PLab1

1.0 Simple programs with core instruction set (I)

1.0.1 Program sections

In this lab we are going to program some simple arithmetical functions to show essential **arithmetic** and flow **control instructions**. We start by the explanation of different program **sections**.

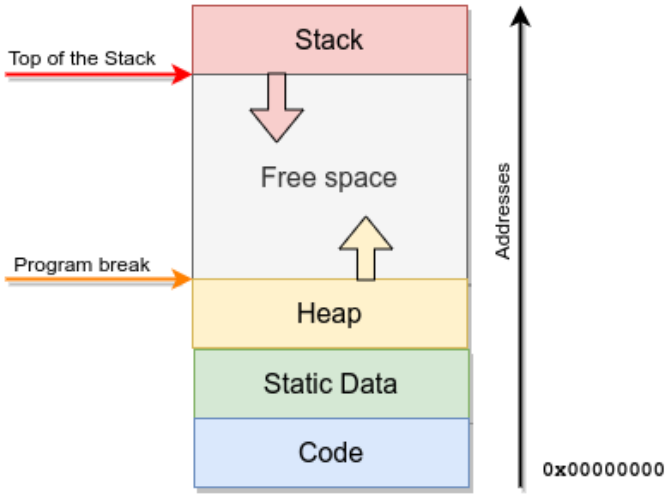

Fig 1.1 Memory layout of the program built from several sections:

- Code - `.text`,
- Static Data (read-write) `.data`, `(read only)` `.rodata`,
- Heap - `.bss`.

In RISC-V assembly programming, different **sections** help organize code, data, and uninitialized storage in memory. These sections follow conventions found in many assembly languages to structure a program effectively. Here's an explanation of the commonly used sections:

1. `.data` Section

The `.data` section is used for declaring initialized data, like constants and global variables, that should be stored in memory. Data in this section is **writable**, meaning it can be **modified at runtime**.

Example:

```
.data
message: .asciiz "Hello, RISC-V!"    # A null-terminated string
value:   .word  10                   # A 32-bit integer initialized to 10
array:   .word  1, 2, 3, 4, 5        # An array of 5 integers
```

`message`: A null-terminated string stored in memory.

- `value`: A 32-bit integer initialized to 10.
- `array`: An array of integers initialized to values 1, 2, 3, 4, and 5.

2. `.text` Section

The `.text` section contains the actual program code (instructions). This section is **typically read-only** and is where the main program logic, functions, and procedures are defined.

```
---------------------------------------------------------------------------
.text
.global _start     # Entry point of the program
_start:
    .option norelax
    la gp, __global_pointer$
    li t0, 10      # Load immediate value 10 into register t0
    la a0, message # Load address of 'message' into a0 for printing
    # Further instructions can follow here, e.g., calls to system functions
---------------------------------------------------------------------------
```

`_start:` The entry point label for the program.

- `li t0, 10`: Loads the **immediate value** 10 into register `t0`.
- `la a0, message:` Loads the **address** of the `message` string into register `a0`.

3. `.bss` Section

The `.bss` section (**Block Started by Symbol**) is used for declaring variables that are **uninitialized** at compile time. The memory for these variables is allocated at runtime and initialized to zero by default. This section is typically used for large **arrays** or **buffers**.

```
-------------------------------------------------------------------------------
.bss
buffer: .space 100      # Reserve 100 bytes for 'buffer', initialized to 0
count:  .word 0         # Reserve space for a 32-bit integer initialized to 0
-------------------------------------------------------------------------------
```

- `buffer:` Reserves 100 bytes of uninitialized space, automatically set to zero**.**
- `count`: Reserves space for a 32-bit integer, also set to zero by default.

---

4. Other Sections

`.rodata` Section (Read-Only Data)

The `.rodata` section is used for constants or **read-only data**. Unlike `.data`, data in `.rodata` **cannot be modified at runtime**.

```
-------------------------------------------------------------------------------
.rodata
pi: .float 3.14159    # A read-only floating-point constant
-------------------------------------------------------------------------------
```

Full Program Example

Here's a simple RISC-V program that uses `.data`, `.text`, and `.bss` sections.

```
-----------------------------------------------------------------------------------------
.data
message: .ascii "The count is: "   # String to print
init_val: .word 5                   # Initialized integer value

.bss
buffer: .space 100                  # Uninitialized space for buffer
count:  .word 0                     # Uninitialized integer

.text
.global _start                      # Main entry point

_start:
    .option norelax
    la  gp, __global_pointer$       # preparing global pointer in gp register
    la a0, message                  # Load address of 'message' into a0
    li a1, 10                       # Load the number 10 into register a1
    sw a1, count                    # Store the value of a1 into 'count'
    # Further program logic here
-----------------------------------------------------------------------------------------
```

1.0.2 Assembly program function system calls

The **IO communication** between the user program and operating systems is done via **function-system calls**. In the first PLab we also use the integrated **C I/O functions** such as `scanf()` and `printf()` to facilitate the input and the output of the data on user terminal.

In second **PLab** we try to **communicate directly** (**I/0**)via **basic system calls**.

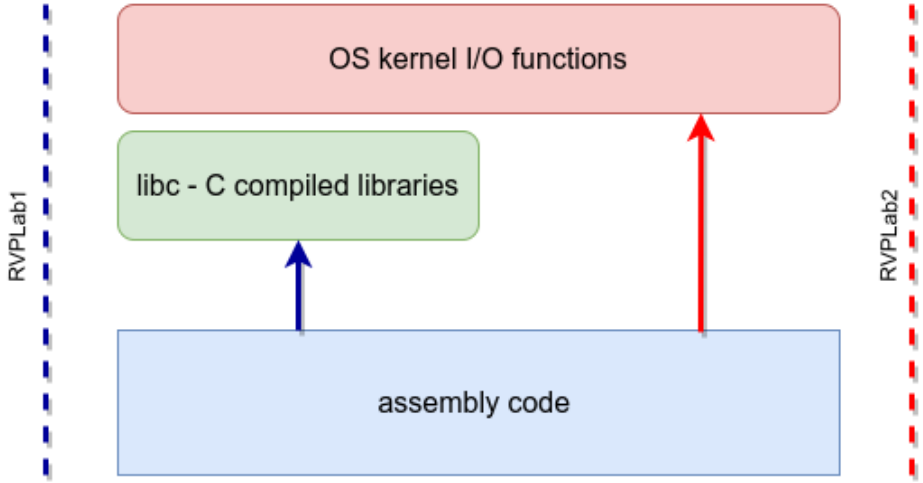

Fig 1.2 I/O calls with C I/O functions (**PLab1**) and direct system I/O functions (**PLab2**)

1.1 Our first program with compilation : `HelloRiscV.s`

Below is an example of a **simple RISC-V assembly program** that prints **`"Hello, RISC-V!"`** on the terminal using **`printf`** - system call function. Since RISC-V doesn't natively have a **`printf`** instruction, we typically use the standard **C library** function to do this by setting up the arguments and making a call to **`printf`**.

