CMPE4003 – CPU AND GPU ARCHITECTURE

CPU Design Assignment

Rashmika Dilshan K. Y. - 20543909
P. A. Sanduni Imasha - 20533463
Thimeth Jayasinghe - 20625506
Hikkaduwage Pasindu Laksara Fernando - 20502557
Samuel Jaden Christy - 20524289
Deshan Jayasinghe - 20819354

OCTOBER 27, 2024

Contents

List of Tables

List of Figures

Introduction

A system cannot exist without a ‘brain’ to controlling its peripherals and functions. Computer is no different. The ‘Central Processing Unit’ or the CPU is the brain of a computer which controls all the functions and performance as well as the other peripherals. Starting from basic arithmetic operations to the control of input and output operations of the entire system, is governed by the CPU of a computer. Additionally, the CPU is responsible for powering up the applications and programs from the level of simple applications all the way up to the complex computations done in the operating system.

With the advancement of technology, CPUs also evolved with a broad range of performance and feature improvements in order to provide the requisites of the users as well as the industries who build the computer system related peripherals. One significant change that has taken place on the development of CPUs over the past few decades is the improvements of core performance and the number of cores per unit. Starting from single-core processors which used in the early-stage CPUs, the number of cores per unit has increased over 30 cores as of today’s processors which are using in most of the computer servers. Being that each of these cores, to be able to perform as individual CPUs themselves, the overall performance of CPUs has increased significantly along with the parallel computing capabilities that facilitate by these multiple cores inside one single chip.

As of this assignment, the primary objective is to design and develop a 16-bit CPU using Quartus II software, where the two key components of the CPU, the Arithmetic Logic Unit (ALU) and Control Unit (CU) are designed using VHDL (Very High-Speed Integrated Circuit Hardware Description Language) in Quartus II software. Specifically, Quartus II 12.0sp2 Web Edition (32-Bit) version was used for this implementation and two designs were developed on where one implementation is evaluated using hardware while the other one is focused on full component-wise build in FPGA. In addition, Logisim Software was used to sketch the design from scratch where all the primitive peripherals of the CPU such as the registers to hold the basic instructions, address data or input/output data. Overall, both designs have used different approaches to address each peripheral of the CPU yet to meet the final goal of working 16-bit CPU including ALU, CU, PC (Program Counter), RAM (Random Access Memory),

Background Information

Introduction of CPUs

Computer architecture refers to the overall design and organization of a computer system, defining how its various components work together to perform tasks such as data processing. It focuses on the functional interactions between components rather than the specific technical details of their implementation [4].

The purpose of computer architecture is to manage all the operations performed by a computer system, from browsing the internet to printing documents. At its core, computer architecture is a framework designed to facilitate the collection, transmission, and interpretation of numerical data. Every task a system undertakes relies on the movement and processing of these numbers. By organizing how data flows and computations are executed, the architecture ensures that the system can efficiently perform a wide range of functions. Essentially, computer architecture is the mathematical structure that allows a computer to handle and process information, enabling it to fulfill various tasks accurately and effectively.

There are two main types of processor architecture:

- Complex Instruction Set Computer (CISC)
- Reduced Instruction Set Computer (RISC)

CISC processors have a single processing unit, auxiliary memory, and a small register set with hundreds of distinct instructions. These processors can execute complex tasks using a single command, which simplifies programming by reducing the lines of code needed. Although this approach requires less memory, it can result in slower instruction execution.

In contrast, a reevaluation of processor design led to the development of RISC architecture for high-performance computers. RISC focuses on simplifying the hardware, enabling faster processing by breaking down complex instructions into simpler ones. This design approach allows for faster execution of instructions, offering higher efficiency compared to CISC.

Both architectures represent different trade-offs between complexity, performance, and memory usage. Below Figure 01 shows the Computer Architecture of Von Neumann Architecture (a) and Harvard Architecture (b)

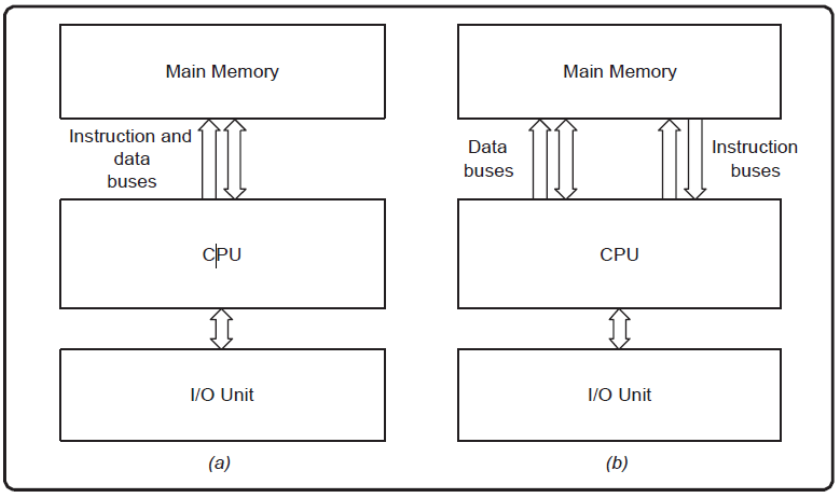


Figure 1 - Computer Architecture of Von Neumann Architecture (a) and Harvard Architecture (b)

The components of a computer architecture can be divided in various ways, depending on the chosen method of categorization. At its core, the main elements include the CPU, memory, and peripherals, all interconnected by the system bus. This system bus consists of the address bus, data bus, and control bus. Within this structure, the architecture is organized into eight key components, which are detailed below Figure 02.

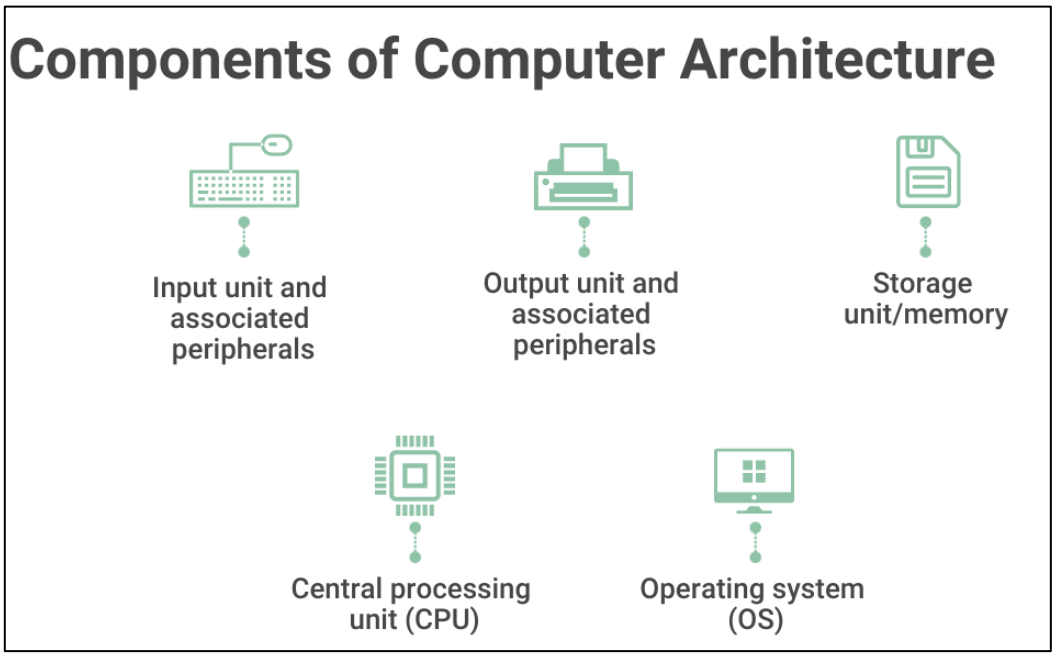


Figure 2 - Key components of a Computer Architecture

In conclusion, computer architecture forms the foundation of how a system operates, determining how data is processed, stored, and transmitted. By organizing the key components—such as the CPU, memory, and peripheralist ensure seamless communication and functionality within the system. Understanding the principles of computer architecture is essential for optimizing performance, enhancing efficiency, and driving technological advancements. As technology continues to evolve, so too will the complexities and capabilities of computer architectures, shaping the future of computing.

The following chapters discussed the primary components that every computer architecture must have.

Data Structure

Data structures are specialized formats designed to organize, process, retrieve, update, and store data efficiently. They provide a framework for arranging data to meet specific requirements or goals. In addition to holding the actual data, data structures manage the relationships between data elements, ensuring that these connections are preserved and easily accessible [3]. Following figure 3 shows the classification of Data structures.

