SAP based Microprocessor Design

Comprehensive Report

Prepared By
Team 5

Version
1.0

Table of Contents

1 Introduction

1.1 Background

Microprocessors stand at the forefront of digital systems and Systems on Chip (SoCs), serving as the foundational building blocks that empower the modern computing landscape. The evolution of microprocessor design has been instrumental in shaping the efficiency, speed, and versatility of digital systems. Rooted in this context, our project focuses on the creation of a 'Basic' microprocessor, drawing inspiration from the Simple As Possible (SAP)-1 architecture.

The SAP architecture, known for its simplicity and educational value, provides a solid framework for students to delve into the core principles of digital design. This project serves as a bridge between theoretical concepts and practical application, offering an immersive experience in crafting a functional microprocessor.

1.2 Objectives

The primary objective of this project is to design and implement an 8-bit microprocessor, adhering rigorously to established design best practices. The microprocessor specifications mandate a minimum of 2 arithmetic operations, 2 logic operations, and one branch operation. Beyond these foundational requirements, students with advanced skills are encouraged to explore additional functionalities, provided they enhance the design without compromising the quality of the design documentation.

A specific focus lies on the control unit, requiring a detailed breakdown of various blocks and a clear articulation of the teamwork plan. The project is designed to not only cultivate technical skills but also to showcase effective team management, reflecting real-world scenarios where collaborative efforts are crucial for success.

1.3 Importance and Applications

The importance of microprocessor design transcends the boundaries of theoretical knowledge, extending into practical applications that drive innovation. Microprocessors serve as the central nervous system of electronic devices, enabling functionalities ranging from simple arithmetic operations to complex computations. The successful design and implementation of a microprocessor not only contributes to advancements in digital systems but also enhances the problem-solving capabilities of the designers.

In the broader context, microprocessors find applications in diverse fields, including embedded systems, IoT devices, communication systems, and beyond. The skills acquired through this project are directly transferable to real-world scenarios, making students well-equipped for challenges in the ever-evolving landscape of digital IC design. The project, therefore, holds significance not only in its educational value but also in its practical implications for future technological advancements.

2 Project Team

In order to design and implement the outlined microprocessor, it is essential to assemble a proficient team consisting of members with expertise in digital IC design, control unit development, RTL implementation, simulation, verification and teamwork. Our team possesses the required skill set, comprising the following members:

Name	AUC ID	Email
Omar Hesham Elshopky	V23010251	omar.elshopky202@gmail.com
Mohamed Ahmed Kamal	V23010268	
Hoda Ashraf Mohamed	V23010471	hodashrafff@gmail.com

2.1 Supervision Team

Name
Dr. Islam Yehia
Eng. Zeina Mohamed Samir

3 Project Scope and Objectives

3.1 Microprocessor Design Overview

In this section, the microprocessor specifications are determined, considering the developed microprocessor as a black box tasked with performing the required functions.

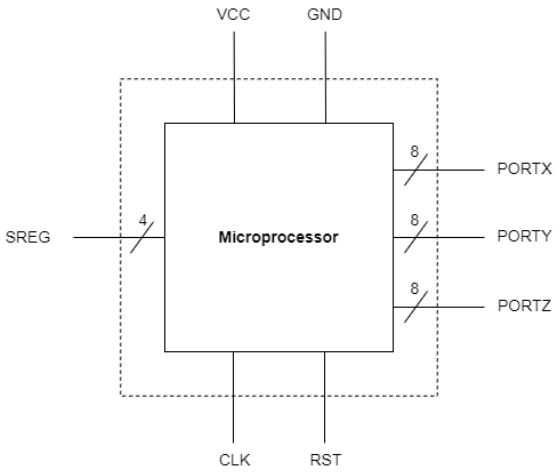


Fig. 1: A block diagram representing the input/output pins of the microprocessor as a black box.

The RISC (Reduced Instruction Set Computing) architecture is followed for the ISA (Instruction Set Architecture) of an 8-bit microprocessor, enabling the execution of multiple arithmetic and logic operations. The microprocessor features four programmer-accessible registers, namely A, B, C, and D, which can be utilized in various operations.

The microprocessor also has three output ports that can be used to display its registers' content to the external world, for instance, through a 7-segment display. This feature allows users to visualize the values stored in these registers during the execution of instructions.

Additionally, a dynamic stack is employed to enhance the capability of executing multiple function calls and branching. This dynamic stack facilitates efficient management of subroutine calls and branching instructions within the microprocessor.

Direct, immediate, and register-based addressing modes are supported in load and store instructions. Furthermore, the microprocessor incorporates a Status Register containing flags obtained from the operations. This register provides information about the status of the microprocessor after each operation.

A single bus is utilized following the Von Neumann Architecture. The technical specifications, including the mentioned features, are outlined in the following table:

Tech Specifications	
Data Width	8 bits
Clock Speed	16 MHz
Memory (RAM)	64 Kb
Registers	9x 8-bit Register File. Among them are 4 programmer-accessible registers, Status Register, and two 16-bit Stack Pointer and Program Counter.
Arithmetic Operations	Addition, Subtraction, Increment, Decrement, Multiplication and Division by 2
Logic Operations	AND, OR, XOR, and Rotation
Branching Operations	Conditional and Unconditional Jump, and Call & Return
Stack	Dynamic Stack Size managed by a 16-bit Stack Pointer (SP)
Input Pins	RST (Reset Pin), CLK (Clock Pin)
Output Pins	Output ALU flags through a 4-bit SREG pins, and display the values of internal registers through three 8-bit {X|Y|Z}PORTs.
Power Consumption	X mW
Instruction Set Architecture	Reduced Instruction Set Computing (RISC)
Bus Architecture	Von Neumann Architecture

Table 1: The technical specifications of the microprocessor.

3.2 Project Goals and Constraints

The project aims to design and implement a microcontroller with the specified characteristics, adhering to best practices in design, Verilog standards, and comprehensive documentation. The preferred approach is to prioritize completeness over complexity.

The SAP-based microprocessor design project concludes upon the completion of the following deliverables:

- Microprocessor design, ranging from high-level conceptualization to detailed sub-block designs.
- A programming guide outlining the instruction set in assembly, hexadecimal, and binary formats.
- Verilog implementation of the designed sub-blocks, integrated to form the desired microprocessor.
- Multiple test benches, including one for each sub-block and another for the top level to verify overall microprocessor functionality.
- An assembler developed in Python to convert assembly instructions into a binary file ready for execution.
- A demonstration video showcasing the microprocessor's verification on FPGA.
- A presentation summarizing the work done, highlighting the characteristics of the microprocessor design.

4 Project Management

4.1 Milestones

The project plan outlines specific milestones that collectively contribute to accomplishing the defined objectives and deliverables presented in the preceding section. These milestones are as follows:

1. **Define Project Objectives**
   Establish the project's goals, scope, and objectives, identifying specific functionalities for the microprocessor.

2. **Select Microprocessor Architecture**
   Explore various versions of SAP, conduct a thorough analysis, and select an appropriate microprocessor architecture that aligns with the project's specific requirements.

3. **Specifications Determination and Instruction Set Architecture**
   Outline the microprocessor specifications, including data width, I/O signals, instruction set architecture, and register configuration.

4. **High-Level Design**
   Create a comprehensive block diagram outlining major components, data paths, and control units in a high-level design.

5. **Control Unit Design**
   Design the control unit along with its Finite State Machines (FSMs), responsible for managing instruction and data flow.

6. **Data Path and ALU Design**
   Design the data path and incorporate the arithmetic logic unit to achieve precise manipulation of data.

7. **Memory Design**
   Design the memory hierarchy components ensuring seamless interfacing and communication within the microprocessor.

8. **Control Unit Components Implementation**
   Implement the FSMs and the components defined during the “Control Unit Design” milestone and perform unit testing on each component to ensure readiness for integration.

9. **Data Path Components Implementation**
   Implement the data path and ALU components defined during the “Data Path and ALU Design” milestone and perform unit testing on each component to ensure readiness for integration.

10. **Memory Components Implementation**
    Implement the memory components defined during the "Memory Design" milestone and perform unit testing on each component to ensure readiness for integration.

11. **Components Integration**
    Integrate units into the complete microprocessor and conduct testing to verify proper communication and coordination.

12. **Simulation and Verification**
    Conduct simulations to validate the RTL design, analyzing microprocessor behavior under various conditions and inputs.

13. **Hardware Implementation**
    Implement the microprocessor on hardware, an FPGA, conducting real-world hardware testing.

14. **Assembler Development**
    Develop a crucial software component, the assembler, by designing algorithms for syntax parsing and object program generation. This facilitates the translation of assembly language programs into machine code, streamlining the programming process for improved efficiency.

15. **Documentation**
    Create comprehensive documentation covering architecture, design specifics, implementation, and testing outcomes in a final report.

4.2 Estimated Timeline

The project progresses through five (**5**) phases, outlined as follows:

I. Phase 1 – Project Initiation
In this initial phase, project objectives, scope, and deliverables are established. The optimal architecture, informed by research and a detailed review of SAP, is selected, and microprocessor specifications are meticulously set, forming a clear guide for the project's trajectory.

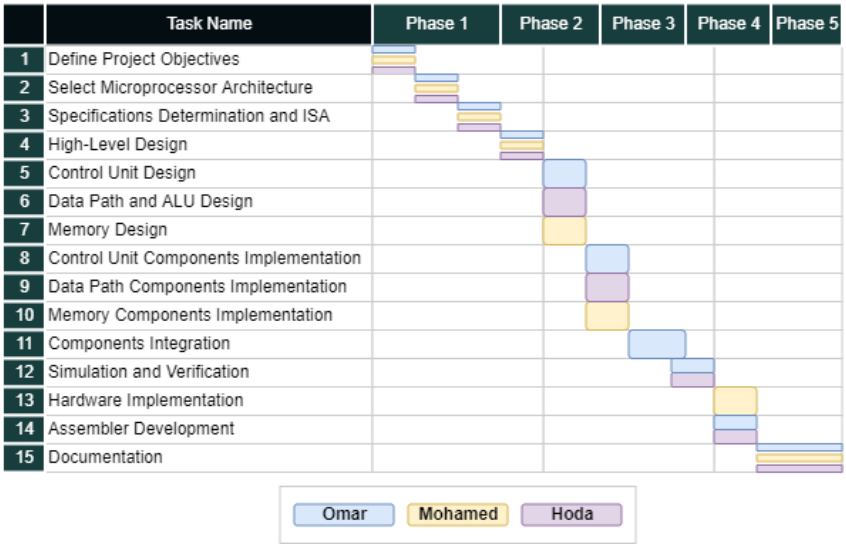
II. Phase 2 – Microprocessor Architecture Design
This phase is centered on the design of the microprocessor, progressing from a detailed block diagram to the construction of subblocks. Simultaneously, the instruction set architecture is defined, specifying opcode assignments, and addressing modes.

III. Phase 3 – RTL Implementation & Unit Testing
During Phase 3, the implementation of microprocessor blocks is undertaken, with rigorous unit testing conducted for each block individually before integration into one system.

IV. Phase 4 – Simulation and Verification
During this phase, simulations are conducted to validate the microprocessor's implementation. Simultaneously, a defined testing plan is executed to ensure the functionality and correctness of the microprocessor.

V. Phase 5 – Final Report Creation
In the final phase, detailed documentation for the microprocessor is created by consolidating sub-documents, offering a comprehensive record of architecture, design, implementation, testing outcomes, and project lessons.

The following is the initial Gantt chart that provides a precise schedule and work plan for each milestone within the project.



4.3 Challenges Faced and Solutions

4.3.1 Instruction Format Creation

As we devise a custom instruction set tailored to meet the specifications of our microprocessor, it becomes imperative to create instruction formats that assist in the design and implementation of control. Initially, we encountered challenges in defining a standardized approach for assigning opcodes to each format, ensuring easy decoding, and encoding of instructions. Ultimately, we drew inspiration from subnetting based on our background knowledge, leading to a valid opcode assignment and defined formats.

For the detailed approach taken see Opcode Assignment Technique.

4.3.2 Limitations of Memory in the Available FPGA

The FPGA provided by the CND for testing imposes a memory limitation of 64 KB for the entire set of blocks. However, our design includes a 64 KB RAM, along with additional ROM for the control unit and other registers, exceeding the available memory capacity. To address this constraint, we opted to reduce the RAM size to 32 KB and set the Stack Pointer to 7FFFH.

5 Microprocessor Architecture

5.1 Choice of Microprocessor Architecture

Before embarking on the design phase and finalizing our microprocessor specifications, a comprehensive review of available microprocessor architectures was conducted. The goal was to select a base model upon which to build our microprocessor. Among the considered options were various variants of the SAP (Simple As Possible) computer, notably SAP-1, SAP-2, and SAP-3 (inspired by the Intel 8080/8085 with some instructions removed).

While SAP-1 offered a simple architecture with essential computer features, it fell short in meeting several points specified in our requirements. Progressing to SAP-2 and SAP-3, we identified advanced capabilities that could enhance our design.

SAP-2 introduced bidirectional registers, reducing wiring capacitance and the count of I/O pins. It also featured a larger memory, providing a more realistic option compared to the 16-byte memory used in SAP-1.

In SAP-3, the introduction of a dynamic stack proved advantageous for call-return applications, surpassing the two slots introduced in SAP-2. Additionally, SAP-3 offered a versatile register file, enabling programmers to reduce the number of memory-reference instructions by leveraging the provided registers in the architecture.

In light of these pivotal considerations, the microprocessor's base models were defined, and the subsequent sections elaborate on how these factors influenced our microprocessor architecture.

5.2 Block Diagram and Components

The high-level design of the microprocessor, depicted in the following diagram, illustrates the key components that constitute its capabilities.

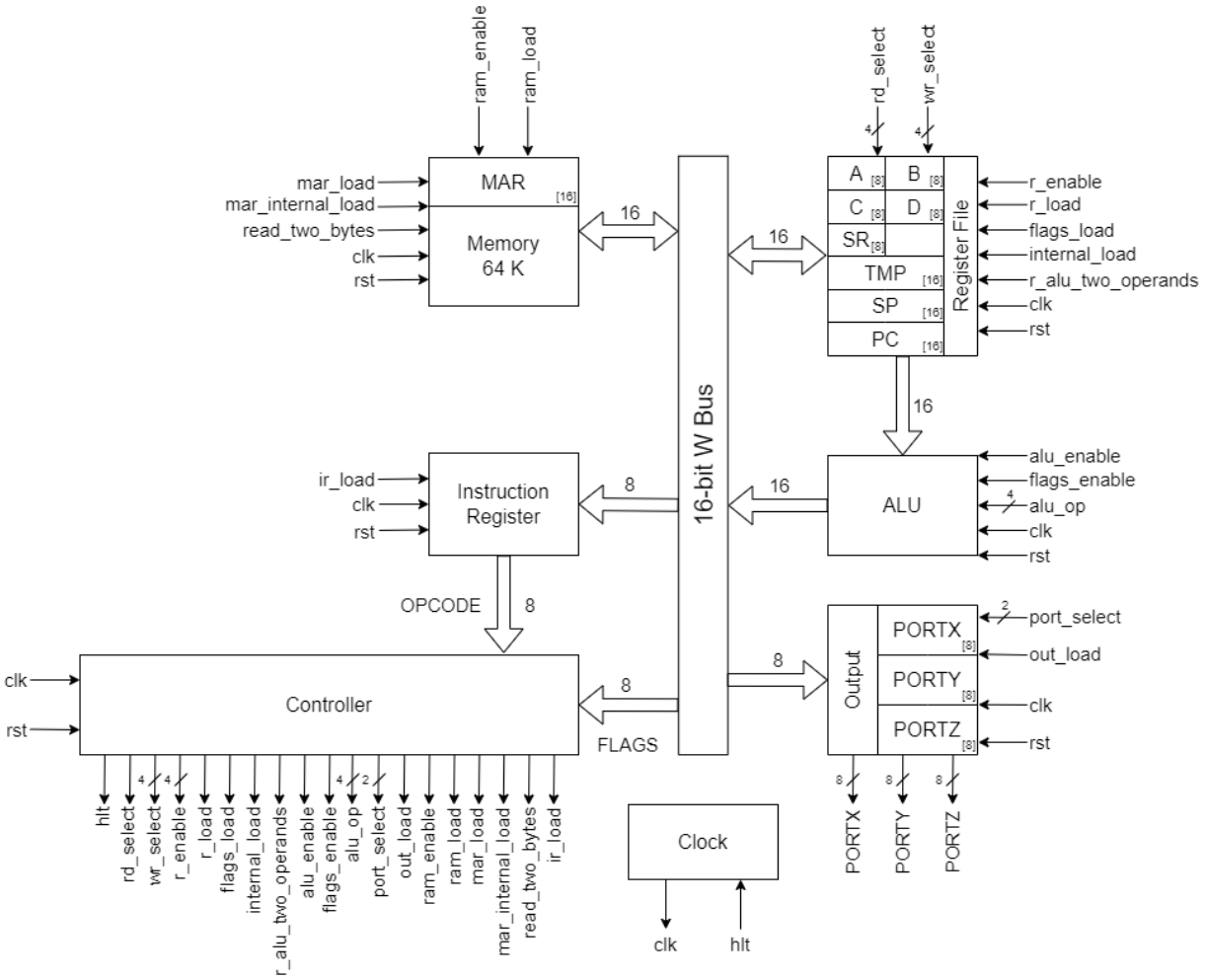


Fig. 2: The block diagram representing main components of the microprocessor.

Input Set
No external source; the microprocessor retrieves its program from the RAM memory, loaded with *'program.bin*,' encompassing both the machine code instructions and the data.

Output Set
Three 8-bit binaries store the content of PORTX, PORTY, and PORTZ, facilitating the display of internal register content. Additionally, a 4-bit binary holds the status register.

Control Signals

31 signals, detailed in the following table, are utilized to control various components of the microprocessor.

