**SAP based Microprocessor Design**

**Comprehensive Report**

**A logo with a number in a square

Description automatically generated**

Prepared By Version

**Team 5**  **1.0**

Table of Contents

[**1** **Introduction** 4](#_Toc154196471)

[**1.1** **Background** 5](#_Toc154196472)

[**1.2** **Objectives** 5](#_Toc154196473)

[**1.3** **Importance and Applications** 5](#_Toc154196474)

[**2** **Project Team** 7](#_Toc154196475)

[**2.1** **Supervision Team** 7](#_Toc154196476)

[**3** **Project Scope and Objectives** 8](#_Toc154196477)

[**3.1** **Microprocessor Design Overview** 8](#_Toc154196478)

[**3.2** **Project Goals and Constraints** 10](#_Toc154196479)

[**4** **Project Management** 11](#_Toc154196480)

[**4.1** **Milestones** 11](#_Toc154196481)

[**4.2** **Estimated Timeline** 13](#_Toc154196482)

[**4.3** **Challenges Faced and Solutions** 14](#_Toc154196483)

[**4.3.1** **Instruction Format Creation** 14](#_Toc154196484)

[**4.3.2** **Limitations of Memory in the Available FPGA** 14](#_Toc154196485)

[**5** **Microprocessor Architecture** 15](#_Toc154196486)

[**5.1** **Choice of Microprocessor Architecture** 15](#_Toc154196487)

[**5.2** **Block Diagram and Components** 16](#_Toc154196488)

[**5.2.1** **Memory** 18](#_Toc154196489)

[**5.2.2** **Instruction Register** 18](#_Toc154196490)

[**5.2.3** **Register File** 19](#_Toc154196491)

[**5.2.4** **ALU** 20](#_Toc154196492)

[**5.2.5** **Output Ports** 21](#_Toc154196493)

[**5.2.6** **Controller** 21](#_Toc154196494)

[**5.2.7** **Clock** 23](#_Toc154196495)

[**6** **Instruction Set Architecture (ISA)** 24](#_Toc154196496)

[**6.1** **Definition of Instruction Set** 24](#_Toc154196497)

[**6.2** **Instruction Formats** 26](#_Toc154196498)

[**6.2.1** **SR-Type Instruction** 27](#_Toc154196499)

[**6.2.2** **DR-Type Instruction** 28](#_Toc154196500)

[**6.2.3** **I-Type Instruction** 29](#_Toc154196501)

[**6.2.4** **J-Type Instruction** 30](#_Toc154196502)

[**6.2.5** **D-Type Instruction** 31](#_Toc154196503)

[**6.2.6** **O-Type Instruction** 32](#_Toc154196504)

[**6.3** **Opcode Assignment Technique** 33](#_Toc154196505)

[**6.4** **Macroinstructions** 35](#_Toc154196506)

[**6.4.1** **LDR** 35](#_Toc154196507)

[**6.4.2** **STR** 37](#_Toc154196508)

[**6.4.3** **MOV** 39](#_Toc154196509)

[**6.4.4** **MVI** 41](#_Toc154196510)

[**6.4.5** **ADD, SUB, ANR, ORR, and XRR** 43](#_Toc154196511)

[**6.4.6** **INR, DER, ROR, and ROL** 45](#_Toc154196512)

[**6.4.7** **JMP** 47](#_Toc154196513)

[**6.4.8** **JZ** 49](#_Toc154196514)

[**6.4.9** **CALL** 51](#_Toc154196515)

[**6.4.10** **RET** 53](#_Toc154196516)

[**6.4.11** **PUSH** 55](#_Toc154196517)

[**6.4.12** **POP** 57](#_Toc154196518)

[**6.4.13** **OUTX, OUTY, and OUTZ** 59](#_Toc154196519)

[**6.4.14** **NOP** 61](#_Toc154196520)

[**6.4.15** **HLT** 63](#_Toc154196521)

[**6.5** **Assembler** 65](#_Toc154196522)

[**6.5.1** **Script Usage** 69](#_Toc154196523)

[**6.5.2** **Script Full Code Snippets** 70](#_Toc154196524)

[**7** **Implementation** 74](#_Toc154196525)

[**7.1** **Memory** 74](#_Toc154196526)

[**7.1.1** **Verilog Code Snippet** 74](#_Toc154196527)

[**7.2** **Instruction Register** 75](#_Toc154196528)

[**7.2.1** **Verilog Code Snippet** 75](#_Toc154196529)

[**7.2.2** **Netlist Schematic** 75](#_Toc154196530)

[**7.2.3** **Test Bench** 76](#_Toc154196531)

[**7.2.4** **Simulation Waveform** 77](#_Toc154196532)

[**7.3** **Register File** 78](#_Toc154196533)

[**7.3.1** **Verilog Code Snippet** 78](#_Toc154196534)

[**7.4** **ALU** 79](#_Toc154196535)

[**7.4.1** **Verilog Code Snippet** 79](#_Toc154196536)

[**7.5** **Output Ports** 80](#_Toc154196537)

[**7.5.1** **Verilog Code Snippet** 80](#_Toc154196538)

[**7.5.2** **Netlist Schematic** 80](#_Toc154196539)

[**7.6** **Controller** 81](#_Toc154196540)

[**7.6.1** **Verilog Code Snippet** 81](#_Toc154196541)

[**7.6.2** **Netlist Schematic** 81](#_Toc154196542)

[**7.7** **Clock** 83](#_Toc154196543)

[**7.7.1** **Verilog Code Snippet** 83](#_Toc154196544)

[**7.7.2** **Netlist Schematic** 83](#_Toc154196545)

[**8** **Testing and Verification** 84](#_Toc154196546)

[**8.1** **Test Plan** 84](#_Toc154196547)

[**8.2** **Simulation Verification** 84](#_Toc154196548)

[**8.3** **Hardware Testing Strategies** 85](#_Toc154196549)

[**9** **Resources** 85](#_Toc154196550)

[**10** **Appendices** 86](#_Toc154196551)

[**10.1** **Appendix 1: Microprocessor Full Instruction Set** 86](#_Toc154196552)

# **Introduction**

## **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.

## **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.

## **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.

# **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](mailto:omar.elshopky202@gmail.com) |
| Mohamed Ahmed Kamal | V23010268 |  |
| Hoda Ashraf Mohamed | V23010471 | [hodashrafff@gmail.com](mailto:hodashrafff@gmail.com) |

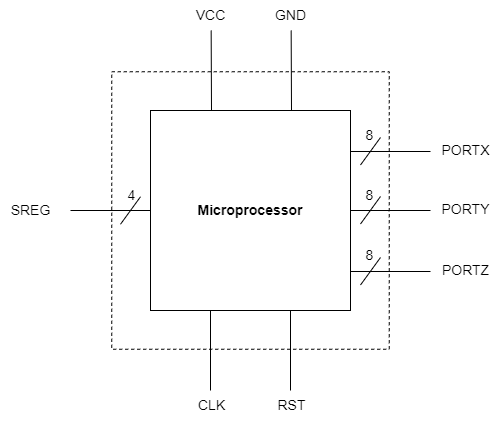
## **Supervision Team**

|  |
| --- |
| **Name** |
| Dr. Islam Yehia |
| Eng. Zeina Mohamed Samir |

# **Project Scope and Objectives**

## **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.*

## **Project Goals and Constraints**

The project aims to design and implement a microcontroller with the [specified characteristics](#_Microprocessor_Design_Overview), 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.

# **Project Management**

## **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.

1. **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.

1. **Specifications Determination and Instruction Set Architecture**

Outline the microprocessor specifications, including data width, I/O signals, instruction set architecture, and register configuration.

1. **High-Level Design**

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

1. **Control Unit Design**

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

1. **Data Path and ALU Design**

Design the data path and incorporate the arithmetic logic unit to achieve precise manipulation of data.

1. **Memory Design**

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

1. **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.

1. **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.

1. **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.

1. **Components Integration**

Integrate units into the complete microprocessor and conduct testing to verify proper communication and coordination.

1. **Simulation and Verification**

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

1. **Hardware Implementation**

Implement the microprocessor on hardware, an FPGA, conducting real-world hardware testing.

1. **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.

1. **Documentation**

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

## **Estimated Timeline**

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

1. **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.

1. **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.

1. **Phase 3 – RTL Implementation & Unit Testing**

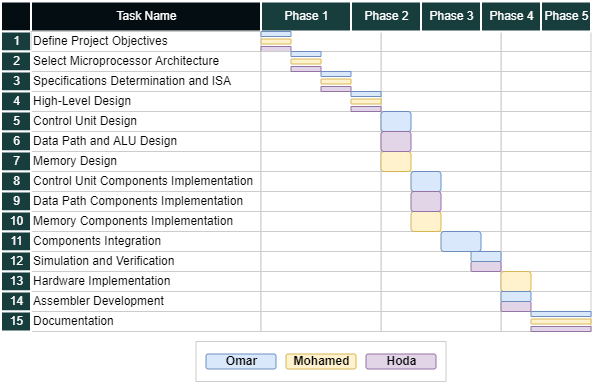
During Phase 3, the implementation of microprocessor blocks is undertaken, with rigorous unit testing conducted for each block individually before integration into one system.

1. **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.

1. **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.

## **Challenges Faced and Solutions**

### **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](#_Opcode_Assignment_Technique).

### **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.

# **Microprocessor Architecture**

## **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.

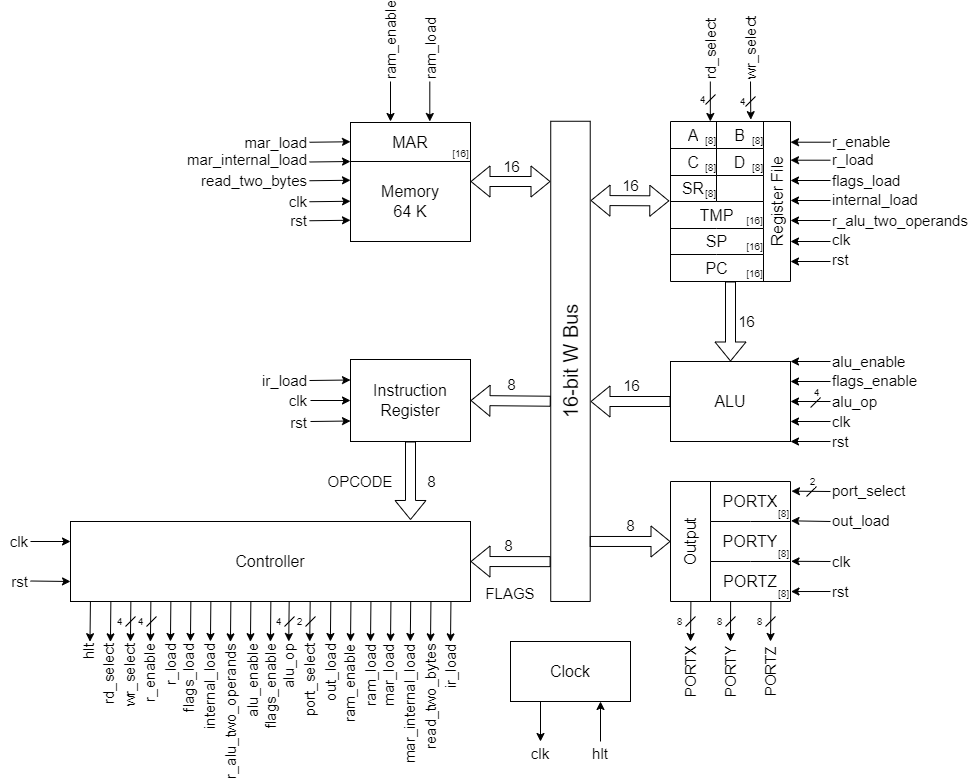
SAP-2 introduced bidirectional registers, reducing wiring capacitance and the count of I/O pins. It also featured a larger memory, providing a more realistic option compared to the 16-byte memory used in SAP-1.

