VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

LSI LOGIC DESIGN

ASSIGNMENT

"Simple RISC CPU Design"

Instructor(s): TÔN HUỲNH LONG

Students: Phạm Viết Huy - 2211269 - L03 - Group 2
Huỳnh Minh Nhật - 2212390 - L03 - Group 2
Đỗ Ôn Trí Quân - 2212783 - L03 - Group 2
Hồ Ngọc Anh Tuấn - 2213768 - L03 - Group 2
Nguyễn Quang Tuấn - 2213790 - L03 - Group 2

HO CHI MINH CITY, MAY 2025

Contents

List of Figures

List of Tables

Member list & Workload

No.	Fullname	Student ID	Problems	% done
1	Phạm Viết Huy	2211269	- Address Mux	100%
2	Huỳnh Minh Nhật	2212390	- Memory	100%
3	Đỗ Ôn Trí Quân	2212783	- Program Counter - ALU	100%
4	Hồ Ngọc Anh Tuấn	2213768	- Instruction Register - Accumulator Register	100%
5	Nguyễn Quang Tuấn	2213790	- Synthesize to RISC - Latex report	100%

Table 1: Member list & workload

1 Introduction

1.1 Overview of the Project

Design of a simple RISC CPU.
In this topic, the group will design a simple RISC CPU with an 8-bit instruction set consisting of 3-bit opcodes and 5-bit operands. The instruction set consists of 8 simple instruction types and 32 address spaces for each instruction memory and data memory.
The processor operates on clock and reset signals. The program will stop when the HALT signal is present.

1.2 Tools and Equipment Used

Tools:

- **Vivado software:** Used for designing, simulating, and synthesizing RISC CPU hardware. Vivado provides a powerful development environment, supporting hardware description languages such as Verilog and VHDL.
- **GitHub:** Used for managing source code, version control, and effective team collaboration. GitHub helps track code changes and facilitates coordination among team members.

1.3 Product Functions The main

Features and functions of the CPU are described in the table below:

Features	RISC CPU
Instruction set architecture	Design an 8-bit instruction set with 3-bit opcode and 5-bit operands
Operating cycle	Multi-cycle (each instruction executes in multiple clock cycles)
Memory	Instruction and data memory with 32-cell address space
Control signal processing	Supports Clock, Reset, and HALT signals Data
processing accuracy	Supports 8-bit data processing and 5-bit operands
Simulation and implementation	Simulation on Vivado and implementation on FPGA Arty-Z7 20
Scalability	Easily edit and extend scripts or functions

Table 2: Features of RISC CPU Multi-cycle

2 Theoretical basis

2.1 Functional blocks in CPU

2.1.1 Program Counter

- Counter is an important counter used to count program instructions. There is also can be used to count program states.
- Counter must be active when clk pulse is rising.
- Reset triggers high, counter returns to 0.

- Counter with counting width of 5.
- Counter has the function of loading any number into the counter. Otherwise, the counter will operate Normal.

2.1.2 Register

- Input signal has width of 8 bits
- rst signal synchronizes and triggers high level.
- Register must be active when clk pulse is up.
- When there is a load signal, the input value will be transferred to the output.
- Otherwise the output value will remain unchanged.

2.1.3 Address Mux

- The Address Mux block with the Mux function will choose between the instruction address in the instruction fetch phase and the operand address in the instruction execution phase.
- Mux will have a default width of 5.
- The width should be used as a parameter so that it can still be changed if needed.

2.1.4 Memory

- Memory will store instructions and data.
- Memory needs to be designed with separate read/write functions using.
- Single bidirectional data port. Cannot read and write at the same time.
- 1-bit read/write enable signal
- Memory must be active when clk pulse is up.

2.1.5 ALU

- The ALU performs arithmetic operations. The calculation performed will depend on the operator of the command.
- The ALU performs 8 operations on 8-bit terms (`inA` and `inB`). The result will be an 8-bit output and 1-bit `is_zero`.
- `is_zero` operates asynchronously to indicate whether the input `inA` is zero or not.
- The 3-bit opcode input determines which operation is used as described in the following table:

Opcode	Code	Operation
HLT	000	Stop the program
SKZ	001	First check if the result of ALU is 0 or not, if it is 0 then skip the next instruction, otherwise continue execution as usual
ADD	010	Add the value in Accumulator to the address memory value in the instruction and the result is returned to Accumulator.
AND	011	AND the value in Accumulator and the address memory value in the instruction and the result is returned to Accumulator.
XOR	100	Performs XOR of the value in the Accumulator and the memory address value in the instruction and the result is returned to the Accumulator.
LDA	101	Performs reading the value from the address in the instruction and putting it into the Accumulator.
STO	110	Performs writing the data of the Accumulator to the address in the instruction.
JMP	111	Unconditional jump instruction, jumps to the destination address in the instruction and continues executing the program.

