|  |  |
| --- | --- |
|  | **Qatar University**  **College of Engineering**  **Department of Computer Science and Engineering** |

**CMPE263 Computer Architecture and Organization I**

**Course Project Report**

**Spring 2025**

Project Title

DesignDesign of 8-bit CPU using Logisim using ALU, Registers, ROM & RAM

Submitted By:

1. Karim Elnaggar – 202307871
2. Amro Arbab – 202309801
3. Ahmed Hassanien – 202307933
4. Albaraa Samak – 202210562

Table of Contents

Part I: [Introduction 3](#_Toc131949761)

[Part II: CPU Architecture and Organization 3](#_Toc131949762)

[Part III: Instruction Set 3](#_Toc131949763)

[Part IV: Instruction Format ………………………………………………………………………………………………………………………………..3](#_Toc131949764)

[Part V: Assembly Programs 3](#_Toc131949764)

Part VI: [Conclusion 3](#_Toc131949767)

# Part I: Introduction

In this project, an 8-bit CPU was designed and implemented using Logisim. The CPU supports eight essential instructions, including *LDR*, *STR*, *ADD*, *SUB*, *LSL*, *LSR*, *CMP*, and *JMP*, enabling arithmetic, logical, and control-flow operations. The architecture incorporates four 8-bit general-purpose registers (R1–R4), an 8-bit ALU for computation, and dedicated memory units (ROM for instructions and RAM for data storage).

The CPU employs an 8-bit data bus and an 8-bit address bus, with a 16-bit fixed-length instruction format (4-bit opcode + 12-bit operands). for efficient decoding, the design utilizes buffers for data routing, reducing hardware complexity while maintaining precise control over data flow. The ALU integrates arithmetic, logical, and shift operations, while a comparator enables conditional jumps (*JMP*) based on equality checks (*CMP*).

A hardwired control unit "controls" operations, synchronizing the Program Counter (PC) for instruction fetching and execution. The PC increments sequentially, accessing instructions from ROM, while the control unit generates signals to coordinate registers, ALU, and memory. The CPU’s functionality is validated through sample programs, demonstrating arithmetic, data movement, and branching operations, therefore reinforcing core principles of computer architecture.

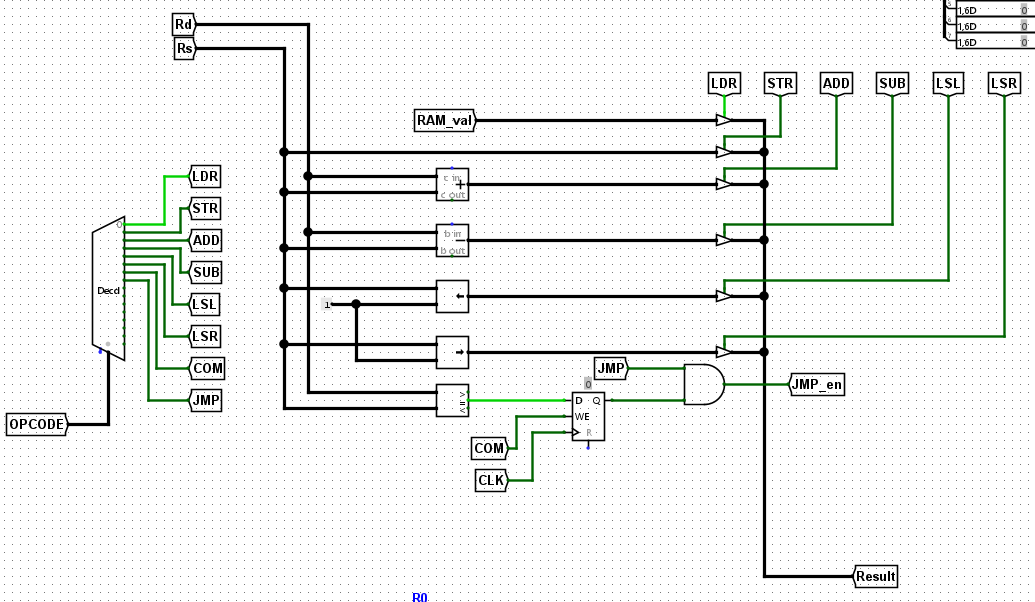
# Part II: CPU Architecture and Organization

A diagram of a computer

AI-generated content may be incorrect.