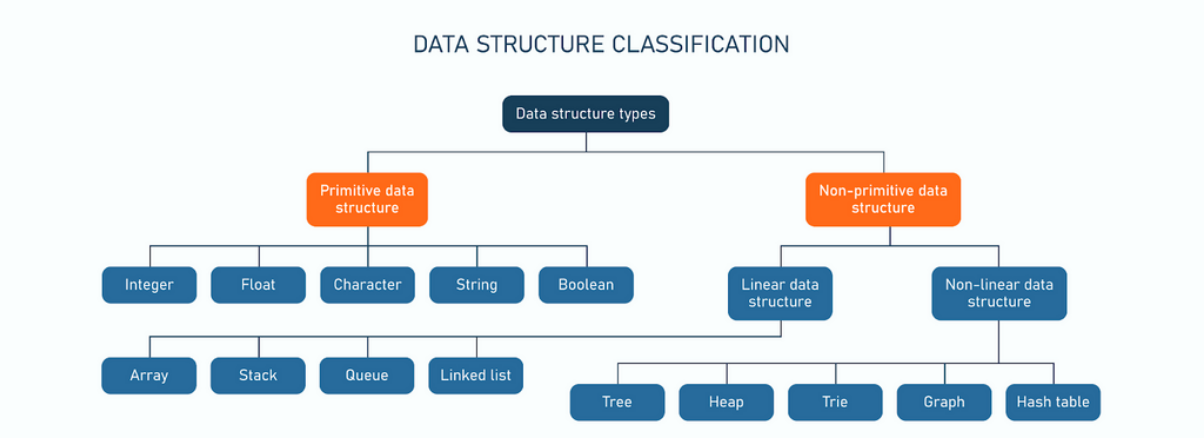


Figure 3 - Data structure classification

Processor

The CPU serves as the core of any computer system, executing instructions and overseeing system resources. As an integrated circuit, the processor handles mathematical computations and logical operations. In earlier computer systems, the CPU was the sole component

responsible for performing all calculations. Modern computers, however, often contain various specialized processors, each optimized for specific tasks.

Memory

Memory serves as the electronic storage area where a computer temporarily holds instructions and data that need to be accessed quickly. It enables the immediate use of information, making it essential for the computer's operation. Without memory, the system would not function effectively. Additionally, memory is utilized by the operating system, hardware, and software to ensure smooth and efficient performance.

In computing, memory is categorized into two main types:

- primary
- secondary

The term memory often refers specifically to primary memory, mainly a type known as random access memory (RAM). RAM is used on microchips that are positioned near the computer's central processing unit (CPU) [1]. Following figure shows the memory hierarchy of a computer system

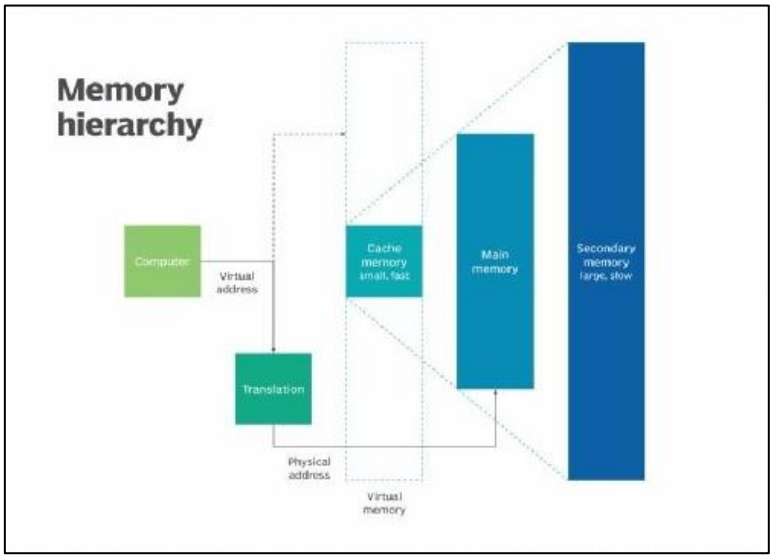


Figure 4 - Memory Hierarchy

Input/Output

In computing, input/output (I/O) refers to any program, device, or operation involved in the transfer of data to or from a computer system. Essentially, I/O encompasses the methods through which data is exchanged between the system and the user. I/O devices include peripherals such as keyboards, mice, monitors, and printers, as well as systems like databases and networks [2].

Summarized History of CPUs

One of the significant parts in a modern computer is the central processing unit (CPU) which set foot to it back into the late 1950s. The initial UNIVAC I computer included a processor, called the UNIVAC 1103. Using vacuum tubes, this unit was significantly slower and bigger than the CPUs that we know presently [5].

The advent of the transistor in the 1960s was a major turning point for CPU design, allowing to optimize faster and reliable smaller processor. The IBM 7090, the first transistorized computer capable of being commercially mass-produced set the stage for further developments in computing technology.

THE 1970s was a tipping point as the first microprocessors were born. Description: The world's first microprocessor 4004 is released by Intel, it opens the door to personal computing. This innovation resulted in the creation of more powerful (and smaller) CPUs, eventually birthing Intel's 8086 in 1978 and providing the seed for x86 architecture to become so common today.

The 1980s and early 1990s continued to bring rapid advances in CPU technology. Weaknesses such as greater clock speeds, more efficient manufacturing processes and the advent of multi-core processors raised digital performance and efficiency. Key processors of this period were the Intel Pentium series and AMD Athlon which dueled almost entirely on selling price.

Modern CPUs are extremely complex systems with multicore designs, advanced power management features and secondary processing units for things like video graphics or AI JsonRequestBehavior The largest manufacturers, like Intel and AMD and ARM, keep advancing their ground-breaking tools in the field of cpu technology that is making things more powerful inside variety buckets from tiny smartphones to enormous supercomputers.

Today, the CPU is still an essential part of modern computing. Researchers and developers are continuously working to enhance its performance, improve energy efficiency, and integrate it with new technologies. Following figures shows some significant moments of the CPU history.

Figure 5 - UNIVAC 1

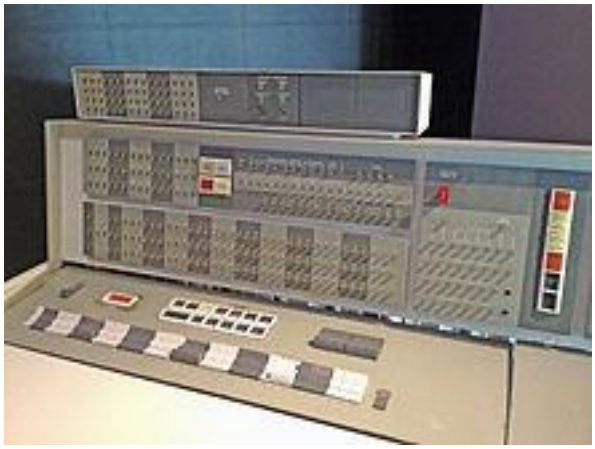


Figure 6 - IBM 7090

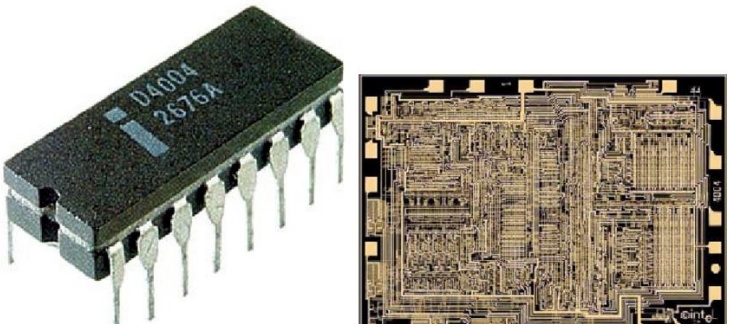


Figure 7 - Microprocessor 4004

Figure 8 - Intel Pentium series and AMD Athlon

Figure 9 - Modern CPU

Types of Computer Architecture

Computer architecture is the design and organization of computer components and systems. It specifies how components and systems of computers interact and process information It considers various conditions and settings. That affects productivity, efficiency, and efficiency. Understanding the Different Types of Computer Architectures, it is important for improving the hardware-software interface and developing new technology. Each architecture has its own unique advantages and applications. which determines the development of computer technology [6].

Summarized history of Computer Architectures

The evolution of computer architecture is an important step in the development of modern computers. The journey began in the early 1940s with von Neumann architecture. Developed by John von Neumann, this design provided a single memory location for all data and instructions. It laid the foundation for most computers today. During the 1950s and 1960s, computers were built using this architecture. As a result, processing power and performance were significantly improved. However, conclusions in terms of speed and performance were short-lived. It encourages researchers to investigate other changes. Harvard architecture emerged in the late 1940s and early 1950s, characterized by separate memory systems for data and text. This difference leads to faster processing speeds and better data processing. This makes it a popular choice in embedded systems and digital signage processing. As technology progressed, during the 1970s, RISC (Reduced Instruction Set Computing) architectures increased. This simplifies the instruction set to improve performance and efficiency. RISC design is based on the rapid execution of simple instructions. Just a few orders This led to advances in CPU design and paved the way for modern microprocessors. In the 1980s and 1990s, it developed multi-core architecture which allows multiple processors to run in parallel. Thus, greatly increasing the processing power. This structure limits the increase in clock speed. This allows for high efficiency without generating too much heat.

