**DEPARTMENT OF**

**NETWORKING AND COMMUNICATION**

**College of Engineering and Technology**

**SRM Institute of Science and Technology**

MINI PROJECT REPORT

ODD Semester, 2023-2024

Lab code & Sub Name : 21CSS201T & Computer Organization and Architecture

Year & Semester : II & III

Project Title : Binary Adder

Lab Supervisor **: Dr.R.Nithya Paranthaman**

Team Members :Saransh Singh (Reg.No:RA2212703010013)

|  |  |  |
| --- | --- | --- |
| **Particulars** | **Max. Marks** | **Marks Obtained** |
| **Name:** |
| **Register No :** |
| Program and Execution | 20 |  |
| Demo verification &viva | 15 |  |
| Project Report | 05 |  |
| **Total** | **40** |  |

**Date:**

**Staff :**

**Name :**

**Signature:**

**BINARY ADDER**

**OBJECTIVE:**

This project aims to implement the Binary Adder algorithm in Python language.

**ABSTRACT:**

This project presents the implementation of a binary adder using the 8085-microprocessor architecture in Python. The 8085 microprocessor is a widely used microprocessor in embedded systems and is known for its simplicity and versatility. The binary adder, a fundamental component in digital systems, performs the addition of two binary numbers.

The Python programming language is utilized to simulate the 8085 microprocessors, enabling the design and execution of the binary addition operation. The project involves the development of a Python program that emulates the instruction set and architecture of the 8085 microprocessors.

The program accepts two binary numbers as input, mimicking the data bus of the 8085, and performs binary addition using the appropriate instructions. The simulation includes the key components of the microprocessor, such as registers, arithmetic and logic unit (ALU), and control unit. The implementation adheres to the instruction set and timing diagrams of the 8085 microprocessors, ensuring accuracy and reliability in the binary addition process.

The program provides a visual representation of the microprocessor's internal states during execution, aiding in understanding the step-by-step process of binary addition.

**INTRODUCTION:**

Binary addition is a fundamental operation in digital electronics, essential for tasks ranging from simple arithmetic calculations to complex data processing. The 8085 microprocessors, a widely utilized processor in the realm of embedded systems, offers a robust platform for implementing binary addition.

This project delves into the creation of a binary adder using the 8085-microprocessor architecture, implemented in the Python programming language. The 8085 microprocessors, known for its simplicity and versatility, features a set of instructions that can be orchestrated to perform various operations. In this context, the goal is to leverage the capabilities of the 8085 to execute binary addition. Python, chosen for its readability and ease of implementation, serves as the programming language for simulating the 8085 microprocessor's behavior.

This project not only aims to create a functional binary adder but also strives to provide a learning platform for individuals interested in digital electronics and microprocessor programming. By emulating 8085's architecture in Python, users can gain a hands-on understanding of the microprocessor's internal workings and its application in basic arithmetic operations.

**HARDWARE/SOFTWARE REQUIREMENTS:**

**Software Requirements:**

Python Interpreter:

Python is required as the programming language to implement the 8085-microprocessor simulation and binary addition algorithm.

**8085 Microprocessor Simulator Library:**

A Python library or module that simulates the behaviour of the 8085 microprocessors. For example, the py8085sim library. Code Editor: A text editor or integrated development environment (IDE) for writing and editing the Python code.

**Operating System:**

The project should be compatible with the operating system on which Python is installed. Common operating systems like Windows, Linux, or macOS are suitable. Documentation and Reference Material: Access to relevant documentation for the 8085-microprocessor architecture, its instruction set, and any additional reference material for Python programming.

**CONCEPTS/WORKING PRINCIPLE**

The project aims to simulate a binary adder using the 8085-microprocessor architecture entirely through Python code. Instead of interacting with physical registers and ALU, the simulation relies on data structures and functions to replicate the behavior of the 8085 microprocessors.

**Working Principle:**

1. **Initialization:**
   * Create virtual registers: Define variables or data structures to represent the registers of the 8085 microprocessors, such as the accumulator, B register, and carry flag register.
   * Load binary numbers into virtual registers: Assign values to these virtual registers, simulating the loading of binary numbers.
2. **Binary Addition Algorithm:**
   * Iterate through each bit position: Use loops to iterate through each bit position, starting from the LSB to the MSB.
   * Simulate ALU operation: Instead of a physical ALU, implement functions or operations that mimic binary addition at each bit position, considering the carry from the previous bit.
   * Update virtual registers: Update the virtual registers based on the simulated ALU operation.
3. **Update Flags:**
   * Simulate flag updates: Implement logic to update virtual flags based on the result of each addition.
4. **Repeat:**
   * Repeat the process for each bit position until the entire binary numbers are added.
5. **Result:**
   * The result of the binary addition is stored in the designated virtual register(s), and virtual flags reflect the status of the result.