Table 3: Supported CPU instructions

2.1.6 Controller

- Controller manages the control signals of the CPU. Including fetching and executing instructions.
- Controller must operate when clk pulse is up.
- Rst signal is synchronized and activated at high level.
- 3-bit opcode signal corresponding to ALU.
- Controller must have 8 or more operating states.

2.2 Design Methodology

- The group designs a state machine for each command in the Controller.

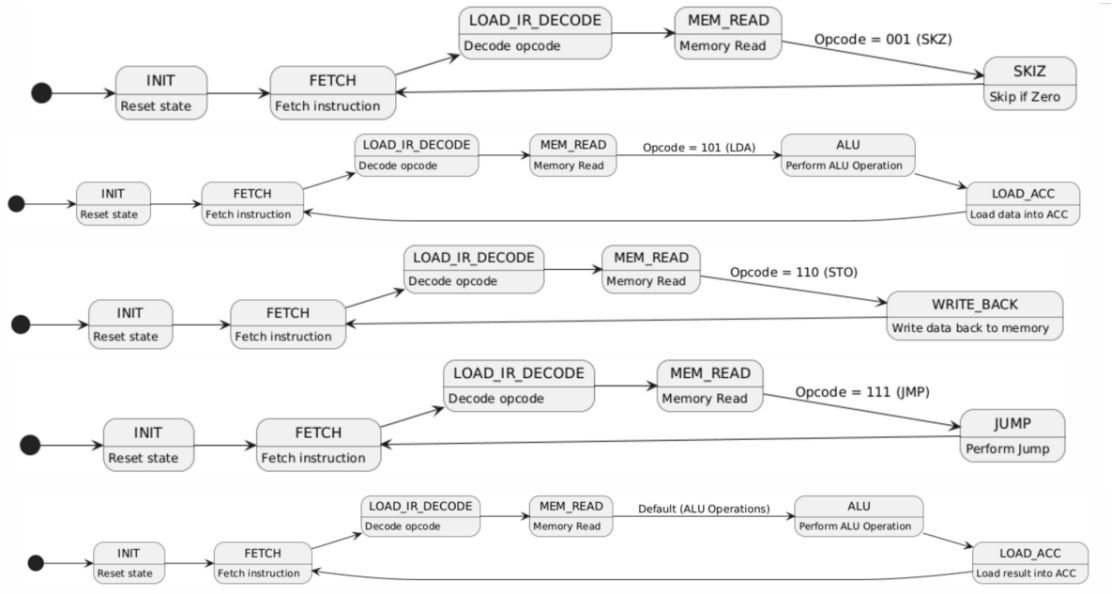

Figure 1: FMS State for Each Instruction

3 System Design

3.1 Overall Block Diagram

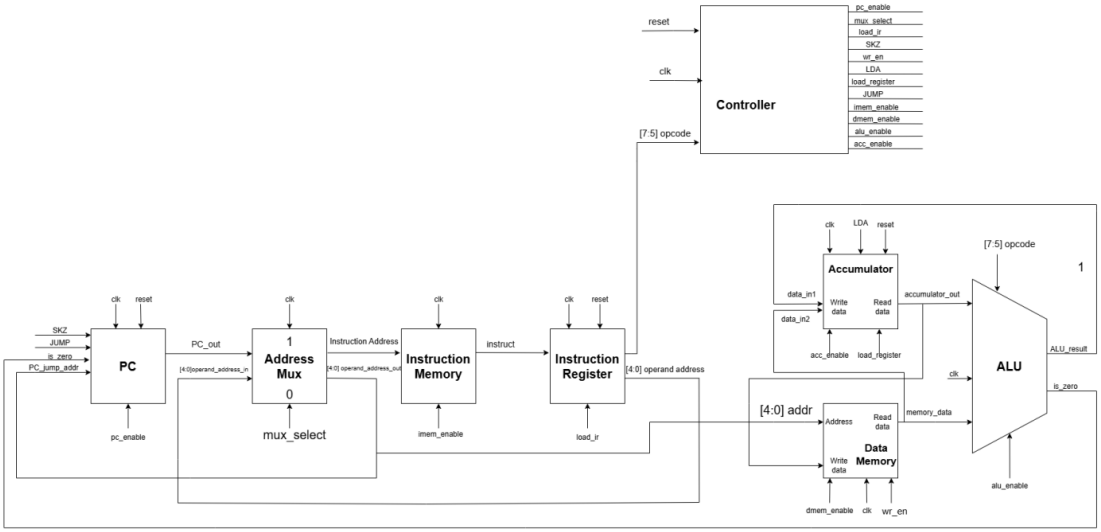

Figure 2: Block diagram of the entire circuit

3.2 Detailed description of functional blocks

3.2.1 Program Counter

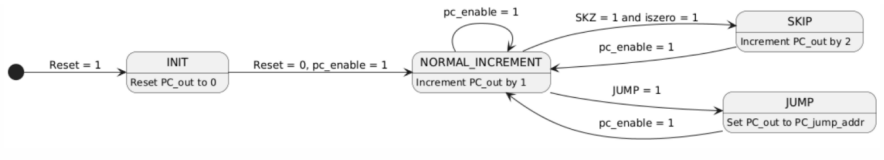

Figure 3: Program Counter (PC) Diagram

- **NORMAL INCREMENT:** The PC register is automatically incremented at the end of each instruction.
- **SKIP AND JUMP:** When special instructions such as SKIZ and JUMP are present, the PC register will wait for a signal and calculate the address before performing the jump.