The following program uses the C standard library, so it relies on linking with the **`libc`**. To make the **`printf`** call, we need to load the address of the string message and use an environment call (**`ecall`**).

```
---------------------------------------------------------------------------------------------------------------------------------
    .section .data
message:
    .asciz "Hello, RISC-V\n"  # Null-terminated string to print

    .section .text
    .globl _start

# Entry point of the program
main:
    # Load address of the message into register a0 (1st argument of printf)
    la      a0, message        # a0 = address of the message
    # Call printf function
    call    printf             # printf call for C implemented function

    # Exit the program
    li      a7, 93             # a7 = 10 (environment call for exit)
    ecall                      # Make ecall to exit

    # Define a start label to ensure compatibility
_start:
    .option norelax
    la  gp, __global_pointer$
    j main                     # Jump to main
---------------------------------------------------------------------------------------------------------------------------------
```

To do:

Analyze the presented code.

Compile (assembly and link) and run the code with:

```
$gcc HelloRiscV.s -nostartfiles -o HelloRiscV
$./HelloRiscV
```

Note that we have to use `gcc` compiler providing standard C library with `prinf`, `scanf`, etc.

1.2 Unsigned multiplication with addition and shift instructions

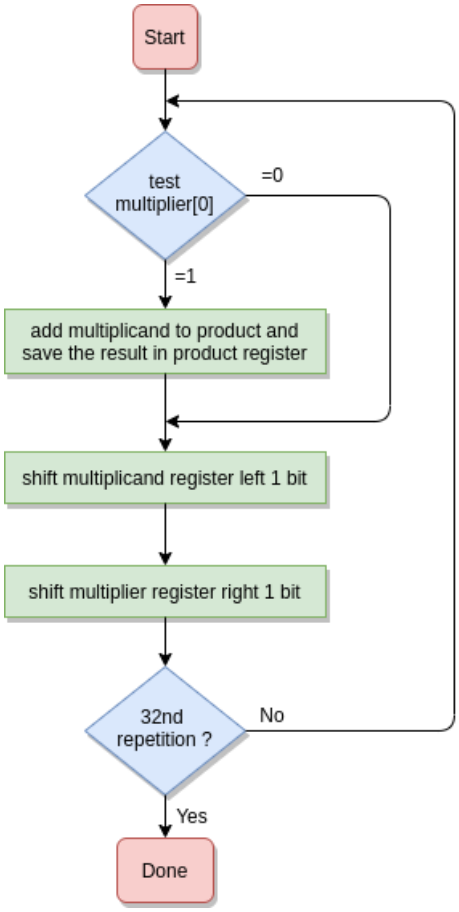

Fig 1.3 The **instructions flow** for unsigned multiplication with addition and shift instructions

The following is assembly function that multiplies **two unsigned 32-bit integers** using only addition and shift instructions. This approach implements a simple version of the **shift-and-add** algorithm (similar to **long multiplication**), which is especially useful for constrained systems that do not have a built-in multiply instruction.

Concept of the Algorithm

- The basic idea is to iterate through each bit of one of the integers (the multiplier).
- For each bit that is set, the other integer (the multiplicand) is added to the result.
- The multiplicand is shifted left for each bit position, while the multiplier is shifted right.
- This process is repeated until all bits of the multiplier have been processed.

The function will take two input arguments in registers `a0` and `a1` and return the result in `a0`.

```
---------------------------------------------------------------------------------------------------------------------------------------
    .text
    .globl multiply_unsigned

# Function: multiply_unsigned
# Description:
#   Multiplies two unsigned integers using only add and shift instructions.
# Arguments:
#   a0: The first unsigned integer (multiplicand)
#   a1: The second unsigned integer (multiplier)
# Returns:
#   a0: The product of the two unsigned integers

multiply_unsigned:
    li t0, 0                # Initialize result to 0 (stored in t0)
    li t1, 1                # Mask to test each bit of the multiplier
loop:
    and t2, a1, t1          # Check if the current bit of the multiplier is set
    beq t2, zero, skip_add  # If the bit is not set, skip the addition
```

```
    add t0, t0, a0          # Add the multiplicand to the result
skip_add:
    sll a0, a0, 1           # Shift the multiplicand left by 1 ( multiplying by 2)
    srl a1, a1, 1           # Shift the multiplier right by 1 to process the next bit
    bnez a1, loop           # If the multiplier is not zero, continue the loop
    mv a0, t0               # Move the result from t0 to a0 (return value)
    ret                     # Return to caller
```

Explanation

1. **Initialization**:
    - `li t0, 0`: Set `t0` to `0`. This register will **accumulate** the result of the multiplication.
    - `li t1, 1`: Set `t1` as the mask to isolate each **bit of the multiplier** (`a1`).
2. **Loop**:
    - **Bit Checking**:
        - `and t2, a1, t1`: Check if the **least significant bi**t (LSB) of the multiplier (`a1`) is set.
        - `beq t2, zero, skip_add`: If the LSB is `0`, skip the addition step.
    - **Addition**:
        - `add t0, t0, a0`: If the bit is set, add the current value of the multiplicand (`a0`) to the result (`t0`).
    - **Shifting**:
        - `sll a0, a0, 1`: Shift the multiplicand (`a0`) left by 1, which is equivalent to multiplying it by 2.
        - `srl a1, a1, 1`: Shift the multiplier (`a1`) right by 1, effectively moving to the next bit.
    - **Loop Condition**:
        - `bnez a1, loop`: If the multiplier (`a1`) **is not zer**o, continue the `loop`.
3. **Return**:
    - `mv a0, t0`: Move the result from `t0` to `a0`, which is the **standard return register**.
    - `ret`: Return to the caller.
    -

How the Algorithm Works

- **Bitwise Processing**: The algorithm processes each bit of the multiplier (`a1`) from least significant to most significant. For each bit that is set (`1`), the current value of the multiplicand (`a0`) is added to the result.
- **Shift Operations**: The multiplicand is shifted left by one position (`sll`) in each iteration to align with the next bit of the multiplier. Similarly, the multiplier is shifted right (`srl`) to check the next bit.
- **Accumulation**: The result (`t0`) accumulates the sum of all **partial products**.