*Full CPU design*

**Starting with the ALU:**

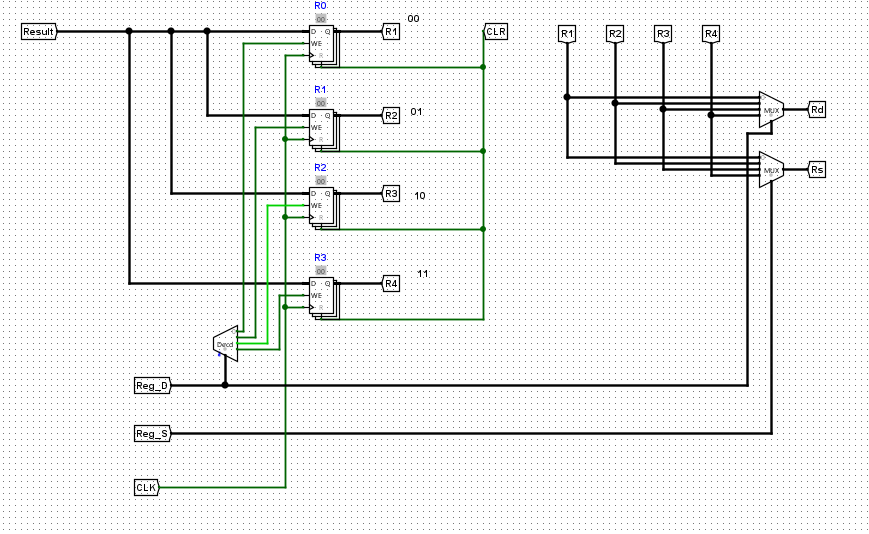


*ALU design*

Our 8-bit ALU (Arithmetic Logic Unit) design is capable of performing a variety of operations, including load, store, addition, subtraction, logical shifts (left and right), a comparison, and jump operations.

1. **Inputs:**
   * **OPCODE:** The operation code input that defines which operation to perform. It is fed into a decoder.
   * **Rd (Destination Register)** and **Rs (Source Register):** These carry the addresses or data needed for the operations.
2. **Decoder (Dec):**
   * The OPCODE is connected to a Decoder block which decodes it into control signals.
   * Each output of the decoder enables one of the operations: LDR (Load), STR (Store), ADD, SUB, LSL (Logical Shift Left), LSR (Logical Shift Right), COM (Compare), or JMP (Jump).
3. **Functional Units:**
   * **RAM\_val:** This fetches a value from RAM to be loaded.
   * **Adders and Subtractors:** Dedicated units perform addition and subtraction (ADD and SUB).
   * **Shift Units:** Two separate units handle shifting operations (LSL for left shift, LSR for right shift).
   * **Comparison Unit (COM):** A comparison unit compares two values and outputs 1 if the two values are equal, based on that the JMP operation is performed.
4. **Result Output:**
   * Each instruction has 1 value per operation.
5. **Jump Handling:**
   * The **JMP** output from the decoder is tied to a D flip-flop (register) that stores the jump condition.
   * The output of the flip-flop is combined with the decoded jump signal through an AND gate, producing a control signal labeled **JMP\_en**.
   * This mechanism ensures that the jump only happens when the appropriate conditions are met and synchronized with the clock (CLK).
6. **Data Flow:**
   * Depending on the decoded operation, data flows from the source registers (Rs) through the selected functional unit (adder, subtractor, shifter, etc.) and into the destination register (Rd).
   * The control signals determine which operation is performed and which result is forwarded to the output.
7. **Special Features (branching):**
   * The **COM** and **JMP** blocks work together for conditional branching, allowing the ALU to participate not only in calculations but also in basic program control flow.
   * Logical shifts (both left and right) multiply or divide numbers by powers of 2, an efficient alternative to multiplication/division units.