3.2.2 Register Instruction

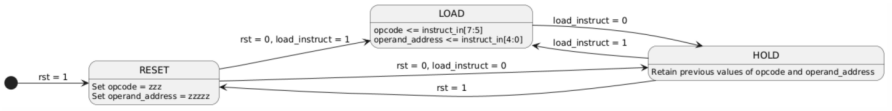

Figure 4: Register Instructions

- Instruction Register is responsible for providing Opcode and instruction from Instruction Memory

3.2.3 Address Mux

3.2.3.a Simple instruction cycle

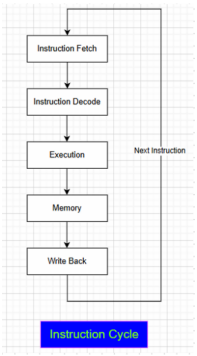

Figure 5: Cycle Instruction

- **Instruction Fetch (IF):** Fetch the instruction from memory at the address specified by the Program Counter (PC). The instruction is then placed in the Instruction Register.
- **Instruction Decode (ID):** Decode the fetched instruction to determine the instruction type and associated operands. The CPU analyzes the instruction to determine what operation it requires, such as math, logic, or memory access.
- **Execution (EX):** Executes the decoded instruction, typically performing math in the ALU (Arithmetic Logic Unit) or testing conditions.
- **Memory Access (MEM):** For memory-related instructions, the CPU accesses memory to read data (if the instruction is a read) or write data (if the instruction is a write).
- **WriteBack (WB):** Write the result of an instruction back to memory or registers (such as the Accumulator) to complete the instruction cycle.

3.2.3.b Flow chart

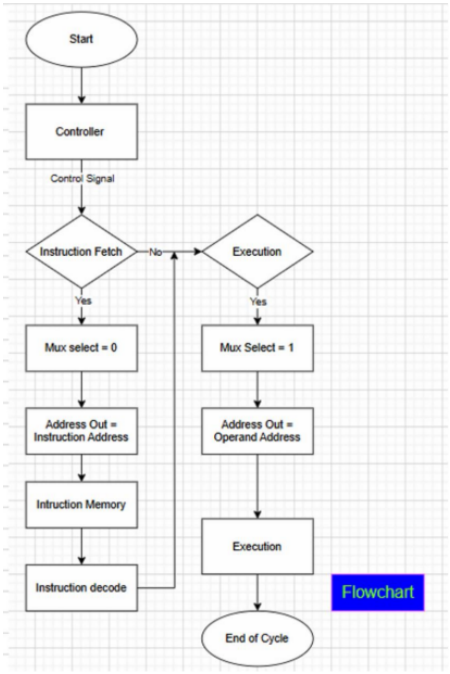

Figure 6: Flowchart Address Mux

- If in the **Fetch** state, the `Mux select` signal is 0, the output of the block will be the instruction address. Through this address, we will enter the Instruction Memory to get the instruction, then Decode and execute the instruction.
- If in the **Execution** stage, the `Mux select` signal is 1, the output of the block will be the operand address. Through this address, we will enter the Data Memory to get the data, then perform the calculation.

3.2.3.c Block diagram design

- Based on the above Flowchart, we can build a block diagram to meet the functional requirements of Address Mux as follows:

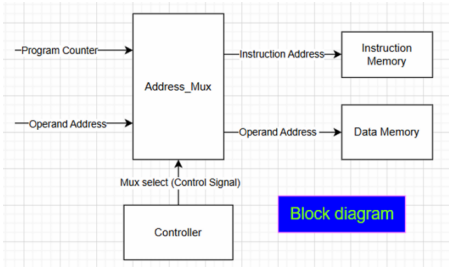

Figure 7: Address Mux block diagram

3.2.4 Memory

3.2.4.a Instruction memory

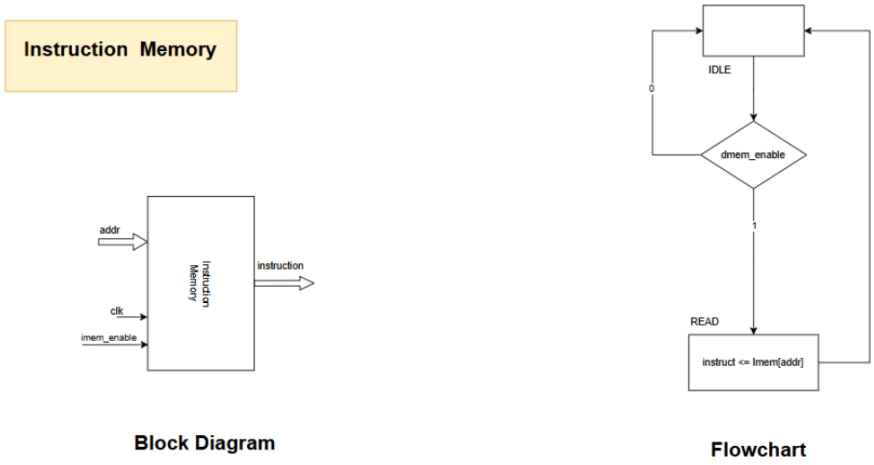

Figure 8: Block diagram and flowchart of the instruction memory block

The instruction memory block has the characteristic **read-only**, which only allows reading and fetching instructions from the memory. Compared to the original specification, there will be an additional signal `imem_enable` because the memory block is divided into two parts: instruction and data.

The `imem_enable` signal is the signal that allows access to the instruction memory block, while `data_enable` is the signal that allows access to the data memory block.

Input:

- Synchronization signal `clk`
- Block access enable signal `imem_enable`

- Instruction address `addr`

Output:

- 8-bit instruction code

3.2.4.b Data memory

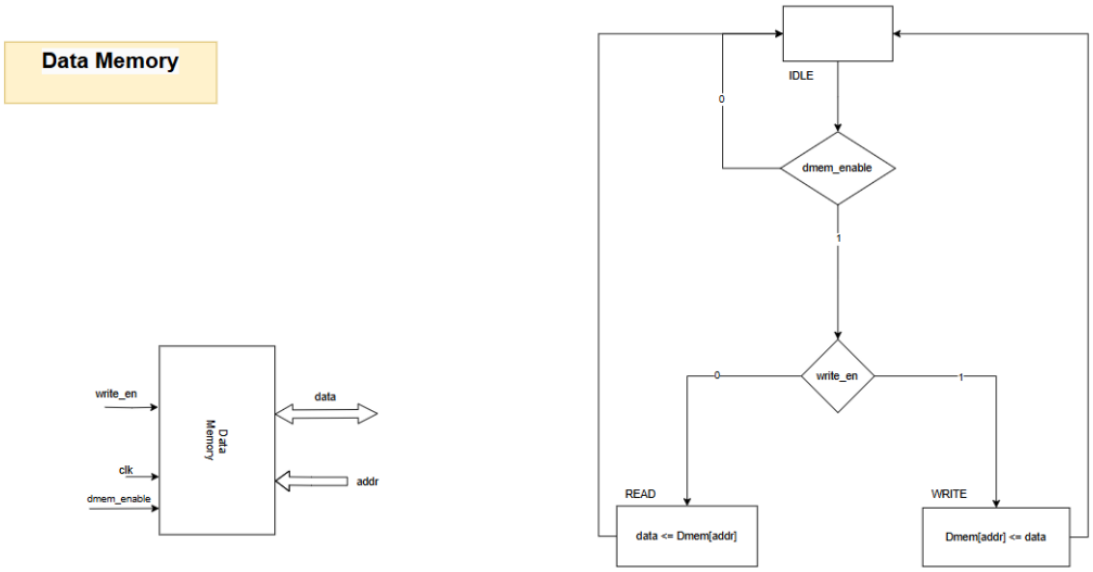

Figure 9: Block diagram and flowchart of data memory block

Similar to instruction memory, `data_enable` is the signal that enables access to the data memory block. The data memory block allows access in both **read** and **write** modes, so the `write_enable` signal is needed to distinguish between the two modes.

When the `write_enable` signal is enabled, data from the `data` input is written to the memory (**write state**). Conversely, when this signal is not enabled, the system will read data from the memory to the `data` output (**read state**).

Input:

- Synchronization signal `clk`
- Signal to enable access to the `dmem_enable` block
- `write_enable` signal to distinguish read/write states
- Data address `addr`
- 8-bit data to be written to `data` memory (if in write state)

Output:

- 8-bit data read from `data` memory (if in read state)