Control Signal	Usage
Instruction Register Controls	
ir_load	Load the instruction register with the lowest 8-bit content from the bus, representing the instruction opcode.
Memory Controls	
ram_load	Load the RAM block addressed by the content of MAR with the content from the bus.
ram_enable	Output the content of the RAM block addressed by the MAR to the bus.
mar_load	Load the MAR with the content of the 16-bit bus.
mar_internal_load	Load the MAR with the content of the RAM block addressed by the current value in MAR.
read_two_bytes	Read two bytes out of the memory instead of reading only one byte.
Output Controls	
port_select[1:0]	Select the port register to store the register content into.
out_load	Load the bus content to one of the port registers.
ALU Controls	
alu_enable	Output the ALU result to the bus lowest 8-bit.
flags_enable	Output the flags to the bus highest 8-bit
alu_op[3:0]	Select the arithmetic/logic operation done by the ALU.
Register File Controls	
rd_select[3:0]	Select the register to be read from.
wr_select[3:0]	Select the register to be write into.
r_enable	Output the selected register content to the bus.
r_load	Load the selected register with the bus lowest 8-bit content.
flags_load	Load the Status Register with the bus highest 8-bit content.
internal_load	Load the selected write register, with the content of the selected read register.
r_alu_two_operands	Output both the write and read register to the alu connection bus.
General Controls	
clk	Synchronize the components.
rst	Reset the state of the components.
hlt	Stop the components' clock.

5.2.1 Memory

The memory block is responsible for storing both the program and data in a 64kb RAM. It includes a 16-bit register known as the Memory Address Register (MAR), which holds the address for reading from or writing into the memory.

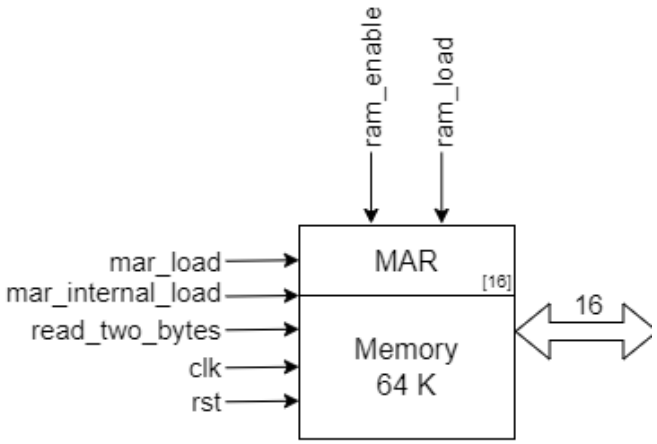


Fig. 3: The memory component.

It has the ability of retrieving two bytes at once, that's why it has 16-bit connection with the bus, and the ability of loading the MAR directly with the value came from the RAM.

5.2.2 Instruction Register

The instruction register (IR) block is tasked with storing the opcode of the instruction received from RAM through the 8-bit bus connection. It then disseminates this opcode to the controller through a combinational 8-bit bus.

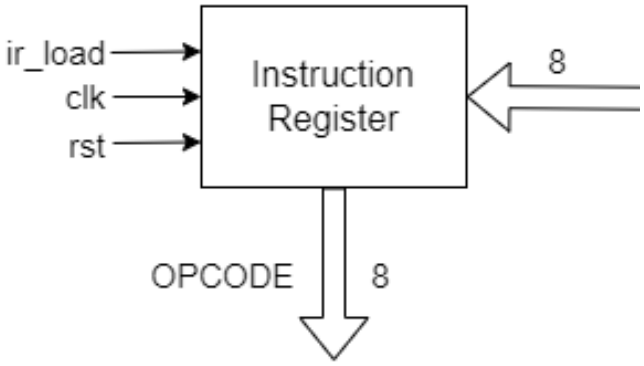


Fig. 4: The instruction regsiter component.

5.2.3 Register File

The register file comprises 12 8-bit registers, categorized as follows:

Programmer Accessible
Four 8-bit registers, denoted as A, B, C, and D, are available for use in various operations.

Programmer Inaccessible

- An 8-bit Status Register comprises four flags: Zero, Carry, Parity, and Sign.

- Three 16-bit addressed registers, namely Temp Register (TMP), Stack Pointer (SP), and Program Counter (PC), are utilized for custom purposes.

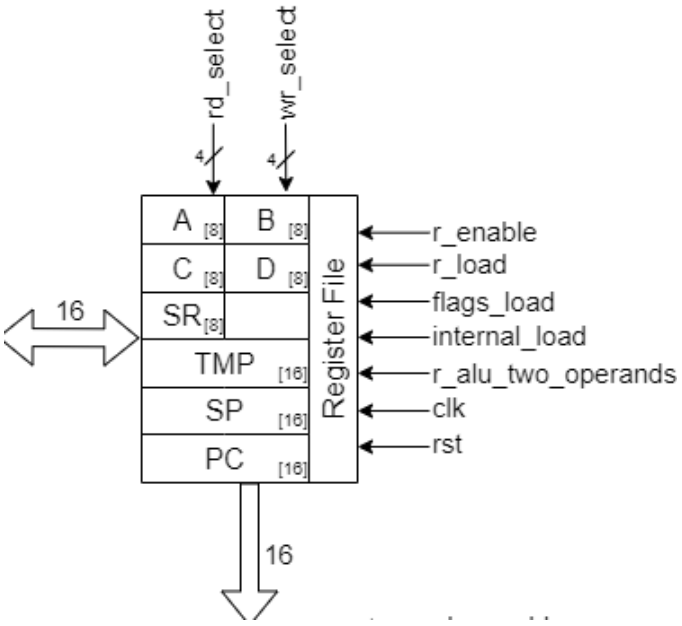


Fig. 5: The register file component.

The register to be read from or written to can be selected through the *rd_select/wr_select* control signals, which encode the registers as follows:

Register	Encoding
A	0000
B	0001
C	0010
D	0011
RS	0100
TMP	0110
SP	1000
PC	1010

It has the capability to internally transfer the content of one register to another and send the content of the registers to the ALU via a 16-bit bus.

5.2.4 ALU

The arithmetic/logic unit takes input from the 16-bit connection with the register file, which contains either two 8-bit data or one 16-bit data. It performs one of the operations on them and outputs the result on the 16-bit bus connection. The output may hold an 8-bit result at the lowest bits and the flags at the higher bits or the total 16-bit result.

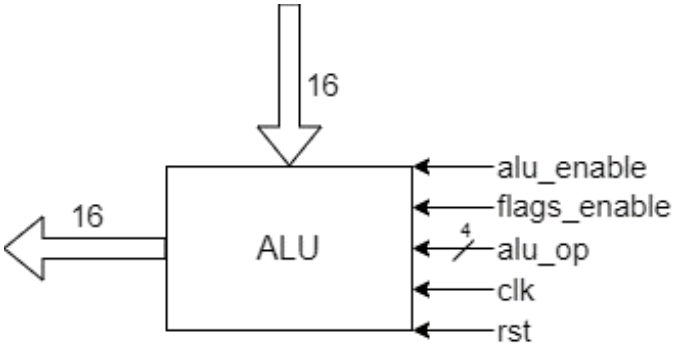


Fig. 6: The arithmetic/logic unit.

The operations that the ALU can perform can be encoded as follows:

Operation	Encoding	Description
ADD	0000	Addition
SUB	0001	Subtraction
INR	0010	Increment by 1
DER	0011	Decrement by 1
ROR	0100	Rotate Right (Division by 2)
ROL	0101	Rotate Left (Multiplication by 2)
AND	0110	Logical AND
OR	0111	Logical OR
XOR	1000	Logical XOR
INR2	1001	Increment by 2

5.2.5 Output Ports

The output ports transmit register content via the 8-bit bus connection to one of their three ports, X, Y, and Z, allowing interaction with the external world.

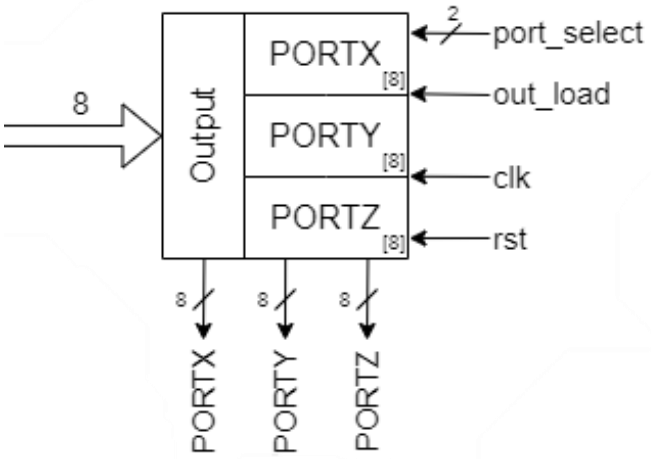


Fig. 7: The output ports component

The output ports can be encoded as follows:

Port	Encoding
PORTX	00
PORTY	01
PORTZ	10

5.2.6 Controller

The controller is tasked with orchestrating the operations of all other components according to the requirements of the current instruction. It generates a control word at the negative edge of the clock, guided by the instruction opcode and its associated substages.

The controller decodes the opcode, determining the necessary microinstructions for executing the instruction. Each microinstruction or T state corresponds to a specific control word value, defining the operation conducted during that stage. Detailed information of the T states of each instruction is provided in the Macroinstructions section.

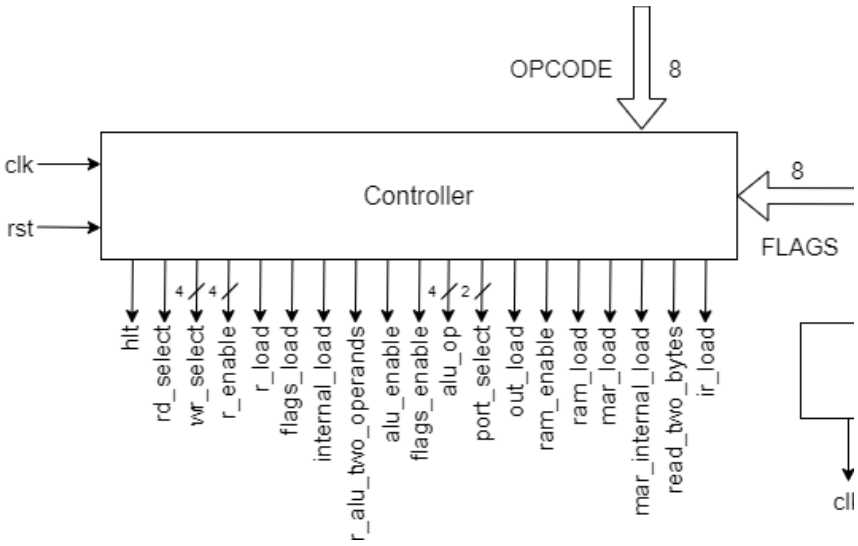


Fig. 8: The controller component

The controller depends on two ROMs: one stores the control words for all stages of each instruction, and the other stores the first address of each instruction in the control ROM.

The controller retrieves the instruction's starting address from the address ROM. It then fetches the control word for this instruction from the control ROM, using the starting address as the base of a presettable counter to iterate over its stages. This method provides more flexibility and power compared to using hard-soldered wires to determine the logic for generating the next control word. The ROM option allows for post-manufacture programming, which is not feasible with the alternative approach.

Control ROM
The ROM has a **width of 29 bits** to accommodate the 29 control signals, with a **length of 256** to hold a total of 256 microinstructions.

Address ROM
The ROM has a **width of 8 bits** to store the address of the 256-byte Control ROM, with a **length of 256** to hold a total of 256 instructions' starting addresses.

The content of both the control ROM and the address ROM is generated by a Python script following the rules defined in the Instruction Formats and the timing diagrams of each instruction described in Macroinstructions. Due to the extensive definition of T states for the distinct 24 operations, no code snippets are inserted in the document. However, the complete script can be found in the project's GitHub repository.

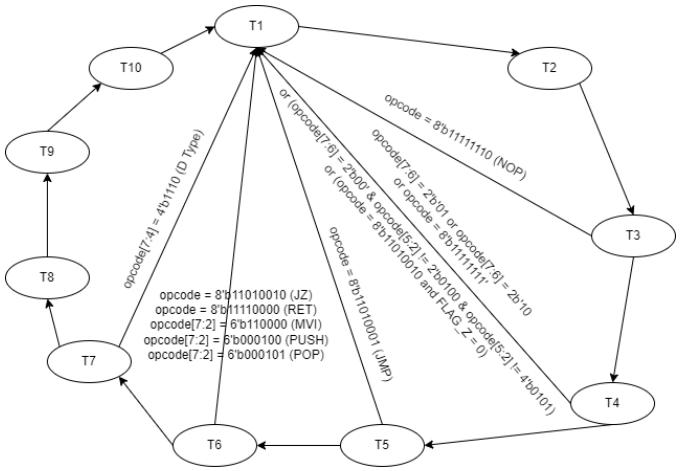
Additionally, the configuration in the Python script allows for easy modification of the control signals for any instruction in a readable manner, providing flexibility for future adjustments and improvements.

```
FETCH_STATES = {
    "T1": {
        "RD_SELECT": "PC",
        "R_ENABLE": 1,
        "MAR_LOAD": 1
    },
    "T2": {
        "RD_SELECT": "PC",
        "WR_SELECT": "PC",
        "R_LOAD": 1,
        "ALU_ENABLE": 1,
        "ALU_OP": "INR"
    },
    "T3": {
        "RAM_ENABLE": 1,
        "IR_LOAD": 1
    }
}
```

```
MNEMONICS = {
    "LDR": {
        "opcode": "111000",
        "states": {
            "T4": {
                "RD_SELECT": "PC",
                "R_ENABLE": 1,
                "MAR_LOAD": 1,
            },
            "T5": {
                "RD_SELECT": "PC",
                "WR_SELECT": "PC",
                "R_LOAD": 1,
                "ALU_ENABLE": 1,
                "ALU_OP": "INR2"
            },
            "T6": {
                "MAR_INTERNAL_LOAD": 1,
                "READ_TWO_BYTES": 1
            },
            "T7": {
                "WR_SELECT": "Rd",
                "R_LOAD": 1,
                "RAM_ENABLE": 1
            }
        }
    },
    "STR": {
        "opcode": "111001",
        "states": {
            "T4": {
                "RD_SELECT": "PC",
```

```
CONTROL_SIGNALS_ENCODING = {
    "IR_LOAD": 0,
    "RAM_LOAD": 1,
    "RAM_ENABLE": 2,
    "MAR_LOAD": 3,
    "MAR_INTERNAL_LOAD": 4,
    "READ_TWO_BYTES": 5,
    "PORT_SELECT_1": 6,
    "PORT_SELECT_0": 7,
    "OUT_LOAD": 8,
    "ALU_ENABLE": 9,
    "FLAGS_ENABLE": 10,
    "ALU_OP_3": 11,
    "ALU_OP_2": 12,
    "ALU_OP_1": 13,
    "ALU_OP_0": 14,
    "RD_SELECT_3": 15,
    "RD_SELECT_2": 16,
    "RD_SELECT_1": 17,
    "RD_SELECT_0": 18,
    "WR_SELECT_3": 19,
    "WR_SELECT_2": 20,
    "WR_SELECT_1": 21,
    "WR_SELECT_0": 22,
    "R_ENABLE": 23,
    "R_LOAD": 24,
    "FLAGS_LOAD": 25,
    "INTERNAL_LOAD": 26,
    "R_ALU_TWO_OPERANDS": 27,
    "HLT": 28
}
```

The controller is also responsible for resetting the counter when the states of the instruction are completed and initiating another fetching cycle, as illustrated in the following finite state machine:



5.2.7 Clock

The clock component is responsible of generating the clock for the other microprocessor components, with hlt control signal which clear the clock and stop the components from run.

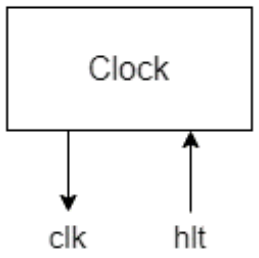


Fig. 9: The clock component

6 Instruction Set Architecture (ISA)

6.1 Definition of Instruction Set

The microprocessor instruction set includes **24** distinct operations that involve diverse initialization, memory-reference, register, arithmetic, logical, branching, stack, and output operations. These operations are outlined in Table 2 and discussed in detail in the subsequent sections.

Appendix 1 contains a comprehensive table detailing each operation across all registers, encompassing a total of **150** instructions.

Instruction	Op Code	Addressing Mode	T states	Flags	Bytes	Type	Main Effect
Memory-Reference Instructions							
LDR ***Rd***, ***address***	111000XX	Direct	7	-	3	D	Rd ← $M_{address}$
STR ***Rs***, ***address***	111001XX	Direct	7	-	3	D	$M_{address}$ ← Rs
Register Instructions							
MOV ***Rd***, ***Rs***	0100XXXX	Register	4	-	1	DR	Rd ← Rs
MVI ***Rd***, ***byte***	110000XX	Immediate	6	-	2	I	Rd ← byte
Arithmetic Instructions							
ADD ***Rd***, ***Rs***	0101XXXX	Register	4	ZCPS	1	DR	Rd ← Rd + Rs
SUB ***Rd***, ***Rs***	0110XXXX	Register	4	ZCPS	1	DR	Rd ← Rd - Rs
INR ***Rd***	000000XX	Register	4	Z-PS	1	SR	Rd ← Rd + 1
DER ***Rd***	000001XX	Register	4	Z-PS	1	SR	Rd ← Rd - 1
Logical Instructions							
ROR ***Rd***	000010XX	Register	4	-C--	1	SR	Rd ← Rd x 2 (Rotate all right)
ROL ***Rd***	000011XX	Register	4	-C--	1	SR	Rd ← Rd / 2 (Rotate all left)
ANR ***Rd***, ***Rs***	0111XXXX	Register	4	ZCPS	1	DR	Rd ← Rd & Rs
ORR ***Rd***, ***Rs***	1000XXXX	Register	4	ZCPS	1	DR	Rd ← Rd | Rs
XRR ***Rd***, ***Rs***	1001XXXX	Register	4	ZCPS	1	DR	Rd ← Rd ^ Rs
Branching Operations							
JMP ***address***	11010001	Immediate	5	-	3	J	PC ← address
JZ ***address***	11010010	Immediate	4/6	-	3	J	PC ← address if Z = 1

Stack Instructions							
CALL ***address***	11010011	Immediate	10	-	3	J	PC ← address
RET	11110000	-	6	-	1	O	PC ← return address
PUSH ***Rs***	000100XX	Register	6	-	1	SR	$M_{stack} - 1 \leftarrow Rs$
POP ***Rd***	000101XX	Register	6	-	1	SR	$Rd \leftarrow M_{stack}$
Misc Instructions							
OUTX ***Rs***	000110XX	Register	4	-	1	SR	PORTX ← Rs
OUTY ***Rs***	000111XX	Register	4	-	1	SR	PORTY ← Rs
OUTZ ***Rs***	001000XX	Register	4	-	1	SR	PORTZ ← Rs
NOP	11111110	-	3	-	1	O	Delay (No Operation)
HLT	11111111	-	4	-	1	O	Stop Processing

Table 2: The distinct operations composed in the instruction set.

