos que implementes la solución iterativa. Puedes asumir que la función `factorial` solo se llamará con valores positivos y con resultados que no desborden un entero de complemento a dos de 32 bits.

Al comienzo de la llamada a `factorial`, el registro `a0` contiene el número cuyo factorial queremos calcular. Luego, coloca tu valor de retorno en el registro `a0` antes de regresar de la función.

**Asegúrate de escribir solo en los registros `t` y `a`. Si usas otros registros, pueden ocurrir cosas extrañas (pronto aprenderás por qué).**

**Además, ¡asegúrate de inicializar los registros que estés utilizando!** Venus podría mostrar que los registros están inicialmente en 0, pero en la vida real pueden contener datos basura. Asegúrate de establecer los valores de los registros que vas a usar a algún número definido antes de usarlos.

## Pruebas

Para probar tu código, puedes asegurarte de que tu función devuelva correctamente la salida esperada. Algunos ejemplos son  $0! = 1$ ,  $3! = 6$ ,  $7! = 5040$  y  $8! = 40320$ .

Para probar tu función, abre `ej5_factorial.s` y ejecútalo en el simulador. Asegúrate de que la prueba se ejecute adecuadamente para los diferentes valores de `n`.

## Entrega

Debes documentar las actividades realizadas dentro de los ejercicios del laboratorio dentro de un informe. El mismo debe contener capturas de pantalla del trabajo realizado y las respuestas. Adicionalmente debes entregar los archivos resultantes, con las soluciones.

Todo esto debe estar dentro de una estructura de carpeta, y zipeado. El archivo zip debe estar denominado de la siguiente manera: `V_<No. de CEDULA>_<Primer Nombre>_<Primer Apellido>.zip`.