3.2.5 Accumulator

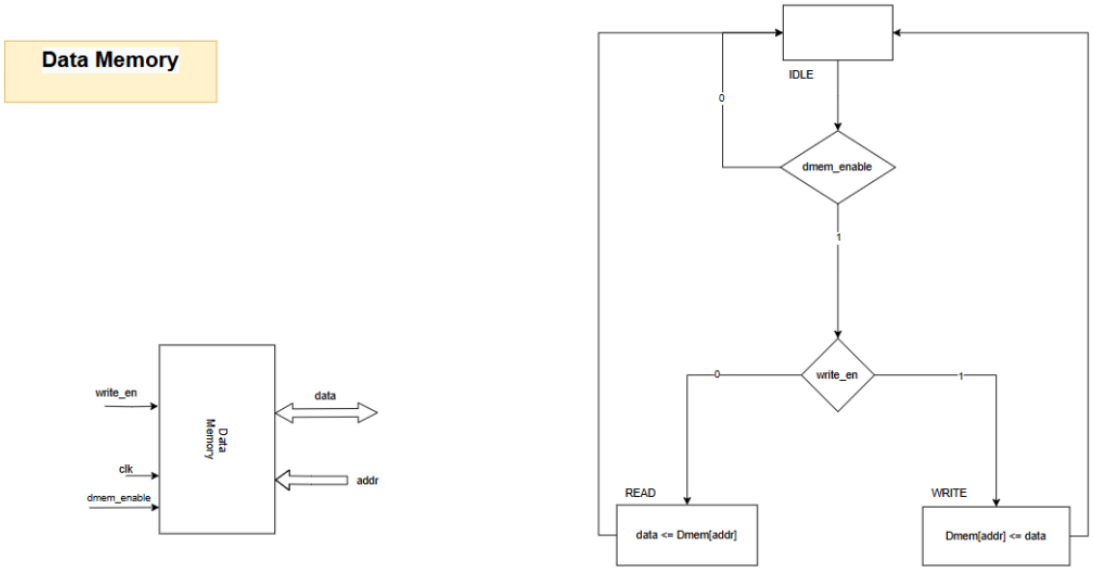

Figure 10: Block diagram of Accumulator Register

- **LDA:** Transfer data from Memory block to Accumulator.
- **LOAD:** Transfer data after calculation in ALU to Accumulator.

3.2.6 ALU

3.2.6.a Flow chart

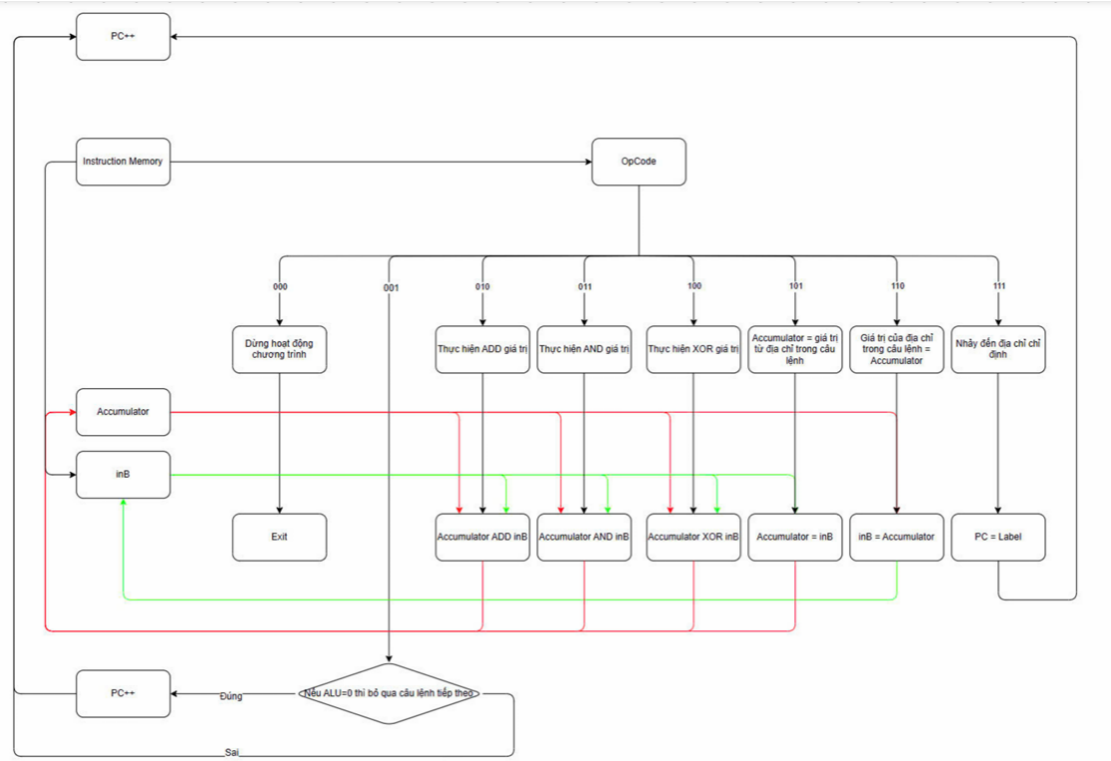

Figure 11: ALU Flowchart

3.2.6.b Block diagram

Based on the Flowchart and block diagram, we have specified the function of the ALU block according to the requirements:

- Perform arithmetic and logic operations.
- The operations depend on the **3-bit opcode** of the instruction, including:
  - **HLT (000):** Stop the program.
  - **SKZ (001):** Check if the ALU result is 0. If it is 0, skip the next instruction; otherwise, continue executing the instruction.
  - **ADD (010):** Add the value in the Accumulator with the value from Memory, save the result to the Accumulator.
  - **AND (011):** Performs AND operation between Accumulator and Memory, stores result in Accumulator.
  - **XOR (100):** Performs XOR operation between Accumulator and Memory, stores result in Accumulator.