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.

## **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. |

### **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.

A diagram of a computer

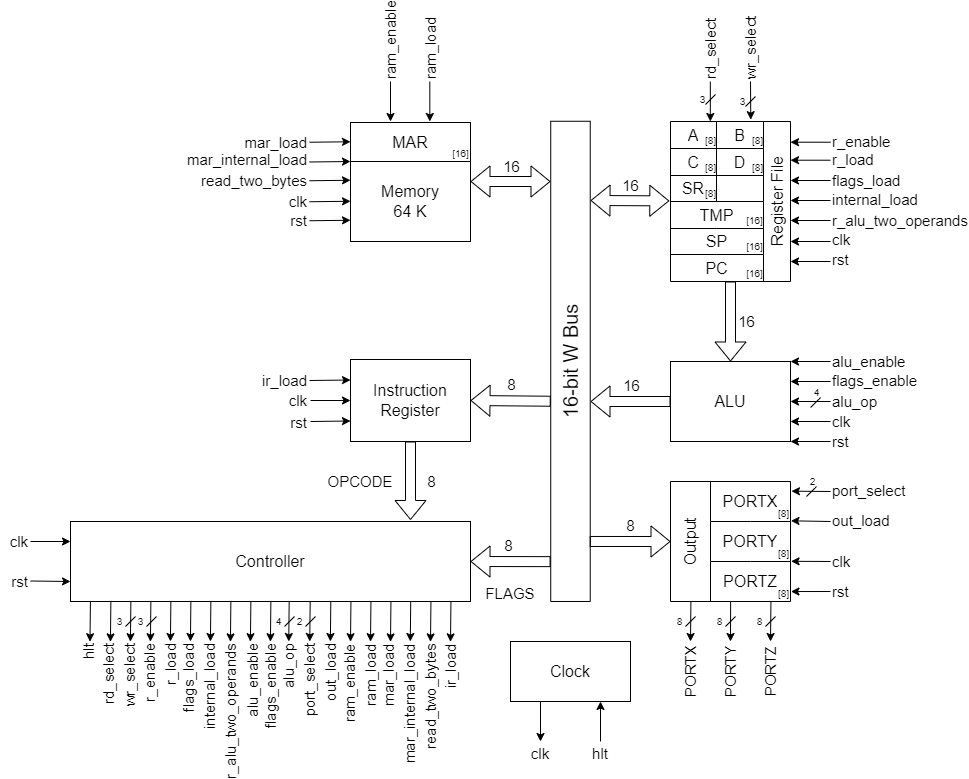
Description automatically generated

*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.

### **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.*

### **Register File**

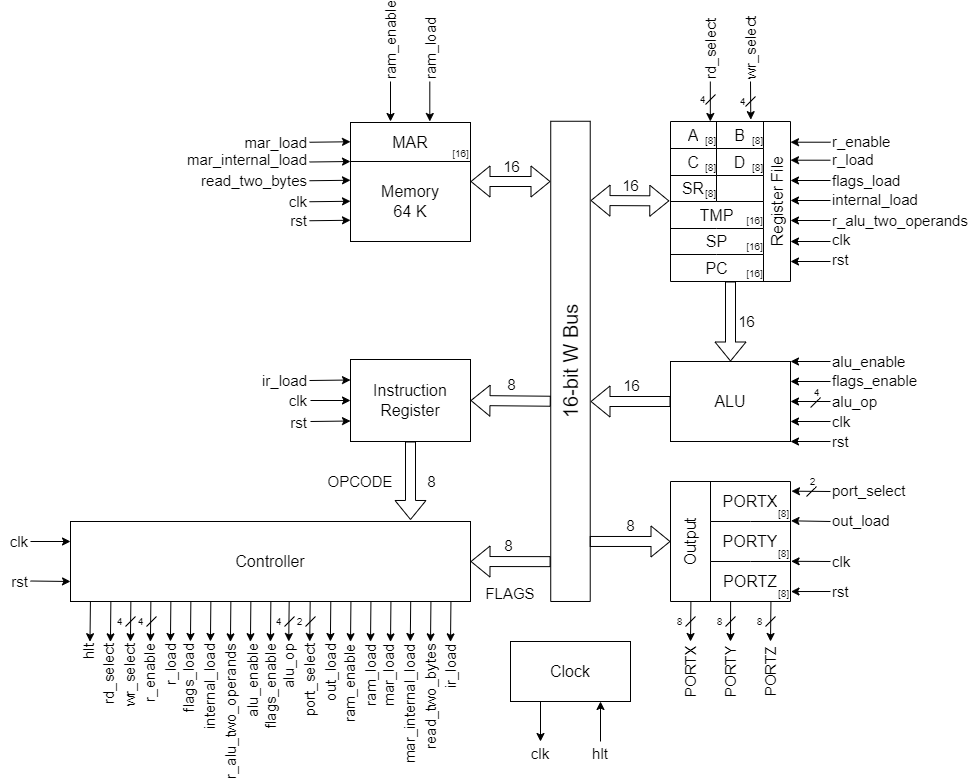
The register file comprises 12 8-bit registers, categorized as follows:

**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.

### **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.

A diagram of a computer

Description automatically generated

*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 |

### **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.

A diagram of a computer

Description automatically generated

*Fig. 7: The output ports component*

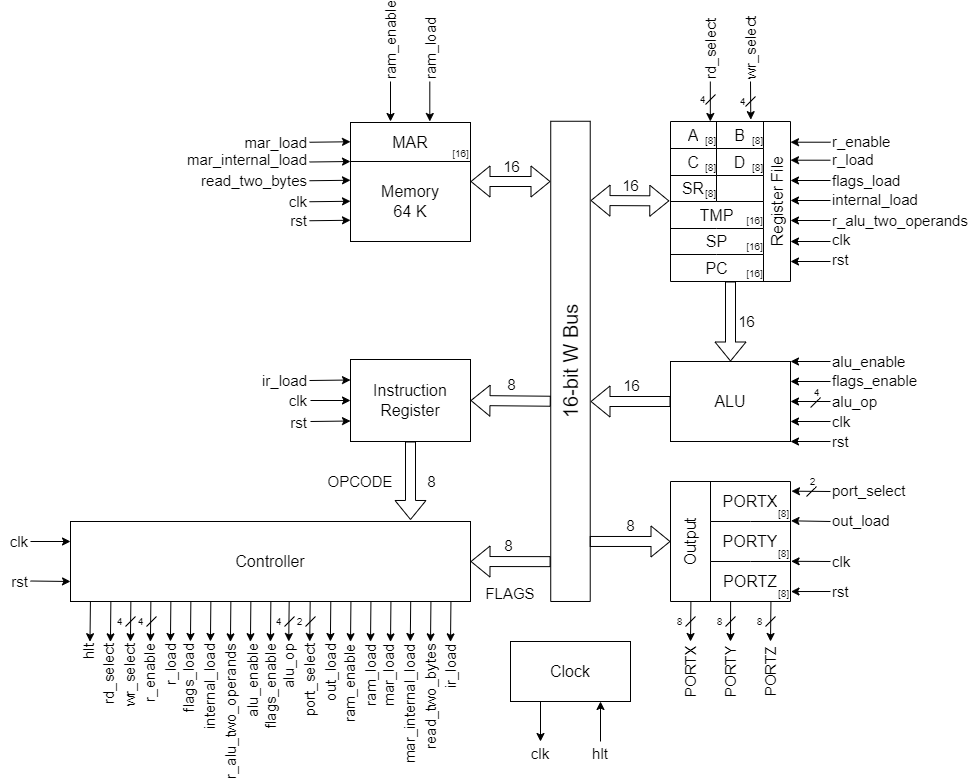
The output ports can be encoded as follows:

|  |  |
| --- | --- |
| **Port** | **Encoding** |
| PORTX | 00 |
| PORTY | 01 |
| PORTZ | 10 |

### **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](#_Macroinstructions) section.



*Fig. 8: The controller component*

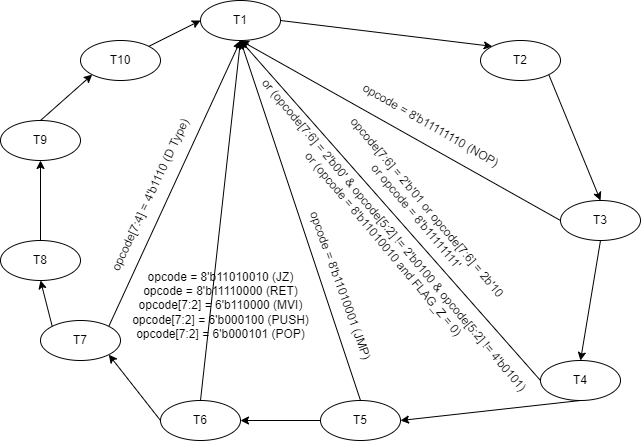
The controller depends on two ROMs: one stores the control words for all stages of each instruction, and the other stores the first address of each instruction in the control ROM.

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 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:



### **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.

A diagram of a computer

Description automatically generated

*Fig. 9: The clock component*

# **Instruction Set Architecture (ISA)**

## **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 ← Maddress |
| STR ***Rs***, ***address*** | 111001XX | Direct | 7 | - | 3 | D | Maddress ← 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 | Mstack – 1 ← Rs |
| POP ***Rd*** | 000101XX | Register | 6 | - | 1 | SR | Rd ← Mstack |
| **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.*

## **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](#_Assembler).

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](#_SR-Type_Instruction) | ASM R | 1 Byte | 00XXXXXX | Register |
| [DR-Type](#_DR-Type_Instruction) | ASM Rd, Rs | 1 Byte | 01XXXXXX  10XXXXXX | Register |
| [I-Type](#_I-Type_Instruction) | ASM Rd, byte | 2 Bytes | 1100XXXX | Immediate |
| [J-Type](#_J-Type_Instruction) | ASM address | 3 Bytes | 1101XXXX | Immediate |
| [D-Type](#_D-Type_Instruction) | ASM Rd, address | 3 Bytes | 1110XXXX | Direct |
| [O-Type](#_O-Type_Instruction) | 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.

### **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 |

### **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 |

### **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 |

### **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 |

### **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 |

### **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 |

## **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 (28).

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.