Example

Consider multiplying 3 (`00000011` in binary) and 5 (`00000101` in binary):

- Initially, `a0 = 3` (multiplicand), `a1 = 5` (multiplier), and `t0 = 0` (result).
- The bits of 5 are processed as follows:
    - **Bit 0 (LSB)**: Set (`1`), so `t0 = t0 + a0 = 0 + 3 = 3`.
    - **Shift**: `a0` is shifted left to `6`, `a1` is shifted right to `2`.
    - **Bit 1**: Not set (`0`), **no addition**.
    - **Shift**: `a0` is shifted left to `12`, `a1` is shifted right to `1`.
    - **Bit 2**: Set (`1`), so `t0 = t0 + a0 = 3 + 12 = 15`.
    - **Shift**: `a0` is shifted left to `24`, `a1` is shifted right to `0`.
- The loop ends, and the result (`t0 = 15`) is returned in `a0`.

Notes

- This implementation is for unsigned multiplication.
- The result is returned in **`a0`**, which is the **standard convention** for returning values in RISC-V.
- The function uses **bitwise shifts** and **conditional addition** to perform multiplication, making it suitable for systems without hardware multiply support.

This assembly code provides an efficient way to multiply two integers using only basic instructions, which is **particularly useful for simple microcontrollers** or environments where a hardware multiplier is not available.

To do:

Analyze the code and understand the principle of operation.

Compile the function with:

```
$ gcc u_mult.s -nostartfiles -c u_mult.o
gcc: warning: u_mult.o: linker input file unused because linking not done
```

At this **stage** we may have the following files in our working directory (**`RVPLabs`**):

```
/RVLabs/RVPLabs/lab1$ ls -l
total 24
-rwxrwxr-x 1 bako bako 6680 11月 29 16:24 HelloRiscV
-rw-rw-r-- 1 bako bako  651 11月 29 16:23 HelloRiscV.s
-rw-rw-r-- 1 bako bako 1176 11月 29 16:31 u_mult.o
-rw-rw-r-- 1 bako bako 1095 11月 29 16:30 u_mult.s
```

1.3 Assembly with `scanf` and `printf` (C functions)

The following is an example of a RISC-V RV64 assembly program that **reads an integer** from the user using the **`scanf`** function provided by the operating system and then outputs it using the **`printf`** function. This program assumes **we are working with the standard C library** (**`libc`**), which provides **`scanf`** and **`printf`**.

```
---------------------------------------------------------------------------------------
    .data
    fmt_scanf:  .asciz "%ld"        # Format string for scanf (reading a long integer)
    fmt_printf: .asciz "You entered: %ld\n"  # Format string for printf
    number:     .quad 0             # Reserve space for a 64-bit integer
    .text
    .globl _start

_start:
    .option norelax
    la  gp, __global_pointer$
    # Step 1: Read an integer from the user using scanf
    # Load the address of the format string ("%ld") into a0 (first argument for scanf)
    la a0, fmt_scanf            # First argument for scanf (the format string)
    # Load the address of the variable 'number' into a1 (second argument for scanf)
    la a1, number               # Second argument for scanf (address of the variable)
    # Call scanf
    call scanf                  # Use the provided scanf function to read input
    # Step 2: Print the entered integer using printf
    # Load the address of the printf format string ("You entered: %ld\n") into a0
    la a0, fmt_printf           # First argument for printf (format string)
    # Load the value of 'number' into a1 (second argument for printf)
    ld a1, number               # Load the integer value from memory (64-bit integer)
    # Call printf
    call printf                 # Use the provided printf function to print output
    # Step 3: Exit the program
    li a7, 93                   # syscall number for exit (in RV64)
    li a0, 0                    # exit code 0 (success)
    ecall                       # Exit the program
---------------------------------------------------------------------------------------
```

Explanation:

1. **Data Section (`.data`):**
   - **`fmt_scanf`**: This is the format string for **`scanf`**, specifying that we want to read a 64-bit integer (**`"%ld"`**).
   - **`fmt_printf`**: This is the format string for `printf`, specifying that we want to print a 64-bit integer with a message (**`"You entered: %ld\n"`**).
   - **`number`**: This reserves 8 bytes of memory (64 bits) to store the integer input from the user.
2. **Text Section (`.text`):**
   - **Step 1: `scanf` call:**
     - Load the format string for **`scanf`** into register **`a0`** (first argument).
     - Load the address of the variable `number` (where the input will be stored) into register **`a1`** (second argument).
     - Call the **`scanf`** function using the **`call scanf`** instruction. This reads a 64-bit integer from the user and stores it in the memory location pointed to by `number`.
   - **Step 2: `printf` call:**
     - Load the format string for **`printf`** into register **`a0`** (first argument).
     - Load the value of the `number` variable from memory into register **`a1`** (second argument).
     - Call the **`printf`** function using the **`call printf`** instruction to print the message and the integer value.
   - **Step 3: Exit the Program:**
     - After printing the integer, the program exits using the **`exit`** **syscall** (**`93`**).

To do:

Analyze the code. Assemble with **`gcc`**

```
$ gcc scanf_and_printf.s -nostartfiles -o scanf_and_printf
$./scanf_and_printf
21
You entered: 21
```

1.4 Assembly multiple files (`u_mult.s` and `scanf_and_printf.s`)

Below is a simple main program in RISC-V assembly that calls the previously defined **`multiply_unsigned`** function to multiply two unsigned integers. The main program sets up the arguments, calls the function, and then makes use of a simple mechanism (like storing the result in memory) to verify that the multiplication works correctly.

Assumptions

- The **`multiply_unsigned`** function is defined in the same file or linked appropriately.
- We use registers **`a0`** and **`a1`** to pass arguments to the function and `a0` to receive the result, following RISC-V calling conventions.
- The main program multiplies two unsigned integers: **6** and **7**.

```
-------------------------------------------------------------------------------------------------------------------------------
    .data
result: .word 0           # Reserve space to store the result of the multiplication

    .text
    .globl _start

# Main Program
_start:
    .option norelax
    la  gp, __global_pointer$
    # Load values to multiply into a0 and a1
    li a0, 6              # Load the first operand (multiplicand) into a0
    li a1, 7              # Load the second operand (multiplier) into a1
    # Call the multiply_unsigned function
    jal ra, multiply_unsigned
    # Store the result in memory for verification
    la t0, result         # Load address of result into t0
    sw a0, 0(t0)          # Store the result (in a0) into the memory location
    # Exit the program
    li a7, 93             # Load the exit ecall code (93) into a7
    ecall                 # Make the exit ecall to terminate the program

multiply_unsigned:
    li t0, 0                # Initialize result to 0 (stored in t0)
    li t1, 1                # Mask to test each bit of the multiplier
loop:
    and t2, a1, t1          # Check if the current bit of the multiplier is set
    beq t2, zero, skip_add  # If the bit is not set, skip the addition
    add t0, t0, a0          # Add the multiplicand to the result
skip_add:
    sll a0, a0, 1           # Shift the multiplicand left by 1 ( multiplying by 2)
    srl a1, a1, 1           # Shift the multiplier right by 1 to process the next bit
    bnez a1, loop           # If the multiplier is not zero, continue the loop
    mv a0, t0               # Move the result from t0 to a0 (return value)
    ret                     # Return to caller
-------------------------------------------------------------------------------------------------
```