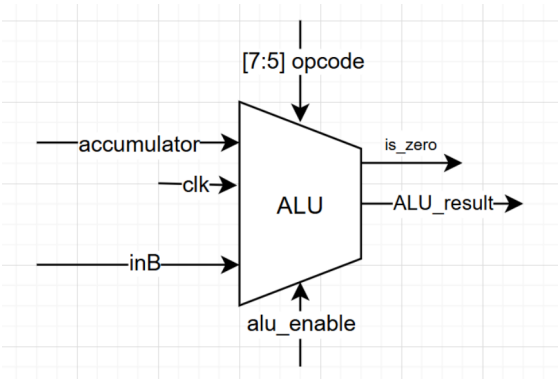

Figure 12: ALU block diagram

- **LDA (101):** Reads value from Memory into Accumulator.
- **STO (110):** Writes value from Accumulator into Memory.
- **JMP (111):** Unconditional jump to target address in instruction.

Input:

- Two 8-bit operands (`accumulator` and `inB`).
- 3-bit `opcode` to identify the type of operation.
- `alu_enable` signal generated from the Controller block to enable the ALU to operate.
- The Controller must operate when the `clk` pulse is raised.

Output:

- 8-bit result.
- `is_zero` flag (1-bit) indicates whether the ALU result is 0 or not.

3.2.7 Controller

The Controller Block's main task is to manage and coordinate the operations of the components in the system based on the control signals and the **3-bit opcode** of the current instruction.

Input:

- Controller operates when the `clk` pulse is up.
- The `reset` signal synchronizes and activates the high level.
- The `opcode` 3-bit signal matches the ALU.

Output:

- The `pc_enable` signal is used to activate the PC block.
- The `mux_select` signal controls the Address Mux to select the instruction address or memory address.

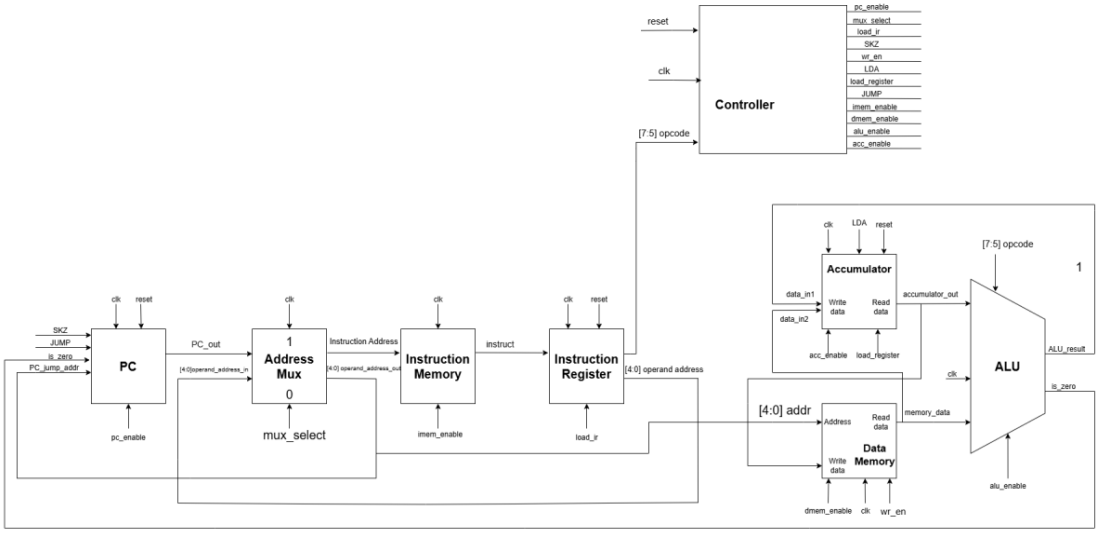

Figure 13: Block diagram and flowchart of the controller block.

- The `imem_enable` signal enables the Instruction Memory to fetch instructions.
- The `load_ir` signal enables the Instruction Register to decode instructions.
- The `SKZ` and `JUMP` signals combine with `pc_enable` to increment the PC address for each corresponding instruction.
- The `dmem_enable` signal enables the Data Memory.
- The `wr_en` signal enables the Data Memory to read or write data.
- The `acc_enable` signal enables the Accumulator Register.