## **Macroinstructions**

### **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**

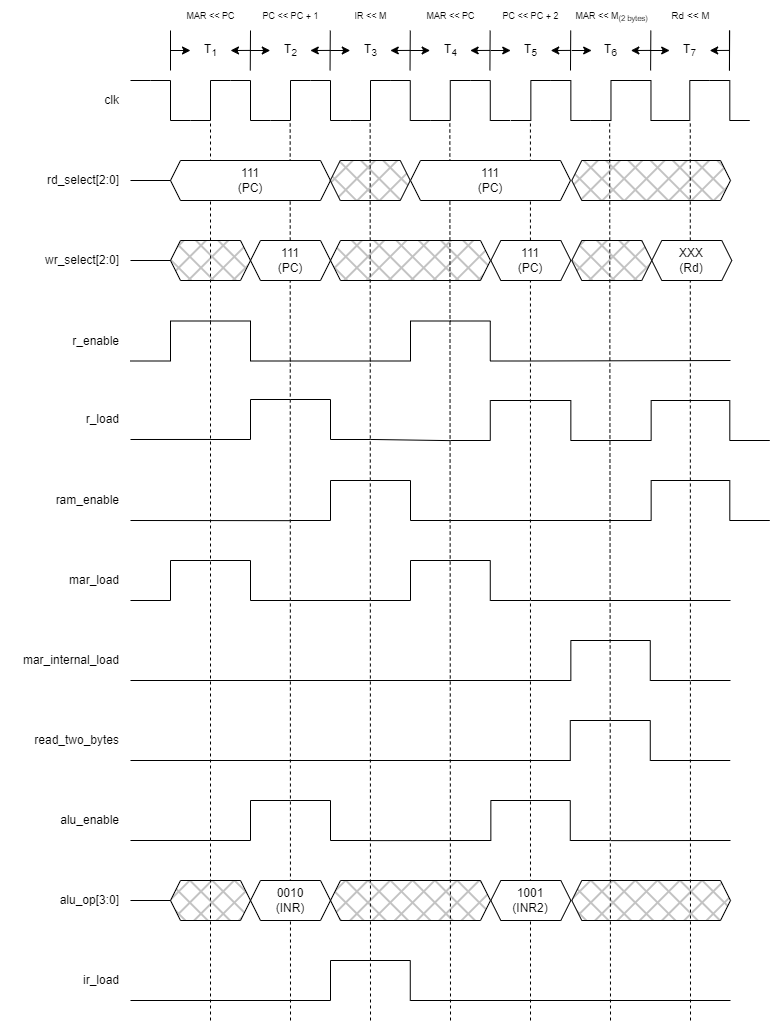
3 Byte

**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.*

### **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**

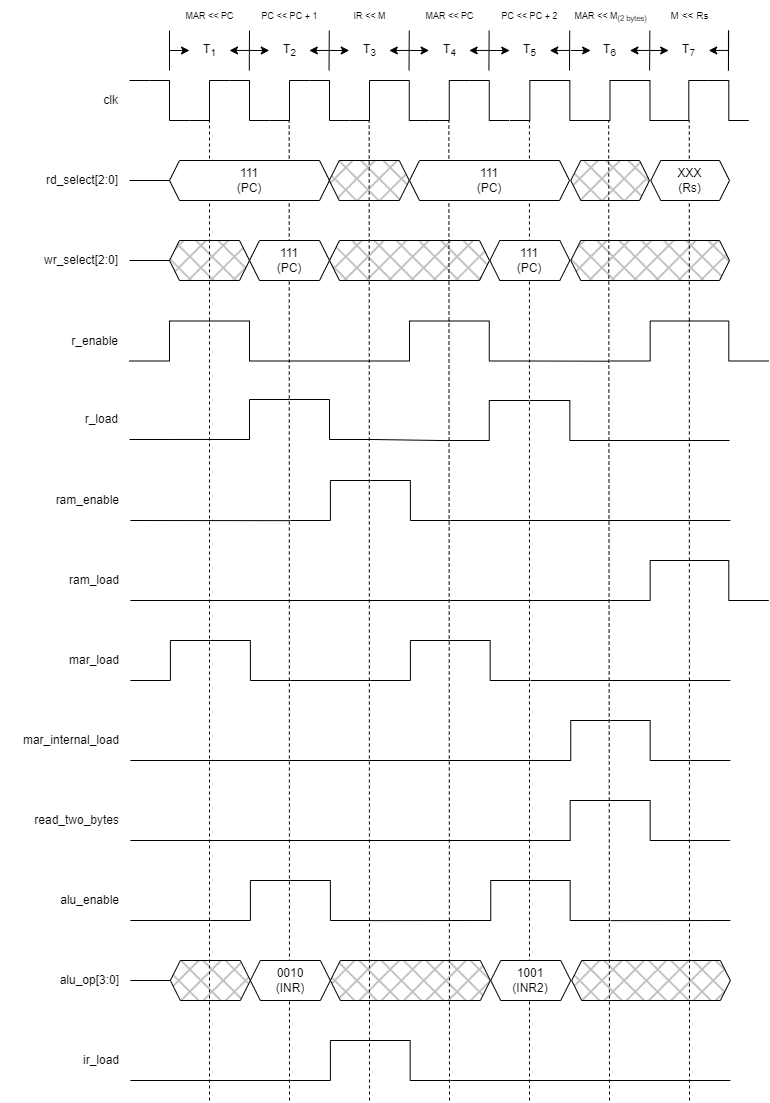
3 Byte

**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.*

### **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**

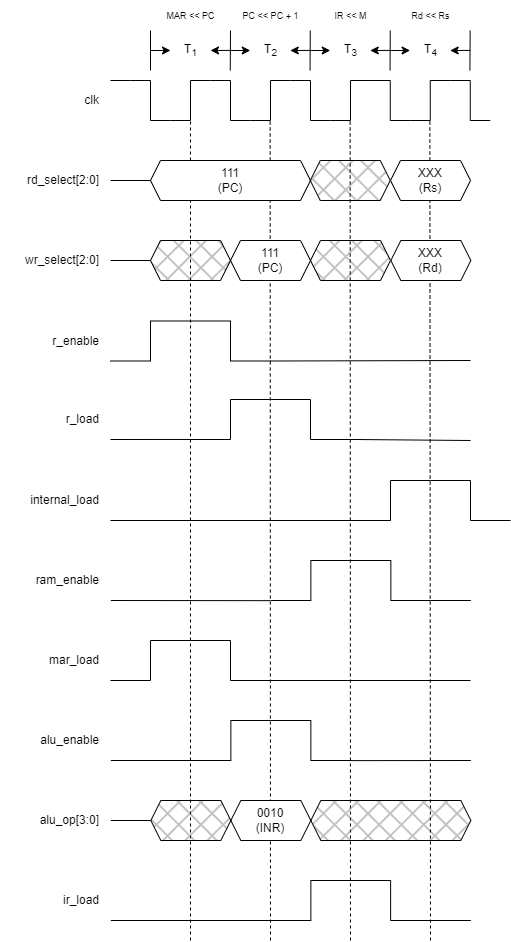
1 Byte

**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.*

### **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**

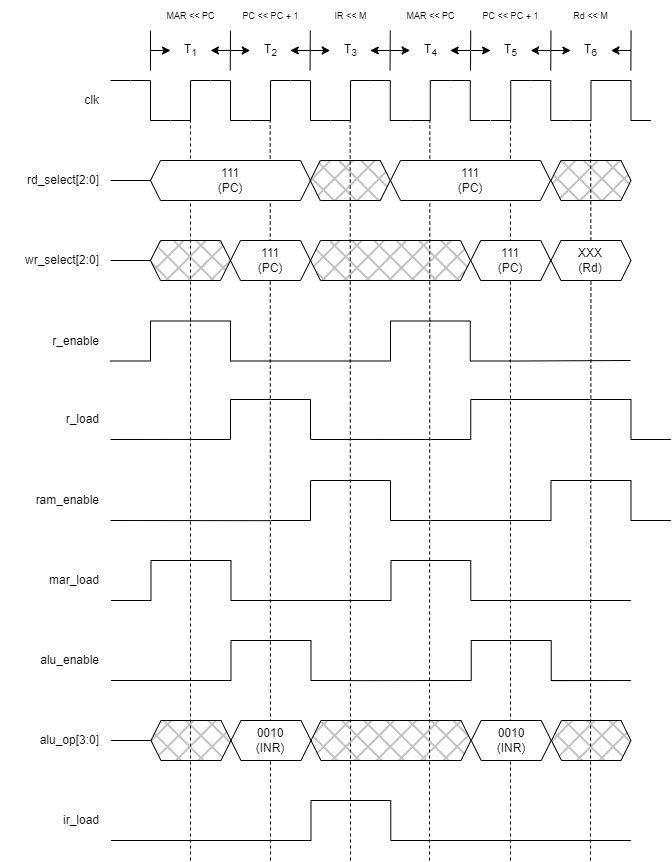
2 Byte

**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.*

### **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**

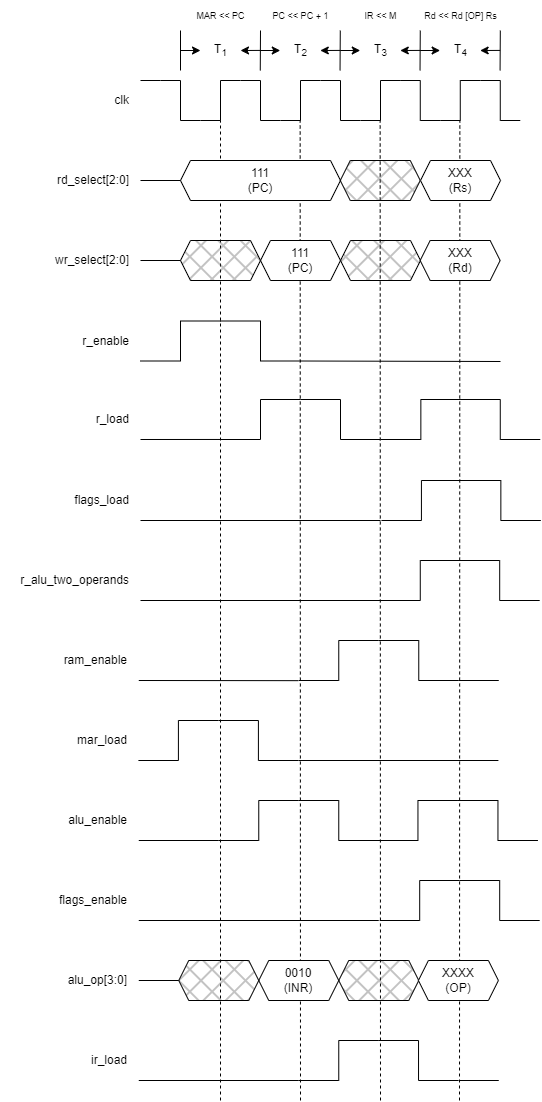
1 Byte

**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.*

### **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**

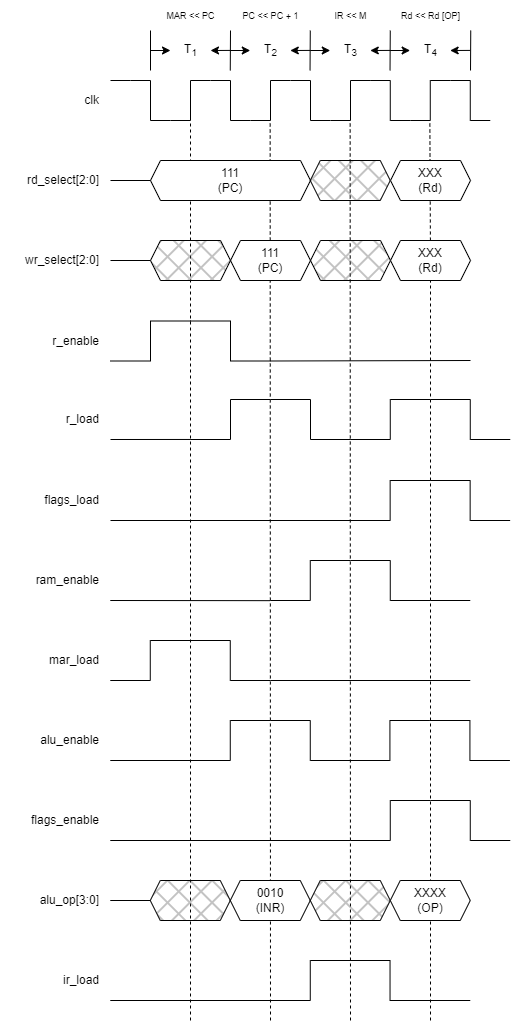
1 Byte

**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.*

### **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**

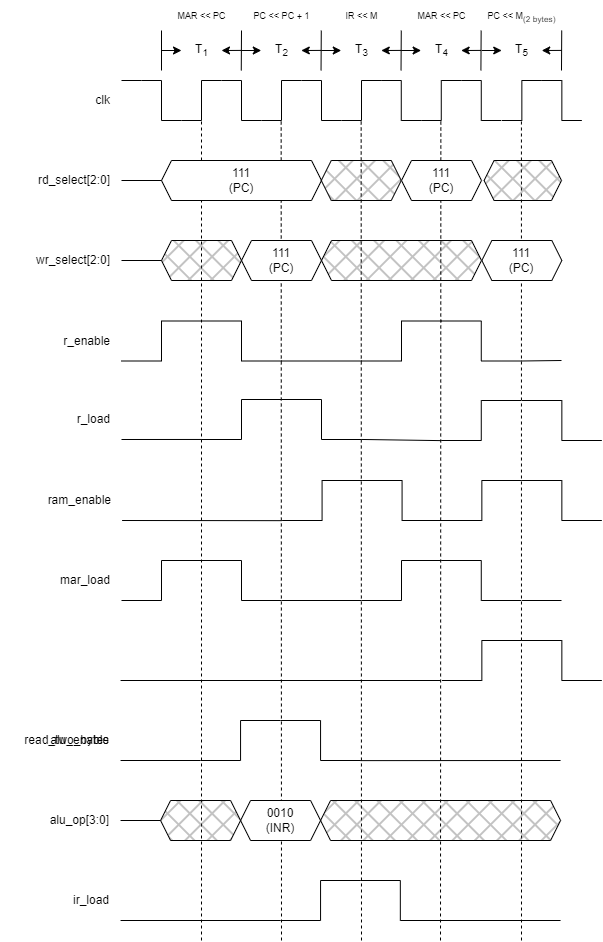
3 Byte

**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.*

### **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**

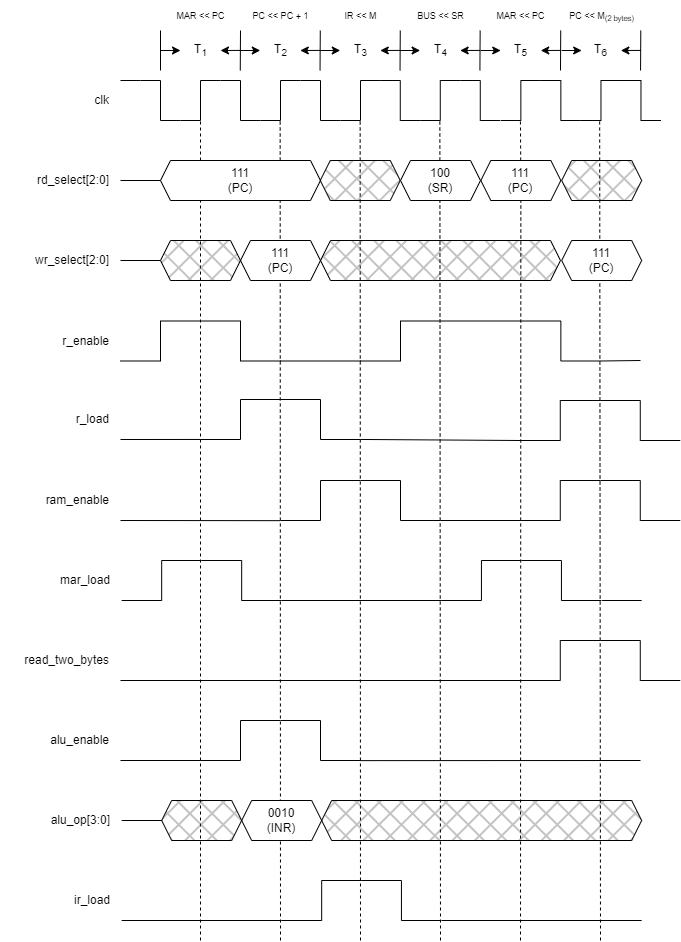