Explanation

1. **Data Section**:
    - **`result: .word 0`**: Reserves a word of space to store the result of the multiplication for verification.
2. **Text Section (`_start`)**:
    - **`_start`** is the entry point of the program, **equivalent** to **`main()`** in high-level languages.
    - **Load Arguments**:
        - **`li a0, 6`**: Loads the value **6** into register **`a0`**.
        - **`li a1, 7`**: Loads the value **7** into register **`a1`**.
    - **Call the Multiply Function**:
        - **`jal ra, multiply_unsigned`**: Calls the **`multiply_unsigned`** function. The **return address** is saved in **`ra`**.
    - **Store the Result**:
        - **`la t0, result:`** Loads the address of the **`result`** into **`t0`**.
        - **`sw a0, 0(t0)`**: Stores the result from `a0` into the memory address held by **`t0`**.
    - **Exit Program**:

- `li a7, 93`: Load the exit system call code (**93**) into `a7`.
- `ecall`: Execute the system call to terminate the program.

3. **Multiply Function (`multiply_unsigned`)**:
   - Implements the multiplication using a simple shift-and-add algorithm, as discussed previously.
   - Uses addition and shifting operations to compute the product, storing the result in `a0`.

Key Points

- **Calling Convention**:
  - The **function arguments** are passed via `a0` and `a1`.
  - The **result** is returned in `a0`.
  - The **return address** is stored in the `ra` register.
- **Register Usage**:
  - `t0` is used as an **accumulator** to store the result.
  - `t1` serves as a **mask** to check each bit of the multiplier.
  - `t2` is used for **temporary** calculations.
- **Exiting the Program**:
  - The **exit** system call (**93**) is used to terminate the program after storing the result.

Running the Program

- The code will multiply **6** and **7**, resulting in **42**.
- The **result** is stored in the memory location labeled `result`.

To do

Analyze and test the above code.

Use `scanf_and_printf.s` program and add the call to `u_mult()` function as external function.

`mult_unsigned_fun.s` and `main_mult_unsigned_fun.s`

Add `scanf` and `printf` functions to read multipler-multiplicand and print the `result`.

```
    .text
    .globl multiply_unsigned

multiply_unsigned:
    li t0, 0                # Initialize result to 0 (stored in t0)
    li t1, 1                # Mask to test each bit of the multiplier
loop:
    and t2, a1, t1          # Check if the current bit of the multiplier is set
    beq t2, zero, skip_add  # If the bit is not set, skip the addition
    add t0, t0, a0          # Add the multiplicand to the result
skip_add:
    sll a0, a0, 1           # Shift the multiplicand left by 1 (multiplying by 2)
    srl a1, a1, 1           # Shift the multiplier right by 1 to process the next bit
    bnez a1, loop           # If the multiplier is not zero, continue the loop
    mv a0, t0               # Move the result from t0 to a0 (return value)
    ret                     # Return to caller
```

```
    .data
    fmt_scanf:  .asciz "%ld"        # Format string for scanf (reading a long integer)
    fmt_printf: .asciz "Product is: %ld\n"  # Format string for printf
    multiplier:     .quad 0             # Reserve space for a 64-bit integer
    multiplicand:     .quad 0             # Reserve space for a 64-bit integer
    result:     .quad 0             # Reserve space for a 64-bit integer
    .text
    .globl _start
    .extern u_mult

_start:
    .option norelax
    la  gp, __global_pointer$
    # Step 1: Read two integers from the user using scanf
    # Load the address of the format string ("%ld") into a0 (first argument for scanf)
    la a0, fmt_scanf            # First argument for scanf (the format string)
    # Load the address of the variable 'multiplier' into a1 (second argument for scanf)
```

```
    la a1, multiplier               # Second argument for scanf (address of the variable)
    # Call scanf
    call scanf                  # Use the provided scanf function to read input
    la a0, fmt_scanf            # First argument for scanf (the format string)
    # Load the address of the variable 'multiplicand' into a1 (second argument for scanf)
    la a1, multiplicand               # Second argument for scanf (address of the variable)
    # Call scanf
    call scanf                  # Use the provided scanf function to read input
    la  t1, multiplier
    ld  a0, 0(t1)
    la  t1, multiplicand
    ld  a1, 0(t1)
    call u_mult
    la  t1, result
    sd  a0, 0(t1)                  # load number - result
    mv  a1, a0
    # Print the  integer using printf
    # Load the address of the printf format string ("Product is: %ld\n") into a0
    la a0, fmt_printf           # First argument for printf (format string)
    # Load the value of 'number' into a1 (second argument for printf)
    # Call printf
    call printf                 # Use the provided printf function to print output
    # Step 3: Exit the program
    li a7, 93                   # syscall number for exit (in RV64)
    li a0, 0                    # exit code 0 (success)
    ecall                       # Exit the program
```

Assembly and execute the program:

```
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ gcc u_mult.s scanf_u_mult_printf.s -nostartfiles -o
scanf_u_mult_printf
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ ./scanf_u_mult_printf
4
6
Product is: 24
```

1.5 Power function with simple multiplication

Below we show RISC-V assembly function to calculate the **power function** using **multiplication** instructions. The function computes **`pow = a^b`** using the **`mul`** instruction, which is part of the **RISC-V M-extension for multiplication**.

```
-------------------------------------------------------------------------------------
    .text
    .globl u_power

# Function: u_power
# Description:
#   Computes the value a^b using a loop and the mul instruction.
# Arguments:
#   a0: The base (a)
#   a1: The exponent (b)
# Returns:
#   a0: The result of a^b

u_power:
    li t0, 1                # Initialize result to 1 (t0 will hold the result)
    # Loop while exponent (a1) > 0
power_loop:
    beq a1, zero, power_end # If exponent is 0, end the loop
    # Multiply t0 by a0
    mul t0, t0, a0          # t0 = t0 * a0
    # Decrement the exponent
    addi a1, a1, -1         # Decrement a1 (exponent) by 1
    # Repeat the loop
    j power_loop            # Jump back to start of loop
power_end:
    mv a0, t0               # Move the final result to a0 (return value)
    ret                     # Return to caller
-------------------------------------------------------------------------------------
```