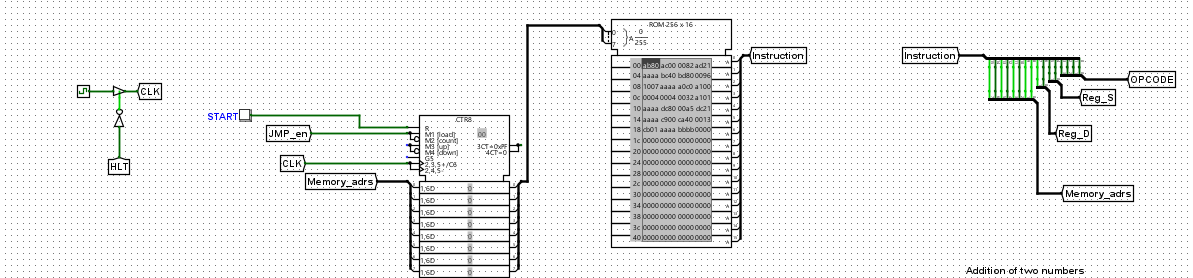
**Secondly registers:**



*Registers design*

1. **Inputs and Control Signals:**
   * **Result:** This input provides the data that will be written into one of the registers.
   * **Reg\_D (Destination Register Select):** This signal selects which register will store the incoming Result data.
   * **Reg\_S (Source Register Select):** This signal selects which register's data will be used as a source for operations.
   * **CLK (Clock):** The clock signal controls when data is written to the registers.
   * **CLR (Clear):** A clear signal resets all registers back to zero when activated.
2. **Decoder (Dec):**
   * The **decoder** takes the Reg\_D input (typically 2 bits) and activates one of four possible outputs.
   * Each output from the decoder is connected to the write enable **(WE)** of a specific register, ensuring that only the selected register is updated with the new data on the next clock cycle.
3. **Registers (R1 to R4):**
   * Each register is an 8-bit D flip-flop array, meaning they store 8 bits of data.
   * When enabled by the decoder and clocked, they latch the incoming Result value.
   * Registers can be independently cleared using the CLR signal.
4. **Data Routing (Multiplexers - MUX):**
   * Two multiplexers are used:
     + One MUX selects the data to be output as **Rd** (Destination Register output).
     + Another MUX selects the data to be output as **Rs** (Source Register output).
   * Both multiplexers are controlled by the Reg\_S signal, allowing dynamic selection of any register for reading without altering stored values.
5. **Operation Flow:**
   * To **write**: The Reg\_D selects which register to write to, and when CLK pulses, the Result is stored into the chosen register.
   * To **read**: The Reg\_S selects which register’s contents will be output to the ALU for operations via **Rs**.
   * The **Rd** output can also be used to feedback into different parts of the system if needed.

**Thirdly PC and ROM:**

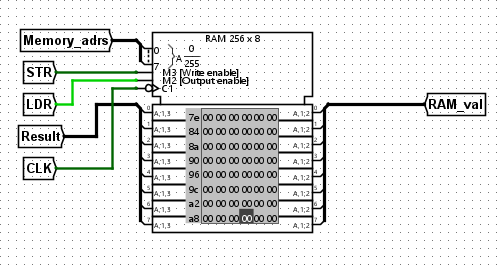
****

*PC and ROM design*

In this section of our 8-bit CPU we designed a system that is responsible for program execution control — fetching instructions from memory and decoding them for execution.

1. **Clock and Control Inputs:**
   * **CLK (Clock)**: Drives the timing for updating the Program Counter (PC) and fetching the next instruction.
   * **HLT (Halt):** A manual control to stop the clock when needed, effectively halting the processor.
   * **START**: Used to reset or initialize the execution, enabling the program to begin from a known starting point.
   * **JMP\_en (Jump Enable):** Controls whether the PC will increment normally (sequential execution) or load a new address (jump).
2. **Program Counter (PC):**
   * Implemented as a 4-bit register connected to an adder.
   * **Normal Operation**: On each clock pulse, the PC increments by 1, pointing to the next instruction in the ROM.
   * **Jump Operation:** If JMP\_en is activated, the PC loads a specific memory address provided through external input, allowing non-linear program flow (branching).
3. **ROM (Read-Only Memory):**
   * **Memory Size:** 256 × 16 bits — meaning 256 addresses, each holding a 16-bit wide instruction.
   * **Instruction Fetch:** The current memory address from the PC is used to access ROM, fetching the 16-bit instruction associated with that address.
   * Instructions are hardcoded into the ROM and represent the program the processor will execute.
4. **Instruction Decoder:**
   * The fetched 16-bit instruction is split into:
     + **Opcode (Operation Code):** Defines what operation the ALU should perform.
     + **Reg\_S (Source Register):** Specifies which register will supply the source operand.
     + **Reg\_D (Destination Register):** Specifies where to store the result.
     + **Memory\_adrs (Memory Address):** For load/store operations that involve memory.
5. **Operation Flow:**
   * **Clock Pulse:** Advances the PC or loads a jump address based on JMP\_en.
   * **PC Address:** Used to fetch an instruction from ROM.
   * **Instruction Split:** The instruction is decoded into its components (**Opcode, Reg\_S, Reg\_D, Memory\_adrs).**
   * These components control the ALU, registers, and memory access as the processor executes each step.