3 Byte

**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.*

### **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 |

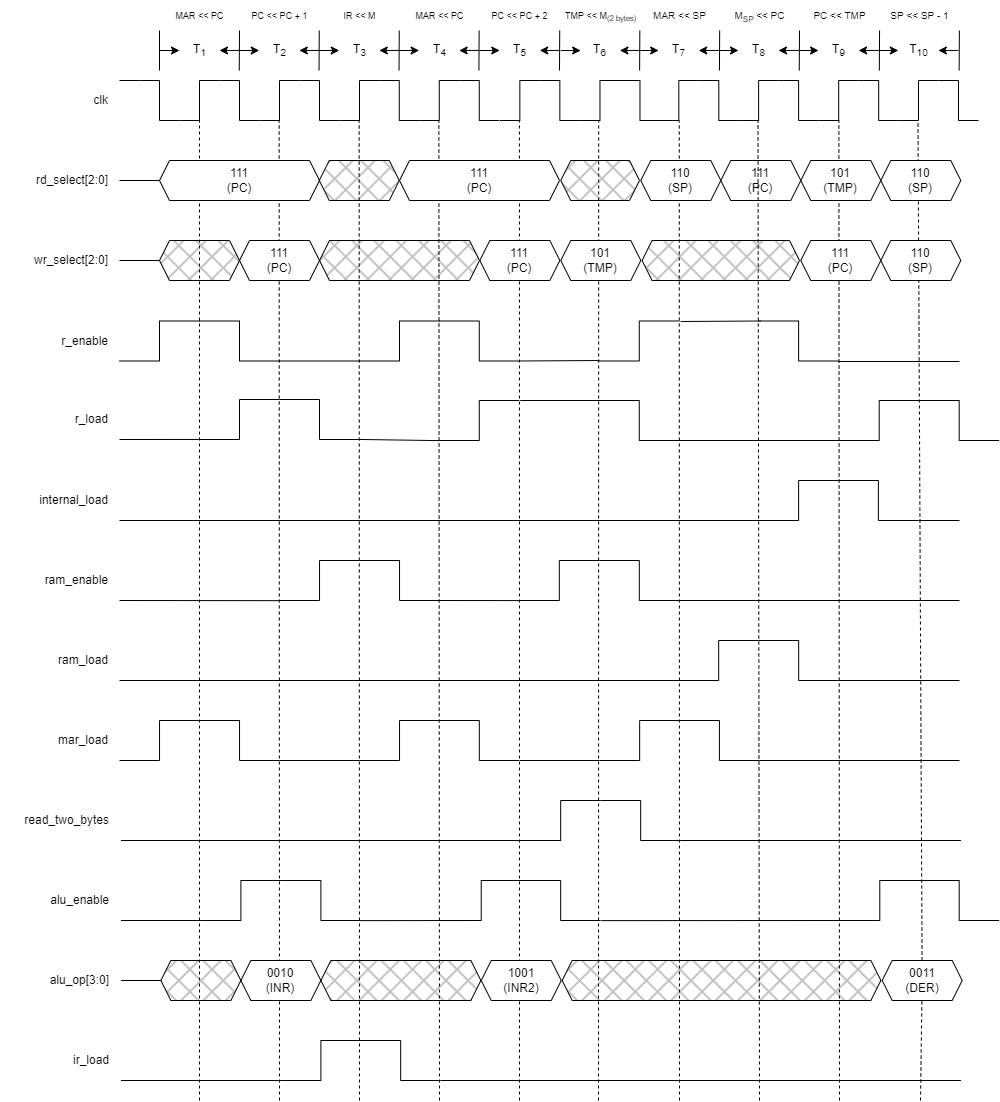
**Instruction Length**

3 Byte

**Example**

CALL 1111H

**T States and Control Signals**



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

### **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 |

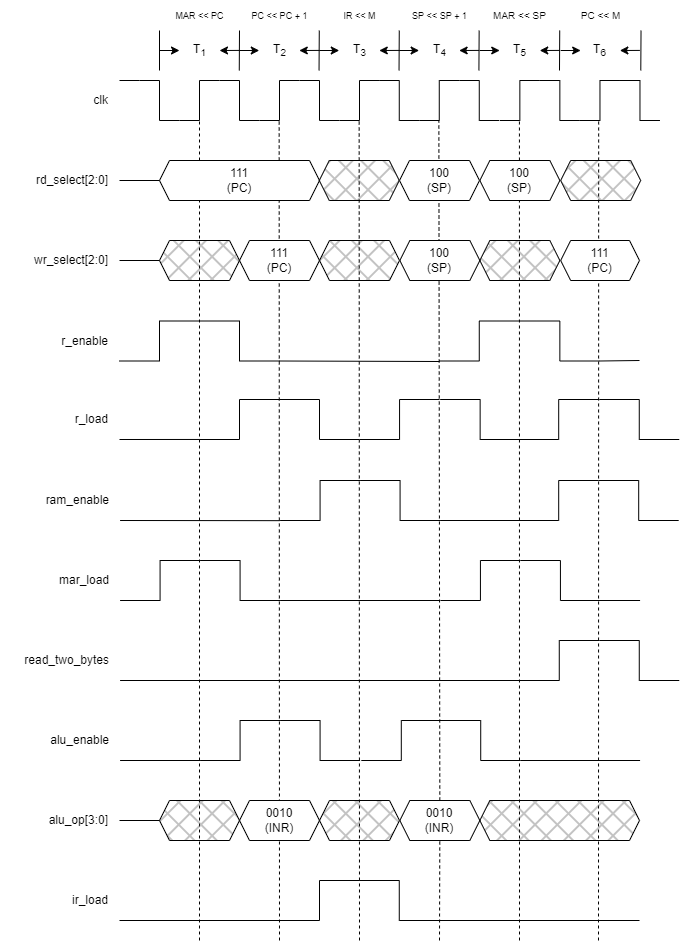
**Instruction Length**

1 Byte

**Example**

RET

**T States and Control Signals**



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

### **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 |

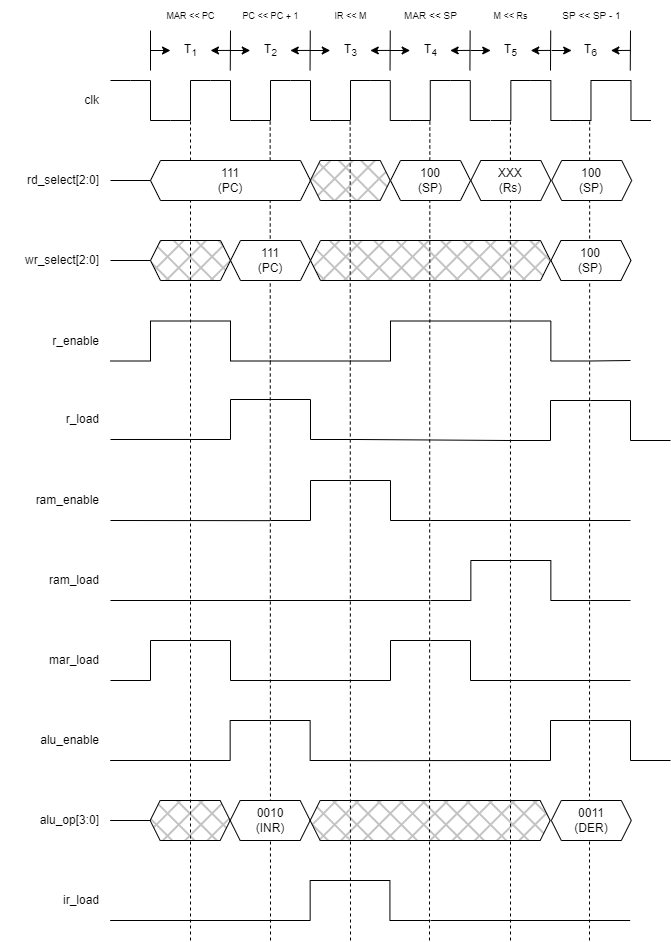
**Instruction Length**

1 Byte

**Example**

PUSH C

**T States and Control Signals**



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

### **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 |

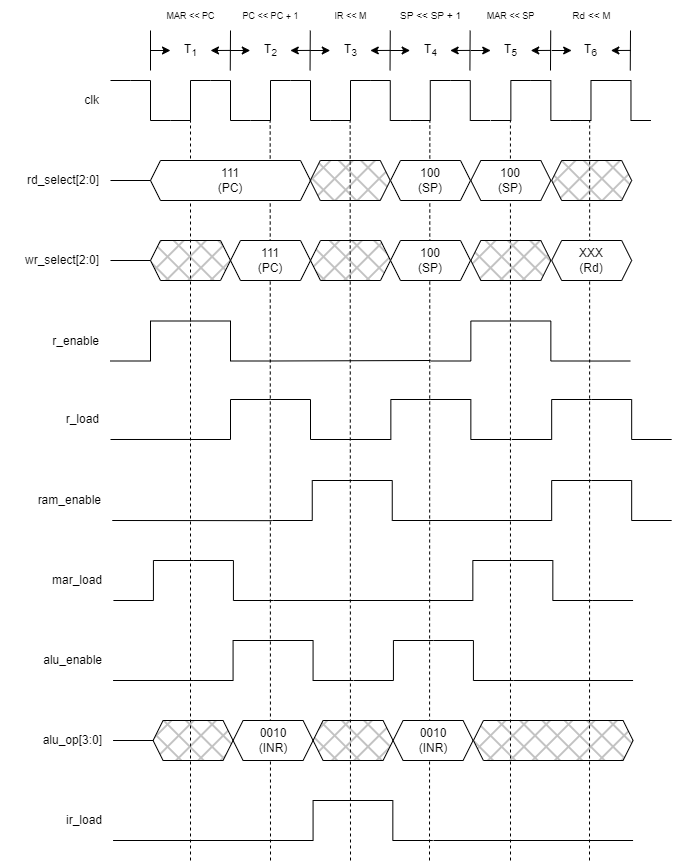
**Instruction Length**

1 Byte

**Example**

POP C

**T States and Control Signals**



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

### **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** | |
| **O-Type** | **Instruction Index** |
| 8 bits | |
| 1111 | 0110|0111|1000 |

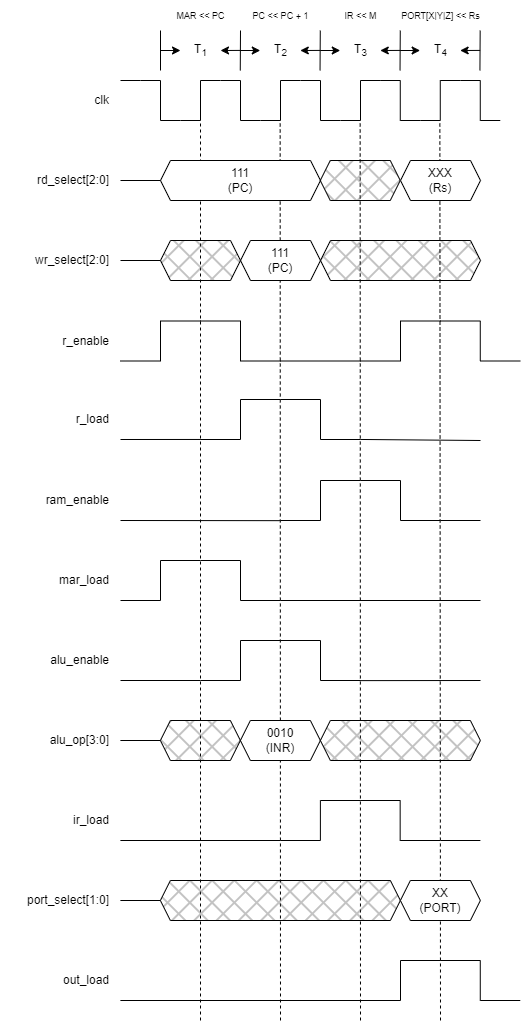
**Instruction Length**

1 Byte

**Example**

OUTY C