Explanation

1. **Initialization**:
   - **`li t0, 1`**: Load the value `1` into register **`t0`**. This is the initial value of the result because anything raised to the power of `0` is `1`.
   - **Why initialize with `1`?**: Starting with `1` ensures that the multiplication does not accidentally result in `0` if the base (**`a0`**) is multiplied with an uninitialized or zero value.
2. **Exponent Zero Check**:
   - **`beq a1, zero, power_end`**: If the exponent (**`a1`**) is `0`, the function directly skips to **`power_end`**, and the result (`1`) is returned.
3. **Power Loop (`power_loop`)**:
   - The function uses a loop to multiply the result (**`t0`**) by the base (**`a0`**) until the exponent (**`a1`**) reaches zero.
   - **Multiplication Step**:
     - **`mul t0, t0, a0`**: Multiply **`t0`** by **`a0`**. Initially, **`t0`** is `1`, so the first iteration of the loop sets **`t0`** to the value of the base (**`a0`**). For subsequent iterations, **`t0`** continues accumulating the result.
   - **Decrement Exponent**:
     - **`addi a1, a1, -1`**: Decrement the value in **`a1`** by `1`. This reduces the exponent on each loop iteration.
   - **Loop Condition**:
     - **`bnez a1, power_loop`**: If the value in **`a1`** is not zero (**`a1 != 0`**), the loop continues.
4. **Return Result (`power_end`)**:
   - **`mv a0, t0:`** Move the value from **`t0`** to **`a0`**. This is because **`a0`** is the **register used to return the result in RISC-V** calling conventions.
   - **`ret`**: Return from the function to the caller.

Key Details

Multiplication Instruction (`mul`):

- The **`mul`** instruction is used to **multiply the base** (**`a0`**) by the **accumulated result** (**`t0`**) in each iteration. This is efficient compared to manually using addition and shifts.

Register Usage:

- **`a0`**: Holds the **base** value (a) and returns the result.
- **`a1`**: Holds the **exponent** value (b).
- **`t0`**: Holds the **running result** of the power calculation.
- The use of **temporary registers** like **`t0`** helps avoid overwriting input values and allows for iterative accumulation.

Edge Cases:

- **Exponent = 0**: The function checks if the exponent (**`a1`**) is **`0`** at the start and directly **returns `1` if true**.
- **Exponent > 0**: The function uses a loop to multiply until the exponent (**`a1`**) becomes **`0`**.

To do:

Analyze the code with its explanation.

Write scanf_u_power_printf.s program, compile and test it.

```
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ gcc u_power.s scanf_u_power_printf.s -nostartfiles -o
scanf_u_power_printf
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ ./scanf_u_power_printf
4
4
Power is: 256
```

1.6 Power function with Exponentiation and Squaring

Below is the RISC-V assembly function that calculates the power of a number using the **Exponentiation by Squaring** method. This method is more efficient than a simple iterative multiplication approach, especially for larger exponents, as it reduces the number of multiplication operations required.

1.6.1 Exponentiation by Squaring

Exponentiation by squaring is based on the mathematical observation:

- **If b is even: $a^b=(a^{b/2})^2$**
- **If b is odd: $a^b=a\times a^{b-1}$**

This approach significantly reduces the number of multiplications required for large exponents, by taking advantage of squaring whenever possible.

```
-----------------------------------------------------------------------------------------------------------------------------------
    .text
    .globl es_power

# Function: power
# Description:
#   Computes the value a^b using exponentiation by squaring and the mul instruction.
# Arguments:
#   a0: The base (a)
#   a1: The exponent (b)
# Returns:
#   a0: The result of a^b

es_power:
    li t0, 1                # Initialize result to 1 (t0 will hold the final result)
    # Check if the exponent is zero
    beq a1, zero, power_end # If exponent is 0, skip to the end (result is 1)
power_loop:
    andi t1, a1, 1          # Check if the current exponent is odd (t1 = a1 & 1)
    beq t1, zero, skip_mul  # If t1 is 0, skip multiplication (even exponent)
    # If exponent is odd, multiply result by base
    mul t0, t0, a0          # t0 = t0 * a0
skip_mul:
    mul a0, a0, a0          # Square the base: a0 = a0 * a0
    srl a1, a1, 1           # Divide the exponent by 2: a1 = a1 >> 1
    bnez a1, power_loop     # If exponent is not zero, continue the loop
power_end:
    mv a0, t0               # Move the final result to a0 (return value)
    ret                     # Return to caller
-----------------------------------------------------------------------------------------
```

To do

Integrate the above "**power**" functions into main program with **`scanf`** and **`printf`** functions.

```
-----------------------------------------------------------------------------------------
    .data
    fmt_scanf:  .asciz "%u"        # Format string for scanf (reading a long integer)
    fmt_printf: .asciz "Power (a^b) is: %u\n"  # Format string for printf
    result: .word 0           # Reserve space to store the result of the multiplication
    base: .word 0           # Reserve space to store the result of the multiplication
    exponent: .word 0           # Reserve space to store the result of the multiplication
    .text
    .globl _start
    .extern power
    .extern power_sq
# Main Program
_start:
    .option norelax
    la gp, __global_pointer$
    # Load the address of the format string ("%ld") into a0 (first argument for scanf)
    la a0, fmt_scanf            # First argument for scanf (the format string)
    la a1, base         # Second argument for scanf (address of the variable)
    call scanf
    la a0, fmt_scanf            # First argument for scanf (the format string)
    la a1, exponent           # Second argument for scanf (address of the variable)
    call scanf
    la  t1, base
    lw  a0, 0(t1)
    la  t1, exponent
    lw  a1, 0(t1)
    # Call the"power"  functions
    # call  power
    call  es_power
    mv t0, a0
    # Load the address of the printf format string ("You entered: %ld\n") into a0
```

```
    la a0, fmt_printf           # First argument for printf (format string)
    mv a1, t0               # Load the product  value
    # Call printf
    call printf                 # Use the provided printf function to print output
    # Exit the program
    li a7, 93             # Load the exit ecall code (93) into a7
    ecall                 # Make the exit ecall to terminate the program
```

To do

Compile and **execute** the above program.

```
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ gcc es_power.s scanf_es_power_printf.s -
nostartfiles -o scanf_es_power_printf
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ ./scanf_es_power_printf
4
4
Power is: 256
```