6.2 Instruction Formats

The microprocessor relies on instructions to guide its sequential execution of tasks. These instructions must be loaded in machine code form at the outset —comprising 0s and 1s— enabling the machine to comprehend and execute them. Programmers commonly use assembly instructions like ADD, SUB, LDR, etc., which are later translated into machine code using an assembler, a software discussed in more detail in a dedicated section.

To standardize the instruction format, a generic structure is adopted for each individual instruction, as outlined below:

Opcode	Operand
8 bits	8 or 16 bits

The 8 bits allocated for the opcode allow the microprocessor to accommodate 255 different instructions. Although our design currently implements only 151 instructions, each may vary in lengths and layouts, making a random assignment of opcodes impractical.

To optimize the instruction encoding and decoding operations, specific formats should be defined to categorize the instructions into cohesive groups or types. Each group or type adheres to standardized method for encoding and decoding the instructions with similarities in length, layout, and memory addressing mode. This systematic approach not only streamlines the encoding and decoding processes but also facilitates smoother operation in the controller, particularly during the decode cycle.

The microprocessor categorizes instructions into six types:

Type	Instruction Layout	Instruction Length	Opcode	Addressing Mode
SR-Type	ASM R	1 Byte	00XXXXXX	Register
DR-Type	ASM Rd, Rs	1 Byte	01XXXXXX 10XXXXXX	Register
I-Type	ASM Rd, byte	2 Bytes	1100XXXX	Immediate
J-Type	ASM address	3 Bytes	1101XXXX	Immediate
D-Type	ASM Rd, address	3 Bytes	1110XXXX	Direct
O-Type	ASM	1 Byte	1111XXXX	-

Table 3: The different instruction types in our microprocessor. ASM stand for Assembly Keyword.

Further elaboration on each type is provided in the following sections.

6.2.1 SR-Type Instruction

The Single Register type instruction typically performs a specific operation on the value stored in the designated register, adhering to the **register addressing mode** paradigm.

Instruction Layout
[INSTRUCTION_KEYWORK] [REGISTER]

Machine Code Format
Utilize the opcode section of the generic format, excluding the operand.

Opcode		
Operation		Register
SR-Type	Instruction Index	
6 bits		2 bits
00	XXXX	XX

Instruction Length
1 Byte

Instructions
9 Instructions: INR, DER, ROR, ROL, PUSH, POP, OUTX, OUTY, OUTZ

Example
INR B

Opcode		
Operation		Register
SR-Type	Instruction Index	
00	0000	01

6.2.2 DR-Type Instruction

The Double Register type instruction typically carries out a specific operation on the values stored in the two provided registers. The result is then stored in the first register, following the **register addressing mode** paradigm.

Instruction Layout
[INSTRUCTION_KEYWORK] [DESTINATION_REGISTER] [SOURCE_REGISTER]

Machine Code Format
Utilize the opcode section of the generic format, excluding the operand.

Opcode			
Operation		Destination Register (Rd)	Source Register (Rs)
DR-Type	Instruction Index		
4 bits		2 bits	2 bits
01 or 10	XX	XX	XX

Instruction Length
1 Byte

Instructions
6 Instructions: MOV, ADD, SUB, AND, ORR, XRR

Example
AND C, B

Opcode			
Operation		Destination Register (Rd)	Source Register (Rs)
DR-Type	Instruction Index		
01	11	10	01

6.2.3 I-Type Instruction

The Immediate type instruction typically executes a specific operation on a specified register, with an immediate value provided as the operand. This follows **the immediate addressing mode** paradigm.

Instruction Layout
[INSTRUCTION_KEYWORK] [DESTINATION_REGISTER] [IMMIDEATE_BYTE]

Machine Code Format
Utilize both the opcode and the operand sections of the generic format.

Opcode			Operand
Operation		Destination Register (Rd)	Immediate
I-Type	Instruction Index		
6 bits		2 bits	8 bits
1100	XX	XX	XXXXXXXX

Instruction Length
2 Bytes

Instructions
1 Instruction: MVI

Example
MVI D, 15H

Opcode			Operand
Operation		Destination Register (Rd)	Immediate
I-Type	Instruction Index		
1100	00	11	00011001

6.2.4 J-Type Instruction

The Jump type instruction typically performs a specific operation on double-byte operand, which follows **the immediate addressing mode** paradigm.

Instruction Layout
[INSTRUCTION_KEYWORK] [DOUBLE_BYTES_IMMIDIATE]

Machine Code Format
Utilize both the opcode and the operand sections of the generic format.

Opcode		Operand
Operation		Immediate
J-Type	Instruction Index	
8 bits		16 bits
1101	XXXX	XXXXXXXX XXXXXXXX

Instruction Length
2 Bytes

Instructions
4 Instructions: JMP, JZ, CALL

Example
JMP FF46H

Opcode		Operand
Operation		Immediate
J-Type	Instruction Index	
1101	0001	11111111 01000110

6.2.5 D-Type Instruction

The Direct type instruction typically performs a specific operation on the specified register and the memory content at the provided address, which follows **the direct addressing mode** paradigm.

Instruction Layout
[INSTRUCTION_KEYWORK] [REGSITER] [MEMORY_ADDRESS]

Machine Code Format
Utilize both the opcode and the operand sections of the generic format.

Opcode			Operand
Operation		Register	Address
D-Type	Instruction Index		
6 bits		2 bits	16 bits
1110	XX	XX	XXXXXXXX XXXXXXXX

Instruction Length
3 Bytes

Instructions
2 Instructions: LDR, STR

Example
LDR A, F037H

Opcode			Operand
Operation		Register	Address
D-Type	Instruction Index		
1110	00	00	11110000 00110111

6.2.6 O-Type Instruction

The Others type instruction typically perform special operations that are hardcoded into the microprocessor's controller.

Instruction Layout
[INSTRUCTION_KEYWORK]

Machine Code Format
Utilize the opcode section of the generic format, excluding the operand.

Opcode	
Operation	
O-Type	Instruction Index
8 bits	
1111	XXXX

Instruction Length
1 Byte

Instructions
3 Instructions: RET, NOP, HLT

Example
HLT

Opcode	
Operation	
O-Type	Instruction Index
1111	1111

6.3 Opcode Assignment Technique

To facilitate a smooth decoding and encoding process, a designated prefix in the opcode indicates the type of instruction. Given the utilization of custom types, specific opcode assignments must be defined based on a particular technique.

Given that the microprocessor is 8-bit, or half-word, an 8-bit opcode is employed, providing the capability for 256 instructions (2^8).

The chosen technique involves embedding the source/destination register directly into the opcode. For this purpose, 2 bits are allocated to encode the four programmer-accessible registers as follows.

Register	Encoding
A	00
B	01
C	10
D	11

Consequently, types that exclusively utilize a source register require two bits of the opcode for their encoding, such as the SR-type. On the other hand, types that involve both source and destination registers necessitate four bits of the opcode to encode the operation registers. More detailed information about the exact format of each type is provided in the previous section. However, for the purpose of illustrating the technique, the following table displays the remaining bits in the opcode after encoding the source/destination registers (if any) and the number of instructions within each type.

Type	Opcode Remaining Bits	Number of Instructions
SR	6	9
DR	4	6
I	6	1
J	8	3
D	6	2
O	8	3

The objective of opcode assignment is to establish a fixed prefix for each type that identifies it, with the remaining bits used to differentiate between instructions within that type.

Initially, one might consider reserving the first 3 bits for identifying the type. However, this approach poses challenges for certain types. For example, the DR type requires at least 4 bits to encode its 9 instructions, leaving only 1 bit for identification, which is insufficient.

A more effective approach involves using a smaller number of groups, which can then be further divided into subgroups if necessary. Initially, 2 bits are reserved to create 4 groups, distributed among the types based on both the available bit count and the number of instructions.

Starting with the SR type, which has 6 available bits and 9 instructions, these can be encoded into 4 bits, leaving the first group (00) with no remaining capacity.

The second type, DR type, has 6 instructions requiring 3 bits for encoding. As it only has 4 available bits, it needs to take two groups (01, 10), with each containing 4 instructions, totaling 8, which accommodates the available 6 instructions.

The remaining group is assigned to I, J, D, and O types. Since all of them have 6 or more available bits, an additional two bits can be reserved to introduce subgroups for further distinction between them.

Group	Subgroups	Type	Instructions		
			Available	Used	Free
00		SR	16	9	7
01		DR	8	6	2
10					
11	00	I	4	1	3
	01	J	16	3	13
	10	D	4	2	2
	11	O	16	3	13

The table above illustrates the distribution of groups and subgroups among instruction types, along with the available number of instructions for each type. It also highlights the free opcodes based on the utilized technique.

6.4 Macroinstructions

6.4.1 LDR

Load specific register with the addressed memory data.

Instruction Layout
LDR *Rd*, *address*

Machine Code Format

Opcode			Operand
Operation		Register	Address
D-Type	Instruction Index		
6 bits		2 bits	16 bits
1110	00	XX	XXXXXXXX XXXXXXXX

Instruction Length
3 Byte

Example
LDR A, 1F15H

Load the data from the memory location with the corresponding address 1F15H into the A register.

T States and Control Signals

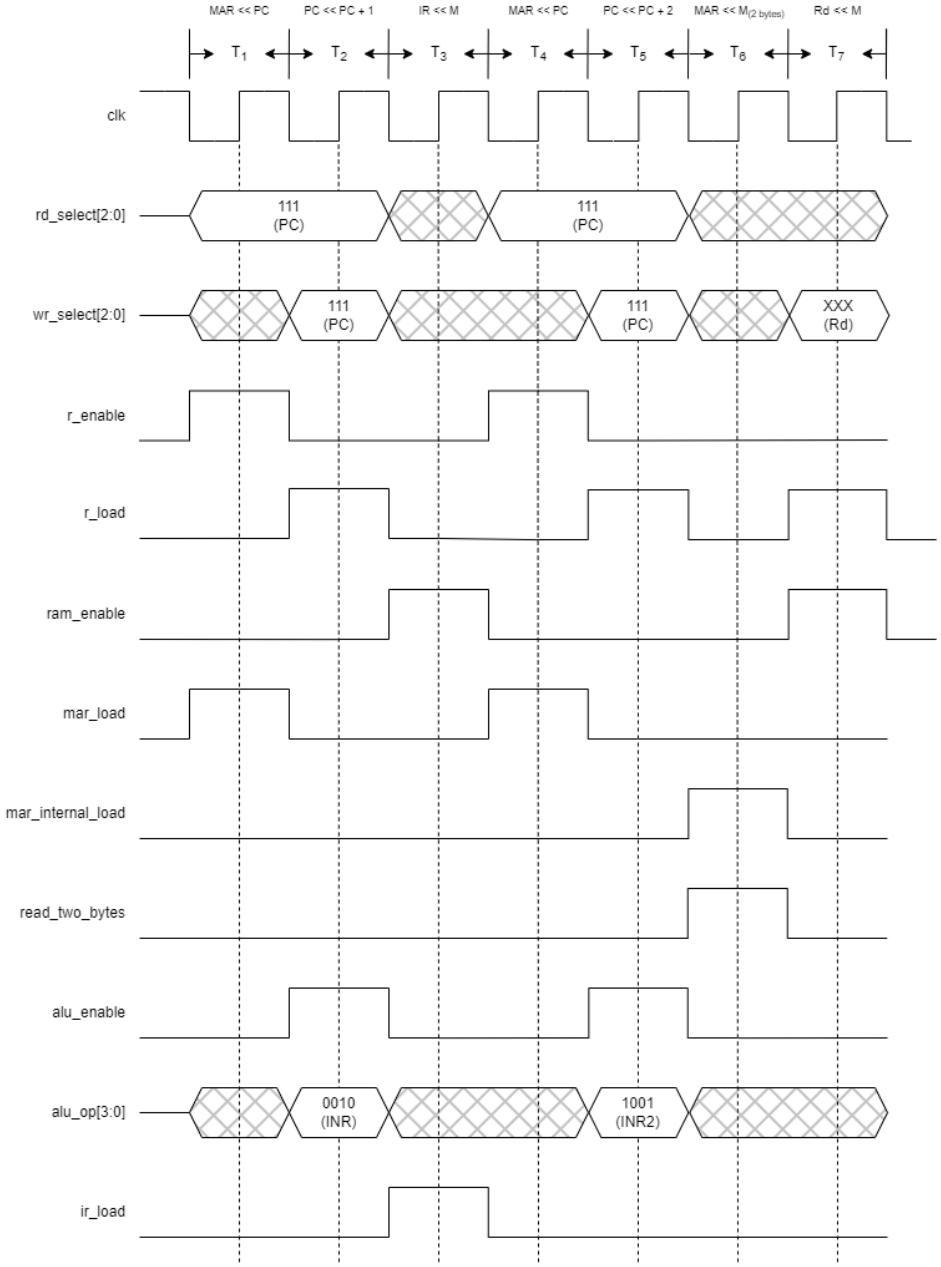


Fig. 11: The timing diagram of the LDR instruction.

6.4.2 STR

Store the value of a specified register into a designated memory address.

Instruction Layout
STR *Rs*, *address*

Machine Code Format

Opcode			Operand
Operation		Register	Address
D-Type	Instruction Index		
6 bits		2 bits	16 bits
1110	01	XX	XXXXXXXX XXXXXXXX

Instruction Length
3 Byte

Example
STR C, 1A55H

Store the content of register C to the memory address 1A55H.

T States and Control Signals

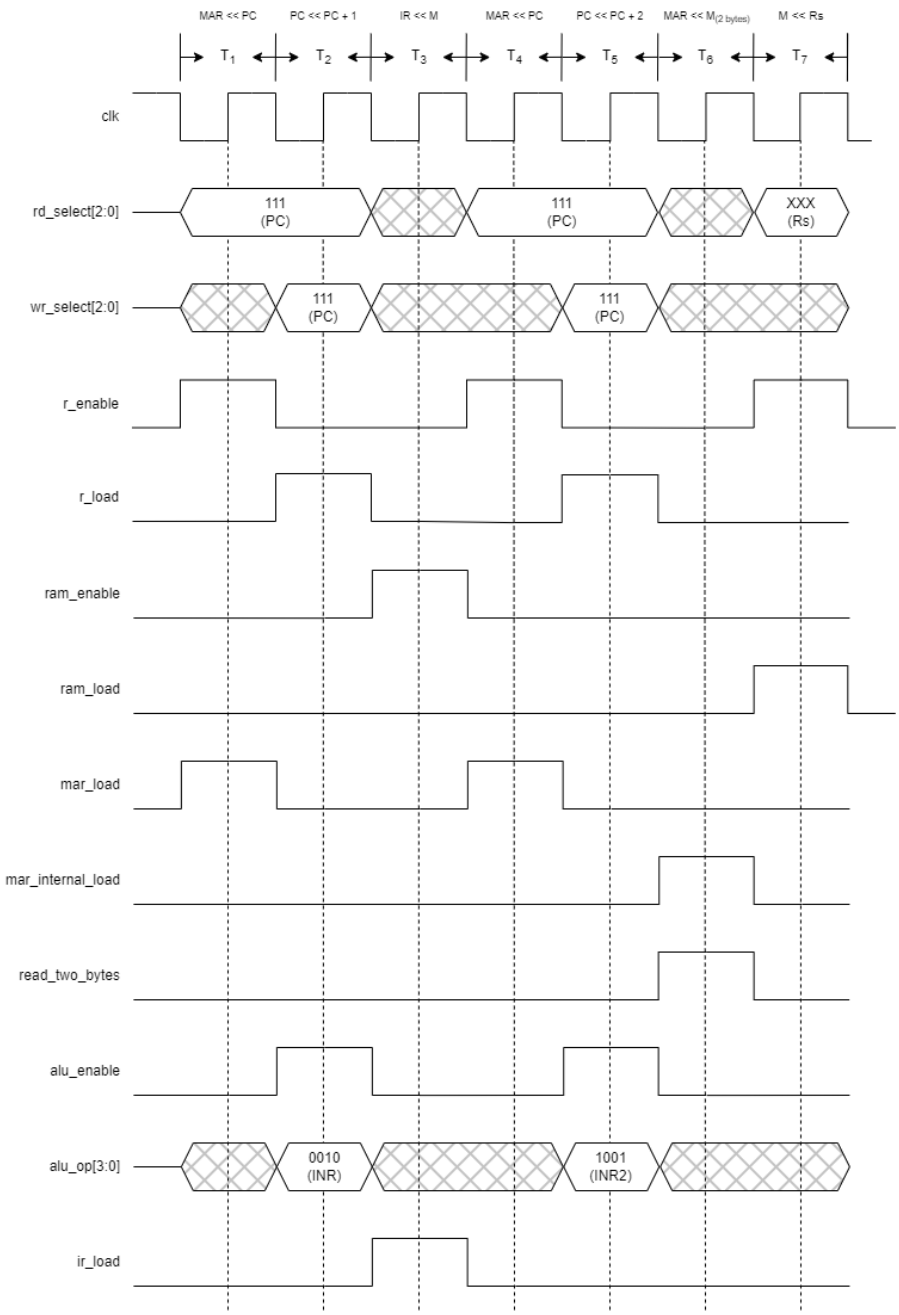


Fig. 12: The timing diagram of the STR instruction.

6.4.3 MOV

Copy the value from one register to another without erasing the content of the source register.

Instruction Layout
MOV *Rd*, *Rs*

Machine Code Format

Opcode			
Operation		Destination Register (Rd)	Source Register (Rs)
DR-Type	Instruction Index		
4 bits		2 bits	2 bits
01	00	XX	XX

Instruction Length
1 Byte

Example
MOV D, B

Copy the data from register B to register D.