**T States and Control Signals**



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

### **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 |

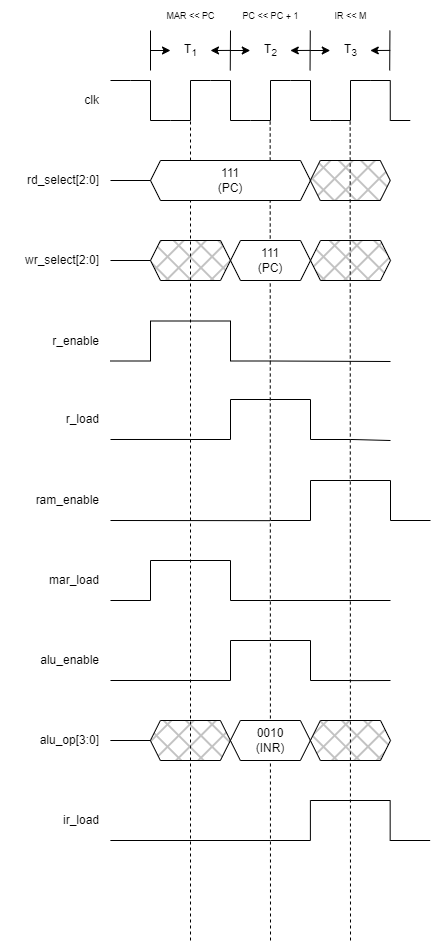
**Instruction Length**

1 Byte

**Example**

**NOP**

**T States and Control Signals**



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

### **HLT**

Stop the processing.

**Instruction Layout**

HLT

**Machine Code Format**

|  |  |
| --- | --- |
| **Opcode** | |
| **Operation** | |
| **O-Type** | **Instruction Index** |
| 8 bits | |
| 1111 | 1111 |

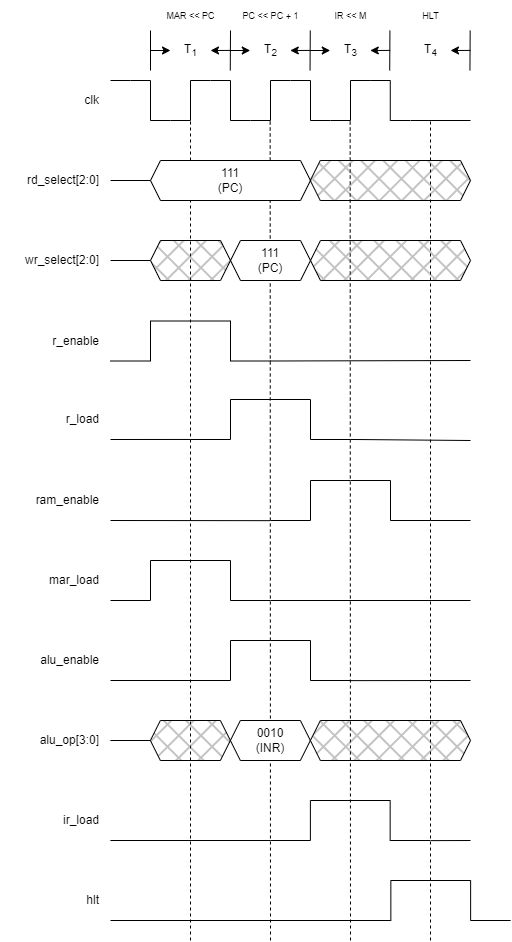
**Instruction Length**

1 Byte

**Example**

**HLT**

**T States and Control Signals**



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

## **Assembler**

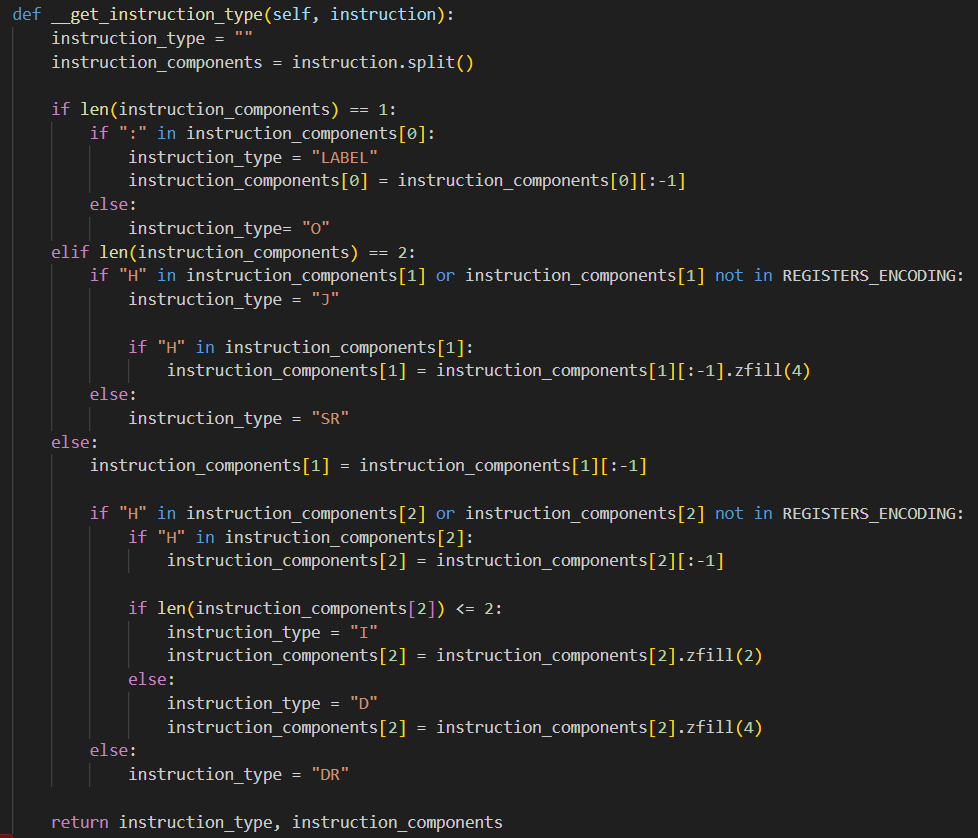
The assembler is a category of computer programs designed to take assembly instructions and convert them into a binary pattern that the microprocessor can utilize for executing operations, commonly known as machine code. While the assembler concept has been introduced in previous sections with examples 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.

A computer screen shot of a program code

Description automatically generated

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.