In the past few years The growth of interconnected and distributed computing applications has changed the way computers are used. These products use multiple applications and connected platforms to efficiently manage large-scale computations. This is important for applications in big data. artificial intelligence and cloud computing

Today, while computer architecture continues to evolve, new technologies such as quantum computing and neuromorphic processing It is starting to become a boundary in this field. Each commercial development has contributed to the growth and incredible capabilities of modern computers. It influences both hardware design and software development [7].

The Von Neumann Architecture

The von Neumann architecture consists of a single shared memory for programs and data. A single bus for accessing memory, arithmetic units, and program control units. The von Neumann processor runs data fetch and cycle operations sequentially [8].

The development of Von-Neumann computer architecture began in 1945. It was later named Von-Neumann architecture. In the past, there were two types of computers:

- Fixed Programming Computers – Computers are specialized in their use and cannot be reprogrammed, such as calculators.
- Stored Program Computers – can be programmed to perform various tasks. that can store applications

Therefore, is the origin of the name. Modern computers are based on the stored programming concept pioneered by John Von Neumann. Programs and data are stored in the same memory. This new concept means that computers built with this architecture will be much easier to reprogram. Following structure shows the schematic overview of Von Neumann architecture,

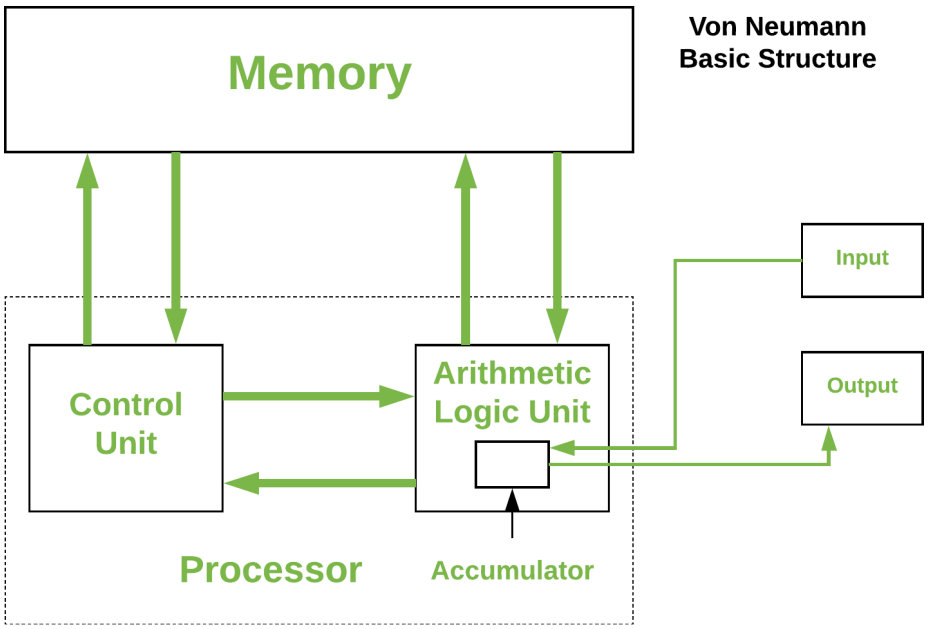


Figure 10 - Von Neumann architecture

The Harvard Architecture

In a normal computer using the von Neumann architecture, both instructions and data are stored in the same memory. Therefore, the same buses are used to transmit messages and data. This means that the CPU cannot do both at the same time. (read text and read/write data) so to overcome this problem the production of Harvard architecture was made.

Harvard Architecture is a kind of computer structure that features distinct storage and separate buses (sign paths) for instructions and datas. It became mostly designed to deal with the constraints of von Neumann's Architecture. One of the important things benefits of getting separate buses for instructions and statistics is that the CPU can concurrently access instructions and examine or write statistics [9]. Following structure shows the schematic overview of Harvard achitecture,

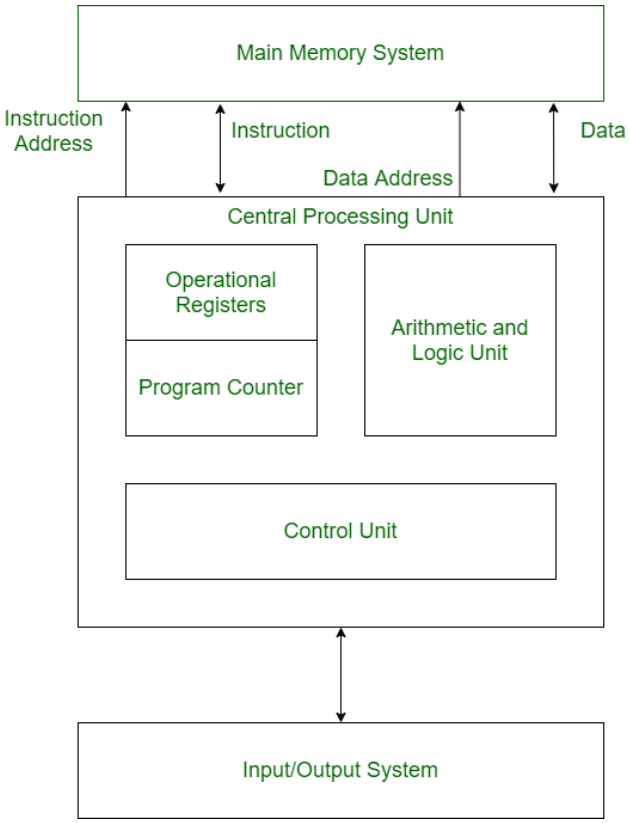


Figure 11 - Harvard architecture

Types of CPU Architecture

The CPU architecture plays an important role in determining how the processor works and interacts with other components of computer system It covers the principles of design organizing and the set of instructions that control the CPU's operation. Understanding the different types of CPU architecture is important to increasing efficiency and effectiveness in computational tasks. The most common architectures include CISC (Complex Instruction Set Computing), which provides a rich set of instructions for complex tasks, RISC (Reduced Instruction Set Computing), which simplifies instructions to increase speed and efficiency, and VLIW (Very Long Instruction Set Computing).), which allows the processing of multiple commands simultaneously. Each architecture has its own unique strengths and applications. It influences everything from embedded systems to high performance computing.

Simple as Possible Architecture

The SAP architecture is an example of how digital electronic computers can be used to design and analyze complex conceptual systems in a digital electronic society. This structure provides a solid framework that helps engineers and computer scientists understand the principles of logic and system design. The SAP architecture evolution has resulted in three product series:

- SAP-1
- SAP-2
- SAP-3

Each edition is built on the original edition. It combines technological improvements and design techniques to improve functionality and performance. SAP-1 provides key components and functions. This supports subsequent iterations. SAP-2 expands on these concepts by increasing complexity, while SAP-3 makes production easier. Helps improve processes and system capabilities In summary, SAP architecture not only serves as a good example for digital computing electronics. But it also plays an important role in the academic environment. Helps both students and practitioners understand the complexities of digital system design. Continuous development through the SAP-1, SAP-2 and SAP-3 versions shows that technology continues to advance in this area. This paves the way for innovative and efficient digital processing solutions [10]. Following figure shows the SAP-1 Architecture.

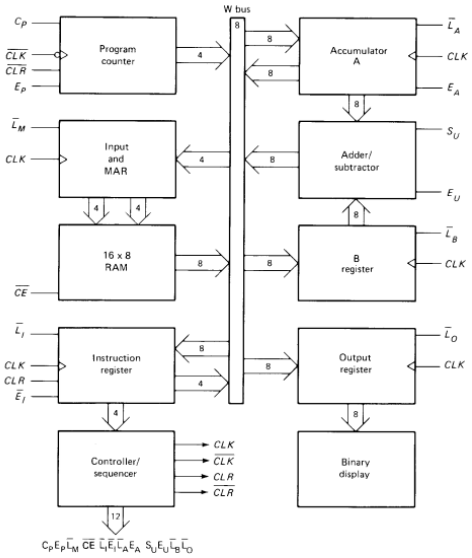


Figure 12 - SAP-1 Architecture

Reduced Instruction Set Computer Architecture

RISC architectures were designed to solve problems related to the increased complexity and length of instruction sets found in traditional CISC architectures. In the late 1970s and early 1980s, computer scientists began to evaluate the need for Re-execute complex instructions in the processor. This has led to the creation of a new design philosophy that focuses on reducing subject complexity while maintaining or improving functionality. Therefore, RISC architecture was born from this idea [11].

The RISC architecture hardware is specially designed for a fast-learning experience. This is done by a small, precise set of instructions. Including many scripts In RISC, data paths are used to store and process data in the computer. It plays an important role in managing data in applications and during its migration between applications and memory. To reduce the time to access main memory Users will use the cache system. Message caches are especially useful for quickly retrieving and storing frequently processed messages. and therefore, it is a quick command execution. Meanwhile The database also serves as a storage space for frequently accessed data from main memory. Following figure shows the overview of RISC.