**Finally RAM:**

****

*RAM design*

We used a RAM module configured as 256 x 8-bit memory. This means it can store 256 words, with each word being 8 bits wide.

**It contains:**

**Memory Address** (**Memory\_adrs**):

This input selects the address in memory where data will be read or written. It connects to the address bus (A1, A2, A3, etc.).

**STR (Store - Write Enable, M3):**

This control line is used to write data into the RAM.

When STR is active (logic HIGH), the RAM allows data to be written into the selected memory location.

**LDR (Load - Output Enable, M2):**

This control line is used to read data from the RAM.

When LDR is active (logic HIGH), the RAM outputs the data stored at the specified address.

Result:

This line provides the input data to be written into the memory during a write operation.

It connects to the data input port of the RAM.

**CLK (Clock):**

The clock signal synchronizes the memory operations, ensuring that write and read operations happen at correct timing intervals.

**RAM\_val (Output Data Bus):**

This is the 8-bit wide output from the RAM, where the read data appears when LDR is active.

Internal View:

The RAM block internally displays a table showing memory addresses (from 00 to FF hexadecimal) with their corresponding 8-bit stored values.

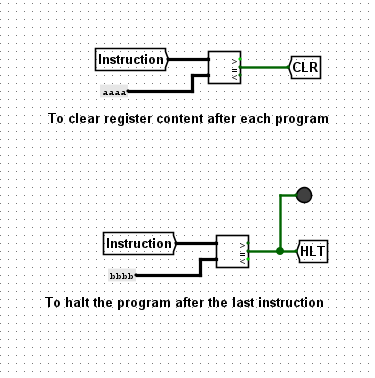
Currently, all memory values are initialized to 00 except for a few addresses (for example, address 00 has 7E, address 01 has 90, etc.).

**Control Signals Mapped:**

M3 (Write Enable): Connected to STR.

M2 (Output Enable): Connected to LDR.

C1 (Chip Select): Always enabled in this diagram (implicitly active).

**Additional components:**

*HLT, CLR instructions*

We implemented two additional instructions to Halt the program or to clear the registers content.

# Part III: Instruction Set

As mentioned in the project requirement, there are 8 instructions that we were tasked to create, and they are:

* LDR (Load)
* STR (Store)
* ADD (Addition)
* SUB (Subtraction)
* LSL (Shift Left)
* LSR (Shift Right)
* CMP (Compare)
* JMP (Jump)

A diagram of a computer

AI-generated content may be incorrect.

*A detailed image of how the instructions were implemented*

***Arithmetic operations:***

The two arithmetic operations discussed — addition and subtraction — are implemented using an adder and a subtractor, respectively. Their operation syntaxes are labeled as (add) and (sub). Both operations are components of the ALU’s arithmetic unit.

* The **ADD** instructions perform **R1 = R1 + R2**, utilizing an 8-bit adder to combine the values stored in two registers.
* The **SUB** instruction computes **R1 = R1 - R2**, achieved through an **8-bit subtractor**

***Shift operations:***

The two required shift operations perform shifting by only 1 bit at a time, meaning multiple shifts are needed for larger movements. These shifts are used to carry out multiplication and division by powers of 2; for example, two left shifts would multiply a value by 4, while two right shifts would divide it by 4. The operations are denoted as **LSL** (Logical Shift Left) and **LSR** (Logical Shift Right), and they are used similarly to arithmetic operations in terms of instruction order. Like addition and subtraction, shifting operations are handled by the ALU. The process involves decoding the instruction, accessing the bus to enable the relevant 7 bits required for the operation, selecting the destination register by setting its enable signal high, retrieving the data, performing the shift, and finally storing the result back into the destination register.

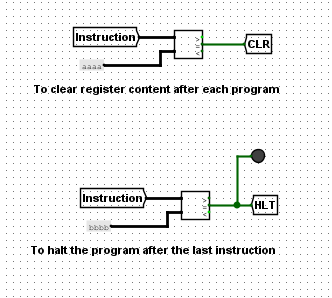
***Memory instructions:***

We implemented **LDR** and **STR** instructions to facilitate data movement between registers and RAM. **LDR** loads an 8-bit value from memory into a register, while **STR** writes a register’s contents to memory. These operations use the 8-bit address bus to specify memory locations and the data bus to transfer values. The RAM values are denoted by a tunnel called “**Ram\_val**” as seen in the above figure. These instructions are fundamental for handling dynamic data in programs, such as storing computational results.