T States and Control Signals

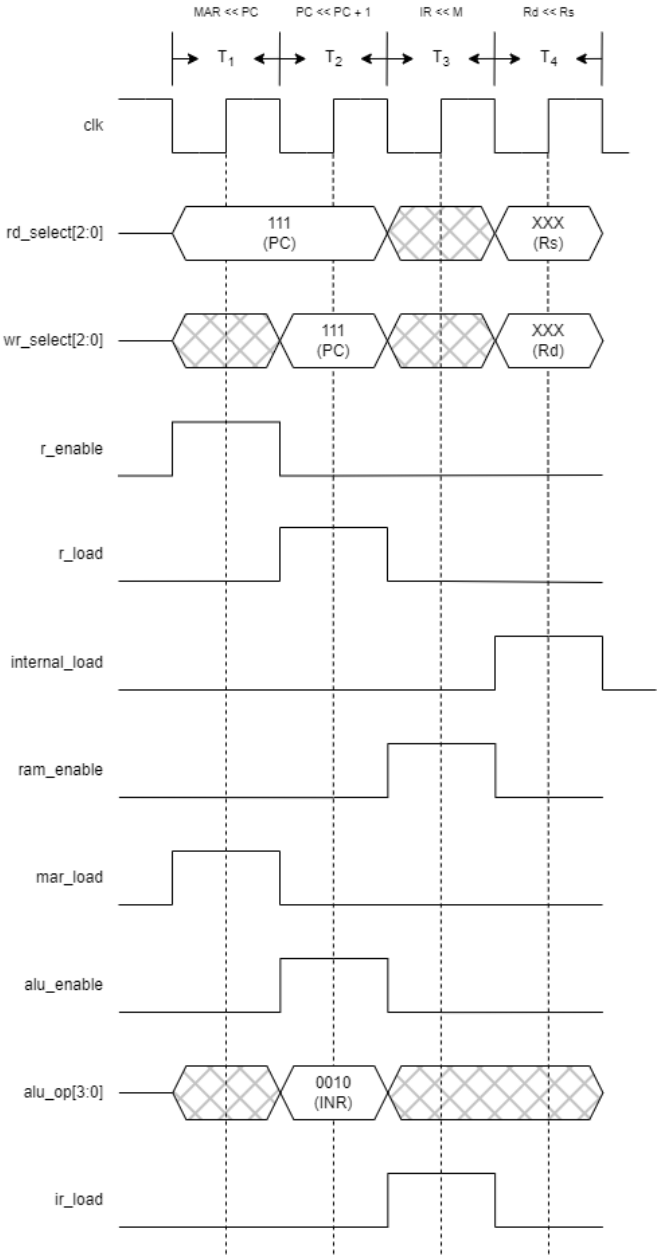


Fig. 13: The timing diagram of the MOV instruction.

6.4.4 MVI

Assign an immediate value to the designated register.

Instruction Layout
MVI *Rd*, *byte*

Machine Code Format

Opcode			Operand
Operation		Destination Register (Rd)	Immediate
I-Type	Instruction Index		
6 bits		2 bits	8 bits
1100	00	XX	XXXXXXXX

Instruction Length
2 Byte

Example
MVI C, 05H

Assign the value 05H to register C.

T States and Control Signals

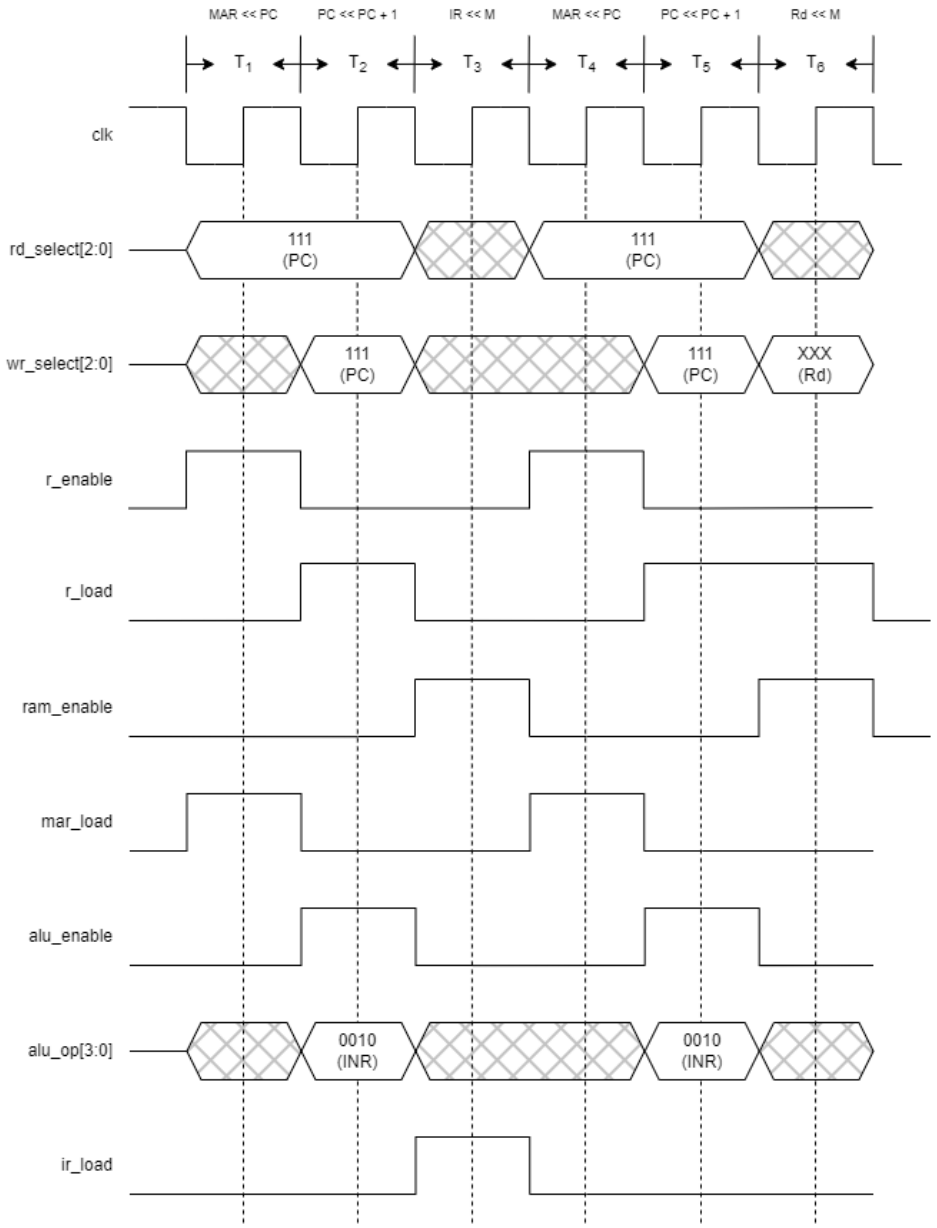


Fig. 14: The timing diagram of the MVI instruction.

6.4.5 ADD, SUB, ANR, ORR, and XRR

Perform arithmetic or logic operation on two registers.

Instruction Layout
[ADD|SUB|ANR|ORR|XRR] *Rd*, *Rs*

Machine Code Format

	Opcode			
	Operation		Destination Register (Rd)	Source Register (Rs)
	DR-Type	Instruction Index		
	4 bits		2 bits	2 bits
ADD	01	01	XX	XX
SUB	01	10		
ANR	01	11		
ORR	10	00		
XRR	10	01		

Instruction Length
1 Byte

Example
ORR A, D

Execute the OR operation between register A and register D, and store the result in register A.

T States and Control Signals

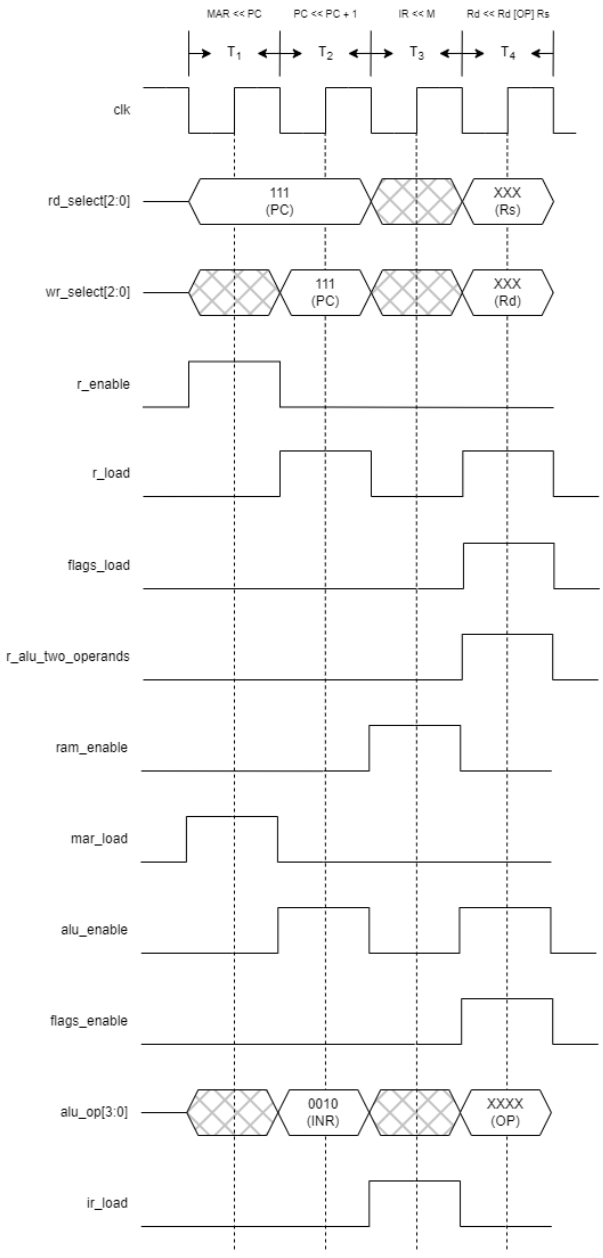


Fig. 15: The timing diagram of the ADD, SUB, ANR, ORR, and XRR instructions.

6.4.6 INR, DER, ROR, and ROL

Perform arithmetic or logic operation on specified register.

Instruction Layout
[INR|DER|ROR|ROL] *Rd*

Machine Code Format

	Opcode		
	Operation		Register
	SR-Type	Instruction Index	
	6 bits		2 bits
INR	00	0000	XX
DER		0001	
ROR		0010	
ROL		0011	

Instruction Length
1 Byte

Example
DER B

Decrease the value of register B by 1.

T States and Control Signals

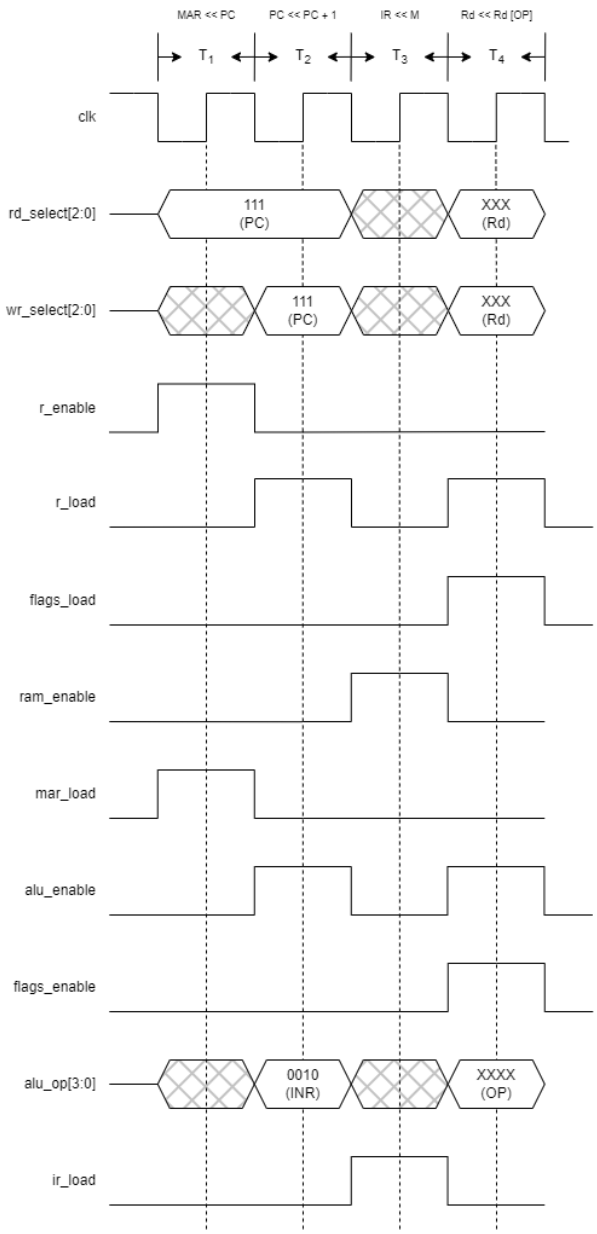


Fig. 16: The timing diagram of the INR, DER, ROR, and ROL instructions.

6.4.7 JMP

Execute an unconditional jump to a specified instruction.

Instruction Layout
JMP *address*

Machine Code Format

Opcode		Operand
Operation		Immediate
J-Type	Instruction Index	
8 bits		16 bits
1101	0001	XXXXXXXX XXXXXXXX

Instruction Length
3 Byte

Example
JMP 1111H

Continue program execution unconditionally, starting from the instruction at memory address 1111H.

T States and Control Signals

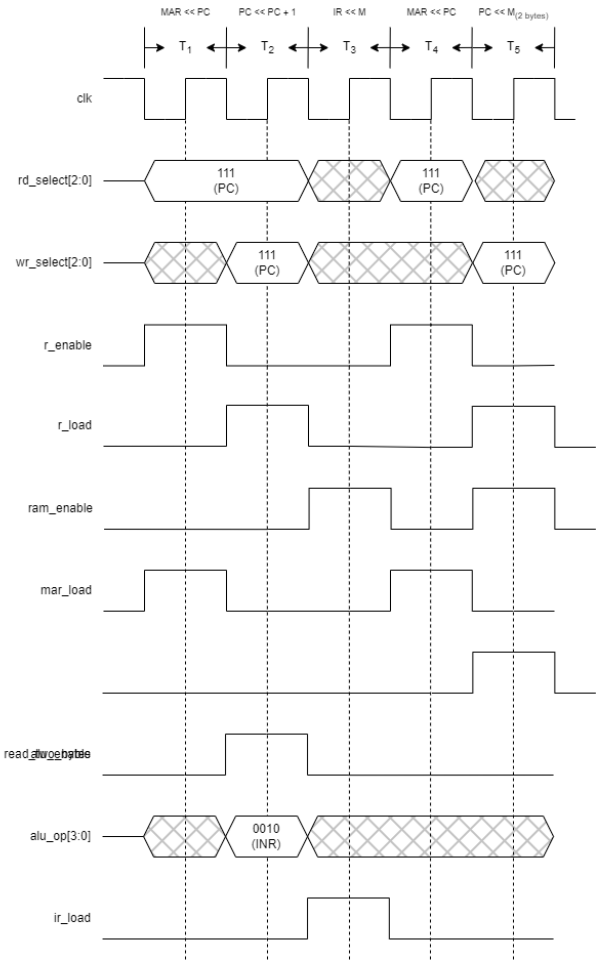


Fig. 17: The timing diagram of the JMP instruction.

6.4.8 JZ

Jump to a specified instruction conditionally, contingent upon the status of the Zero flag.

Instruction Layout
JZ *address*

Machine Code Format

Opcode		Operand
Operation		Immediate
J-Type	Instruction Index	
8 bits		16 bits
1101	0010	XXXXXXXX XXXXXXXX

Instruction Length
3 Byte

Example
JZ 1111H

Continue the program execution conditionally, proceeding from the instruction located at memory address 1111H, based on the status of the Zero flag.

T States and Control Signals

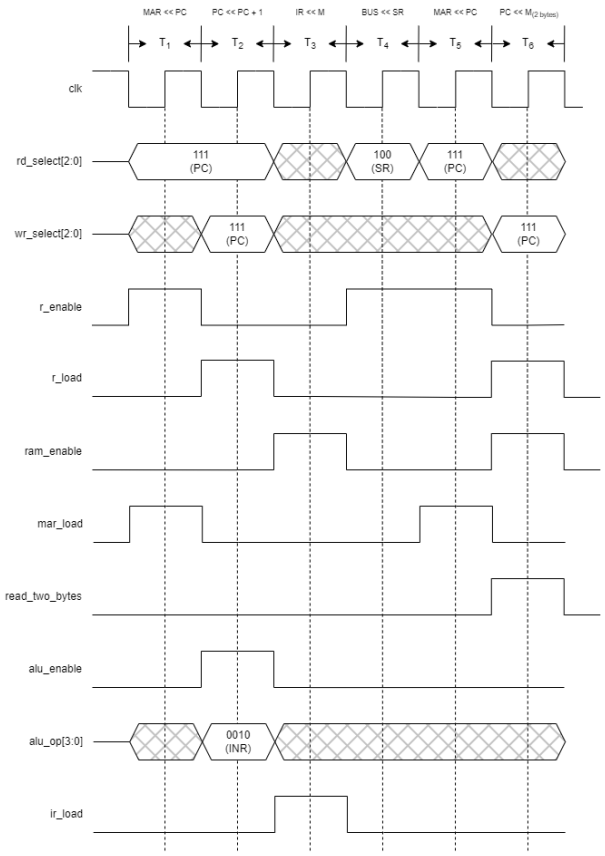


Fig. 18: The timing diagram of the JZ instruction.

6.4.9 CALL

Perform a subroutine call by jumping to a specific instruction while storing the current Program Counter (PC) in the stack. This allows for a return to the original PC after completing the execution of the instructions block.