Instruction Control:

- The `LDA` signal combines with `acc_enable` to cause the LDA instruction to write to the Accumulator Register.
- The `load_register` signal combines with `acc_enable` to write to the Accumulator Register (except for the LDA instruction).
- The `alu_enable` signal activates the ALU to perform the calculation.

4 System implementation

4.1 Program Counter

- PC is designed to provide instruction addresses to Instruction Memory in each instruction cycle.
- Main function: increment PC address after each clock cycle.

4.2 Address Mux

- Main function: used to select instruction addresses in the instruction fetch phase and operand addresses in the instruction execution phase.

4.3 Instruction Memory

- Main function: retrieve the current instruction from the address provided by the PC so that it can be used.

4.4 Instruction Register

- Main function: retrieve the instruction provided from Instruction Memory for decoding including the operand address and opcode of the instruction.

4.5 Data Memory

- Main function:
  - For arithmetic instructions, the logic takes the provided operand address to retrieve the data provided to the ALU to perform the calculation.
  - For the LDA instruction, the value in memory will be read to provide to the Accumulator.
  - For the STO instruction, the value will be written to the memory.

4.6 Accumulator

- Accumulator is a temporary register to store the result.
- Main function: Receive the result from the ALU or data from the Data Memory.

4.7 ALU

- ALU performs 8 operations based on the opcode.
- Main function:
  - Combines two data (Input A and Input B) and returns the result.
  - The `is_zero` signal indicates whether the result is 0 or not.

4.8 Controller

- The controller ensures that the instructions are executed in the correct order in each clock cycle.
- Operation sequence:
  - Fetch instruction: PC increments, the selected address is placed in Instruction Memory.
  - Decode: Opcode is transferred to the controller.
  - Execute: ALU operates and transfers the result.
  - Write result: The result is stored in the Accumulator or written to Data Memory.

5 Testing and Simulation

Test case:

```
/*****************************************************************************
 * Test program
 *
 * Kết quả cần có: Chương trình sau kết thúc (halt) ở lệnh địa chỉ 17(hex)
 *****************************************************************************/

//opcode_operand  // addr                  assembly code
//--------------  // ----  ----------------------------------------------
@00 111_11110     //  00   BEGIN:   JMP TST_JMP
    000_00000     //  01            HLT
    000_00000     //  02            HLT
    101_11010     //  03   JMP_OK:  LDA DATA_1
    001_00000     //  04            SKZ
    000_00000     //  05            HLT
    101_11011     //  06            LDA DATA_2
    001_00000     //  07            SKZ
    111_01010     //  08            JMP SKZ_OK
    000_00000     //  09            HLT
    110_11100     //  0A   SKZ_OK:  STO TEMP
    101_11010     //  0B            LDA DATA_1
    110_11100     //  0C            STO TEMP
    101_11100     //  0D            LDA TEMP
    001_00000     //  0E            SKZ
    000_00000     //  0F            HLT
    100_11011     //  10            XOR DATA_2
    001_00000     //  11            SKZ
    111_10100     //  12            JMP XOR_OK
    000_00000     //  13            HLT
    100_11011     //  14   XOR_OK:  XOR DATA_2
    001_00000     //  15            SKZ
    000_00000     //  16            HLT
    000_00000     //  17   END:     HLT
    111_00000     //  18            JMP BEGIN

@1A 00000000      //  1A   DATA_1:            (giá trị hằng 0x00)
    11111111      //  1B   DATA_2:            (giá trị hằng 0xFF)
    10101010      //  1C   TEMP:              (biến khởi tạo với giá trị 0xAA)

@1E 111_00011     //  1E   TST_JMP: JMP JMP_OK
    000_00000     //  1F            HLT
```

Figure 14: Test Case

TestBench:

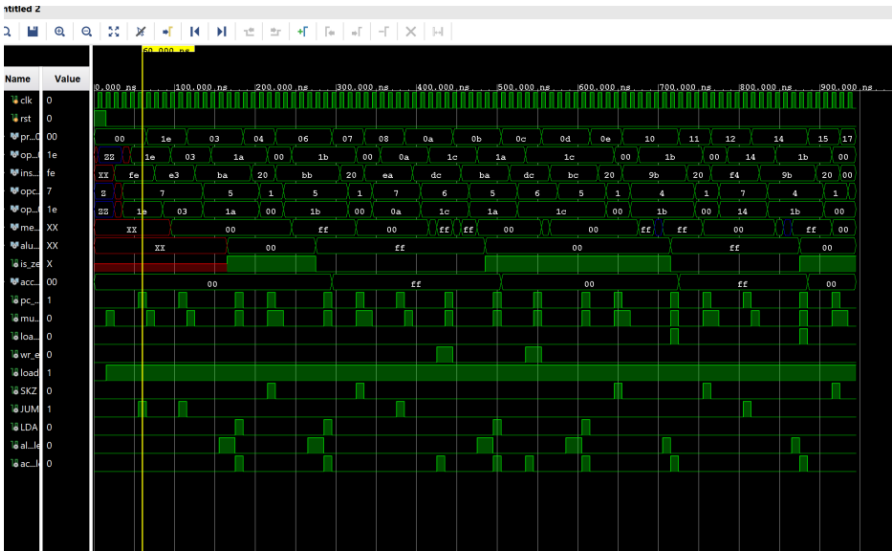

Figure 15: Testbench RTL

- First we will encounter the **JUMP** instruction (Opcode is 111) at address `0x00`. The address JUMP to will be `0x1E`.
- At address `0x1E` we continue to encounter the **JUMP** instruction and jump to address `0x03`.
- At address `0x03` we encounter the **LDA** instruction (Opcode is 101), loading the value `0x00` into the Accumulator.
- At address `0x04` we encounter the **SKZ** instruction (Opcode 001). Since the ALU result is now `0x00`, the PC will jump to address `0x06`.
- At address `0x06` we encounter the **LDA** instruction (Opcode is 101), loading the value `0xFF` into the Accumulator.
- At address `0x07` we encounter the **SKZ** instruction (Opcode 001), since the ALU value is not 0, the PC continues to the next address `0x08`.
- At address `0x08` we encounter the **JUMP** instruction (Opcode 111), jumping to address `0x0A`.
- At address `0x0A` we encounter the **STO** instruction (Opcode 110), which stores the Accumulator value (`0xFF`) in address `0x1C`.
- At address `0x0B` we encounter the **LDA** instruction (Opcode 101), which loads the value `0x00` into the Accumulator.
- At address `0x0C` we encounter the **STO** instruction (Opcode 110), which stores the Accumulator value (`0x00`) in address `0x1C`.

- At address `0x0D` we encounter the **LDA** instruction (Opcode 101), which loads the value `0x00` into the Accumulator.
- At address `0x0E` we encounter the instruction **SKZ** (Opcode 001). Since the ALU result is `0x00`, the PC jumps to address `0x10`.
- At address `0x10` we encounter the instruction **XOR** (Opcode 100), XORing the value `0x00` in the Accumulator with the value at address `0x1B` (`0xFF`), giving the result `0xFF`.
- At address `0x11` we encounter the instruction **SKZ** (Opcode 001), since the ALU value is non-zero, the PC continues to the next address `0x12`.
- At address `0x12` we encounter the **JUMP** instruction, which jumps to address `0x14`.
- At address `0x14` we encounter the **XOR** instruction (Opcode 100), which XORs the value `0xFF` in the Accumulator with the value at address `0x1B` (`0xFF`), giving the result `0x00`.
- At address `0x15` we encounter the **SKZ** instruction (Opcode 001). Since the ALU result is `0x00`, the PC jumps to address `0x17`.
- At address `0x17` we encounter the **HAL** instruction (Opcode 000), the program ends at this address.

6 Design Evaluation and Future Development Directions

6.1 Design Evaluation

Evaluating with the proposed test case set, the team's results are as follows:

- Passed the behaviour simulation.
- Synthesis simulation does not work as expected (caused by timing warnings, the schematic in synthesis is much different from the original schematic).
- The result running on the real circuit does not work properly (the result is similar to synthesis).

6.2 Difficulties encountered

During the project implementation, the team encountered some professional and non-professional difficulties:

- Problems related to synthesis and timing cause the circuit to work incorrectly.
- Knowledge of CPU architecture is still limited, resources have not been optimized.
- Changes in team members, delayed plans and led to poor results.

6.3 Future Development Directions

- **Improve knowledge and skills:**
  - Further study of CPU architecture, focusing on resource optimization techniques.
  - Learn more about timing methods and how to troubleshoot synthesis simulation issues.
- **Refine and optimize circuits:**
  - Revise circuit designs to better handle timing issues.
  - Conduct more real-world testing on a variety of hardware to evaluate overall performance.
- **Functional extensions:**
  - Add new instructions and functions to the CPU to increase its usability, such as supporting complex calculations or peripheral communication.
  - Design a simple cache to improve data access speed.
- **Practical applications:**
  - Deploy this CPU model into simple embedded systems or small applications to evaluate real-world performance.

7 Conclusion

Overall, the group has realized the project's requirements at the RTL level, but there are still many limitations when synthesizing logic from RTL through Synthesis, leading to circuit latency. Through this, the group learned more ways to complete a project by running multi-part analysis, to accurately evaluate the circuit's applicability when running in practice. The group also understood that writing code according to rules is extremely important and has a great influence on the implementation on the real circuit.

The GitHub link is updated by the group here.

8 References

- https://www.fpga4student.com/
- https://www.fpga4fun.com/
- https://sg.element14.com/
- https://www.hackster.io/
- https://www.instructables.com/circuits/howto/fpga/