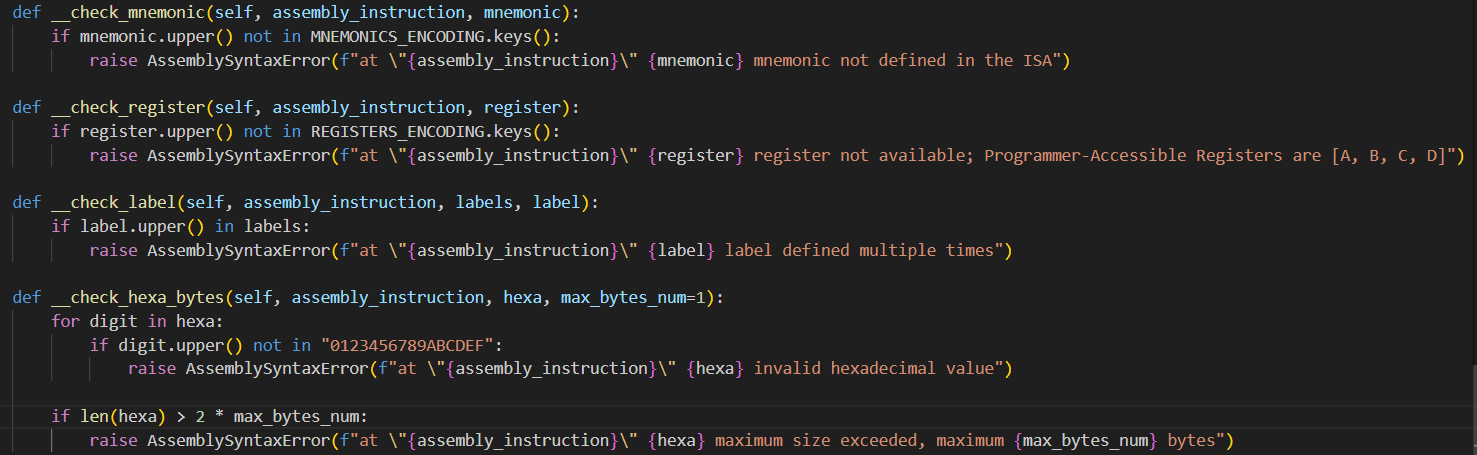
The next step involves checking for potential assembly syntax errors, such as:

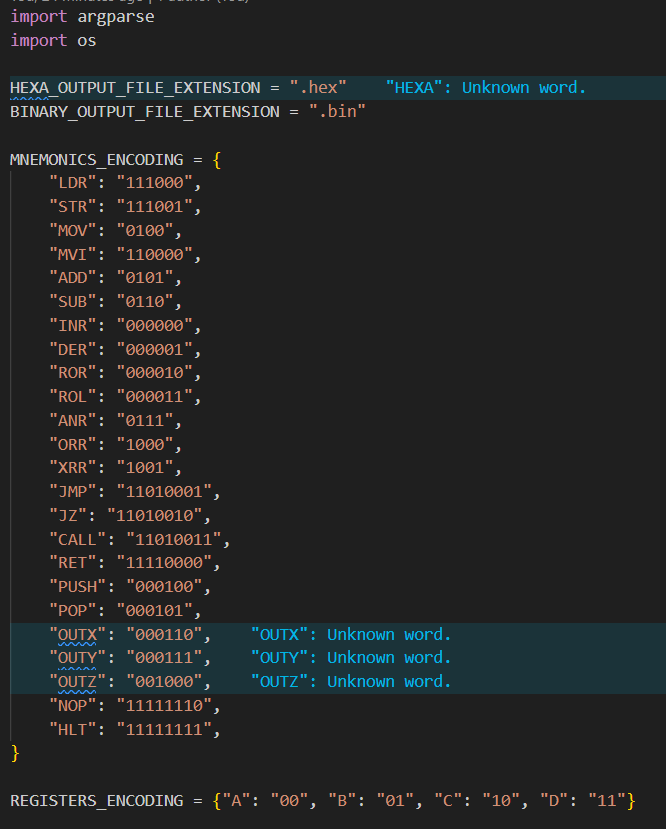
**Mnemonic Not Defined in the ISA**  
This occurs when the mnemonic in the instruction is not defined in the instruction set. It's important to note that the checking is case-insensitive.  
  
**Register Not Available; Programmer-Accessible Registers are [A, B, C, D]**This occurs when the source/destination 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.



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

A screenshot of a computer program

Description automatically generated

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.

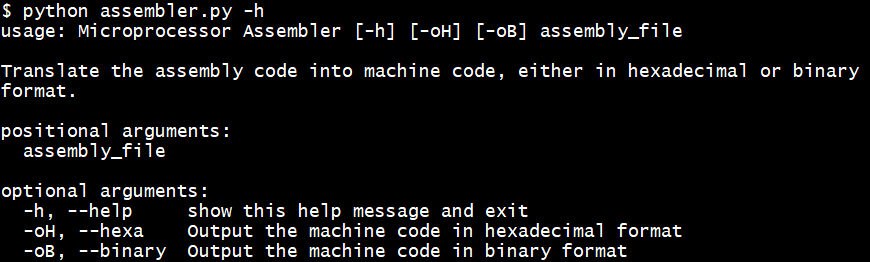
A screen shot of a computer program

Description automatically generated

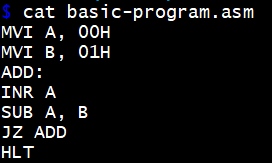
A screen shot of a computer code

Description automatically generated

### **Script Usage**

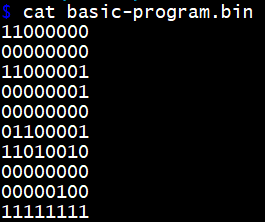
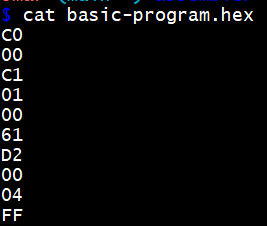
The assembler script can be utilized from the command line, and its usage is outlined in the help page:  
  


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



1. 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.



1. 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.

### **Script Full Code Snippets**

The complete code for the script is available in the [project's GitHub repository](https://github.com/omarelshopky/SAP_based_Microprocessor_Design/blob/main/assembler/assembler.py).