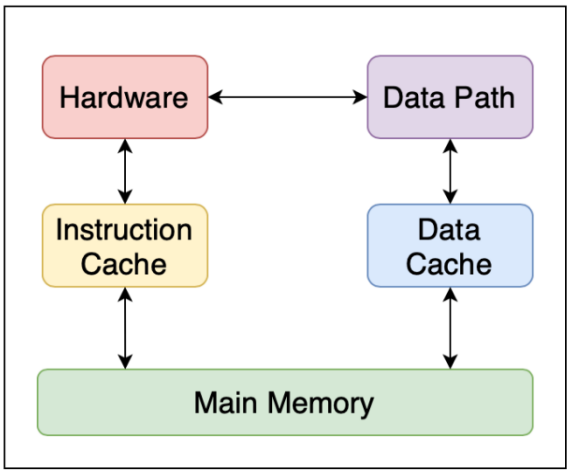


Figure 13 - RISC architecture

Complex Instruction Set Computer Architecture

Complex Instruction Set Computing (CISC) is at the critical part of computer application development. It represents a unique design philosophy. Unlike Lower Instruction Set Computing (RISC), CISC users emphasize flexibility by combining multiple complex instructions. As a result, the decoding and execution process becomes more complex. The objective of this report is to provide an objective analysis of the CISC architecture, focusing on its main features, features, and historical developments. and its relevance in modern computing environments.

In contrast, the CISC architecture is designed to provide a wide range of complex and versatile instructions within a single instruction set for computer processors. This approach differs from RISC architecture, which focuses on simplicity and efficient execution. The multi-step instructions found in CISC systems make them well-suited for handling a variety of tasks. Following figure shows the overview of CISC.

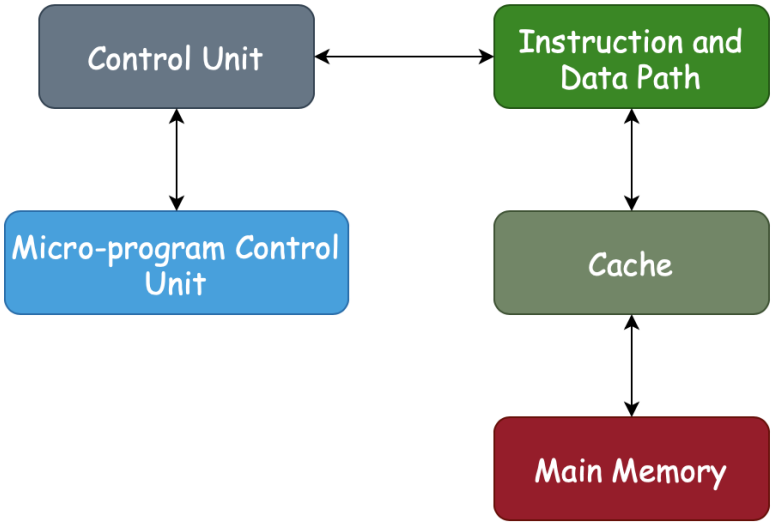


Figure 14 - CISC architecture

DESIGN OVERVIEW OF THE 16- BIT CPU

Approach 01 – Design of a 16-BIT CPU with output display

Design Approach and Overview

This design was focused on developing a critically analysed 16-bit CPU. This CPU is designed based on a simple Accumulator-based architecture with a Finite State Machine included Control Unit to perform all the primary arithmetic and logical operations that a 16-bit CPU can perform. Due to the availability of the hardware and lab facilities has been changed during the time of this assignment, this approach was set to build a simplified model of a 16-bit CPU with an ALU entity, capable of performing a series of arithmetic operations in a structured and sequential manner. Following Specifications and considerations were taken into account during the period of this design development. The CPU was developed with essential components like the Arithmetic Logic Unit (ALU), Registers, a simplified Control Unit (CU), state control using a Finite State Machine (FSM), and intermediate signals for managing data flow arbitrary RAM and operations. Each component plays a unique role in supporting the CPU's functionality and controlling multi-step calculations. Functionality of this CPU design is demonstrated in the Youtube link attached at Appendices at the end of this document.

Finite State Machine (FSM)

For the purpose of mathematical modelling of this ALU, Finite State Machine (FSM) theories were used. All the four key concepts, which are states, transitions, inputs and outputs were thoroughly considered into each portion of the design while the development was progressed. A **Moore Machine FSM** was used for this design, where the outputs are independent from the inputs and depends only on the current state of the machine. Outputs were set to change only when the state is changed. The FSM's current state dictates which operation is performed, providing a structured approach to managing complex calculations.

ALU (Arithmetic Logic Unit)

- The ALU performs sequential arithmetic operations using a finite set of states controlled by a FSM (calc_state). Each state corresponds to a mathematical operation: addition, multiplication, subtraction, and division.
- The ALU accepts two 5-bit inputs (reg_a and reg_b) and produces an intermediate result through several stages
    - value1: Result of the addition of reg_a and reg_b.
    - value2: Result of multiplying value1 by reg_a.
    - value3: Result of subtracting reg_a from value2.
    - alu_result: Final result, where value3 is divided by reg_b (if non-zero).
- Limited Overflow Detection is available in the ALU. Since the ALU operates on values that can expand up to 10 bits, overflow management is essential to ensure each operation's integrity which is included here.

Sequential Processing

The sequential instruction handling is an arbitrary approach to a Program Counter. Sequential processing steps of this CPU design are synchronous, where all the tasks are executed in a step-by-step manner which is coordinated by clock cycles of the CPU. Discrete time intervals are defined by the clock signal and referred to as ‘clock cycles’ in general. This ensures the integrity of the computational steps of an operation that is performed by the ALU.

Control Unit(CU)

The Control Unit in this CPU manages multi-step operations using a State Machine (FSM) and a state variable (calc_state). Button control initiates the sequence, selecting mathematical operations based on button press and sequence logic. Data flow control manages input and output, while result display on LEDs allows real-time output. Division by Zero handling prevents errors, enabling the functionality in the CPU.

Input Handling

The CPU can accept 10-bit inputs through switches, enabling it to work with a range of values (0 to 1023 in unsigned representation or -512 to +511 in signed representation). This was a design constraint since the boards available on campus were DEO Cyclone 3.

Error Handling

The code includes basic error handling for division by zero, ensuring that the ALU does not produce undefined behavior in such cases.

Random Access Memory (RAM)

The intermediate signals (value1, value2, value3) serve a similar purpose to RAM by temporarily storing data during computation. These registers allow for the retention of data values across multiple clock cycles, enabling complex arithmetic operations to be executed step-by-step.

Volatile Storage: The intermediate values are reset to zero upon a system reset (rst), similar to how RAM loses its data when power is lost or during a reset operation.

Dynamic Access: Data stored in these registers can be read and written in any order.

Accumulator Register, Operand Register and Intermediate Value Register

Operand Register – Stores 5-bit values and receive input from switches.

Intermediate Value Registers – Temporary storage for intermediate results during multi-step operations.

Accumulator Register – This unit holds the operation results of the computations. This unit Stores the final computed result after all operations are completed. Outputs the final result to the LEDs (both external and onboard).

LED Output Display

The results of the computations are displayed on the LEDs, providing real-time feedback of the ALU's output, which is crucial for debugging and monitoring the CPU's operations.

The use of a finite state machine in this design enhances the clarity and efficiency of the computational process, while the LED outputs provide immediate visual feedback on the results of the computations. This code serves as a foundational building block for more complex CPU designs and can be expanded upon to include additional functionality or enhance the existing operations.

Approach 02 – Multilevel 16-bit CPU

In this approach the development process was initiated to design a CPU consisting of all primary functional components such as ALU, Registers, memory (RAM), Control Unit (CU), Buffers and the Program Counter (PC). The role and the functionalities of each component are as follows,

ALU (Arithmetic Logic Unit)

- The opcode (S) and the Carry in signal (C_IN) contributes on performing the arithmetic and logical operations
- The result of the operation (G) as well as the carry-out bit (C_OUT) contributes to manage the mathematical operations as well as the bitwise logics such as OR, AND, XOT, NOT etc.
- Having a carry bit for the result outputs, enables the ALU to manipulate operations that even exceeds 16 bits which is a crucial requirement to detect and manipulate the overflows accurately.

Buffer and Register Units

- **Buffer16Bit –** This unit controls the dataflow by setting the data output (S) when it is enabled (E) . But if it is not enabled then a high impedance will be initiated (Z)
- **BufferedRegister4Bit –** This register is consisting of 4 bits to do the primary manipulations such as load, clear and enable; to the intermediate data that holds during the computations.
- These two subunits were used to ensure the data integrity of operations, with the intention of enabling functional help to manage dataflow and bus connections.

ALU16BitBuffered

- This unit encircling the whole ALU unit, by adding the buffered outputs, zero and carry flags together (ZF and CF)
- Both ZF and CF are crucial to be implemented into the system as they are important for the process of implementing conditional instructions and for the control logics as well.

Control Unit

- In this unit decoding opcodes and generating the control signals such as MI, RO and RI are manipulated to coordinate the movement of data in CPU as well as the instruction execution steps.
- Each opcode is consisting of a FSM that organizes the control signals based on the 'Step' input.