1.7 Factorial function – iterative method

The following is a RISC-V assembly function to calculate the factorial of an integer using an **iterative method** and **multiplication instructions**. The factorial function computes:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

The iterative approach starts from 1 and multiplies up to n to compute the factorial value.

```
-----------------------------------------------------------------------------------------
    .text
    .globl factorial
# Function: factorial
# Description:
#   Computes the factorial of a given unsigned integer n using an iterative method.
# Arguments:
#   a0: The input number n
# Returns:
#   a0: The result of n!

factorial:
    li t0, 1                # Initialize result to 1 (t0 will hold the factorial value)
    # Check if n is zero or one (0! and 1! are both 1)
    beq a0, t1, factorial_end # If a0 (n) <= 1, skip to the end (result is 1)
factorial_loop:
    mul t0, t0, a0          # t0 = t0 * a0 (multiply the current result by n)
    addi a0, a0, -1         # a0 = a0 - 1 (decrement n by 1)
    bnez a0, factorial_loop # If a0 != 0, continue looping
factorial_end:
    mv a0, t0               # Move the final result to a0 (return value)
    ret                     # Return to caller
-----------------------------------------------------------------------------------------
```

Explanation

1. **Initialization**:
    - **Set result to `1`**:
        - **`li t0, 1`**: The initial result is set to `1` in **`t0`**, since the factorial calculation always starts with `1`.
2. **Check for Base Case (`0!` or `1!`)**:
    - **`ble a0, t1, factorial_end`**: If **`a0`** (the input n) is less than or equal to `1`, the factorial value is `1`. In this case, the function directly jumps to **`factorial_end`**.
3. **Factorial Loop (`factorial_loop`)**:
    - **Multiplication Step**:
        - **`mul t0, t0, a0`**: Multiply the current value of **`t0`** by the current value of **`a0`**. This accumulates the result.
    - **Decrement `a0`**:
        - **`addi a0, a0, -1`**: Decrement the value of **`a0`** by `1` to proceed to the next iteration.
    - **Loop Condition**:
        - **`bnez a0, factorial_loop`**: Continue the loop if **`a0`** is not `0`.
4. **Return the Result (`factorial_end`)**:
    - **`mv a0, t0`**: Move the final result (**`t0`**) to **`a0`** to be returned to the caller.
    - **`ret`**: Return from the function.

Example of Factorial Calculation

Let's take the example of calculating **`5!`**:

- **Initialization**:
    - **`t0`** (result) = `1`
    - **`a0`** (input n) = 5
- **Loop Iterations**:
    - **Iteration 1**:
        - **Multiply**: **`t0 = 1 * 5 = 5`**

        - **Decrement**: `a0 = 5 - 1 = 4`
    - **Iteration 2**:
        - **Multiply**: `t0 = 5 * 4 = 20`
        - **Decrement**: `a0 = 4 - 1 = 3`
    - **Iteration 3**:
        - **Multiply**: `t0 = 20 * 3 = 60`
        - **Decrement**: `a0 = 3 - 1 = 2`
    - **Iteration 4**:
        - **Multiply**: `t0 = 60 * 2 = 120`
        - **Decrement**: `a0 = 2 - 1 = 1`
    - **Iteration 5**:
        - **Multiply**: `t0 = 120 * 1 = 120`
        - **Decrement**: `a0 = 1 - 1 = 0`
- **Result**: The loop ends when **`a0`** becomes `0`, and **`t0`** holds the **final result of `120`**.

Key Details

1. **Edge Case Handling**:
    - **`0!` and `1!`:** The factorial of **`0`** and **`1`** is **`1`**. The **`ble`** (**branch if less than or equal**) instruction ensures that the function immediately returns `1` for these values without entering the loop.
2. **Register Usage**:
    - **`a0`**: Initially holds the input **n** and returns the final factorial value.
    - **`t0`**: Holds the running result of the factorial calculation.
    - This approach minimizes register usage, making it straightforward and efficient.

To do:

Analyze the above code. Compile and execute it:

```
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ gcc factorial.s scanf_factorial_printf.s -nostartfiles -o
scanf_factorial_printf
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ ./scanf_factorial_printf
6

Factorial (a!) is: 720
-------------------------------------------------------------------------------------
```

1.8 Factorial function – recursive method

Below is a RISC-V assembly function to calculate the factorial of an integer using a **recursive method** and **multiplication instructions**. In a recursive implementation of the factorial function:

$$n! = n \times (n-1)!$$

With base cases:

- $0! = 1$
- $1! = 1$

Important Considerations

- This recursive method makes use of the **call stack** to store intermediate values.
- Each recursive call r**educes n by 1** until reaching the base case (`n = 0` or `n = 1`).

```
    .section .text
    .globl factorial_rec

# Function to calculate factorial recursively for RV64
factorial_rec:
    # Base Case: if (n == 0) return 1
    addi    t0, a0, 0           # Copy a0 to t0 (n to t0)
    beqz    t0, factorial_base  # If t0 == 0, jump to base case
    # Recursive Case: n * factorial(n-1)
    addi    sp, sp, -32         # Allocate space on the stack (RV64 uses 8-byte alignment)
    sd      ra, 24(sp)          # Save return address on the stack
    sd      a0, 16(sp)          # Save n (a0) on the stack
    addi    a0, a0, -1          # Calculate n - 1
    jal     ra, factorial_rec       # Recursive call: factorial(n-1)
    # Multiply result by n (saved on stack)
    ld      t1, 16(sp)          # Load saved n from stack
    mul     a0, a0, t1          # a0 = a0 * t1 (factorial(n-1) * n)
    ld      ra, 24(sp)          # Restore return address
    addi    sp, sp, 32          # Deallocate space from the stack
    jr      ra                  # Return to caller
factorial_base:
    # Base Case: return 1
    li      a0, 1               # a0 = 1 (factorial(0) = 1)
    jr      ra                  # Return to caller
```

Explanation

1. **Base Case Check**:
    - **Set `t0 = 1`**:
        - `li t0, 1`: Set `t0` to `1` to use it as a reference for the base case.
    - **Branch if Less Than or Equal (`ble`)**:
        - `ble a0, t0, factorial_end`: If **n** (`a0`) is less than or equal to `1`, skip the recursive call and **return `1`**.
2. **Recursive Case**:
    - **Stack Allocation**:
        - `addi sp, sp, -32`: Allocate space on the stack to store the return address (**8 bytes for RV64**).
        - `sd ra, 24(sp)`: Store the return address (`ra`) on the stack so that it can be restored after the recursive call.
    - **Recursive Call Preparation**:
        - `addi a0, a0, -1`: Decrement the value of n by `1` in preparation for the recursive call.
        - `jal ra, factorial_rec`: Make the recursive call to `factorial_rec(n - 1)`. The return address will be stored in `ra`.
    - **Restoring Stack and Multiplying**:
        - `ld ra, 24(sp)`: Load the saved return address from the stack to `ra`.
        - `addi sp, sp, 32`: Restore the stack pointer (`sp`) to its previous value by **deallocating the 8 bytes used for storing `ra`**.
        - **Multiplication**:

- **`mul a0, a0, t1`**: Multiply the current value of n (original value of `a0`) by the result of **`factorial_rec(n - 1)`**, which is in **`t1`**.

