**Experiment No 4**

RISC-V assembly programming using Data Transfer, Logical and

Arithmetic Instructions

# Objectives

After completing this experiment, student will be able to:

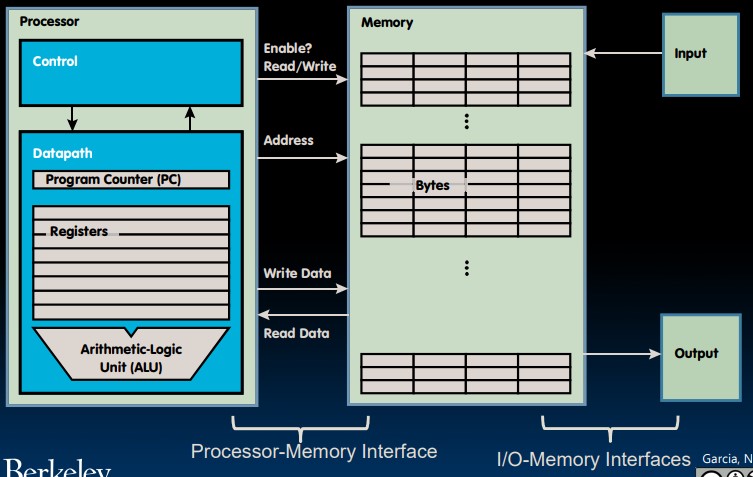
* Explore addi and lui instruction to store a 32 bit constant in a register
* Gain proficiency in using data transfer instructions (**lw** and **sw**) to move data between memory and registers.
* Understand the syntax and functionality of data transfer instructions in RISC-V assembly programming.
* Explore arithmetic instructions (**add**, **sub**, **mul**, and **div**) for performing basic arithmetic operations on registers.
* Explore logical instructions (**and**, **or**, **xor**, and **not**) for performing the bitwise operations on registers.
* Learn how to specify memory addresses and register operands in data transfer and arithmetic instructions.
* Practice writing RISC-V assembly code to load data from memory, perform arithmetic operations, and store results back into memory.
* Enhance understanding of memory organization, register manipulation, and program execution flow in RISC-V architecture.

# Background Theory

This lab on "RISC-V assembly programming using Data Transfer and Arithmetic Instructions in Venus" aims to provide students with a foundational understanding of low-level programming concepts within the RISC-V architecture. RISC-V, as an open-source instruction set architecture, offers a versatile platform for learning assembly language programming. Through this lab, students will explore the fundamentals of data transfer and arithmetic operations, essential for manipulating data within a RISC-V environment. By leveraging instructions like **lw** and **sw** for memory operations and **add**, **sub**, **mul**, and **div** for arithmetic computations, students will gain hands-on experience in moving data between registers and memory, performing basic arithmetic operations, and storing results back into memory. This lab not only familiarizes students with the syntax and usage of these instructions but also cultivates a deeper understanding of memory organization, register manipulation, and the execution flow of RISC-V programs. Through practical exercises conducted in the Venus IDE, students will develop proficiency in writing efficient and effective RISC-V assembly code, laying a solid groundwork for further exploration into computer architecture and systems programming.

# RISC-V Architecture

RISC-V, an open-source instruction set architecture (ISA), embodies the principles of simplicity and modularity, making it highly adaptable across a wide range of computing devices. It features a clean and elegant design with a small set of fixed-length instructions, categorized into base and optional extensions, allowing for flexibility and scalability. The architecture defines a set of registers, including general-purpose registers and special-purpose registers, providing a foundation for efficient data manipulation and control flow. RISC-V supports various addressing modes and instruction formats, facilitating ease of programming and efficient code generation. Its modular design allows for the inclusion of custom extensions tailored to specific application domains, promoting innovation and specialization. Overall, RISC-V architecture stands as a versatile and extensible framework, poised to drive innovation in computing across diverse applications and industries.



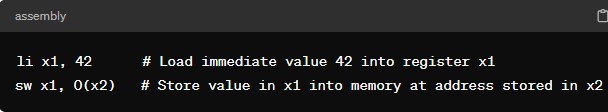
*Figure 4.1: RISC-V Architecture*

# Basics of RISC-V assembly programming

Following are the foundational concepts and instructions for writing RISC-V assembly programs. By combining these elements, you can create programs to perform various computational tasks, manipulate data, control program flow, and interact with memory. Practice with these basics will help you gain proficiency in RISC-V assembly programming.