***Branch operations:***

Our CPU supports CMP and JMP instructions to enable program branching. CMP compares two registers, while JMP redirects execution to a specified address. Though distinct—CMP is an ALU operation, and JMP is a control-flow instruction—they work in synch to implement conditional logic (loops or if-else structures).

***Stop and reset:***

We implemented two additional pseudo-instructions, each designed without an opcode or parameters, to serve two specific purposes: one to clear the contents of registers after each program, and another to halt program execution after the final instruction. The pseudo-instruction **'aaaa'** is used for clearing, while **'bbbb'** is used for stopping. These instructions are distinguished from the rest of the instructions set by a unique bit — specifically, a designated bit set to 1 in both pseudo-instructions, whereas in all other instructions, this bit remains 0.

# Part IV: Instruction Format

We were given 16 bits to work with in terms of the length of each instruction, so we decided to go with the following split:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Memory address | Register  D | Register  S | OOP code | Hexadecimal |
| 8-bits | 2-bit | 2-bit | 4-bits |  |

* Opcode: 4 bits (the instruction itself)
* destination register: 2 bits (the select for which register to store the result in)
* source register: 2 bits (the select for which register we will copy the data from to store in the destination register)
* Memory address: 8 bits

Once this is written it should be translated into hex code in order to be placed in the ROM for execution. The values of each operation HEX:

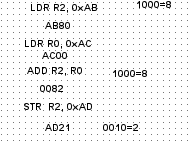
|  |  |
| --- | --- |
| 000 (000) | LDR |
| 001 (001) | STR |
| 002 (010) | ADD |
| 003(011) | SUB |
| 004(100) | LSL |
| 005(101) | LSR |
| 006(110) | COM |
| 007(111) | JMP |

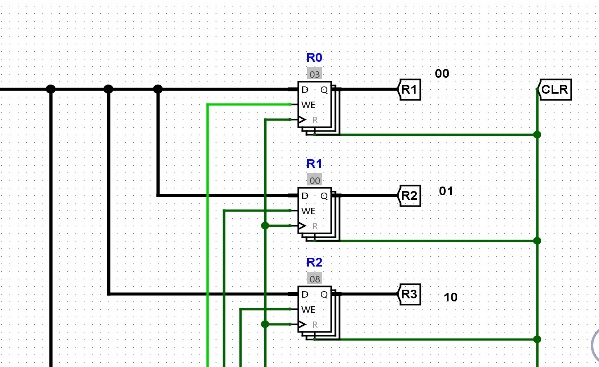
The values for the register select s:

|  |  |
| --- | --- |
| 00 | R1 |
| 01 | R2 |
| 10 | R3 |
| 11 | R4 |

# Part V: Assembly Programs

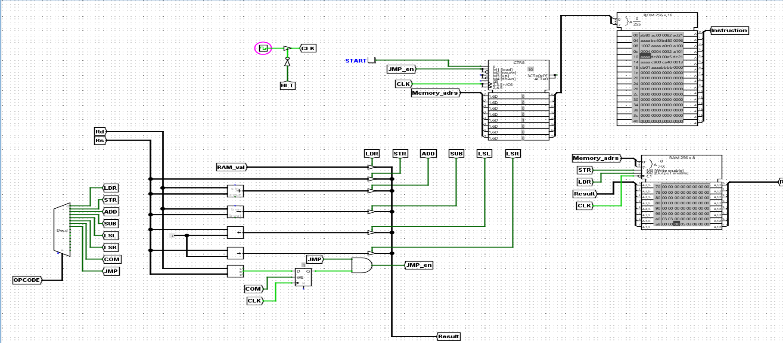
Program 1: Addition of two numbers (5+3)

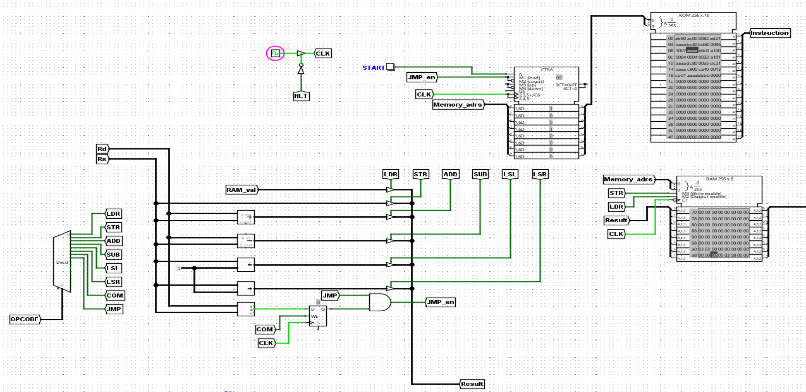


A computer screen shot of a computer

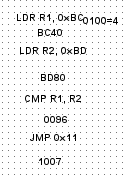
AI-generated content may be incorrect.