3. **Return to Caller**:
    - **`jr`**: Return to the caller, using the restored return address in **`ra`**.

To do

Analyze the above code.

Write test program for factorial functions with **`scanf`** and **`printf`** functions.

```
    .data
    fmt_scanf:  .asciz "%u"        # Format string for scanf (reading a long integer)
    fmt_printf: .asciz "Factorial (a!) is: %u\n"  # Format string for printf
    result: .word 0           # Reserve space to store the result of the multiplication
    base: .word 0           # Reserve space to store the result of the multiplication
    .text
    .globl _start
    .extern factorial
    .extern factorial_rec
# Main Program
_start:
    .option norelax
    la gp, __global_pointer$
    # Load the address of the format string ("%ld") into a0 (first argument for scanf)
    la a0, fmt_scanf            # First argument for scanf (the format string)
    la a1, base         # Second argument for scanf (address of the variable)
    call scanf
    la  t1, base
    lw  a0, 0(t1)
    call  factorial
    #call  factorial_rec
    mv t0, a0
    # Load the address of the printf format string ("You entered: %ld\n") into a0
    la a0, fmt_printf           # First argument for printf (format string)
    mv a1, t0               # Load the product  value
    # Call printf
    call printf                 # Use the provided printf function to print output
    # Exit the program
    li a7, 93             # Load the exit ecall code (93) into a7
    ecall                 # Make the exit ecall to terminate the program
```

To do

Analyze the use of stack (**RV64**) with then, **compile** and **execute** the program.

```
    addi    sp, sp, -32         # Allocate space on the stack (RV64 uses 8-byte alignment)
    sd      ra, 24(sp)          # Save return address on the stack
    sd      a0, 16(sp)          # Save n (a0) on the stack

musepi@musepipro:~/RVLabs/RVPLabs/lab1$ gcc factorial_rec.s scanf_factorial_printf.s -nostartfiles -
o scanf_factorial_rec_printf
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ ./scanf_factorial_rec_printf
6
Factorial (a!) is: 720
```

1.9 Circle surface – using floating point extension (F)

Below is a simple RISC-V assembly program for **RV64G** that calculates the surface area of a circle from a given radius. The formula for the surface area of a circle is:

$$\mathbf{Area = \pi \times r2}$$

In this example, we'll assume the value of **π** as **3.14159265359** and calculate the area using the formula. The result will be stored in **floating-point registers.**

```
    .data
radius:  .float 5.0                # Radius of the circle, adjust this value as needed
pi:      .float 3.1415927          # Approximation of pi in single precision
result:  .float 0.0                # Variable to store the result

    .text
    .global _start

_start:
    # Load the radius and pi into floating-point registers
    flw     ft0, radius            # Load radius (r) into ft0 (single-precision)
    flw     ft1, pi                # Load pi into ft1 (single-precision)
    # Calculate r^2 (square of the radius)
    fmul.s  ft2, ft0, ft0          # ft2 = r * r (single-precision)
    # Calculate area = pi * r^2
    fmul.s  ft3, ft1, ft2          # ft3 = pi * (r * r) (single-precision)
    # Store the result back to memory
    fsw     ft3, result            # Store the result in 'result' (single-precision)
    # Exit the program (using system call for exit)
    li      a7, 93                 # System call number for exit in RISC-V (64-bit)
    ecall                           # Exit the program
```

Explanation:

1. **Data Section**:
   - `radius` contains the value of the radius of the circle (e.g., **5.0** in this example).
   - `pi` contains the approximation of π (**3.14159265359**).
   - `result` is where the computed area will be stored.
2. **Text Section**:
   - `fld` instructions load floating-point numbers (64-bit `double`) into floating-point registers.
   - `fmul.d` is used to multiply floating-point numbers (`double`).
   - The area is calculated using **π×r2** and the result is stored in the `result` memory location.
   - `ecall` is used to exit the program.

To do

Analyze, **assembly** and **test** the program. Add `scanf` and `printf` functions.

The following is the **modified** and **completed** code:

```
    .data
fmt_scanf:  .asciz "%f"        # Format string for scanf (reading a long integer)
radius:  .float 10.0           # Radius of the circle, adjust this value as needed
pi:      .float 3.1415927      # Approximation of pi in single precision
area:  .float 0.0              # Variable to store the result
fmt_printf: .asciz "Surface: %f\n"

    .text
    .global _start

_start:
    .option   norelax
    la  gp, __global_pointer$
    la a0, fmt_scanf            # First argument for scanf (the format string)
    la a1, radius         # Second argument for scanf (address of the variable)
    call scanf

# Load the radius and pi into floating-point registers
    lla     a4 , radius
    flw     ft0, 0(a4)            # Load radius (r) into ft0 (single-precision)
    lla     a5, pi
    flw     ft1, 0(a5)                # Load pi into ft1 (single-precision)
```

```
    # Calculate r^2 (square of the radius)
    fmul.s  ft2, ft0, ft0          # ft2 = r * r (single-precision)

    # Calculate area = pi * r^2
    fmul.s  ft3, ft1, ft2          # ft3 = pi * (r * r) (single-precision)

    # Store the result back to memory
    lla     a4, area
    fsw     ft3, 0(a4)            # Store the result in 'result' (single-precision)

    fcvt.d.s     ft3, ft3
    fmv.x.d      a1, ft3
    la      a0, fmt_printf
    call    printf

    # Exit the program (using system call for exit)
    li      a7, 93                 # System call number for exit in RISC-V (64-bit)
    ecall                           # Exit the program
```

Explanation

The instruction **`fcvt.d.s ft3,ft3`** in RISC-V performs a **floating-point conversion** from a **single-precision (32-bit)** floating-point value to a **double-precision (64-bit)** floating-point value.