Instruction Layout
CALL *address*

Machine Code Format

Opcode		Operand
Operation		Immediate
J-Type	Instruction Index	
8 bits		16 bits
1101	0011	XXXXXXXX XXXXXXXX

Instruction Length
3 Byte

Example
CALL 1111H

T States and Control Signals

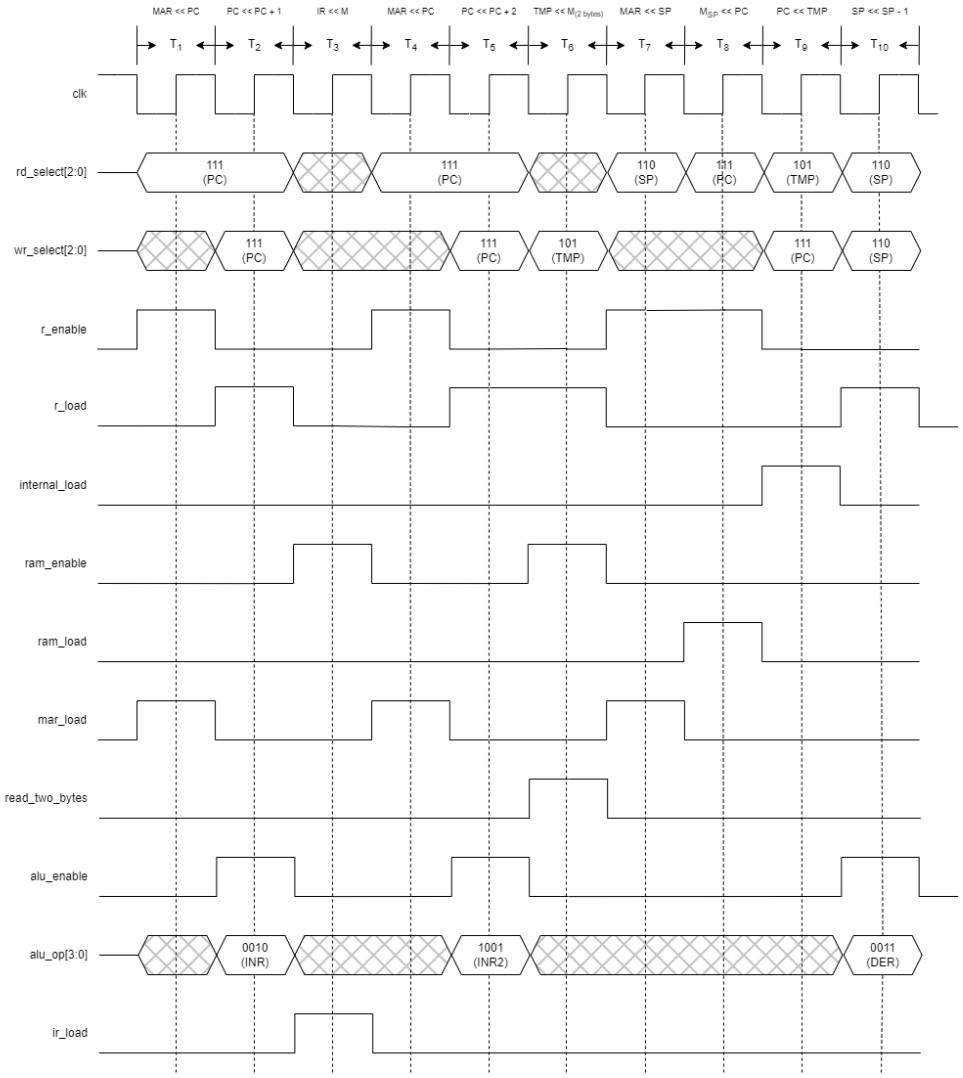


Fig. 19: The timing diagram of the CALL instruction.

6.4.10 RET

Return to the instruction that precedes the CALL instruction in the program execution sequence.

Instruction Layout
RET

Machine Code Format

Opcode	
Operation	
O-Type	Instruction Index
8 bits	
1111	0000

Instruction Length
1 Byte

Example
RET

T States and Control Signals

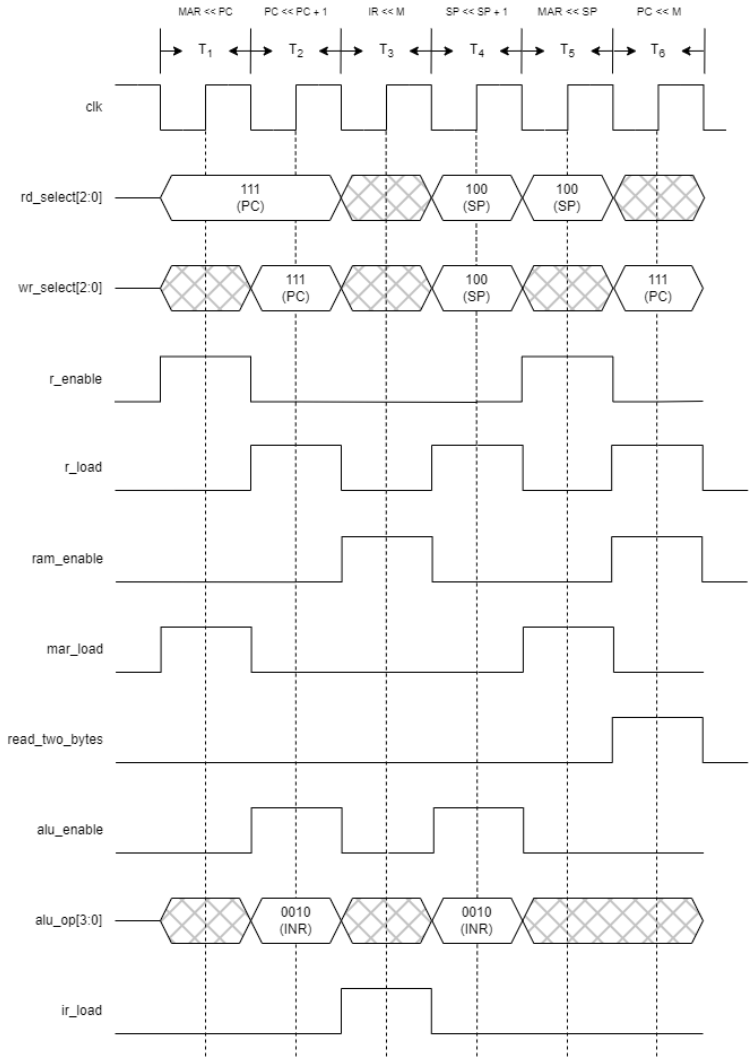


Fig. 20: The timing diagram of the RET instruction.

6.4.11 PUSH

Store the content of a register in the stack memory space, which is beneficial for preserving the program status after executing a subroutine.

Instruction Layout
PUSH *Rs*

Machine Code Format

Opcode		
Operation		Register
SR-Type	Instruction Index	
6 bits		2 bits
00	0100	XX

Instruction Length
1 Byte

Example
PUSH C

T States and Control Signals

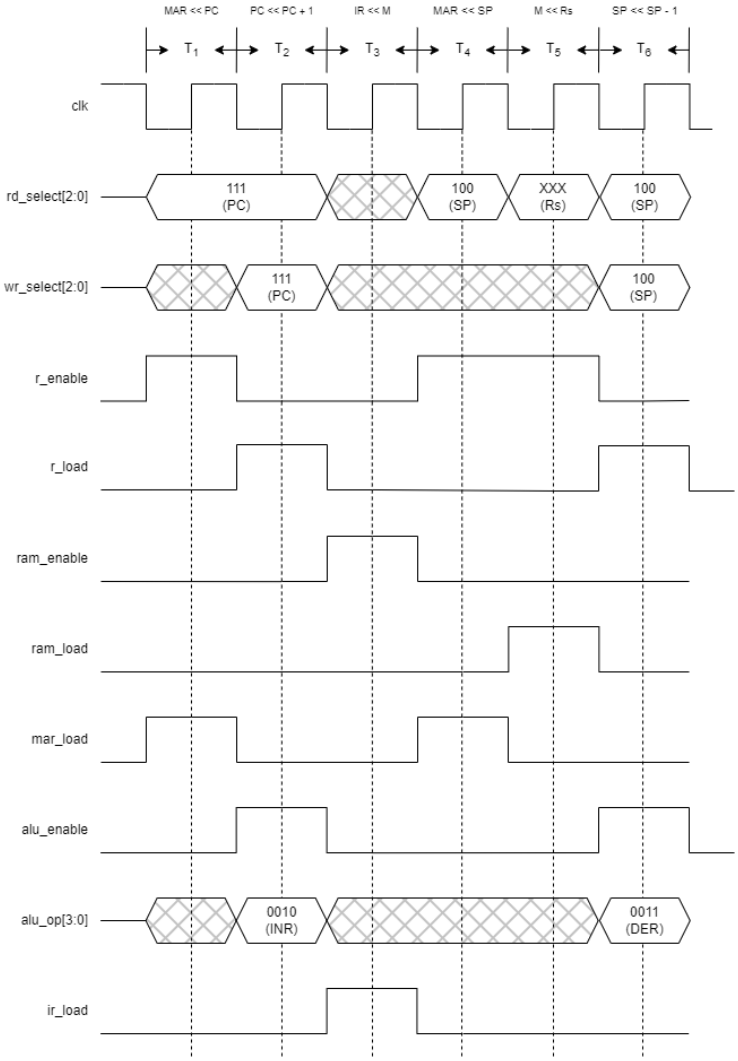


Fig. 21: The timing diagram of the PUSH instruction.

6.4.12 POP

Store a half-word from the stack memory space to a register, which is beneficial for preserving the program status after executing a subroutine.

Instruction Layout
POP *Rd*

Machine Code Format

Opcode		
Operation		Register
SR-Type	Instruction Index	
6 bits		2 bits
00	0101	XX

Instruction Length
1 Byte

Example
POP C

T States and Control Signals

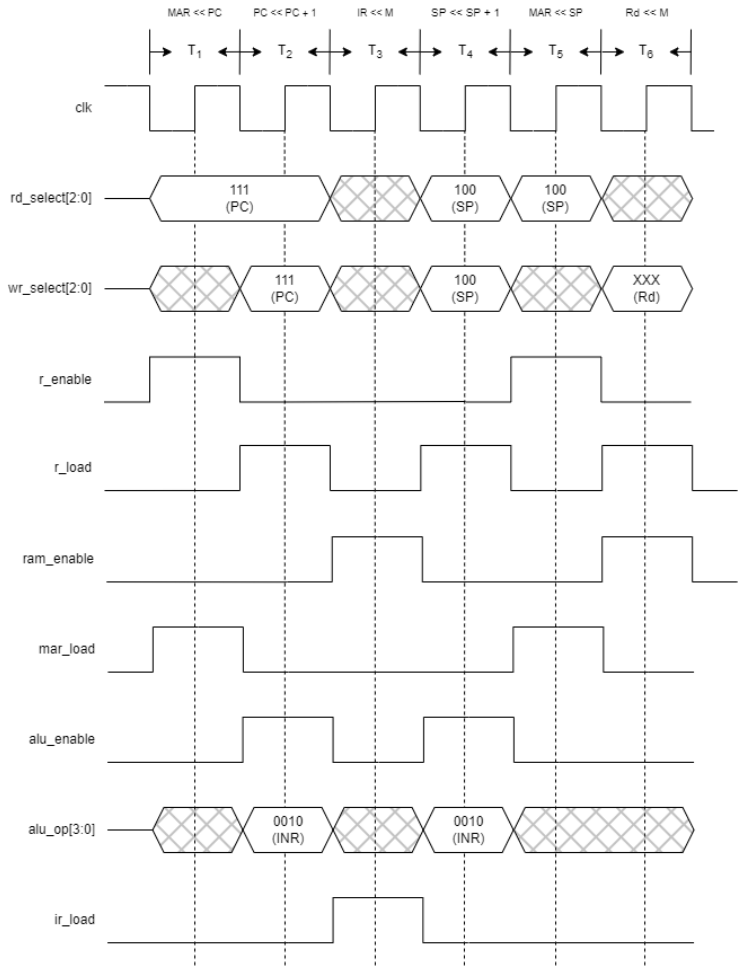


Fig. 21: The timing diagram of the POP instruction.

6.4.13 OUTX, OUTY, and OUTZ

Output a specified register content to one of the output ports.

Instruction Layout
OUT[X|Y|Z] *Rs*

Machine Code Format

	Opcode		
	Operation		Register
	SR-Type	Instruction Index	
	6 bits		2 bits
OUTX	00	0110	XX
OUTY		0111	
OUTZ		1000	

Instruction Length
1 Byte

Example
OUTY C

T States and Control Signals

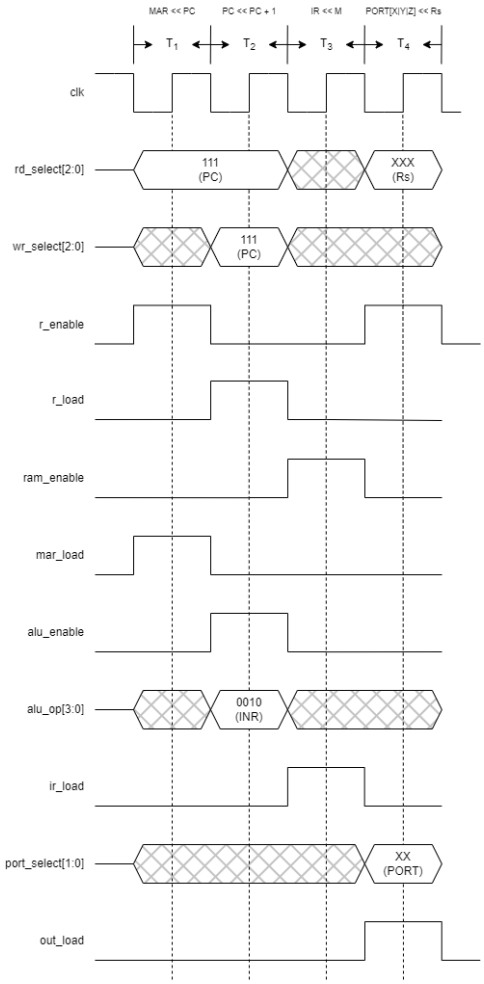


Fig. 22: The timing diagram of the OUTX, OUTY, and OUTZ instructions.

6.4.14 NOP

Do nothing; used to introduce a delay in the execution.

Instruction Layout
NOP

Machine Code Format

Opcode	
Operation	
O-Type	Instruction Index
8 bits	
1111	1110

Instruction Length
1 Byte

Example
NOP

T States and Control Signals

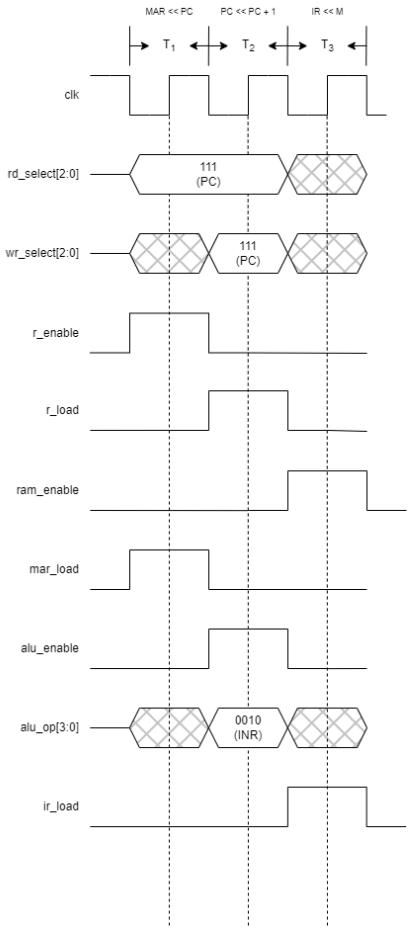


Fig. 22: The timing diagram of the NOP instruction.

6.4.15 HLT

Stop the processing.

Instruction Layout
HLT

Machine Code Format

Opcode	
Operation	
O-Type	Instruction Index
8 bits	
1111	1111

Instruction Length
1 Byte

Example
HLT

T States and Control Signals

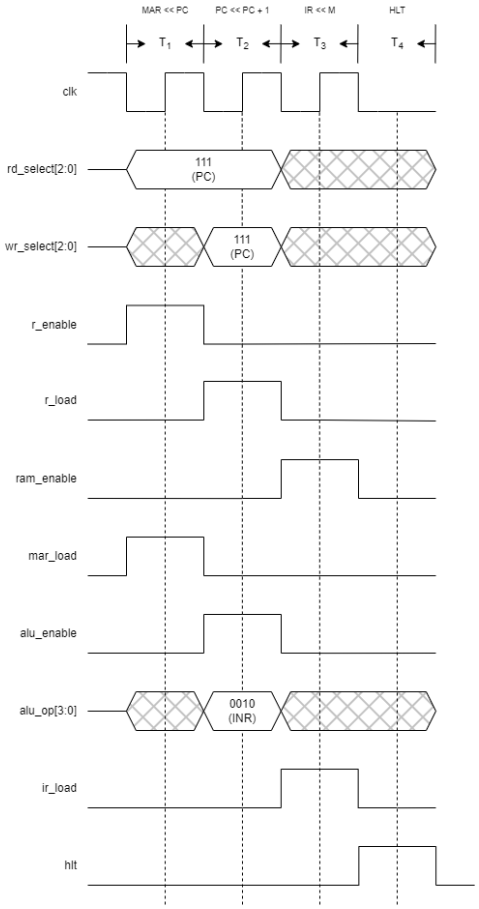


Fig. 23: The timing diagram of the HLT instruction.

6.5 Assembler

The assembler is a category of computer programs designed to take assembly instructions and convert them into a binary pattern that the microprocessor can utilize for executing operations, commonly known as machine code. While the assembler concept has been introduced in previous sections with examples for each instruction, often referred to as hand-assembly, the focus here is on the automated process. This involves providing a Python script capable of carrying out this conversion.

The initial step for an assembler involves reading the file containing assembly instructions, removing comments, extra whitespaces, and lines, resulting in a list of lines where each line contains a single instruction or label.

```
def __read_assembly_file(self, assembly_file_path):
    with open(assembly_file_path, "r") as assembly_file:
        assembly_file_lines = assembly_file.readlines()
        assembly_instructions = []

        for line in assembly_file_lines:
            # Remove comments and empty lines from the start and end of the line
            line = line.split("#")[0].rstrip().strip()
            if line:
                assembly_instructions.append(line)

        return assembly_instructions
```

Once this list is established, the translation process commences. It iterates through each instruction, determines its type based on the categories defined in the Instruction Formats, and extracts the instruction components.

```
def __get_instruction_type(self, instruction):
    instruction_type = ""
    instruction_components = instruction.split()

    if len(instruction_components) == 1:
        if ":" in instruction_components[0]:
            instruction_type = "LABEL"
            instruction_components[0] = instruction_components[0][:-1]
        else:
            instruction_type= "O"
    elif len(instruction_components) == 2:
        if "H" in instruction_components[1] or instruction_components[1] not in REGISTERS_ENCODING:
            instruction_type = "J"

            if "H" in instruction_components[1]:
                instruction_components[1] = instruction_components[1][:-1].zfill(4)
        else:
            instruction_type = "SR"
    else:
        instruction_components[1] = instruction_components[1][:-1]

        if "H" in instruction_components[2] or instruction_components[2] not in REGISTERS_ENCODING:
            if "H" in instruction_components[2]:
                instruction_components[2] = instruction_components[2][:-1]

            if len(instruction_components[2]) <= 2:
                instruction_type = "I"
                instruction_components[2] = instruction_components[2].zfill(2)
            else:
                instruction_type = "D"
                instruction_components[2] = instruction_components[2].zfill(4)
        else:
            instruction_type = "DR"

    return instruction_type, instruction_components
```

The next step involves checking for potential assembly syntax errors, such as:

Mnemonic Not Defined in the ISA
This occurs when the mnemonic in the instruction is not defined in the instruction set. It's important to note that the checking is case-insensitive.