Accumulator Register, Instruction Register and B Register

- **AccumulatorRegister –** This unit temporarily holds the intermediate operation results of the computations. This unit receives data only when the load is enabled beforehand, and the reset command has been issued to wipe out any previously holding bits.
- **B Register –** The second operand for operations are holding by this unit
- **Instruction Register –** This unit is responsible for holding the instructions for the currently executing operations. This unit ensures the instruction flow is consistent and interpreted with each executing opcode.
- The existence of **AccumulatorRegister** and **B Register** in this design enhances the capability of handling complex operations.

Program Counter (PC) and Memory Address Register (MAR)

- **Program Counter –** This unit provides program flow management, instruction incrementations for computations as well as address loading steps based on INC and LOAD control signals.
- **MAR –** This unit holds addresses for the memory operations. And this unit is important memory access tasks that are based on the addresses.

RAM

- RAM unit in this CPU is 16-bit wide and is capable of handling write and output enables (WE and OE) signals separately. There two signals provide dynamic switching between each mode which ensures that memory is not accessible unless it is necessary.
- To control memory locations, it uses ADDR and both the DATA_IN and DATA_OUT are set to perform the data input and output operations.

Top_lvl_enty_CPU

- This VHDL code integrates all the components of CPU and manages the functionality of the CPU. At the same time it coordinates the data paths between each component ensuring that the communication between these units/components are well balanced.

PROGRESS REPORT

Work Breakdown Structure

To ensure the design and development of this CPU project holds a properly balanced time and task management, strategical approach was made under seven different major deliberations such as Research Definition, Design Approach, Component Analysis, Finalizing the architecture and design strategies, Experiments and Evaluation, Design and Demonstration and the Conclusion of the project as the final stage to complete the project. The following table shows the work breakdown structure that used for this project.

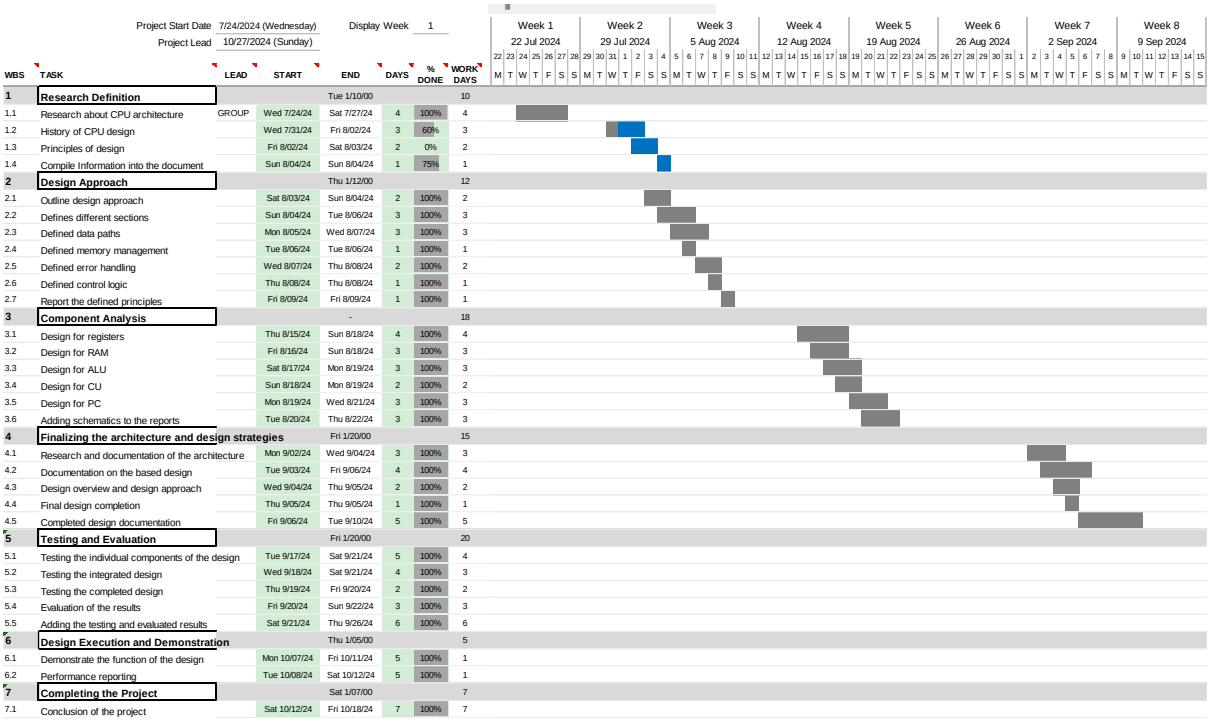
Table 1 - Work Breakdown Structure

WBS Level	Task	Duration (Days)	Start Date	Finish Date
1	**Research Definition**	**10**	2024-07-24 00:00:00	2024-08-02 00:00:00
1.1	Research about CPU architecture	4		
1.2	History of CPU design	3		
1.3	Principles of design	2		
1.4	Compile Information into the document	1		
2	**Design Approach**	**12**	2024-08-03 00:00:00	2024-08-14 00:00:00
2.1	Outline design approach	2		
2.2	Define different sections	2		
2.3	Defined data paths	3		
2.4	Defined memory management	1		
2.5	Defined error handling	2		
2.6	Defined control logic	1		
2.7	Report the defined principles	1		
3	**Component Analysis**	**18**	2024-08-15 00:00:00	2024-09-01 00:00:00
3.1	Design for registers	4		
3.2	Design for RAM	3		
3.3	Design for ALU	3		
3.4	Design for CU	2		
3.5	Design for PC	3		
3.6	Adding schematics to the reports	3		
4	**Finalizing the architecture and design strategies**	**15**	2024-09-02 00:00:00	2024-09-16 00:00:00
4.1	Research and documentation of the architecture	3		
4.2	Documentation on the based design	4		
4.3	Design overview and design approach	2		
4.4	Final design completion	1		

4.5	Completed design documentation	5		
5	**Experiments and Evaluation**	**20**	2024-09-17 00:00:00	2024-10-06 00:00:00
5.1	Testing the individual components of the design	5		
5.2	Testing the integrated design	4		
5.3	Testing the completed design	2		
5.4	Evaluation of the results	3		
5.5	Adding the testing and evaluated results	6		
6	**Design and Demonstration**	**5**	2024-10-07 00:00:00	2024-10-11 00:00:00
6.1	Demonstrate the function of the design	5		
6.2	Performance reporting	5		
7	**Conclusion of the Project**	**7**	2024-10-12 00:00:00	2024-10-27 00:00:00
7.1	Conclusion of the project	7		

Gantt Chart

The following Figure 15 shows the Gantt Chart that used for the design and development of this project.



Project Start Date 7/24/2024 (Wednesday) — Display Week 1
Project Lead 10/27/2024 (Sunday)

WBS	TASK	LEAD	START	END	DAYS	% DONE	WORK DAYS
1	**Research Definition**			Tue 1/10/00			10
1.1	Research about CPU architecture	GROUP	Wed 7/24/24	Sat 7/27/24	4	100%	4
1.2	History of CPU design		Wed 7/31/24	Fri 8/02/24	3	60%	3
1.3	Principles of design		Fri 8/02/24	Sat 8/03/24	2	0%	2
1.4	Compile Information into the document		Sun 8/04/24	Sun 8/04/24	1	75%	1
2	**Design Approach**			Thu 1/12/00			12
2.1	Outline design approach		Sat 8/03/24	Sun 8/04/24	2	100%	2
2.2	Defines different sections		Sun 8/04/24	Tue 8/06/24	3	100%	3
2.3	Defined data paths		Mon 8/05/24	Wed 8/07/24	3	100%	3
2.4	Defined memory management		Tue 8/06/24	Tue 8/06/24	1	100%	1
2.5	Defined error handling		Wed 8/07/24	Thu 8/08/24	2	100%	2
2.6	Defined control logic		Thu 8/08/24	Thu 8/08/24	1	100%	1
2.7	Report the defined principles		Fri 8/09/24	Fri 8/09/24	1	100%	1
3	**Component Analysis**			-			18
3.1	Design for registers		Thu 8/15/24	Sun 8/18/24	4	100%	4
3.2	Design for RAM		Fri 8/16/24	Sun 8/18/24	3	100%	3
3.3	Design for ALU		Sat 8/17/24	Mon 8/19/24	3	100%	3
3.4	Design for CU		Sun 8/18/24	Mon 8/19/24	2	100%	2
3.5	Design for PC		Mon 8/19/24	Wed 8/21/24	3	100%	3
3.6	Adding schematics to the reports		Tue 8/20/24	Thu 8/22/24	3	100%	3
4	**Finalizing the architecture and design strategies**			Fri 1/20/00			15
4.1	Research and documentation of the architecture		Mon 9/02/24	Wed 9/04/24	3	100%	3
4.2	Documentation on the based design		Tue 9/03/24	Fri 9/06/24	4	100%	4
4.3	Design overview and design approach		Wed 9/04/24	Thu 9/05/24	2	100%	2
4.4	Final design completion		Thu 9/05/24	Thu 9/05/24	1	100%	1
4.5	Completed design documentation		Fri 9/06/24	Tue 9/10/24	5	100%	5
5	**Testing and Evaluation**			Fri 1/20/00			20
5.1	Testing the individual components of the design		Tue 9/17/24	Sat 9/21/24	5	100%	4
5.2	Testing the integrated design		Wed 9/18/24	Sat 9/21/24	4	100%	3
5.3	Testing the completed design		Thu 9/19/24	Fri 9/20/24	2	100%	2
5.4	Evaluation of the results		Fri 9/20/24	Sun 9/22/24	3	100%	3
5.5	Adding the testing and evaluated results		Sat 9/21/24	Thu 9/26/24	6	100%	6
6	**Design Execution and Demonstration**			Thu 1/05/00			5
6.1	Demonstrate the function of the design		Mon 10/07/24	Fri 10/11/24	5	100%	1
6.2	Performance reporting		Tue 10/08/24	Sat 10/12/24	5	100%	1
7	**Completing the Project**			Sat 1/07/00			7
7.1	Conclusion of the project		Sat 10/12/24	Fri 10/18/24	7	100%	7

Figure 15 - Gantt Chart

SOFTWARE AND HARDWARE

DE0 OVERVIEW

The DE0 board is a development platform that utilizes the Altera Cyclone III FPGA (EP3C16F484). It is designed for digital logic design and embedded systems education. The board provides various components and interfaces that can be used to implement different digital systems.