**A screen shot of a computer

Description automatically generated**

A screen shot of a computer

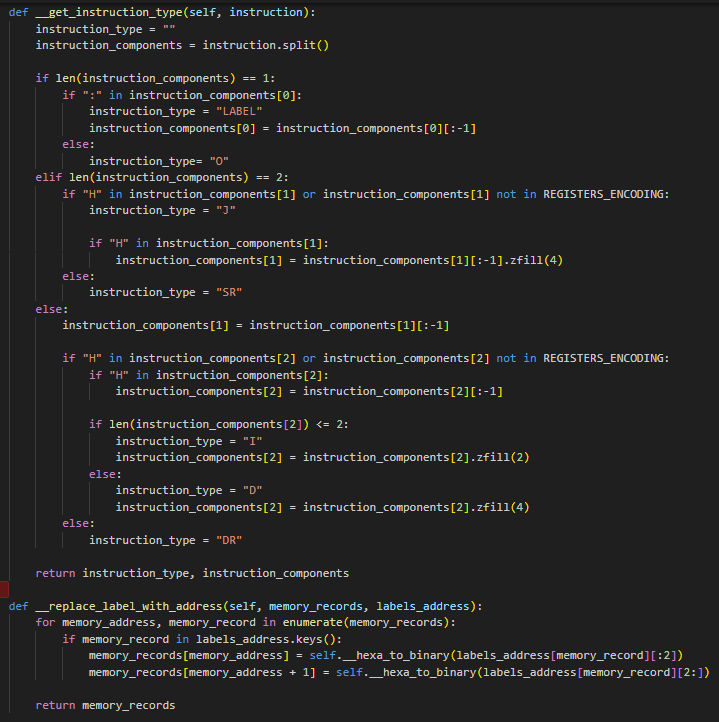
Description automatically generated

A computer screen shot of a program

Description automatically generated

A screen shot of a computer program

Description automatically generated



A screen shot of a computer program

Description automatically generated

# **Implementation**

## **Memory**

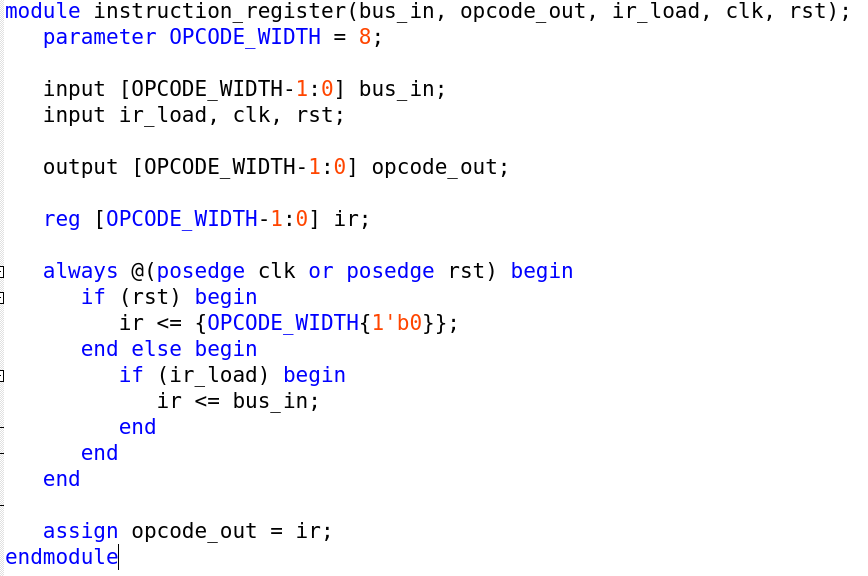
### **Verilog Code Snippet**

A screenshot of a computer

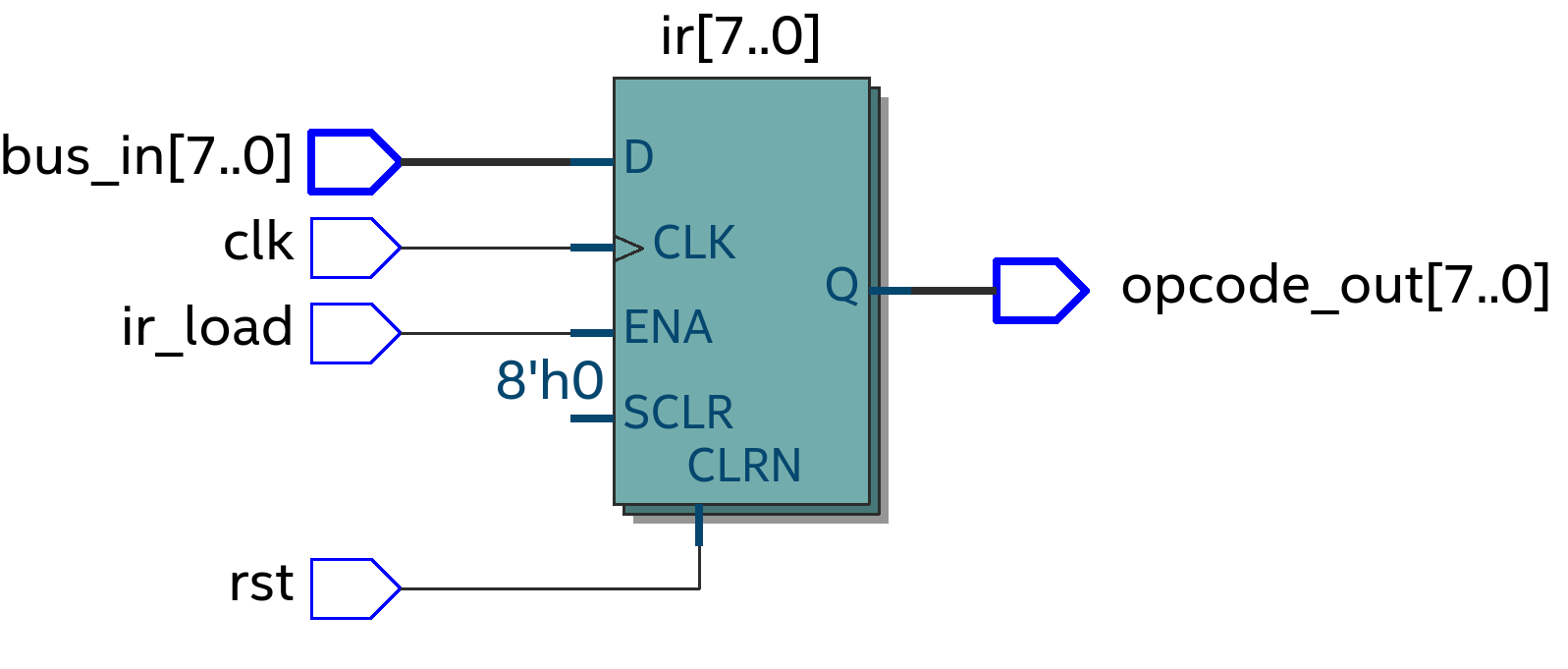
Description automatically generated

## **Instruction Register**

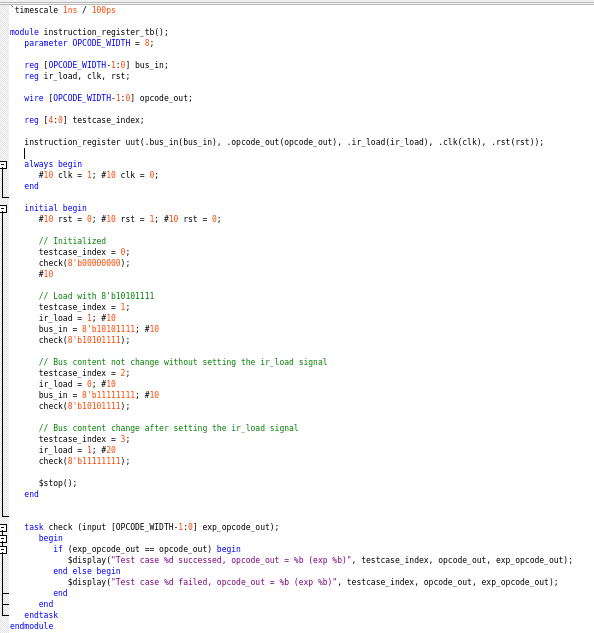
### **Verilog Code Snippet**



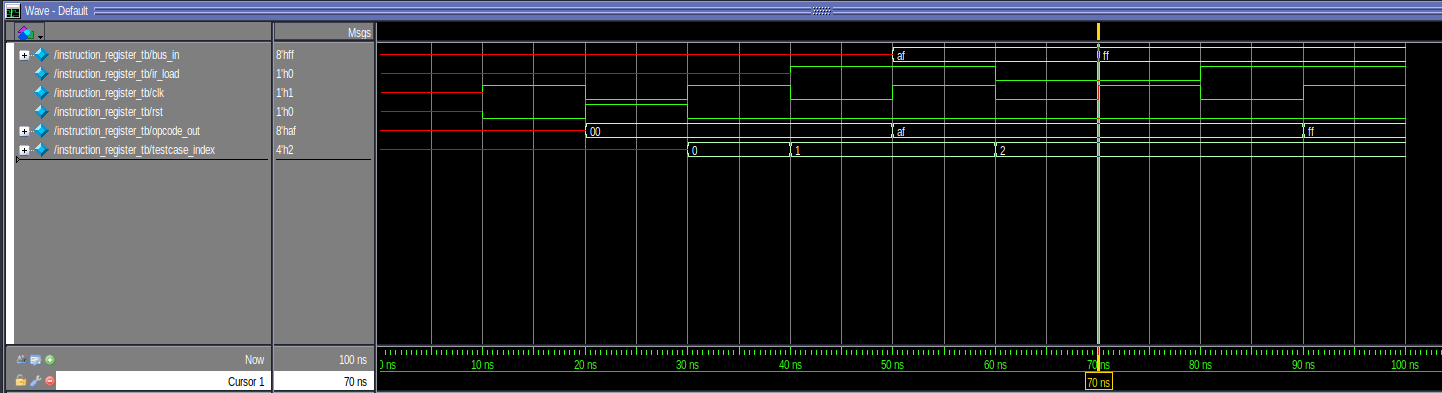
### **Netlist Schematic**



### **Test Bench**



### **Simulation Waveform**

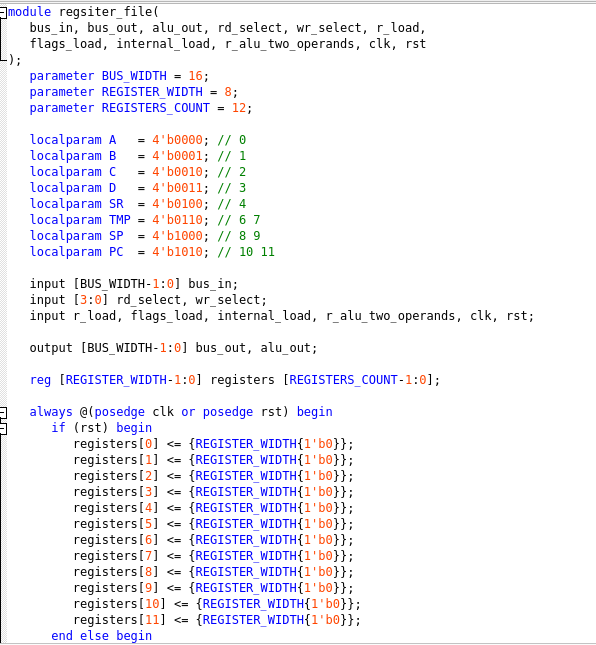


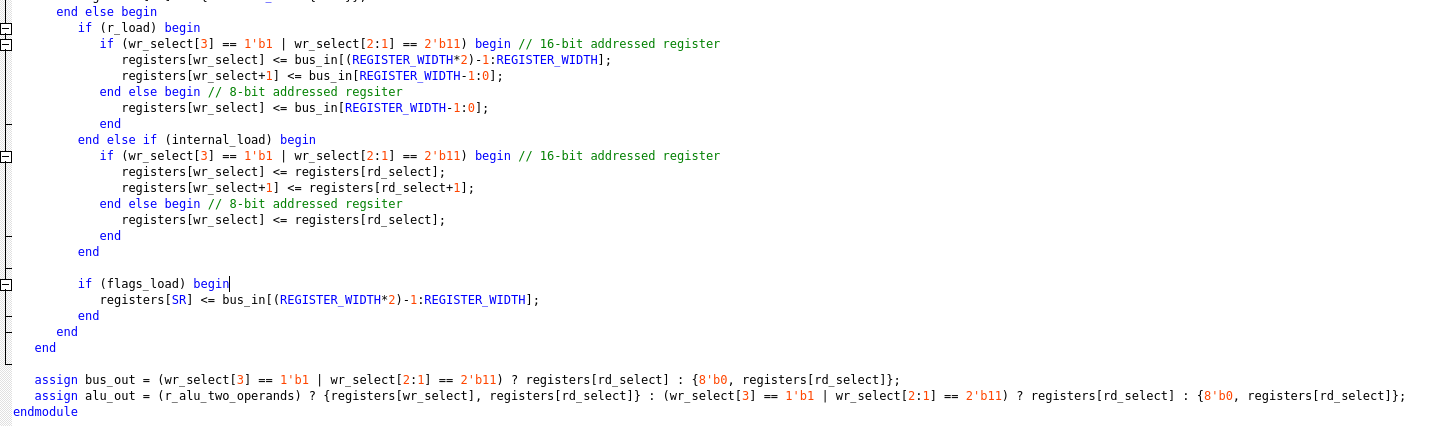
A close-up of a code

Description automatically generated

## **Register File**

### **Verilog Code Snippet**



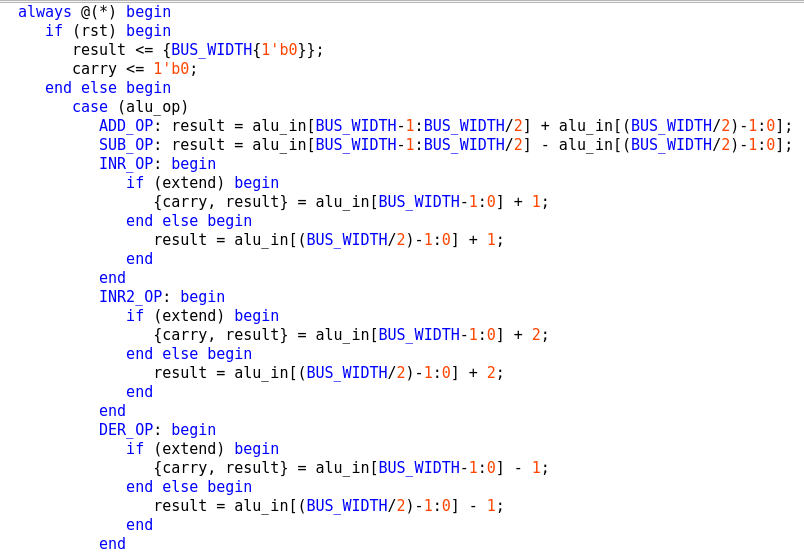


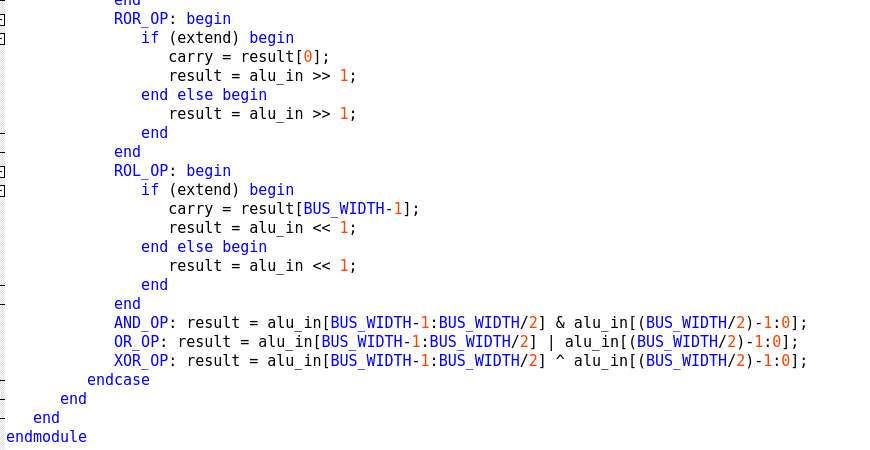
## **ALU**

### **Verilog Code Snippet**

A screenshot of a computer program

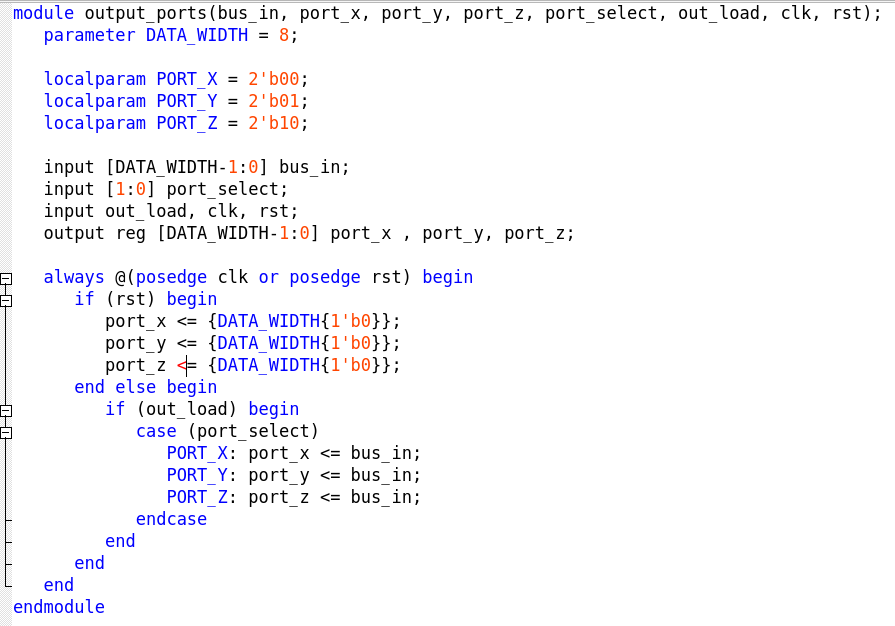
Description automatically generated





## **Output Ports**

### **Verilog Code Snippet**



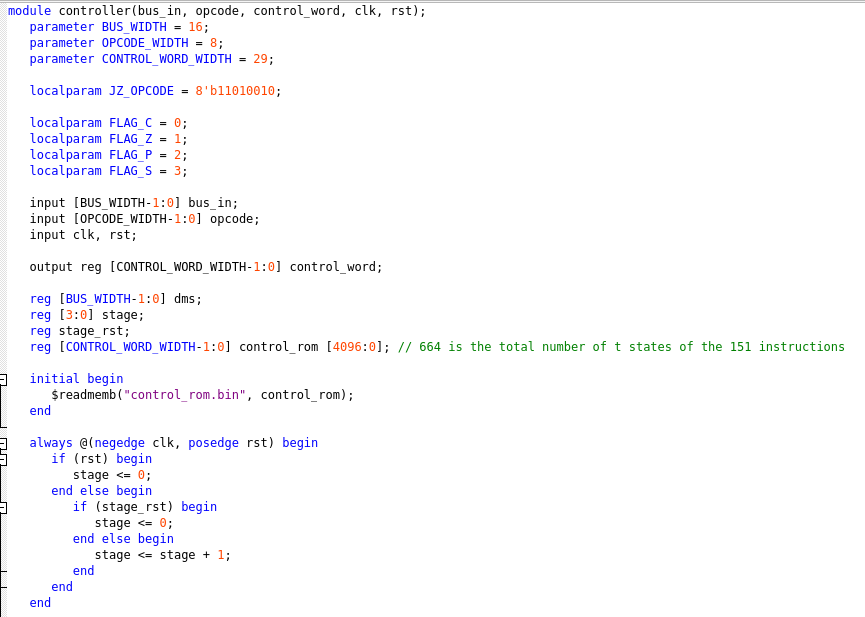
### **Netlist Schematic**

A diagram of a computer

Description automatically generated

## **Controller**

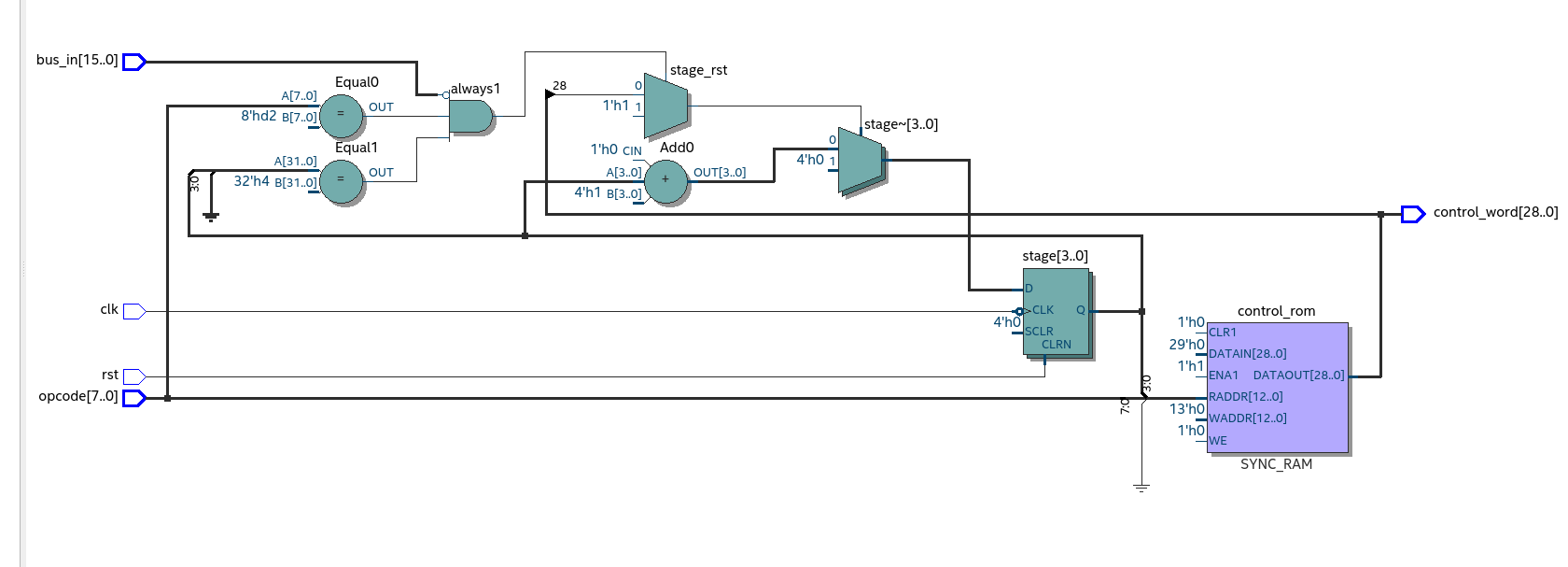
### **Verilog Code Snippet**



A computer code with blue and red text

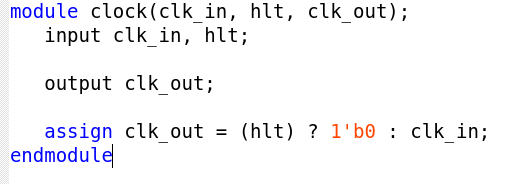
Description automatically generated

### **Netlist Schematic**

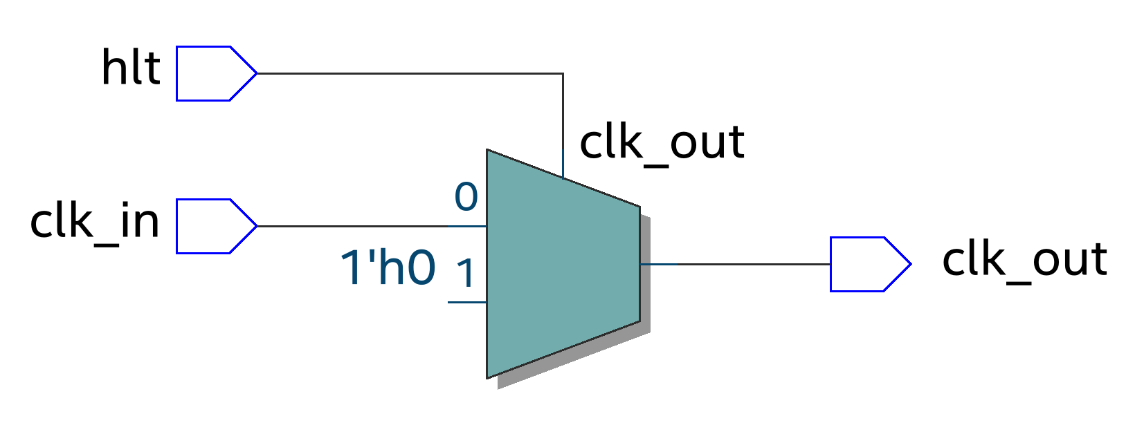


## **Clock**

### **Verilog Code Snippet**



### **Netlist Schematic**



# **Testing and Verification**

## **Test Plan**

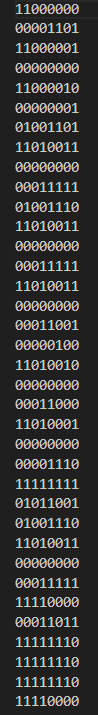
## **Simulation Verification**

## **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.  
     
    A screenshot of a computer