Program 2: Conditional jump (compare)

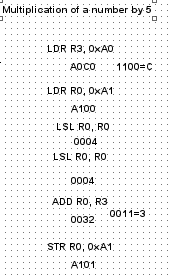
A diagram of a computer

AI-generated content may be incorrect.A diagram of a computer

AI-generated content may be incorrect.A diagram of a machine

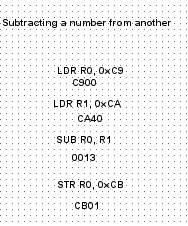
AI-generated content may be incorrect.

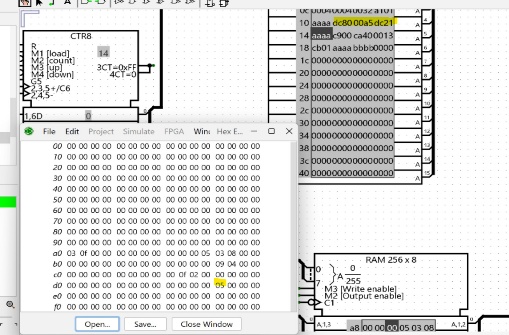
# 

A screenshot of a computer program

AI-generated content may be incorrect.Program 3: Multiplication of a number by 5 (LSL)

A screenshot of a computer

AI-generated content may be incorrect.Program 4: subtracting a number from another

A close up of a paper

AI-generated content may be incorrect.Program 5: Shifting a number to the right (LSR)

# Part VI: Conclusion

Our project involved designing an 8-bit CPU using a logical circuit simulator, during which we encountered significant challenges, particularly in integrating the individual components into a cohesive, functional system. Translating assembly programs into microcode proved especially demanding due to the precision required—a single misplaced bit could disrupt the entire program. However, this process deepened our understanding of computer architecture, from the critical role of components like ROM to problem-solving strategies for technical obstacles. Collaboratively, we honed our teamwork skills by distributing tasks effectively and refining our approach through iterative troubleshooting. Ultimately, the experience not only strengthened our technical proficiency but also highlighted the value of careful planning and adaptability in complex engineering endeavors. Finally, seeing the CPU execute instructions flawlessly after countless iterations was an immensely rewarding feeling,resulting from our collective effort and perseverance.

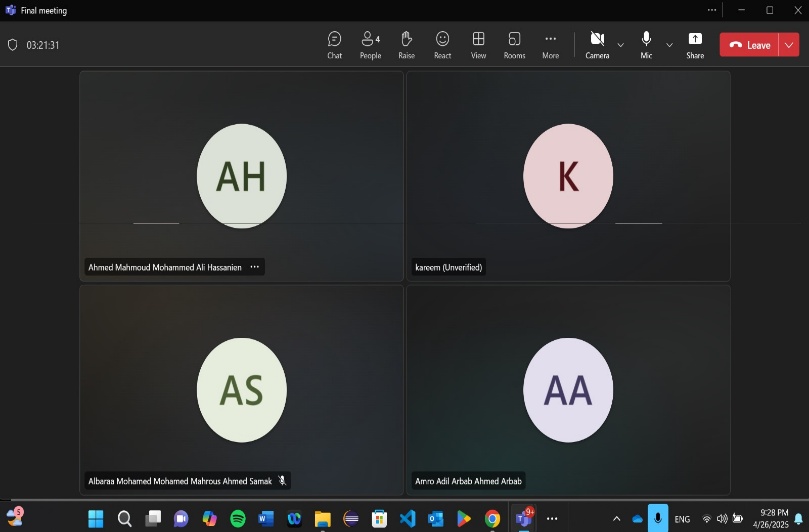
**Note:**

**to load test data to the RAM, we used two data files: one where JMP is executed (RAMvaluesWithJump) and one where it is not (RAMvaluesNoJump). All data is identical in both files apart from the value at address 0xBD, where it was 09 in one file and 04 in the other.**

**Proof of Collaboration and Project Effort**

As part of our course project, our team held multiple meetings to coordinate, plan, and complete the required tasks. The screenshot below is taken from our different meetings, where we discussed our progress, finalized the project details, and ensured that all objectives were met. This demonstrates our commitment to teamwork, consistent communication, and dedication to delivering quality work.

A computer screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.