Key Features

Cyclone III 3C16 FPGA with,

- 15,408 Logic Elements (LEs)
- 56 M9K Embedded Memory Blocks (504K total RAM bits)
- 56 embedded multipliers
- 4 Phase-Locked Loops (PLLs)
- 346 user I/O pins

Memory Modules

SDRAM

- 8 MB single data rate Synchronous Dynamic RAM (SDRAM)
- Supports a 16-bit data bus

Flash Memory

- 4 MB NOR Flash memory
- Supports Byte (8-bit) and Word (16-bit) modes

SD Card Socket

- Supports both SPI and SD 1-bit mode for accessing SD cards

I/O Interfaces

The DE0 board is equipped with a wide range of input and output interfaces, allowing for versatile applications.

User Interfaces:

- Pushbutton switches: 3 pushbutton switches that provide one active-low pulse when pressed.
- Slide switches: 10 toggle switches that provide logic 0 when in the DOWN position and logic 1 when in the UP position.
- LEDs: 10 green user-controllable LEDs.
- 7-segment Displays: 4 seven-segment displays used for numeric output.
- LCD Module Interface: 16x2 character display interface (LCD module not included).

VGA Output:

- Supports VGA resolution of up to 1280x1024 at 60 Hz refresh rate.
- Uses a 4-bit resistor-network DAC for red, green, and blue color signals.

RS-232 Serial Port: For serial communication, though it does not include a DB-9 connector.
PS/2 Port: Allows connection to a PS/2 mouse or keyboard.

Expansion Headers:

- Two 40-pin expansion headers provide 72 I/O pins to the FPGA, allowing for additional peripherals and custom circuits.

Clock Sources

The DE0 board provides the following clock sources for use with the FPGA:

- 50 MHz Oscillator: This clock signal is connected to the FPGA and can be used for clocking user logic and Phase-Locked Loop (PLL) circuits.

This setup allows the DE0 board to function as a versatile development tool for a variety of digital design and embedded system applications.

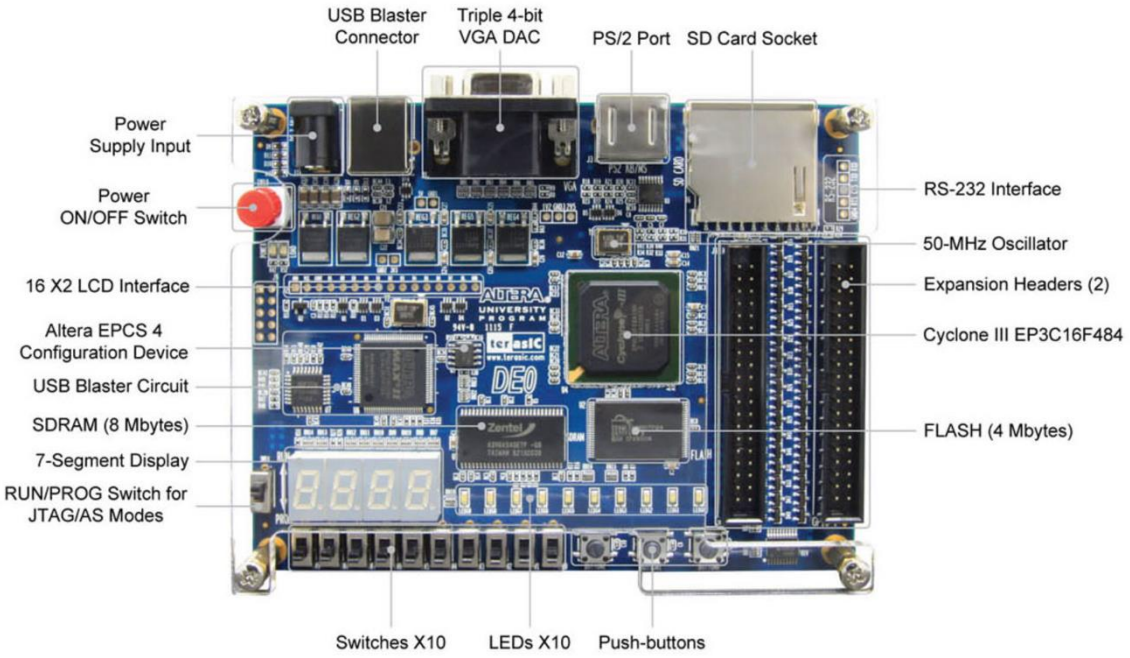


Figure 16 - Layout and Components

The following hardware is provided on the DEO board:

- Altera Cyclone® III 3C16 FPGA device
- Altera Serial Configuration device – EPCS4
- USB Blaster (on board) for programming and user API control; both JTAG and Active Serial
- (AS) programming modes are supported
- 8-Mbyte SDRAM
- 4-Mbyte Flash memory
- SD Card socket
- 3 pushbutton switches
- 10 toggle switches
- 10 green user LEDs
- 50-MHz oscillator for clock sources
- VGA DAC (4-bit resistor network) with VGA-out connector

- RS-232 transceiver
- PS/2 mouse/keyboard connector
- Two 40-pin Expansion Headers

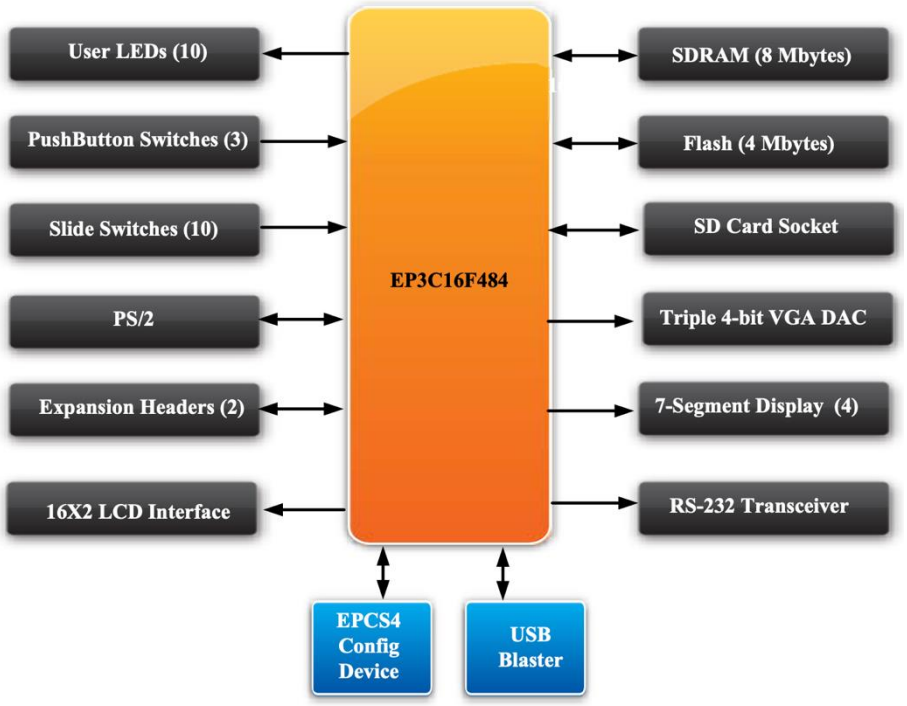


Figure 17 - Block diagram of the DE0 board

SOFTWARE OVERVIEW

Quartus II

For this project ‘Quartus II 12.0sp2 Web Edition (32-Bit)’ software was used to develop the VHDL codes for each component of the CPU and to design the schematic overview of the implementation of each approach. For approach 1 design, Altera’s DE0 board was used along with the Cyclone III family as it was the available version of the hardware at the laboratory to physical implementations. For approach 2 design, DE2-70 board was used along with Cyclone II family. All the software wise experiments were done during each approach to ensure the compatibility of the design performs well with the Quartus II 12.0sp2 Web Edition that used for the implementations.