Register Not Available; Programmer-Accessible Registers are [A, B, C, D]
This occurs when the source/destination register is not one of the available programmer-accessible registers A, B, C, and D. Similar to the mnemonic check, the register checking is case-insensitive.

Label Defined Multiple Times
This occurs when a label is defined more than once.

Invalid Hexadecimal Value
This occurs when the hexadecimal value contains non-hexadecimal digits.

Maximum Size Exceeded; Maximum X Bytes

This occurs when the written hexadecimal value exceeds the maximum size for the instruction type.

```
def __check_mnemonic(self, assembly_instruction, mnemonic):
    if mnemonic.upper() not in MNEMONICS_ENCODING.keys():
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {mnemonic} mnemonic not defined in the ISA")

def __check_register(self, assembly_instruction, register):
    if register.upper() not in REGISTERS_ENCODING.keys():
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {register} register not available; Programmer-Accessible Registers are [A, B, C, D]")

def __check_label(self, assembly_instruction, labels, label):
    if label.upper() in labels:
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {label} label defined multiple times")

def __check_hexa_bytes(self, assembly_instruction, hexa, max_bytes_num=1):
    for digit in hexa:
        if digit.upper() not in "0123456789ABCDEF":
            raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {hexa} invalid hexadecimal value")

    if len(hexa) > 2 * max_bytes_num:
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {hexa} maximum size exceeded, maximum {max_bytes_num} bytes")
```

After ensuring the validity of each instruction component, encoding takes place based on each instruction type.

```
MNEMONICS_ENCODING = {
    "LDR": "111000",
    "STR": "111001",
    "MOV": "0100",
    "MVI": "110000",
    "ADD": "0101",
    "SUB": "0110",
    "INR": "000000",
    "DER": "000001",
    "ROR": "000010",
    "ROL": "000011",
    "ANR": "0111",
    "ORR": "1000",
    "XRR": "1001",
    "JMP": "11010001",
    "JZ": "11010010",
    "CALL": "11010011",
    "RET": "11110000",
    "PUSH": "000100",
    "POP": "000101",
    "OUTX": "000110",     "OUTX": Unknown word.
    "OUTY": "000111",     "OUTY": Unknown word.
    "OUTZ": "001000",     "OUTZ": Unknown word.
    "NOP": "11111110",
    "HLT": "11111111",
}

REGISTERS_ENCODING = {"A": "00", "B": "01", "C": "10", "D": "11"}
```

```
for assembly_instruction in self.assembly_instructions:
    instruction_type, instruction_components = self.__get_instruction_type(assembly_instruction)

    if instruction_type == "LABEL":
        labels_address[instruction_components[0].upper()] = self.__decimal_to_hexa(len(memory_records))    "hexa": Unknown word.
    else:
        self.__check_mnemonic(assembly_instruction, instruction_components[0])

        if instruction_type == "SR":
            self.__check_register(assembly_instruction, instruction_components[1])
            memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]])
        elif instruction_type == "DR":
            self.__check_register(assembly_instruction, instruction_components[1])
            self.__check_register(assembly_instruction, instruction_components[2])
            memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]] + REGISTERS_ENCODING[instruction_components[2]])
        elif instruction_type == "I":
            self.__check_register(assembly_instruction, instruction_components[1])
            memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]])
            self.__check_hexa_bytes(assembly_instruction, instruction_components[2])    "hexa": Unknown word.
            memory_records.append(self.__hexa_to_binary(instruction_components[2]))    "hexa": Unknown word.
        elif instruction_type == "J":
            memory_records.append(MNEMONICS_ENCODING[instruction_components[0]])
            if instruction_components[1] in labels_address.keys():
                memory_records.append(instruction_components[1])
                memory_records.append(instruction_components[1])
            else:
                self.__check_hexa_bytes(assembly_instruction, instruction_components[1], 2)    "hexa": Unknown word.
                memory_records.append(self.__hexa_to_binary(instruction_components[1][:2]))    "hexa": Unknown word.
                memory_records.append(self.__hexa_to_binary(instruction_components[1][2:]))
        elif instruction_type == "D":
            self.__check_register(assembly_instruction, instruction_components[1])
            memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]])
            self.__check_hexa_bytes(assembly_instruction, instruction_components[2], 2)
            memory_records.append(self.__hexa_to_binary(instruction_components[2][:2]))
            memory_records.append(self.__hexa_to_binary(instruction_components[2][2:]))
        elif instruction_type == "O":
            memory_records.append(MNEMONICS_ENCODING[instruction_components[0]])
```

Lastly, labels are replaced with the corresponding instruction addresses. Once this stage is reached, the memory content is ready and can be saved as a binary file or converted to hexadecimal before saving, providing initialization for the microprocessor RAM.

```
# Extract Labels
for assembly_instruction in self.assembly_instructions:
    if ":" in assembly_instruction:
        label = assembly_instruction[:-1]
        self.__check_label(assembly_instruction, labels_address.keys(), label)
        labels_address[label.upper()] = 0
```

```
def __replace_label_with_address(self, memory_records, labels_address):
    for memory_address, memory_record in enumerate(memory_records):
        if memory_record in labels_address.keys():
            memory_records[memory_address] = self.__hexa_to_binary(labels_address[memory_record][:2])
            memory_records[memory_address + 1] = self.__hexa_to_binary(labels_address[memory_record][2:])

    return memory_records
```

6.5.1 Script Usage

The assembler script can be utilized from the command line, and its usage is outlined in the help page:

```
$ python assembler.py -h
usage: Microprocessor Assembler [-h] [-oH] [-oB] assembly_file

Translate the assembly code into machine code, either in hexadecimal or binary
format.

positional arguments:
  assembly_file

optional arguments:
  -h, --help     show this help message and exit
  -oH, --hexa    Output the machine code in hexadecimal format
  -oB, --binary  Output the machine code in binary format
```

1. Create the file containing the assembly instructions and save I to a location on your computer.

```
$ cat basic-program.asm
MVI A, 00H
MVI B, 01H
ADD:
INR A
SUB A, B
JZ ADD
HLT
```

2. Execute the assembler Python script by providing the full path of the assembly file and choose the desired output file format—either binary or hexadecimal. Note that binary is the default format.

```
$ python assembler.py basic-program.asm -oB -oH
```

3. Press Enter and check the directory where the assembly file is located. You will find a '*.bin*' file containing the binary machine code and a '*.hex*' file containing the hexadecimal machine code.

```
$ cat basic-program.bin
11000000
00000000
11000001
00000001
00000000
01100001
11010010
00000000
00000100
11111111
```

```
$ cat basic-program.hex
C0
00
C1
01
00
61
D2
00
04
FF
```

6.5.2 Script Full Code Snippets

The complete code for the script is available in the project's GitHub repository.

```
import argparse
import os

HEXA_OUTPUT_FILE_EXTENSION = ".hex"    "HEXA": Unknown word.
BINARY_OUTPUT_FILE_EXTENSION = ".bin"

MNEMONICS_ENCODING = {
    "LDR": "111000",
    "STR": "111001",
    "MOV": "0100",
    "MVI": "110000",
    "ADD": "0101",
    "SUB": "0110",
    "INR": "000000",
    "DER": "000001",
    "ROR": "000010",
    "ROL": "000011",
    "ANR": "0111",
    "ORR": "1000",
    "XRR": "1001",
    "JMP": "11010001",
    "JZ": "11010010",
    "CALL": "11010011",
    "RET": "11110000",
    "PUSH": "000100",
    "POP": "000101",
    "OUTX": "000110",    "OUTX": Unknown word.
    "OUTY": "000111",    "OUTY": Unknown word.
    "OUTZ": "001000",    "OUTZ": Unknown word.
    "NOP": "11111110",
    "HLT": "11111111",
}

REGISTERS_ENCODING = {"A": "00", "B": "01", "C": "10", "D": "11"}
```

```
class AssemblySyntaxError(Exception):
    pass

parser = argparse.ArgumentParser(
    prog="Microprocessor Assembler",
    description="Translate the assembly code into machine code, either in hexadecimal or binary format.",
)

parser.add_argument("assembly_file")
parser.add_argument("-oH", "--hexa" , action="store_true", default=False, help="Output the machine code in hexadecimal format")
parser.add_argument("-oB", "--binary", action="store_true", default=False, help="Output the machine code in binary format")

args = parser.parse_args()

assembler = Assembler(args.assembly_file)

if args.binary or (not args.binary and not args.hexa):
    assembler.generate_binary_machine_code()

if args.hexa:
    assembler.generate_hexa_machine_code()      You, 23 minutes ago • Uncommitted changes
```

```python
class Assembler():
    def __init__(self, assembly_file_path):
        self.assembly_instructions = self.__read_assembly_file(assembly_file_path)
        self.output_file_path = self.__get_file_path_without_extension(assembly_file_path)
        self.memory_records = self.__translate_assembly_instructions()

    def __get_file_path_without_extension(self, assembly_file_path):
        file_name_with_extension = os.path.basename(assembly_file_path)
        file_name_without_extension, file_extension = os.path.splitext(file_name_with_extension)
        directory_path = os.path.dirname(assembly_file_path)

        return os.path.join(directory_path, file_name_without_extension)

    def __read_assembly_file(self, assembly_file_path):
        with open(assembly_file_path, "r") as assembly_file:
            assembly_file_lines = assembly_file.readlines()
            assembly_instructions = []

            for line in assembly_file_lines:
                # Remove comments and empty lines from the start and end of the line
                line = line.split("#")[0].rstrip().strip()
                if line:
                    assembly_instructions.append(line)

            return assembly_instructions
```

```python
def __translate_assembly_instructions(self):
    labels_address = {}
    memory_records = []

    # Extract Labels
    for assembly_instruction in self.assembly_instructions:
        if ":" in assembly_instruction:
            label = assembly_instruction[:-1]
            self.__check_label(assembly_instruction, labels_address.keys(), label)
            labels_address[label.upper()] = 0

    for assembly_instruction in self.assembly_instructions:
        instruction_type, instruction_components = self.__get_instruction_type(assembly_instruction)

        if instruction_type == "LABEL":
            labels_address[instruction_components[0].upper()] = self.__decimal_to_hexa(len(memory_records))    "hexa": Unknown word.
        else:
            self.__check_mnemonic(assembly_instruction, instruction_components[0])

            if instruction_type == "SR":
                self.__check_register(assembly_instruction, instruction_components[1])
                memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]])
            elif instruction_type == "DR":
                self.__check_register(assembly_instruction, instruction_components[1])
                self.__check_register(assembly_instruction, instruction_components[2])
                memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]] + REGISTERS_ENCODING[instruction_components[2]])
            elif instruction_type == "I":
                self.__check_register(assembly_instruction, instruction_components[1])
                memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]])
                self.__check_hexa_bytes(assembly_instruction, instruction_components[2])    "hexa": Unknown word.
                memory_records.append(self.__hexa_to_binary(instruction_components[2]))    "hexa": Unknown word.
            elif instruction_type == "J":
                memory_records.append(MNEMONICS_ENCODING[instruction_components[0]])
                if instruction_components[1] in labels_address.keys():
                    memory_records.append(instruction_components[1])
                    memory_records.append(instruction_components[1])
                else:
                    self.__check_hexa_bytes(assembly_instruction, instruction_components[1], 2)    "hexa": Unknown word.
                    memory_records.append(self.__hexa_to_binary(instruction_components[1][:2]))    "hexa": Unknown word.
                    memory_records.append(self.__hexa_to_binary(instruction_components[1][2:]))
            elif instruction_type == "D":
                self.__check_register(assembly_instruction, instruction_components[1])
                memory_records.append(MNEMONICS_ENCODING[instruction_components[0]] + REGISTERS_ENCODING[instruction_components[1]])
                self.__check_hexa_bytes(assembly_instruction, instruction_components[2], 2)
                memory_records.append(self.__hexa_to_binary(instruction_components[2][:2]))
                memory_records.append(self.__hexa_to_binary(instruction_components[2][2:]))
            elif instruction_type == "O":
                memory_records.append(MNEMONICS_ENCODING[instruction_components[0]])

    return self.__replace_label_with_address(memory_records, labels_address)
```

```
def __get_instruction_type(self, instruction):
    instruction_type = ""
    instruction_components = instruction.split()

    if len(instruction_components) == 1:
        if ":" in instruction_components[0]:
            instruction_type = "LABEL"
            instruction_components[0] = instruction_components[0][:-1]
        else:
            instruction_type= "O"
    elif len(instruction_components) == 2:
        if "H" in instruction_components[1] or instruction_components[1] not in REGISTERS_ENCODING:
            instruction_type = "J"

            if "H" in instruction_components[1]:
                instruction_components[1] = instruction_components[1][:-1].zfill(4)
        else:
            instruction_type = "SR"
    else:
        instruction_components[1] = instruction_components[1][:-1]

        if "H" in instruction_components[2] or instruction_components[2] not in REGISTERS_ENCODING:
            if "H" in instruction_components[2]:
                instruction_components[2] = instruction_components[2][:-1]

            if len(instruction_components[2]) <= 2:
                instruction_type = "I"
                instruction_components[2] = instruction_components[2].zfill(2)
            else:
                instruction_type = "D"
                instruction_components[2] = instruction_components[2].zfill(4)
        else:
            instruction_type = "DR"

    return instruction_type, instruction_components

def __replace_label_with_address(self, memory_records, labels_address):
    for memory_address, memory_record in enumerate(memory_records):
        if memory_record in labels_address.keys():
            memory_records[memory_address] = self.__hexa_to_binary(labels_address[memory_record][:2])
            memory_records[memory_address + 1] = self.__hexa_to_binary(labels_address[memory_record][2:])

    return memory_records
```

```python
def __check_mnemonic(self, assembly_instruction, mnemonic):
    if mnemonic.upper() not in MNEMONICS_ENCODING.keys():
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {mnemonic} mnemonic not defined in the ISA")

def __check_register(self, assembly_instruction, register):
    if register.upper() not in REGISTERS_ENCODING.keys():
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {register} register not available; Programmer-Accessible Registers are [A, B, C, D]")

def __check_label(self, assembly_instruction, labels, label):
    if label.upper() in labels:
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {label} label defined multiple times")

def __check_hexa_bytes(self, assembly_instruction, hexa, max_bytes_num=1):
    for digit in hexa:
        if digit.upper() not in "0123456789ABCDEF":
            raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {hexa} invalid hexadecimal value")

    if len(hexa) > 2 * max_bytes_num:
        raise AssemblySyntaxError(f"at \"{assembly_instruction}\" {hexa} maximum size exceeded, maximum {max_bytes_num} bytes")

def generate_binary_machine_code(self):
    self.__generate_machine_code_file(self.memory_records, BINARY_OUTPUT_FILE_EXTENSION)

def generate_hexa_machine_code(self):
    memory_hexa_records = [self.__binary_to_hexa(memory_binary_record) for memory_binary_record in self.memory_records]
    self.__generate_machine_code_file(memory_hexa_records, HEXA_OUTPUT_FILE_EXTENSION)    "HEXA": Unknown word.

def __binary_to_hexa(self, binary):
    return hex(int(binary, 2))[2:].zfill(2).upper()

def __hexa_to_binary(self, hexa):
    return bin(int(hexa, 16))[2:].zfill(8)

def __decimal_to_hexa(self, decimal):
    return hex(decimal)[2:].zfill(4).upper()

def __generate_machine_code_file(self, machine_code_lines, extension):
    with open(self.output_file_path + extension, "w") as output_file:
        output_file.write("\n".join(machine_code_lines))
```

7 Implementation

7.1 Memory

7.1.1 Verilog Code Snippet

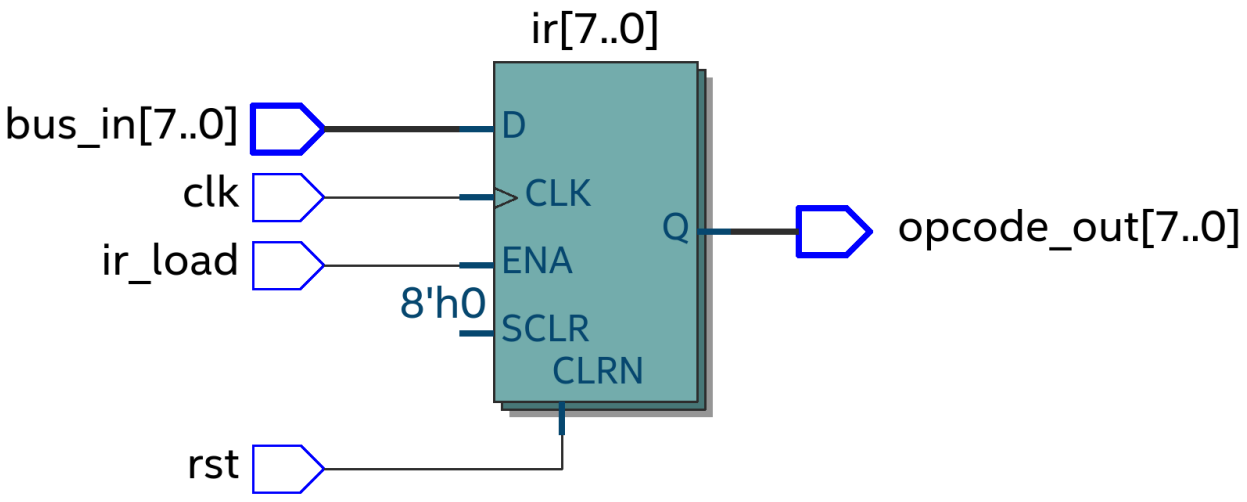
```
module memory(bus_in, bus_out, ram_load, mar_load, mar_internal_load, read_two_bytes, clk, rst);
   parameter BUS_WIDTH = 16;
   parameter MEMORY_WIDTH = 8;
   parameter MEMORY_LENGTH = 65536; // 64kb

   input [BUS_WIDTH-1:0] bus_in;
   input ram_load, mar_load, mar_internal_load, read_two_bytes, clk, rst;
   output [BUS_WIDTH-1:0] bus_out;

   reg [BUS_WIDTH-1:0] mar;
   reg [MEMORY_WIDTH-1:0] ram [MEMORY_LENGTH-1:0];

   initial begin
      $readmemb("mem.bin", ram);
   end

   always @(posedge clk or posedge rst) begin
      if (rst) begin
         mar <= {BUS_WIDTH{1'b0}};
      end else begin
         if (ram_load) begin
            ram[mar] <= bus_in[MEMORY_WIDTH-1:0];
         end else if (mar_load) begin
            mar <= bus_in;
         end else if (mar_internal_load) begin
            mar <= {ram[mar], ram[mar+1]};
         end
      end
   end

   assign bus_out = (read_two_bytes) ? {ram[mar], ram[mar+1]} : {8'b0, ram[mar]};
endmodule
```

