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.

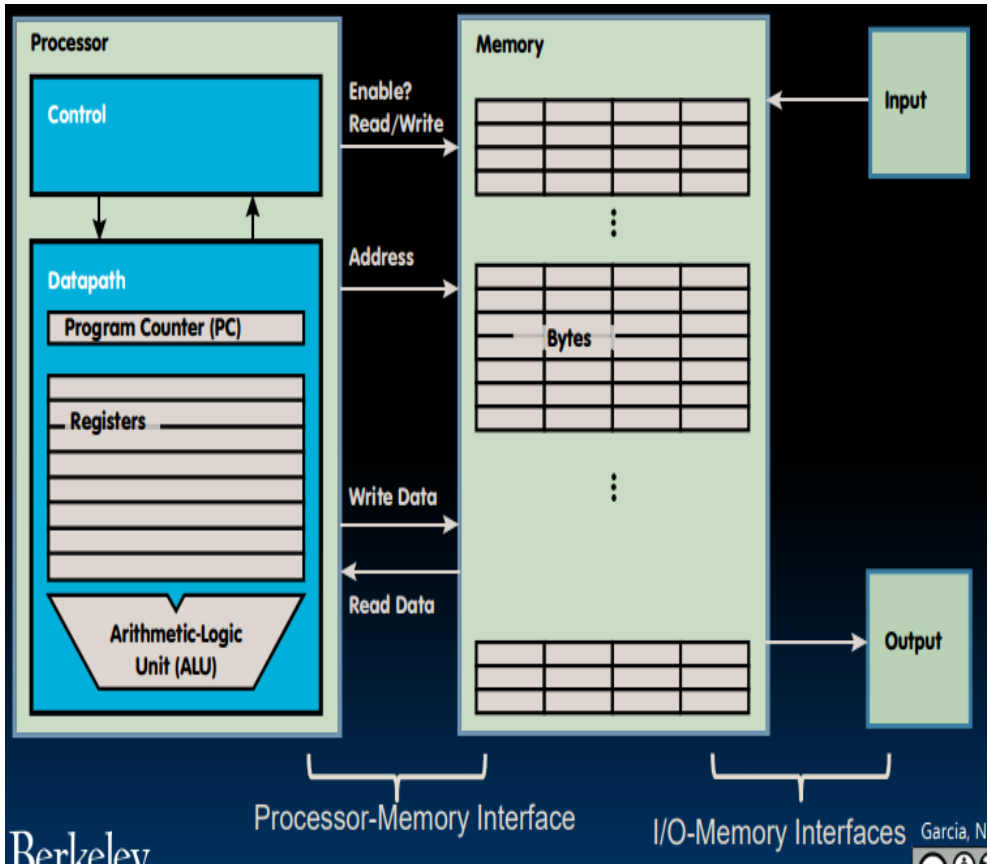

Figure 4.1: RISC-V Architecture

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:

```assembly
li x1, 42      # Load immediate value 42 into register x1
sw x1, 0(x2)   # Store value in x1 into memory at address stored in x2
```

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:

```
lw x3, 100(x4)   # Load value from memory into register x3
sw x5, 200(x6)   # Store value from register x5 into memory
```

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:

```
add x7, x8, x9   # Add values in x8 and x9, store result in x7
sub x10, x11, x12 # Subtract value in x12 from x11, store result in x10
```

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:

```
.data
    operand1:   .word 5      # Define operand1 as 5
    operand2:   .word 8      # Define operand2 as 8
    result:     .word 0      # Define a space to store the result

.text
    lw  t0, operand1   # Load operand1 into register t0
    lw  t1, operand2   # Load operand2 into register t1
    mul t2, t0, t1     # Multiply operand1 and operand2, store result in t2
    sw  t2, result     # Store the result in memory
```