Originally the Quartus II software was built and developed by Altera although it is a part of intel as of today. Quartus II is a systematic and comprehensive design software tool that uses for design, simulation and digital circuit verification purposes of FPGA (Field Programmable Gate Array) designs and CPLD (Complex Programmable Logic Devices) designs. This software is capable of facilitating the design options from the initial design sketches to advanced integrated circuit designs that are based on FPGA and CPLD concepts. Following Figure 18 shows the interface of Quartus II software.

Figure 18 - Quartus II interface

Logisim

Logisim is a complete software package designed to create and simulate digital logic circuits. It provides the user with a computational interface. It allows the user to build circuits by placing components such as logic gates, flip-flops, multiplexers, and many more. And to interface with errors, Logisim is widely used. Widely used in educational settings to teach the basics of digital memory and computer architecture. This is due to its user-friendly interface and real-time simulation capabilities. Support for creating arrays This allows users to create complex arrays by combining simpler arrays. It is also possible to simulate timed arrays. This makes it a versatile tool for beginners and advanced students alike. Following figure 19 shows the interface of Logisim software.

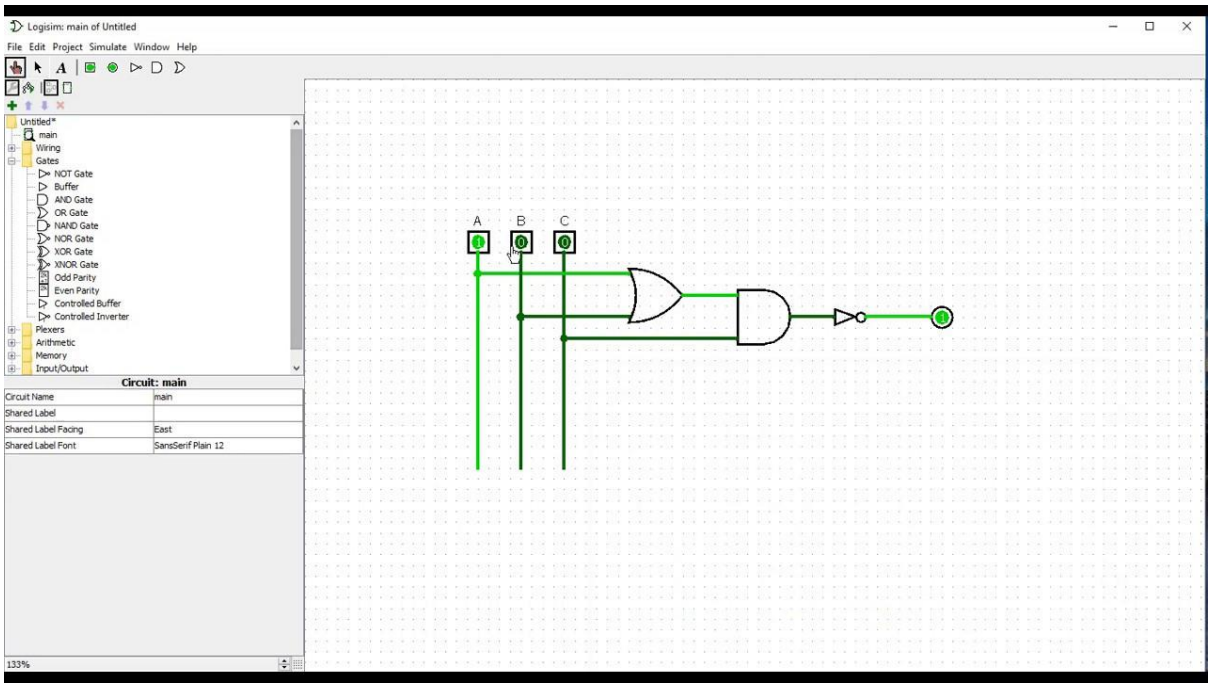


Figure 19 - Logisim software interface

CPU PERIPHERALS

Registers

Registers serve as fundamental components of a CPU, which facilitates computational steps by providing temporary storage to hold the data, addresses and computational instructions. The primary objective of registers is to act as small memory cells in the CPU to ensure that the data flow of the operations maintain their desired high speed, such that the consistency of operations and computational steps are well balanced and organized. [12] Following parts of this section provides a comprehensive overview of the registers and their roles inside of a CPU.

Primary roles of Registers

- Data Storage – Holds the operands of an operation to perform the computations done in ALU
- Control Information Storage – Holds the status flags and control information that governs by the control unit to facilitate the data processing and manipulations in CPU
- Instruction Storage – Holds the instructions that uses for command executions as well as decoding processes
- Address Storage – Holds the memory addresses of each executional segment on current operation to facilitate the coordination for the proper dataflow

Types of CPU registers

Based on the functionality and the usage, the registers in CPU can be divided into several categories as follows,

1. Data Registers
    a. General Purpose Registers (GPR) – A type of adaptable registers that holds the intermediate results as well as the operands of the current operation
    b. Input-Output Registers (I/O_R) – These registers are used to transfer the final results and incoming information/data inside the CPU between each of the connected peripherals.

    c. Accumulator Register (ACC) – These registers are used to hold the temporary results of arithmetic and logic operations done by the ALU

2. Address Registers
    a. Program Counter (PC) – This register holds the address of next instruction of current sequency to be executed in the current operation.
    b. Memory Address Register (MAR) – This register holds the addresses of memory locations to access data for both memory read, and memory write options.
    c. Index Registers (IR) – These registers are used in array processing computational steps or loop constructions to hold the address values of each index of the loop or array list
    d. Stack Pointer (SP) - This register is used to manage and coordinate functions by pointing to the top of the stack in currently loaded memory.
3. Control and Status Registers
    a. Status Register (SR) – This register also known as the 'Flag Register' of CPU. The primary objective of this register is to falg the operational results as zero (Z), carry (C), overflow (V) or sign (S) for the current operation. These flags are used to branching and decision-making steps for the currently executing operation to reach to the desired result
    b. Instruction Register (IR) – This register holds the currently executing instruction
4. Special Purpose Registers (SPR)
    a. Base Register – The primary objective of this register is to hold the starting address of a memory sequence that needs to be executed for current operation. This register is very useful in segmented memory archicture.
    b. Segment Registers – This register used to support the memory segmentation processes where it holds the base addresses of each segmented memory portion, that enables the CPU to handle complex computations.
    c. Floating Point Registers (FPR) – These registers are used to store floating point values for intermediate calculations done in the currently executing computation sequences.

While addressing the registers it is important to discuss about the building blocks of these components as well. Latches and flip-flops are the primary building blocks of CPU registers as

well as most other digital electronic components. These two fundamental units are based on binary bit-wise operations.

Both the Latches and Flip-flops are single bit storing elements which have the nature of sequential logic circuits. They only show the difference between them based on the previous inputs and outputs, but not the current status of the inputs, where flip-flops use a clock signal to trigger its status while the latch does not use any clock signal. The following sections are discussing different types of flip-flops and latches that supports to the development of above-mentioned registers.

1. Latches
    a. S-R (Set-Reset) Latch

SR latch is made using NAND or NOR gates and has only two inputs. ‘S’ input stands as the ‘Set’ and ‘R’ input stands as the ‘Reset’ in the input side of the latch. At the output side, Q and Q’ are performed as the output variables. Following Figure 20 shows the schematic build of S-R latch made in Logisim software.

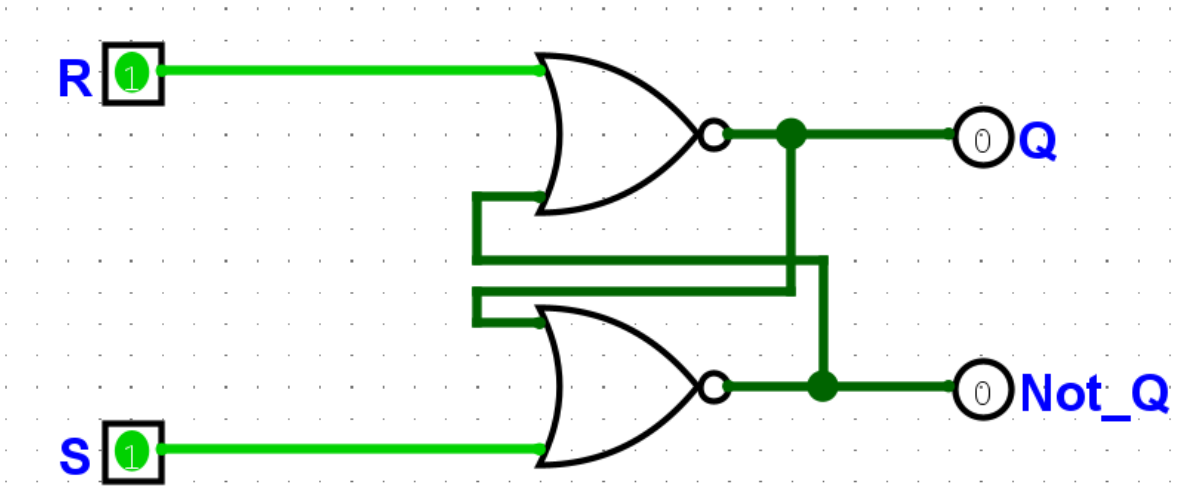


Figure 20 - S-R Latch

The following table shows the truth table for the SR latch were the 0-0 state is not defined (ND) by default. Here, when S is set as 1, the ‘Set’ operation is initiate and when R is set as 1, the Reset operation initiates.