7.2 Instruction Register

7.2.1 Verilog Code Snippet

```
module instruction_register(bus_in, opcode_out, ir_load, clk, rst);
   parameter OPCODE_WIDTH = 8;

   input [OPCODE_WIDTH-1:0] bus_in;
   input ir_load, clk, rst;

   output [OPCODE_WIDTH-1:0] opcode_out;

   reg [OPCODE_WIDTH-1:0] ir;

   always @(posedge clk or posedge rst) begin
      if (rst) begin
         ir <= {OPCODE_WIDTH{1'b0}};
      end else begin
         if (ir_load) begin
            ir <= bus_in;
         end
      end
   end

   assign opcode_out = ir;
endmodule
```

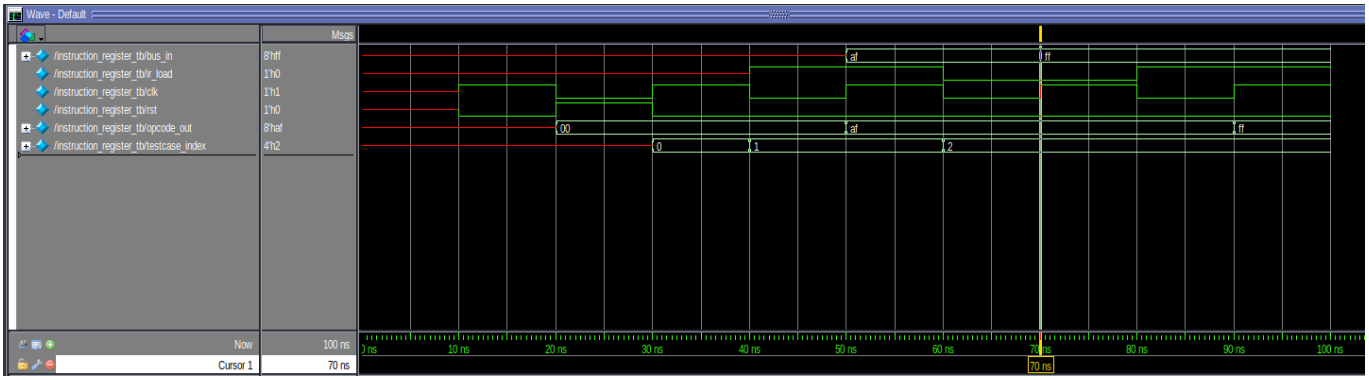
7.2.2 Netlist Schematic



7.2.3 Test Bench

```
`timescale 1ns / 100ps

module instruction_register_tb();
   parameter OPCODE_WIDTH = 8;

   reg [OPCODE_WIDTH-1:0] bus_in;
   reg ir_load, clk, rst;

   wire [OPCODE_WIDTH-1:0] opcode_out;

   reg [4:0] testcase_index;

   instruction_register uut(.bus_in(bus_in), .opcode_out(opcode_out), .ir_load(ir_load), .clk(clk), .rst(rst));

   always begin
      #10 clk = 1; #10 clk = 0;
   end

   initial begin
      #10 rst = 0; #10 rst = 1; #10 rst = 0;

      // Initialized
      testcase_index = 0;
      check(8'b00000000);
      #10

      // Load with 8'b10101111
      testcase_index = 1;
      ir_load = 1; #10
      bus_in = 8'b10101111; #10
      check(8'b10101111);

      // Bus content not change without setting the ir_load signal
      testcase_index = 2;
      ir_load = 0; #10
      bus_in = 8'b11111111; #10
      check(8'b10101111);

      // Bus content change after setting the ir_load signal
      testcase_index = 3;
      ir_load = 1; #20
      check(8'b11111111);

      $stop();
   end


   task check (input [OPCODE_WIDTH-1:0] exp_opcode_out);
      begin
         if (exp_opcode_out == opcode_out) begin
            $display("Test case %d successed, opcode_out = %b (exp %b)", testcase_index, opcode_out, exp_opcode_out);
         end else begin
            $display("Test case %d failed, opcode_out = %b (exp %b)", testcase_index, opcode_out, exp_opcode_out);
         end
      end
   endtask
endmodule
```

7.2.4 Simulation Waveform




```
# run -all
# Test case  0 successed, opcode_out = 00000000 (exp 00000000)
# Test case  1 successed, opcode_out = 10101111 (exp 10101111)
# Test case  2 successed, opcode_out = 10101111 (exp 10101111)
# Test case  2 successed, opcode_out = 11111111 (exp 11111111)
```

7.3 Register File

7.3.1 Verilog Code Snippet

```
module regsiter_file(
    bus_in, bus_out, alu_out, rd_select, wr_select, r_load,
    flags_load, internal_load, r_alu_two_operands, clk, rst
);
    parameter BUS_WIDTH = 16;
    parameter REGISTER_WIDTH = 8;
    parameter REGISTERS_COUNT = 12;

    localparam A   = 4'b0000; // 0
    localparam B   = 4'b0001; // 1
    localparam C   = 4'b0010; // 2
    localparam D   = 4'b0011; // 3
    localparam SR  = 4'b0100; // 4
    localparam TMP = 4'b0110; // 6 7
    localparam SP  = 4'b1000; // 8 9
    localparam PC  = 4'b1010; // 10 11

    input [BUS_WIDTH-1:0] bus_in;
    input [3:0] rd_select, wr_select;
    input r_load, flags_load, internal_load, r_alu_two_operands, clk, rst;

    output [BUS_WIDTH-1:0] bus_out, alu_out;

    reg [REGISTER_WIDTH-1:0] registers [REGISTERS_COUNT-1:0];

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            registers[0] <= {REGISTER_WIDTH{1'b0}};
            registers[1] <= {REGISTER_WIDTH{1'b0}};
            registers[2] <= {REGISTER_WIDTH{1'b0}};
            registers[3] <= {REGISTER_WIDTH{1'b0}};
            registers[4] <= {REGISTER_WIDTH{1'b0}};
            registers[5] <= {REGISTER_WIDTH{1'b0}};
            registers[6] <= {REGISTER_WIDTH{1'b0}};
            registers[7] <= {REGISTER_WIDTH{1'b0}};
            registers[8] <= {REGISTER_WIDTH{1'b0}};
            registers[9] <= {REGISTER_WIDTH{1'b0}};
            registers[10] <= {REGISTER_WIDTH{1'b0}};
            registers[11] <= {REGISTER_WIDTH{1'b0}};
        end else begin
```

```
    end else begin
        if (r_load) begin
            if (wr_select[3] == 1'b1 | wr_select[2:1] == 2'b11) begin // 16-bit addressed register
                registers[wr_select] <= bus_in[(REGISTER_WIDTH*2)-1:REGISTER_WIDTH];
                registers[wr_select+1] <= bus_in[REGISTER_WIDTH-1:0];
            end else begin // 8-bit addressed regsiter
                registers[wr_select] <= bus_in[REGISTER_WIDTH-1:0];
            end
        end else if (internal_load) begin
            if (wr_select[3] == 1'b1 | wr_select[2:1] == 2'b11) begin // 16-bit addressed register
                registers[wr_select] <= registers[rd_select];
                registers[wr_select+1] <= registers[rd_select+1];
            end else begin // 8-bit addressed regsiter
                registers[wr_select] <= registers[rd_select];
            end
        end

        if (flags_load) begin
            registers[SR] <= bus_in[(REGISTER_WIDTH*2)-1:REGISTER_WIDTH];
        end
    end
end

assign bus_out = (wr_select[3] == 1'b1 | wr_select[2:1] == 2'b11) ? registers[rd_select] : {8'b0, registers[rd_select]};
assign alu_out = (r_alu_two_operands) ? {registers[wr_select], registers[rd_select]} : (wr_select[3] == 1'b1 | wr_select[2:1] == 2'b11) ? registers[rd_select] : {8'b0, registers[rd_select]};
endmodule
```

7.4 ALU

7.4.1 Verilog Code Snippet

```
module alu(bus_out, alu_in, alu_op, flags_enable, extend, rst);
   parameter BUS_WIDTH = 16;

   localparam ADD_OP    = 4'b0000;
   localparam SUB_OP    = 4'b0001;
   localparam INR_OP    = 4'b0010;
   localparam DER_OP    = 4'b0011;
   localparam ROR_OP    = 4'b0100;
   localparam ROL_OP    = 4'b0101;
   localparam AND_OP    = 4'b0110;
   localparam OR_OP     = 4'b0111;
   localparam XOR_OP    = 4'b1000;
   localparam INR2_OP   = 4'b1001;

   localparam FLAG_C = 8;
   localparam FLAG_Z = 9;
   localparam FLAG_P = 10;
   localparam FLAG_S = 11;

   input [BUS_WIDTH-1:0] alu_in;
   input [3:0] alu_op;
   input flags_enable, extend, rst;

   output [BUS_WIDTH-1:0] bus_out;

   reg [BUS_WIDTH-1:0] result;
   reg carry;

   assign bus_out[FLAG_C] = (flags_enable) ? ((extend) ? carry : result[FLAG_C]) : result[FLAG_C];
   assign bus_out[FLAG_Z] = (flags_enable) ? ((extend) ? result[BUS_WIDTH-1:0] == 16'b0 : result[(BUS_WIDTH/2)-1:0] == 8'b0) : result[FLAG_Z];
   assign bus_out[FLAG_P] = (flags_enable) ? ((extend) ? ~^result[BUS_WIDTH-1:0] : ~^result[(BUS_WIDTH/2)-1:0]) : result[FLAG_P];
   assign bus_out[FLAG_S] = (flags_enable) ? ((extend) ? result[BUS_WIDTH-1] : result[(BUS_WIDTH/2)-1]) : result[FLAG_S];
```

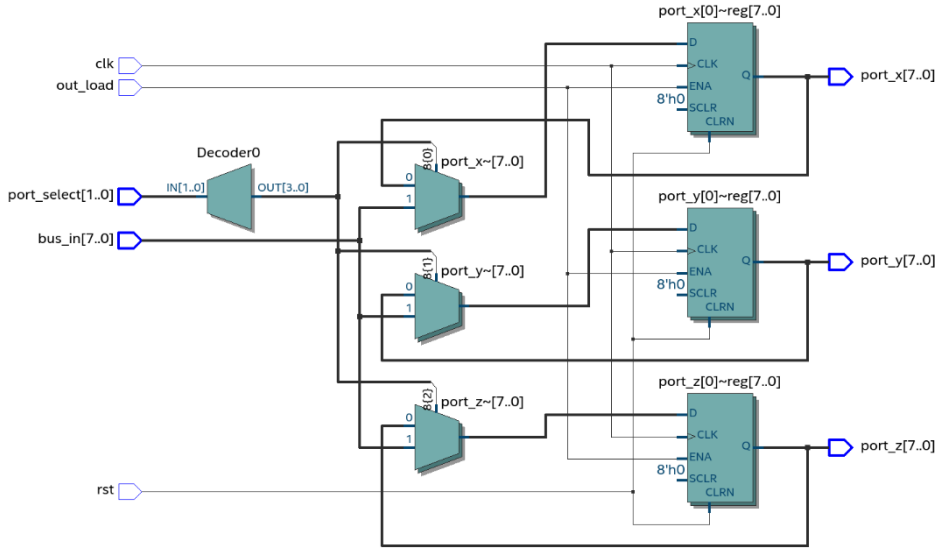
```
  always @(*) begin
     if (rst) begin
         result <= {BUS_WIDTH{1'b0}};
         carry <= 1'b0;
     end else begin
         case (alu_op)
            ADD_OP: result = alu_in[BUS_WIDTH-1:BUS_WIDTH/2] + alu_in[(BUS_WIDTH/2)-1:0];
            SUB_OP: result = alu_in[BUS_WIDTH-1:BUS_WIDTH/2] - alu_in[(BUS_WIDTH/2)-1:0];
            INR_OP: begin
               if (extend) begin
                  {carry, result} = alu_in[BUS_WIDTH-1:0] + 1;
               end else begin
                  result = alu_in[(BUS_WIDTH/2)-1:0] + 1;
               end
            end
            INR2_OP: begin
               if (extend) begin
                  {carry, result} = alu_in[BUS_WIDTH-1:0] + 2;
               end else begin
                  result = alu_in[(BUS_WIDTH/2)-1:0] + 2;
               end
            end
            DER_OP: begin
               if (extend) begin
                  {carry, result} = alu_in[BUS_WIDTH-1:0] - 1;
               end else begin
                  result = alu_in[(BUS_WIDTH/2)-1:0] - 1;
               end
            end
            ROR_OP: begin
               if (extend) begin
                  carry = result[0];
                  result = alu_in >> 1;
               end else begin
                  result = alu_in >> 1;
               end
            end
            ROL_OP: begin
               if (extend) begin
                  carry = result[BUS_WIDTH-1];
                  result = alu_in << 1;
               end else begin
                  result = alu_in << 1;
               end
            end
            AND_OP: result = alu_in[BUS_WIDTH-1:BUS_WIDTH/2] & alu_in[(BUS_WIDTH/2)-1:0];
            OR_OP: result = alu_in[BUS_WIDTH-1:BUS_WIDTH/2] | alu_in[(BUS_WIDTH/2)-1:0];
            XOR_OP: result = alu_in[BUS_WIDTH-1:BUS_WIDTH/2] ^ alu_in[(BUS_WIDTH/2)-1:0];
         endcase
      end
   end
endmodule
```

7.5 Output Ports

7.5.1 Verilog Code Snippet

```
module output_ports(bus_in, port_x, port_y, port_z, port_select, out_load, clk, rst);
   parameter DATA_WIDTH = 8;

   localparam PORT_X = 2'b00;
   localparam PORT_Y = 2'b01;
   localparam PORT_Z = 2'b10;

   input [DATA_WIDTH-1:0] bus_in;
   input [1:0] port_select;
   input out_load, clk, rst;
   output reg [DATA_WIDTH-1:0] port_x , port_y, port_z;

   always @(posedge clk or posedge rst) begin
      if (rst) begin
         port_x <= {DATA_WIDTH{1'b0}};
         port_y <= {DATA_WIDTH{1'b0}};
         port_z <= {DATA_WIDTH{1'b0}};
      end else begin
         if (out_load) begin
            case (port_select)
               PORT_X: port_x <= bus_in;
               PORT_Y: port_y <= bus_in;
               PORT_Z: port_z <= bus_in;
            endcase
         end
      end
   end
endmodule
```

7.5.2 Netlist Schematic



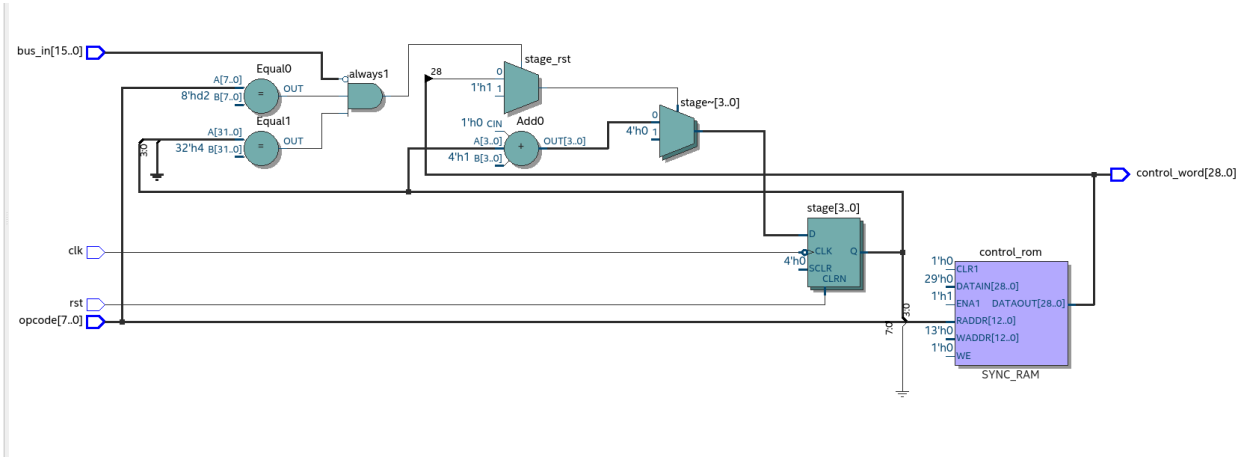
7.6 Controller

7.6.1 Verilog Code Snippet

```
module controller(bus_in, opcode, control_word, clk, rst);
   parameter BUS_WIDTH = 16;
   parameter OPCODE_WIDTH = 8;
   parameter CONTROL_WORD_WIDTH = 29;

   localparam JZ_OPCODE = 8'b11010010;

   localparam FLAG_C = 0;
   localparam FLAG_Z = 1;
   localparam FLAG_P = 2;
   localparam FLAG_S = 3;

   input [BUS_WIDTH-1:0] bus_in;
   input [OPCODE_WIDTH-1:0] opcode;
   input clk, rst;

   output reg [CONTROL_WORD_WIDTH-1:0] control_word;

   reg [BUS_WIDTH-1:0] dms;
   reg [3:0] stage;
   reg stage_rst;
   reg [CONTROL_WORD_WIDTH-1:0] control_rom [4096:0]; // 664 is the total number of t states of the 151 instructions

   initial begin
      $readmemb("control_rom.bin", control_rom);
   end

   always @(negedge clk, posedge rst) begin
      if (rst) begin
         stage <= 0;
      end else begin
         if (stage_rst) begin
            stage <= 0;
         end else begin
            stage <= stage + 1;
         end
      end
   end
```

```
   always @(*) begin
      control_word = control_rom[{opcode, stage}];

      if (opcode == JZ_OPCODE && stage == 4 && bus_in[FLAG_Z] == 0) begin
         stage_rst = 1;
      end else begin
         stage_rst = control_word[CONTROL_WORD_WIDTH-1];
      end
   end
endmodule
```