**Input and Output Handling:**

* Define a mechanism for inputting the target value and the sorted dataset into the 8085.
* Establish a way to output the result, such as whether the target value was found or not.

**Code Optimization:**

* Optimize the code for space and time efficiency, considering the limited resources of the 8085 microprocessors.
* Minimize the number of instructions and efficiently use registers and memory.

**Communication Between Python and 8085:**

* Establish a communication interface between the Python program and the 8085 microprocessors.
* This could involve using a hardware interface or a simulator that allows Python to control the 8085.

**PROGRAMS:**

import tkinter as tk

from tkinter import ttk

from tkinter import messagebox

import time

def ripple\_carry\_adder(A, B):

carry = 0

result = [0] \* len(A)

start\_time = time.perf\_counter\_ns()

for i in range(len(A)):

sum\_bit = A[i] ^ B[i] ^ carry

carry = (A[i] & B[i]) | (carry & (A[i] ^ B[i]))

result[i] = sum\_bit

execution\_time\_ns = time.perf\_counter\_ns() - start\_time

return result, execution\_time\_ns

def generate\_g\_and\_p(A, B):

g = [A[i] & B[i] for i in range(len(A))]

p = [A[i] | B[i] for i in range(len(A))]

return g, p

def carry\_lookahead\_adder(A, B, G, P):

c = [0] \* (len(A) + 1)

s = [0] \* len(A)

start\_time = time.perf\_counter\_ns()

c[0] = 0 # Initial carry-in

for i in range(len(A)):

c[i + 1] = G[i] | (P[i] & c[i])

s[i] = A[i] ^ B[i] ^ c[i]

execution\_time\_ns = time.perf\_counter\_ns() - start\_time

return s, execution\_time\_ns

def calculate():

try:

A = [int(bit) for bit in entry\_a.get()]

B = [int(bit) for bit in entry\_b.get()]

# Check if A and B have more than 6 bits

if len(A) > 6 or len(B) > 6:

messagebox.showerror("Error", "Binary operands must have a maximum of 6 bits.")

return # Exit the function

# Check if A and B have the same length

if len(A) != len(B):

messagebox.showerror("Error", "Binary operands must have the same number of bits.")

return # Exit the function

G, P = generate\_g\_and\_p(A, B) # Generate G and P here

result\_ripple, execution\_time\_ripple\_ns = ripple\_carry\_adder(A, B)

result\_cla, execution\_time\_cla\_ns = carry\_lookahead\_adder(A, B, G, P) # Pass G and P

result\_frame = ttk.LabelFrame(root, text="Results")

result\_frame.grid(column=0, row=3, padx=10, pady=10)

ttk.Label(result\_frame, text="Ripple Carry Adder Result:").grid(column=0, row=0, sticky='w', padx=5)

ttk.Label(result\_frame, text="Operand A:").grid(column=0, row=1, sticky='w', padx=5)

ttk.Label(result\_frame, text="Operand B:").grid(column=0, row=2, sticky='w', padx=5)

ttk.Label(result\_frame, text="Result:").grid(column=0, row=3, sticky='w', padx=5)

ttk.Label(result\_frame, text="Execution Time:").grid(column=0, row=4, sticky='w', padx=5)

ttk.Label(result\_frame, text=''.join(map(str, A)).ljust(30)).grid(column=1, row=1, columnspan=4, padx=5)

ttk.Label(result\_frame, text=''.join(map(str, B)).ljust(30)).grid(column=1, row=2, columnspan=4, padx=5)

ttk.Label(result\_frame, text=''.join(map(str, result\_ripple)).ljust(30)).grid(column=1, row=3, columnspan=4, padx=5)

ttk.Label(result\_frame, text=f'{execution\_time\_ripple\_ns} ns'.ljust(30)).grid(column=1, row=4, columnspan=4, padx=5)

ttk.Label(result\_frame, text="").grid(column=0, row=5) # Empty row for spacing

ttk.Label(result\_frame, text="Carry Look-Ahead Adder Result:").grid(column=0, row=6, sticky='w', padx=5)

ttk.Label(result\_frame, text="Operand A:").grid(column=0, row=7, sticky='w', padx=5)

ttk.Label(result\_frame, text="Operand B:").grid(column=0, row=8, sticky='w', padx=5)

ttk.Label(result\_frame, text="Result:").grid(column=0, row=9, sticky='w', padx=5)

ttk.Label(result\_frame, text="Execution Time:").grid(column=0, row=10, sticky='w', padx=5)

ttk.Label(result\_frame, text=''.join(map(str, A)).ljust(30)).grid(column=1, row=7, columnspan=4, padx=5)

ttk.Label(result\_frame, text=''.join(map(str, B)).ljust(30)).grid(column=1, row=8, columnspan=4, padx=5)

ttk.Label(result\_frame, text=''.join(map(str, result\_cla)).ljust(30)).grid(column=1, row=9, columnspan=4, padx=5)

ttk.Label(result\_frame, text=f'{execution\_time\_cla\_ns} ns'.ljust(30)).grid(column=1, row=10, columnspan=4, padx=5)

except ValueError:

messagebox.showerror("Error", "Please enter valid binary operands (0s and 1s only).")