Breakdown of the instruction:

- **`fcvt.d.s`**: This is the RISC-V floating-point conversion instruction that converts a value from **single-precision (32-bit)** floating-point (denoted by **`.s`**) to **double-precision (64-bit)** floating-point (denoted by **`.d`**).
- **`ft3`**: This is both the source and destination floating-point register. In this case, the source register contains a single-precision value, and the destination will contain the **converted double-precision value**.

Compilation and execution:

```
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ gcc circle_surface_all.s -nostartfiles -o circle_surface_all
musepi@musepipro:~/RVLabs/RVPLabs/lab1$ ./circle_surface_all
5
Surface: 78.539818
```

1.10 Vector add program with V extension instructions

Our RISC-V board integrates **X60 SoC** from SpacemiT. This SoC implements **standard V** (vector) extension operating on **256-bit vector**s. It means that we can operate in parallel on **8 32-bit** data (integer, floating point) or even on **32 8-bit** data.

The following is an assembly program for RISC-V RV64 with **Vector Extension (RVV)** to add two vectors with 8 elements each, where each element is a 32-bit integer. The program will **load the vectors** into vector registers, perform the **addition using vector instructions**, and **store** the result in memory.

```
-----------------------------------------------------------------------------------------
    .section .data
vector_a:
    .word 1, 2, 3, 4, 5, 6, 7, 8   # First input vector with 8 elements (32-bit integers)

vector_b:
    .word 8, 7, 6, 5, 4, 3, 2, 1   # Second input vector with 8 elements (32-bit integers)

result_vector:
    .space 32                          # Space to store the result (8 elements * 4 bytes)

    .section .text
    .globl _start
_start:
    # Set up the vector length to 8 elements (each 32-bit wide)
    li t0, 8                             # Set vl (vector length) to 8 elements
    vsetvli t0, t0, e32, m1              # Set vector length (VL) to 8 elements, 32-bit wide
    # Load vector_a into vector register v1
    la t1, vector_a                      # Load address of vector_a into t1
    vle32.v v1,0(t1)                 # Load 8 elements (32-bit each) from vector_a into v1
    # Load vector_b into vector register v2
    la t2, vector_b                      # Load address of vector_b into t2
    vle32.v v2,0(t2)                 # Load 8 elements (32-bit each) from vector_b into v2
    # Perform vector addition: v3 = v1 + v2
    vadd.vv v3, v1, v2                   # Add vectors in v1 and v2, store result in v3
    # Store the result vector from v3 into memory
    la t3, result_vector                 # Load address of result_vector into t3
    vse32.v v3,0(t3)               # Store the result (32-bit elements) from v3 into memory
    # Exit the program
    li a0, 0                             # Exit code 0
    li a7, 93                            # Syscall number for exit
    ecall                                # Make the system call
-----------------------------------------------------------------------------------------
```

Explanation:

1. **Data Section**:
   - **`vector_a`** and **`vector_b`** contain two input vectors, each with 8 elements of 32-bit integers.
   - **`result_vector`** is reserved to store the result, with 32 bytes of space (8 elements × 4 bytes each).
2. **Main Program**:
   - **Vector Length Setup**:
     - The **`vsetvli`** instruction sets the **vector length** (**`vl`**) to 8 elements, where each element is 32 bits wide (e32). **`m1`** indicates **single-width elements**.
   - **Vector Load**:
     - The **`vle32.v`** instruction loads 8 elements (32-bit integers) from **`vector_a`** and **`vector_b`** into **vector registers** **`v1`** and **`v2`**.
   - **Vector Addition**:
     - The **`vadd.vv`** instruction performs element-wise addition of the two vectors stored in **`v1`** and **`v2`**, and stores the result in **`v3`**.
   - **Vector Store**:
     - The **`vse32.v`** instruction stores the result from **`v3`** (which contains the 8 element-wise sums) into the memory location pointed to by **`result_vector`**.
3. **Program Exit**:
   - After the addition is complete, the program exits using the **`exit`** system call with **exit code `0`**.

To do

The following, is corrected, code that must be assembled with (V) architectural extension.

Analyze, compile and execute the example. Note that we have added **`printf`** instruction to output the **twice the four elements** of the result vector.

```
-----------------------------------------------------------------------------------------
   .data
vector1:  .word 1, 2, 3, 4, 5, 6, 7, 8          # First 8-word vector
vector2:  .word 1, 2, 3, 4, 5, 6, 7, 8          # Second 8-word vector
result:   .space 32                             # Space for the result (8 x 4 bytes)
result_msg:
    .string   "%d\n"  # Format string for result
format_string1:
    .string "4 values [0-3]: %d,%d,%d,%d\n"       # Format string to print an integer with newline
format_string2:
    .string "4 values [4-7]: %d,%d,%d,%d\n"       # Format string to print an integer with newline

    .text
    .global _start

_start:
    .option     norelax
    la gp, __global_pointer$
    # Set up vector length register(VLEN = 8 elements, each 32 bits)
    li t0, 8                          # vector length is 8
    vsetvli t0,t0,e32,m1    # vector length for 32-bit elements, one operation per element
    # Load the vectors into vector registers
    la t1, vector1                  # Load address of vector1 into t1
    vle32.v v0,0(t1)                # Load 8-word vector1 into vector register v0
    la t2, vector2                  # Load address of vector2 into t2
    vle32.v v1,0(t2)                # Load 8-word vector2 into vector register v1
    # Perform vector addition
    vadd.vv v2, v0, v1              # v2 = v0 + v1 (element-wise vector addition)
    # Store the result back to memory
    la t3, result                   # Load address of result into t3
    vse32.v v2,0(t3)                # Store the result from vector register v2 into memory
    lw  a1,0(t3)                        # load last element to print
    lw  a2,4(t3)                        # load last element to print
    lw  a3,8(t3)                        # load last element to print
    lw  a4,12(t3)                       # load last element to print
    la  a0, format_string1
    call printf
    la t3, result                     # Load address of result into t3
    lw  a1,16(t3)                       # load last element to print
    lw  a2,20(t3)                       # load last element to print
    lw  a3,24(t3)                       # load last element to print
    lw  a4,28(t3)                       # load last element to print
    la  a0, format_string2
    call printf

    # Exit the program (using system call for exit)
    li a0,0
    li a7, 93                       # System call number for exit in RISC-V (64-bit)
    ecall                           # Exit the program
-----------------------------------------------------------------------------------------
```

Note the use of **`-march=rv64gcv` option** to call the vector extension !

```
$gcc vector_add_printf.s -nostartfiles -march=rv64gcv -o vector_add_printf
$./vector_add_printf
4 values: 2,4,6,8
4 values: 10,12,14,16
-----------------------------------------------------------------------------------------
```