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:

```
.data
    dividend:   .word 20     # Define dividend as 20
    divisor:    .word 5      # Define divisor as 5
    quotient:   .word 0      # Define a space to store the quotient

.text
    lw  a0, dividend   # Load dividend into register a0
    lw  a1, divisor    # Load divisor into register a1
    div a0, a1         # Divide dividend by divisor, quotient stored in a0
    mflo t3            # Move the quotient from lo register to t3
    sw  t3, quotient   # Store the quotient in memory
```

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 on individual bits. The AND instructions are commonly used to mask parts of a register.

```
and  rd, rs1, rs2  # rd = rs1 & rs2
andi rd, rs1, imm  # rd = rs1 & imm
```

- **OR**

The **or** and **ori** instructions perform logical OR on individual bits.

```
or  rd, rs1, rs2  # rd = rs1 | rs2
ori rd, rs1, imm  # rd = rs1 | imm
```

- **XOR**
  The **xor** and **xori** instructions perform logical XOR (exclusive OR) on individual bits.

```
xor  rd, rs1, rs2  # rd = rs1 ^ rs2
xori rd, rs1, imm  # rd = rs1 ^ imm
```

- **NOT**

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

```
not  rd, rs1     # rd = ~rs1 (pseudoinstruction)
```

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:

```
loop_start:
    addi x18, x18, 1   # Increment x18
    bne x18, x19, loop_start   # Branch back to loop_start if x18 != x19
```

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:

```
.data
my_data: .word 42, 56, -10   # Define an array of three 32-bit integers
```

RISC-V operands

Name	Example	Comments
32 registers	x0-x31	Fast locations for data. In RISC-V, data must be in registers to perform arithmetic. Register x0 always equals 0.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4,294,967,292]	Accessed only by data transfer instructions. RISC-V uses byte addresses, so sequential word accesses differ by 4. Memory holds data structures, arrays, and spilled registers.

RISC-V Operands

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	add x5, x6, x7	x5 = x6 + x7	Three register operands; add
	Subtract	sub x5, x6, x7	x5 = x6 - x7	Three register operands; subtract
	Add immediate	addi x5, x6, 20	x5 = x6 + 20	Used to add constants
Data transfer	Load word	lw x5, 40(x6)	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	sw x5, 40(x6)	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	lh x5, 40(x6)	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	sh x5, 40(x6)	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	lb x5, 40(x6)	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	Byte unsigned from memory to register
	Store byte	sb x5, 40(x6)	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	lr.d x5, (x6)	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	lui x5, 0x12345	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits
Logical	And	and x5, x6, x7	x5 = x6 & x7	Three reg. operands; bit-by-bit AND
	Inclusive or	or x5, x6, x8	x5 = x6 | x8	Three reg. operands; bit-by-bit OR
	Exclusive or	xor x5, x6, x9	x5 = x6 ^ x9	Three reg. operands; bit-by-bit XOR
	And immediate	andi x5, x6, 20	x5 = x6 & 20	Bit-by-bit AND reg. with constant
	Inclusive or immediate	ori x5, x6, 20	x5 = x6 | 20	Bit-by-bit OR reg. with constant
	Exclusive or immediate	xori x5, x6, 20	x5 = x6 ^ 20	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	sll x5, x6, x7	x5 = x6 << x7	Shift left by register
	Shift right logical	srl x5, x6, x7	x5 = x6 >> x7	Shift right by register
	Shift right arithmetic	sra x5, x6, x7	x5 = x6 >> x7	Arithmetic shift right by register
	Shift left logical immediate	slli x5, x6, 3	x5 = x6 << 3	Shift left by immediate
	Shift right logical immediate	srli x5, x6, 3	x5 = x6 >> 3	Shift right by immediate
	Shift right arithmetic immediate	srai x5, x6, 3	x5 = x6 >> 3	Arithmetic shift right by immediate
Conditional branch	Branch if equal	beq x5, x6, 100	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	bne x5, x6, 100	if (x5 != x6) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	blt x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	bge x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	bltu x5, x6, 100	if (x5 < x6) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	jal x1, 100	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	Procedure return; indirect call

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.

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.

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.

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)