root = tk.Tk()

root.title("Binary Adder")

main\_frame = ttk.Frame(root, padding=20)

main\_frame.grid(column=0, row=0, sticky=("N", "W", "E", "S"))

root.columnconfigure(0, weight=1)

root.rowconfigure(0, weight=1)

ttk.Label(main\_frame, text="Binary Adder", font=("Helvetica", 16)).grid(column=0, row=0, columnspan=2, pady=10)

ttk.Label(main\_frame, text="Enter binary operand A (e.g., 1011, max 6 bits):").grid(column=0, row=1, sticky='w', padx=5)

entry\_a = ttk.Entry(main\_frame)

entry\_a.grid(column=1, row=1, padx=5)

ttk.Label(main\_frame, text="Enter binary operand B (e.g., 0111, max 6 bits):").grid(column=0, row=2, sticky='w', padx=5)

entry\_b = ttk.Entry(main\_frame)

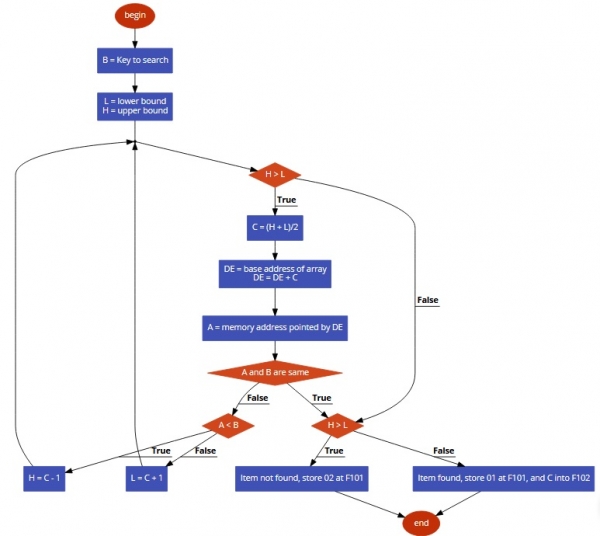
entry\_b.grid(column=1, row=2, padx=5)

calculate\_button = ttk.Button(main\_frame, text="Calculate", command=calculate)

calculate\_button.grid(column=0, row=3, columnspan=2, pady=10)

root.mainloop()

**FLOWCHART:**



**OUTPUT:**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**CONCLUSIONS:**

In conclusion, the implementation of a binary adder using Python demonstrates the fundamental principles of digital arithmetic and programming. Through this project, we have explored the binary number system, bitwise operations, and the logical steps involved in adding two binary numbers.

The binary adder serves as a practical example of how computers perform arithmetic operations at the most basic level. By breaking down the addition process into individual bitwise operations, we gain insights into the inner workings of digital circuits and the efficient manipulation of binary data. Moreover, this project showcases the versatility of Python as a programming language for handling binary operations.

Python's simplicity and readability make it an excellent choice for beginners to understand and experiment with fundamental concepts in computer science. As we wrap up this binary adder implementation, it's important to recognize the broader implications of such projects. Understanding binary addition is not only crucial for programming but also provides a foundation for more advanced topics in computer science and digital electronics. This project, therefore, serves as a steppingstone for those interested in delving deeper into the intricate world of computer architecture and algorithm design.

**REFRENCES:**

1. [Javatpoint: This page provides a detailed explanation of binary adders, including the digital components required for arithmetic addition and the construction of a binary adder using full-adder circuits1](https://www.javatpoint.com/coa-binary-adder).
2. [Studocu: This lab manual from Deen Dayal Upadhyay University provides practical implementation of half adders and full adders using basic logic gates2](https://www.studocu.com/in/document/deen-dayal-upadhyay-gorakhpur-university/computer-organisation/coa-lab-manualdoc-practical/43935057).
3. [The Developer Blog](https://thedeveloperblog.com/coa/coa-binary-adder)[: This blog post shows the interconnections of four full-adder circuits to provide a 4-bit binary adder](https://www.javatpoint.com/coa-binary-adder)[3](https://thedeveloperblog.com/coa/coa-binary-adder).