   Description automatically generated
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.



1. 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.
2. The Verilog code files were synthesized, placed, and routed using the Lattice iCEstick2 application tailored for the iCE40HX FPGA.
3. 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 |

1. The generated .bin file was flashed onto the FPGA using the Diamond Programmer.
2. 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](https://drive.google.com/file/d/1jr80LdNZxRnlyRhyvE5Ms-m3UUTPdf-F/view?usp=sharing).

# **Resources**

1. [Albert P. Malvino, Jerald A. Brown. “Digital Computer Electronics” (3rd Edition)](https://ia803000.us.archive.org/8/items/367026792DigitalComputerElectronicsAlbertPaulMalvinoAndJeraldABrownPdf1/367026792-Digital-Computer-Electronics-Albert-Paul-Malvino-and-Jerald-A-Brown-pdf%20%281%29.pdf)

# **Appendices**

## **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 ← Maddress |
| LDR ***B***, ***address*** | E1 | Direct |  | - | 3 | D | B ← Maddress |
| LDR ***C***, ***address*** | E2 | Direct |  | - | 3 | D | C ← Maddress |
| LDR ***D***, ***address*** | E3 | Direct |  | - | 3 | D | D ← Maddress |
| STR ***A***, ***address*** | E4 | Direct |  | - | 3 | D | Maddress ← A |
| STR ***B***, ***address*** | E5 | Direct |  | - | 3 | D | Maddress ← B |
| STR ***C***, ***address*** | E6 | Direct |  | - | 3 | D | Maddress ← C |
| STR ***D***, ***address*** | E7 | Direct |  | - | 3 | D | Maddress ← 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 | Mstack – 1 ← A |
| PUSH ***B*** | 11 | Register |  | - | 1 | SR | Mstack – 1 ← B |
| PUSH ***C*** | 12 | Register |  | - | 1 | SR | Mstack – 1 ← C |
| PUSH ***D*** | 13 | Register |  | - | 1 | SR | Mstack – 1 ← D |
| POP ***A*** | 14 | Register |  | - | 1 | SR | A ← Mstack |
| POP ***B*** | 15 | Register |  | - | 1 | SR | B ← Mstack |
| POP ***C*** | 16 | Register |  | - | 1 | SR | C ← Mstack |
| POP ***D*** | 17 | Register |  | - | 1 | SR | D ← Mstack |
| **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 |