7.6.2 Netlist Schematic

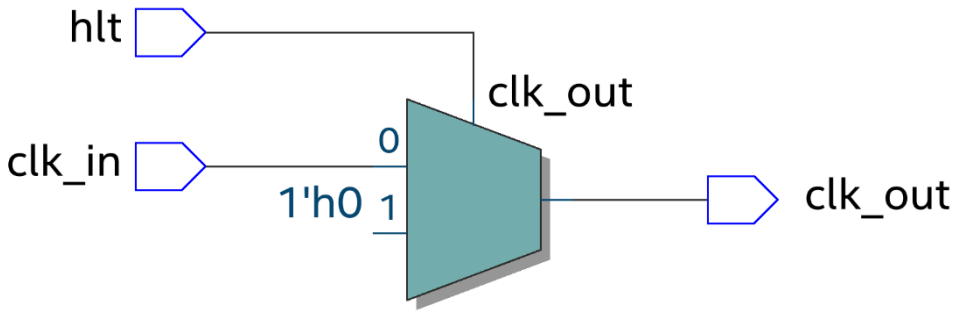

bus_in[15..0]
Equal0
A[7..0]
8'hd2 B[7..0]
OUT
always1
28
stage_rst
0
1'h1 1
Equal1
A[31..0]
32'h4 B[31..0]
OUT
1'h0 CIN
Add0
A[3..0]
4'h1 B[3..0]
OUT[3..0]
stage~[3..0]
0
4'h0 1
control_word[28..0]
stage[3..0]
D
CLK
Q
SCLR
CLRN
4'h0
clk
rst
opcode[7..0]
control_rom
1'h0
29'h0
1'h1
13'h0
1'h0
CLR1
DATAIN[28..0]
ENA1
DATAOUT[28..0]
RADDR[12..0]
WADDR[12..0]
WE
SYNC_RAM

7.7 Clock

7.7.1 Verilog Code Snippet

```
module clock(clk_in, hlt, clk_out);
   input clk_in, hlt;

   output clk_out;

   assign clk_out = (hlt) ? 1'b0 : clk_in;
endmodule
```

7.7.2 Netlist Schematic



8 Testing and Verification

8.1 Test Plan

8.2 Simulation Verification

8.3 Hardware Testing Strategy

The available FPGA for testing is the iCE40HX, utilized to validate the microprocessor design on hardware. The strategy involved the following steps:

1. Writing an assembly program designed to incorporate as many capabilities of the microprocessor as possible. The chosen application was a Fibonacci sequence generator, showcasing the use of multiple subroutines, conditions, loops, arithmetic operations, and output ports.

```
MVI A, 0DH # 13th Fib number (N = 13)
MVI B, 00H # 0th Fib = 0
MVI C, 01H # 1th Fib = 1

MOV D, B
CALL DISPLAY
MOV D, C
CALL DISPLAY

FIB_LOOP:
CALL FIB
DER A
JZ STOP
JMP FIB_LOOP

STOP:
HLT

FIB:
ADD C, B
MOV D, C
CALL DISPLAY
RET

DISPLAY:
OUTX D
NOP
NOP
NOP
RET
```

2. The assembly program was translated into machine code using the previously developed assembler from the development phases. This was done by executing the command: *python assembler.py fibonacci.asm -oB,* resulting in the creation of the *fibonacci.bin* file, which was utilized to initialize the RAM.

```
11000000
00001101
11000001
00000000
11000010
00000001
01001101
11010011
00000000
00011111
01001110
11010011
00000000
00011111
11010011
00000000
00011001
00000100
11010010
00000000
00011000
11010001
00000000
00001110
11111111
01011001
01001110
11010011
00000000
00011111
11110000
00011011
11111110
11111110
11111110
11110000
```

3. Due to FPGA constraints, the RAM size was reduced to 32kb, and the Stack Pointer was set to 7FFFH to fit within the 64kb memory limitation for all design components on the iCE40HX FPGA.

4. The Verilog code files were synthesized, placed, and routed using the Lattice iCEstick2 application tailored for the iCE40HX FPGA.

5. Output pins, internally linked to PORTX, were configured for the pins available on the iCE40HX.

PORTX Bit	iCEstick Pin
0	62
1	61
2	60
3	56
4	48
5	47
6	45
7	44

6. The generated .bin file was flashed onto the FPGA using the Diamond Programmer.

7. The Fibonacci sequence started to be displayed in binary format on the eight LEDs connected to the pins of the PORTX.

The video showcasing the program running on the FPGA can be accessed through this Google Drive link.

9 Resources

[1] Albert P. Malvino, Jerald A. Brown. “Digital Computer Electronics” (3rd Edition)

10 Appendices

10.1 Appendix 1: Microprocessor Full Instruction Set

Instruction	Op Code	Addressing Mode	T states	Flags	Bytes	Type	Main Effect
Memory-Reference Instructions							
LDR ***A***, ***address***	E0	Direct		-	3	D	A ← $M_{address}$
LDR ***B***, ***address***	E1	Direct		-	3	D	B ← $M_{address}$
LDR ***C***, ***address***	E2	Direct		-	3	D	C ← $M_{address}$
LDR ***D***, ***address***	E3	Direct		-	3	D	D ← $M_{address}$
STR ***A***, ***address***	E4	Direct		-	3	D	$M_{address}$ ← A
STR ***B***, ***address***	E5	Direct		-	3	D	$M_{address}$ ← B
STR ***C***, ***address***	E6	Direct		-	3	D	$M_{address}$ ← C
STR ***D***, ***address***	E7	Direct		-	3	D	$M_{address}$ ← D
Register Instructions							
MOV ***A***, ***A***	40	Register		-	1	DR	A ← A
MOV ***A***, ***B***	41	Register		-	1	DR	A ← B
MOV ***A***, ***C***	42	Register		-	1	DR	A ← C
MOV ***A***, ***D***	43	Register		-	1	DR	A ← D
MOV ***B***, ***A***	44	Register		-	1	DR	B ← A
MOV ***B***, ***B***	45	Register		-	1	DR	B ← B
MOV ***B***, ***C***	46	Register		-	1	DR	B ← C
MOV ***B***, ***D***	47	Register		-	1	DR	B ← D
MOV ***C***, ***A***	48	Register		-	1	DR	C ← A
MOV ***C***, ***B***	49	Register		-	1	DR	C ← B
MOV ***C***, ***C***	4A	Register		-	1	DR	C ← C
MOV ***C***, ***D***	4B	Register		-	1	DR	C ← D
MOV ***D***, ***A***	4C	Register		-	1	DR	D ← A
MOV ***D***, ***B***	4D	Register		-	1	DR	D ← B
MOV ***D***, ***C***	4E	Register		-	1	DR	D ← C
MOV ***D***, ***D***	4F	Register		-	1	DR	D ← D
MVI ***A***, ***byte***	C0	Immediate		-	2	I	A ← byte
MVI ***B***, ***byte***	C1	Immediate		-	2	I	B ← byte

MVI ***C***, ***byte***	C2	Immediate		-	2	I	C ← byte
MVI ***D***, ***byte***	C3	Immediate		-	2	I	D ← byte
Arithmetic Instructions							
ADD ***A***, ***A***	50	Register		ZCPS	1	DR	A ← A + A
ADD ***A***, ***B***	51	Register		ZCPS	1	DR	A ← A + B
ADD ***A***, ***C***	52	Register		ZCPS	1	DR	A ← A + C
ADD ***A***, ***D***	53	Register		ZCPS	1	DR	A ← A + D
ADD ***B***, ***A***	54	Register		ZCPS	1	DR	B ← B + A
ADD ***B***, ***B***	55	Register		ZCPS	1	DR	B ← B + B
ADD ***B***, ***C***	56	Register		ZCPS	1	DR	B ← B + C
ADD ***B***, ***D***	57	Register		ZCPS	1	DR	B ← B + D
ADD ***C***, ***A***	58	Register		ZCPS	1	DR	C ← C + A
ADD ***C***, ***B***	59	Register		ZCPS	1	DR	C ← C + B
ADD ***C***, ***C***	5A	Register		ZCPS	1	DR	C ← C + C
ADD ***C***, ***D***	5B	Register		ZCPS	1	DR	C ← C + D
ADD ***D***, ***A***	5C	Register		ZCPS	1	DR	D ← D + A
ADD ***D***, ***B***	5D	Register		ZCPS	1	DR	D ← D + B
ADD ***D***, ***C***	5E	Register		ZCPS	1	DR	D ← D + C
ADD ***D***, ***D***	5F	Register		ZCPS	1	DR	D ← D + D
SUB ***A***, ***A***	60	Register		ZCPS	1	DR	A ← A – A
SUB ***A***, ***B***	61	Register		ZCPS	1	DR	A ← A – B
SUB ***A***, ***C***	62	Register		ZCPS	1	DR	A ← A – C
SUB ***A***, ***D***	63	Register		ZCPS	1	DR	A ← A – D
SUB ***B***, ***A***	64	Register		ZCPS	1	DR	B ← B – A
SUB ***B***, ***B***	65	Register		ZCPS	1	DR	B ← B – B
SUB ***B***, ***C***	66	Register		ZCPS	1	DR	B ← B – C
SUB ***B***, ***D***	67	Register		ZCPS	1	DR	B ← B – D
SUB ***C***, ***A***	68	Register		ZCPS	1	DR	C ← C – A
SUB ***C***, ***B***	69	Register		ZCPS	1	DR	C ← C – B
SUB ***C***, ***C***	6A	Register		ZCPS	1	DR	C ← C – C
SUB ***C***, ***D***	6B	Register		ZCPS	1	DR	C ← C – D
SUB ***D***, ***A***	6C	Register		ZCPS	1	DR	D ← D – A
SUB ***D***, ***B***	6D	Register		ZCPS	1	DR	D ← D – B
SUB ***D***, ***C***	6E	Register		ZCPS	1	DR	D ← D – C
SUB ***D***, ***D***	6F	Register		ZCPS	1	DR	D ← D – D

INR ***A***	0	Register		Z-PS	1	SR	A ← A + 1
INR ***B***	1	Register		Z-PS	1	SR	B ← B + 1
INR ***C***	2	Register		Z-PS	1	SR	C ← C + 1
INR ***D***	3	Register		Z-PS	1	SR	D ← D + 1
DER ***A***	4	Register		Z-PS	1	SR	A ← A - 1
DER ***B***	5	Register		Z-PS	1	SR	B ← B - 1
DER ***C***	6	Register		Z-PS	1	SR	C ← C - 1
DER ***D***	7	Register		Z-PS	1	SR	D ← D - 1
Logical Instructions							
ROR ***A***	8	Register		-C--	1	SR	A ← A x 2 (Rotate all right)
ROR ***B***	9	Register		-C--	1	SR	B ← B x 2 (Rotate all right)
ROR ***C***	A	Register		-C--	1	SR	C ← C x 2 (Rotate all right)
ROR ***D***	B	Register		-C--	1	SR	D ← D x 2 (Rotate all right)
ROL ***A***	C	Register		-C--	1	SR	A ← A / 2 (Rotate all left)
ROL ***B***	D	Register		-C--	1	SR	B ← B / 2 (Rotate all left)
ROL ***C***	E	Register		-C--	1	SR	C ← C / 2 (Rotate all left)
ROL ***D***	F	Register		-C--	1	SR	D ← D / 2 (Rotate all left)
ANR ***A***, ***A***	70	Register		ZCPS	1	DR	A ← A & A
ANR ***A***, ***B***	71	Register		ZCPS	1	DR	A ← A & B
ANR ***A***, ***C***	72	Register		ZCPS	1	DR	A ← A & C
ANR ***A***, ***D***	73	Register		ZCPS	1	DR	A ← A & D
ANR ***B***, ***A***	74	Register		ZCPS	1	DR	B ← B & A
ANR ***B***, ***B***	75	Register		ZCPS	1	DR	B ← B & B
ANR ***B***, ***C***	76	Register		ZCPS	1	DR	B ← B & C
ANR ***B***, ***D***	77	Register		ZCPS	1	DR	B ← B & D
ANR ***C***, ***A***	78	Register		ZCPS	1	DR	C ← C & A
ANR ***C***, ***B***	79	Register		ZCPS	1	DR	C ← C & B
ANR ***C***, ***C***	7A	Register		ZCPS	1	DR	C ← C & C
ANR ***C***, ***D***	7B	Register		ZCPS	1	DR	C ← C & D
ANR ***D***, ***A***	7C	Register		ZCPS	1	DR	D ← D & A
ANR ***D***, ***B***	7D	Register		ZCPS	1	DR	D ← D & B
ANR ***D***, ***C***	7E	Register		ZCPS	1	DR	D ← D & C
ANR ***D***, ***D***	7F	Register		ZCPS	1	DR	D ← D & D
ORR ***A***, ***A***	80	Register		ZCPS	1	DR	A ← A | A
ORR ***A***, ***B***	81	Register		ZCPS	1	DR	A ← A | B

ORR ***A***, ***C***	82	Register		ZCPS	1	DR	A ← A | C
ORR ***A***, ***D***	83	Register		ZCPS	1	DR	A ← A | D
ORR ***B***, ***A***	84	Register		ZCPS	1	DR	B ← B | A
ORR ***B***, ***B***	85	Register		ZCPS	1	DR	B ← B | B
ORR ***B***, ***C***	86	Register		ZCPS	1	DR	B ← B | C
ORR ***B***, ***D***	87	Register		ZCPS	1	DR	B ← B | D
ORR ***C***, ***A***	88	Register		ZCPS	1	DR	C ← C | A
ORR ***C***, ***B***	89	Register		ZCPS	1	DR	C ← C | B
ORR ***C***, ***C***	8A	Register		ZCPS	1	DR	C ← C | C
ORR ***C***, ***D***	8B	Register		ZCPS	1	DR	C ← C | D
ORR ***D***, ***A***	8C	Register		ZCPS	1	DR	D ← D | A
ORR ***D***, ***B***	8D	Register		ZCPS	1	DR	D ← D | B
ORR ***D***, ***C***	8E	Register		ZCPS	1	DR	D ← D | C
ORR ***D***, ***D***	8F	Register		ZCPS	1	DR	D ← D | D
XRR ***A***, ***A***	90	Register		ZCPS	1	DR	A ← A ^ A
XRR ***A***, ***B***	91	Register		ZCPS	1	DR	A ← A ^ B
XRR ***A***, ***C***	92	Register		ZCPS	1	DR	A ← A ^ C
XRR ***A***, ***D***	93	Register		ZCPS	1	DR	A ← A ^ D
XRR ***B***, ***A***	94	Register		ZCPS	1	DR	B ← B ^ A
XRR ***B***, ***B***	95	Register		ZCPS	1	DR	B ← B ^ B
XRR ***B***, ***C***	96	Register		ZCPS	1	DR	B ← B ^ C
XRR ***B***, ***D***	97	Register		ZCPS	1	DR	B ← B ^ D
XRR ***C***, ***A***	98	Register		ZCPS	1	DR	C ← C ^ A
XRR ***C***, ***B***	99	Register		ZCPS	1	DR	C ← C ^ B
XRR ***C***, ***C***	9A	Register		ZCPS	1	DR	C ← C ^ C
XRR ***C***, ***D***	9B	Register		ZCPS	1	DR	C ← C ^ D
XRR ***D***, ***A***	9C	Register		ZCPS	1	DR	D ← D ^ A
XRR ***D***, ***B***	9D	Register		ZCPS	1	DR	D ← D ^ B
XRR ***D***, ***C***	9E	Register		ZCPS	1	DR	D ← D ^ C
XRR ***D***, ***D***	9F	Register		ZCPS	1	DR	D ← D ^ D
Branching Operations							
JMP ***address***	D1	Immediate		-	3	J	PC ← address
JZ ***address***	D2	Immediate		-	3	J	PC ← address if Z = 0
Stack Instructions							
CALL ***address***	D3	Immediate		-	3	J	PC ← address

RET	F0	-		-	1	O	PC ← return address
PUSH ***A***	10	Register		-	1	SR	$M_{stack} - 1 \leftarrow A$
PUSH ***B***	11	Register		-	1	SR	$M_{stack} - 1 \leftarrow B$
PUSH ***C***	12	Register		-	1	SR	$M_{stack} - 1 \leftarrow C$
PUSH ***D***	13	Register		-	1	SR	$M_{stack} - 1 \leftarrow D$
POP ***A***	14	Register		-	1	SR	$A \leftarrow M_{stack}$
POP ***B***	15	Register		-	1	SR	$B \leftarrow M_{stack}$
POP ***C***	16	Register		-	1	SR	$C \leftarrow M_{stack}$
POP ***D***	17	Register		-	1	SR	$D \leftarrow M_{stack}$
Misc Instructions							
OUTX ***A***	18	Register		-	1	SR	PORTX ← A
OUTX ***B***	19	Register		-	1	SR	PORTX ← B
OUTX ***C***	1A	Register		-	1	SR	PORTX ← C
OUTX ***D***	1B	Register		-	1	SR	PORTX ← D
OUTY ***A***	1C	Register		-	1	SR	PORTY ← A
OUTY ***B***	1D	Register		-	1	SR	PORTY ← B
OUTY ***C***	1E	Register		-	1	SR	PORTY ← C
OUTY ***D***	1F	Register		-	1	SR	PORTY ← D
OUTZ ***A***	20	Register		-	1	SR	PORTZ ← A
OUTZ ***B***	21	Register		-	1	SR	PORTZ ← B
OUTZ ***C***	22	Register		-	1	SR	PORTZ ← C
OUTZ ***D***	23	Register		-	1	SR	PORTZ ← D
NOP	FE	-		-	1	O	Delay (No Operation)
HLT	FF	-		-	1	O	Stop Processing