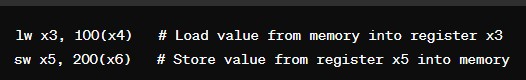
# Registers

* RISC-V architecture includes 32 general-purpose registers (**x0** to **x31**) and a few special- purpose registers like the program counter (**pc**) and the stack pointer (**sp**).
* Example: Loading a value into a register and storing it in memory:



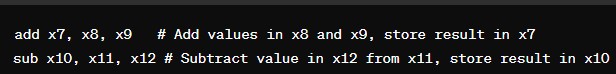
# Data Transfer Instructions

* **lw** (load word) loads a 32-bit value from memory into a register.
* **sw** (store word) stores a 32-bit value from a register into memory.
* Example:



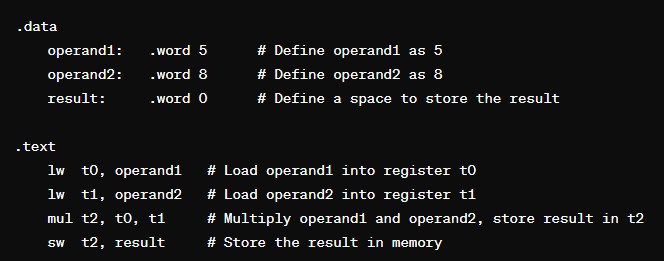
# Arithmetic Instructions

* **add** adds two registers and stores the result in another register.
* **sub** subtracts one register from another and stores the result in another register.
* Example:



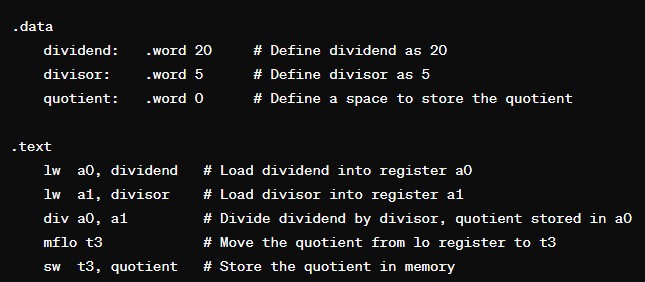
**Multiplication:**

* Load the first operand (**operand1**) into register **t0** using the **lw** instruction.
* Load the second operand (**operand2**) into register **t1** using the **lw** instruction.
* Multiply the values in registers **t0** and **t1** using the **mul** instruction, and store the result in register **t2**.
* Store the result from register **t2** into memory using the **sw** instruction.
* Example:



**Division:**

* Load the dividend (**dividend**) into register **a0** using the **lw** instruction.
* Load the divisor (**divisor**) into register **a1** using the **lw** instruction.
* Divide the value in register **a0** by the value in register **a1** using the **div** instruction. The quotient is stored in the special register **lo**.
* Move the quotient from the special register **lo** to a general-purpose register **t3** using the **mflo** instruction.
* Store the quotient from register **t3** into memory using the **sw** instruction.
* Example:

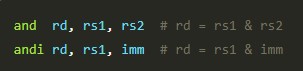


# Logical Instructions

In RISC-V, logical operations are fundamental operations used to manipulate individual bits in registers. These operations are commonly used in various types of bitwise manipulation, such as masking, setting, clearing, and toggling specific bits.

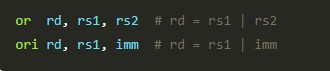
#  AND

The **and** and **andi** instructions perform logical [AND](https://en.wikipedia.org/wiki/AND_gate) on individual bits. The AND instructions are commonly used to mask parts of a register.



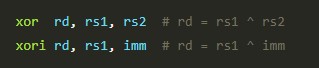
##  OR

The **or** and **ori** instructions perform logical [OR](https://en.wikipedia.org/wiki/OR_gate) on individual bits.



##  XOR

The **xor** and **xori** instructions perform logical [XOR](https://en.wikipedia.org/wiki/XOR_gate) (exclusive OR) on individual bits.



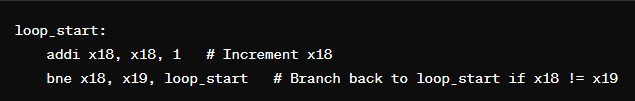
##  NOT

The **not** pseudoinstruction inverts the bits in a register (0 → 1 and 1 → 0). The assembler converts **not** to **xori.**



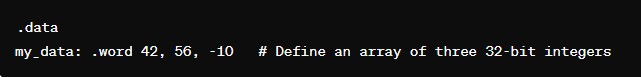
## Labeling and Comments

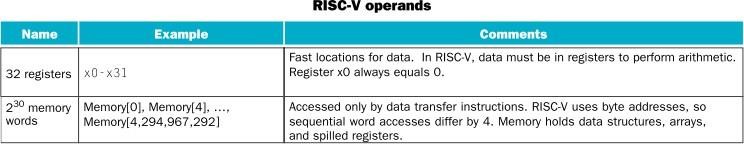
* Labels are used to mark specific locations in the code, facilitating branching and jumping.
* Comments provide explanatory notes within the code.
* Example:



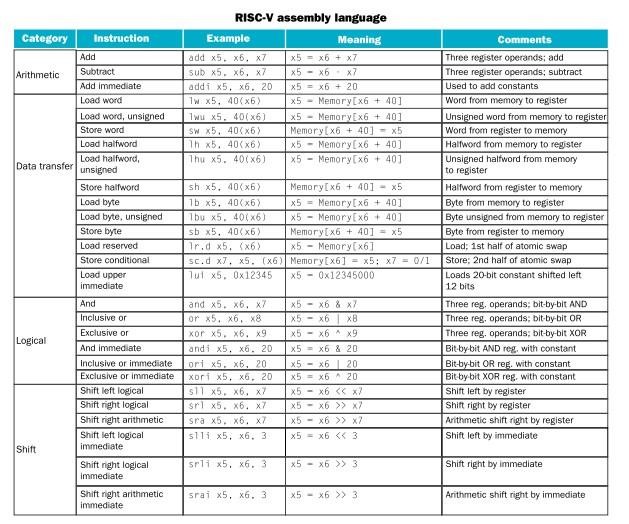
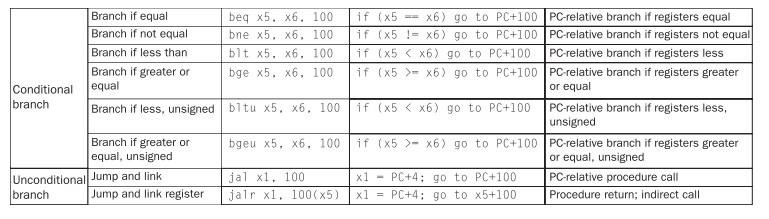
## Data Directives

* **.data** directive marks the beginning of the data section, where data is initialized.
* **.word** directive allocates memory for a 32-bit word and initializes it with specified values.
* Example:





*RISC-V Operands*



*RISC-V Assembly Language*

**Lab Tasks:**

**Task-1: Write a RISC-V assembly program that store a full 32-bit constant.**

* Use the lui instruction to load a 20-bit upper immediate value of your registration number into the upper 20 bits of a register.
* Use the addi instruction to add the lower 12-bit immediate value of your registration number to a register.
* Combine both instructions to load a full 32-bit constant into a register.

# Storing first 20 bits using lui

lui s0 , 0x02023

# Adding in last 12 bits using addi into same register

addi s0 , s0 , 0x775

**Task-2: Develop a RISC-V assembly program to demonstrate the use of arithmetic instructions in the Venus IDE. The program should accomplish the following objectives:**

* Load Values into Registers (t0, t1, t2, t3)
* Perform addition and subtraction operations on predetermined values stored in registers using the **add** and **sub** instructions.
* Implement multiplication and division operations between specified registers using the **mul** and **div** instructions.
* Verify the correctness of the arithmetic operations by inspecting the results stored in registers.

# Initialise data

.data

num\_1 : .word 1

num\_2 : .word 2

num\_3 : .word 3

num\_4 : .word 4

.text

# Load values into register

lw t0 , num\_1

lw t1 , num\_2

lw t2 , num\_3

lw t3 , num\_4

# Addition

add s2 , t0 , t1

add s3 , t2 , t3

# Subtraction

sub s4 , t1 , t0

sub s5 , t3 , t2

# Multiplication

mul s6 , t1 , t2

mul s7 , t2 , t3

# Division

div s8 , t1 , t0

div s9 , t3 , t1

**Task-3: Develop a RISC-V assembly program in the Venus IDE that combines arithmetic and data transfer instructions to perform a simple computational task. The program should achieve the following objectives:**

* Load two integers from memory into registers using the lw instruction.
* Perform an arithmetic operation (addition, subtraction, multiplication, or division) between the two integers using the corresponding arithmetic instruction (add, sub, mul, or div).
* Store the result of the arithmetic operation back into memory at a specified memory location using the sw instruction.

# Initialise data

.data

num\_1 : .word 2

num\_2 : .word 4

add\_op : .word 0

sub\_op : .word 0

mul\_op : .word 0

div\_op : .word 0

.text

# Load values into register

lw t0 , num\_1

lw t1 , num\_2

# Arithemetic Operations

add s1 , t0 , t1

sub s2 , t1 , t0

mul s3 , t0 , t1

div s4 , t1 , t0

# Storing result back into memory

sw s1 , add\_op , s5

sw s2 , sub\_op , s6

sw s3 , mul\_op , s7

sw s4 , div\_op , s8

# Verifying stored results

lw t3 , add\_op

lw t4 , sub\_op

lw t5 , mul\_op

lw t6 , sub\_op

**Task-4: Write a RISC-V assembly program in the Venus IDE that perform bitwise logical operations (AND, OR, XOR, and NOT) to manipulate the bits of registers. You will perform a series of operations on two registers (t0 and t1) and observe how each operation affects the binary values in those registers.**

* Load Values into Registers (t0, t1)
* Perform Bitwise AND Operation (and, andi).
* Perform Bitwise OR Operation (or, ori).
* Perform Bitwise XOR Operation (xor, xori) Perform Bitwise NOT Operation (~not)

# Load values into register

li t0 , 9 # 1001

li t1 , 15 # 1111

# Logical Operations

and s1 , t0 , t1 # 1001 = 9

or s2 , t0 , t1 # 1111 = 15

xor s3 , t0 , t1 # 0110 = 6

not s4 , t0 # 111111110110 = -10

not s5 , t1 # 111111110000 = -16