Table 2 - SR latch Truth Table

S	R	Q	Q’
0	0	ND	ND
0	1	1	0
1	0	0	1
1	1	0	0

b. D Latch

This is also known as the ‘Data Latch’ or the ‘Transparent Latch’ and has only one input (D) and one Enable pin, which performs as the secondary operator. The following Figure 21 shows the schematic build of D latch made in Logisim software.

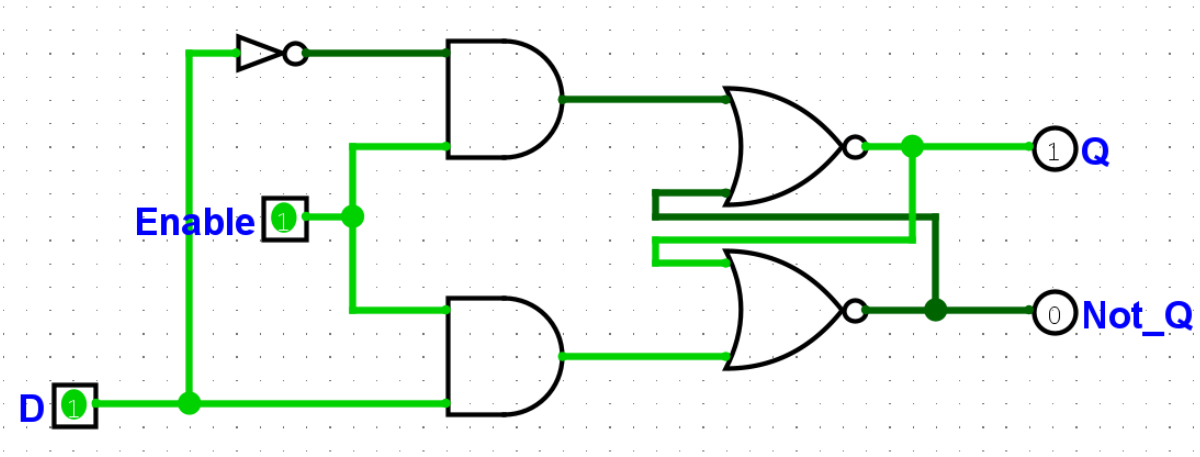


Figure 21 - D Latch

The following table 3 shows the truth table of a D latch.

Table 3 - D latch Truth Table

D	Enable	Q	Q’
0	0	ND	ND
0	1	ND	ND
1	0	0	1
1	1	1	0

2. Flip flops

    a. S-R Flip flop

The S-R flip flop consists of two inputs as S-R latch and two outputs as Q and Q' as in S-R latch. The following Figure 22 shows the schematic build of S-R flipflop made in Logisim software.

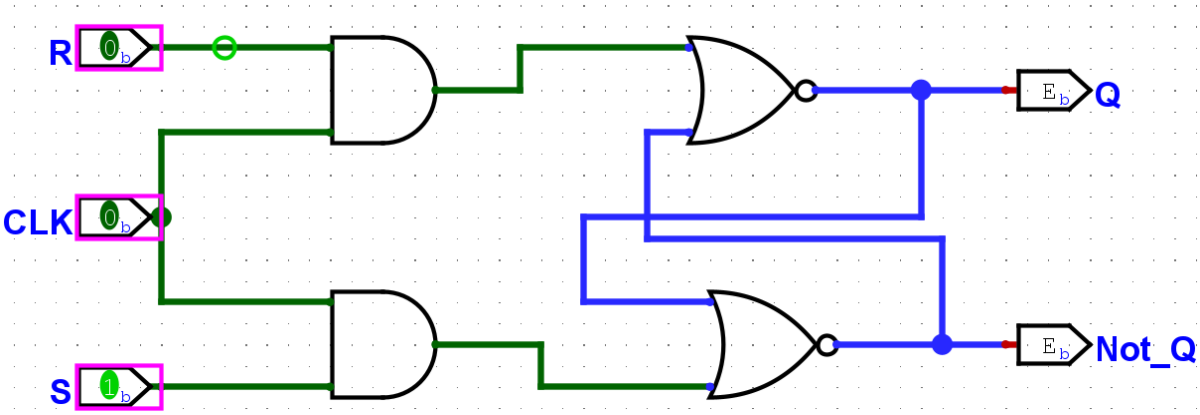


Figure 22 - SR Flipflop

The following table 4 shows the truth table of SR flip flop, where the output state of Q and Q' becomes invalid when both Set and Reset have set as 1.

Table 4 - SR flipflop Truth Table

S	R	Qn+1	State
0	0	Qn	Holds
0	1	0	Reset
1	0	1	Set
1	1	x	Invalid

    b. D – Flip flop

D flip flop also known as the 'delay flip flop' or the 'data flip flop' as it use for store the data bit-by-bit order. Although the D flip flops are as synchronous for most of the time, the asynchronous types also in use for different purposes. The following Figure 23 shows the schematic build of D flipflop made in Logisim software.

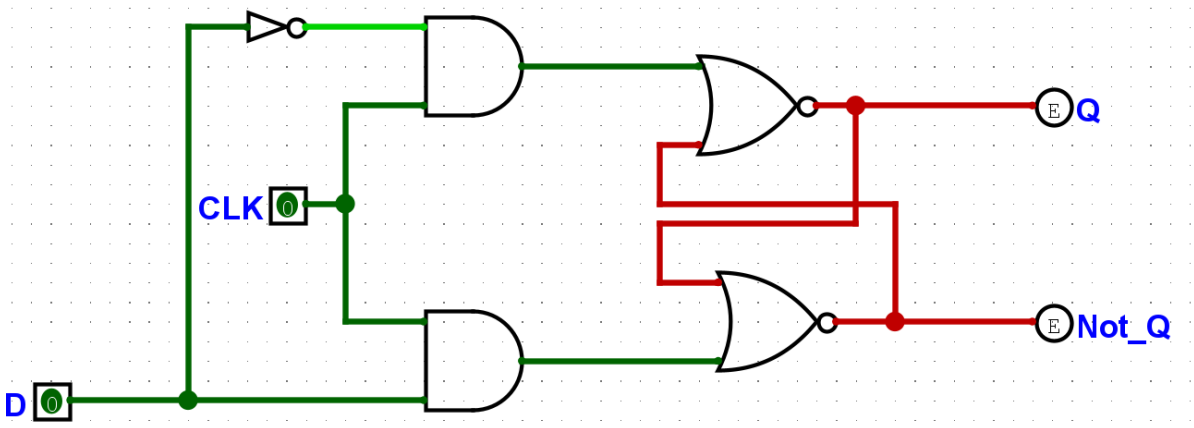


Figure 23 - D flip flop

The following table 5 shows the truth table of D flip flop.

Table 5 - D flip flop truth table

D	Q	Qn+1
0	0	0
0	1	0
1	0	1
1	1	1

The following section discusses the 16-bit register development starting from 1-bit register

1-Bit Register

The ground concept of D-flip flop can be used to design and illustrate the 1-bit register. As shown in the following figure, with a D-flip flop, an additional pin can be used as the 'LOAD' pin to store or write the bit values for the current operation. Pin P is configured as the actual data input. Here the LOAD pin acts as the controller for input to determine whether the input should be loaded into flipflop or not.

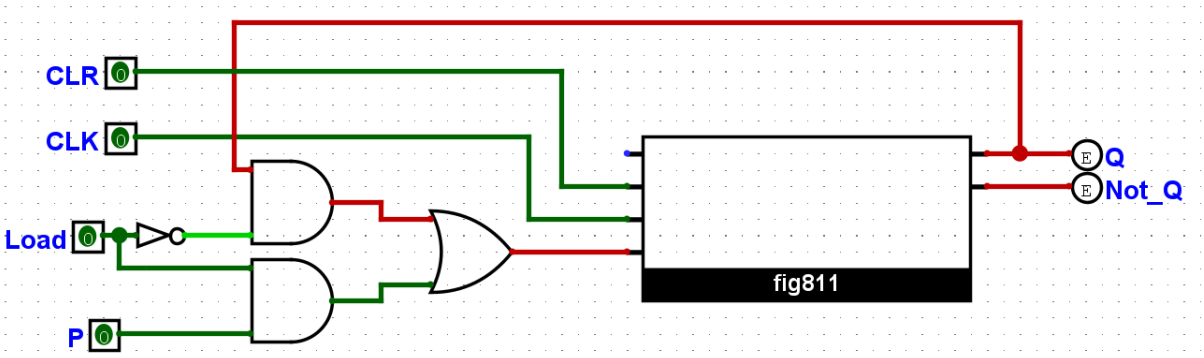


Figure 24 - Schematic for 1 bit register

4-Bit Register

The following figure shows the development of 4-Bit Register using 4 of 1 Bit Registers. Here the LOAD, CLK and CLR pins are commonly used for all registers to ensure that the registers are synchronized.

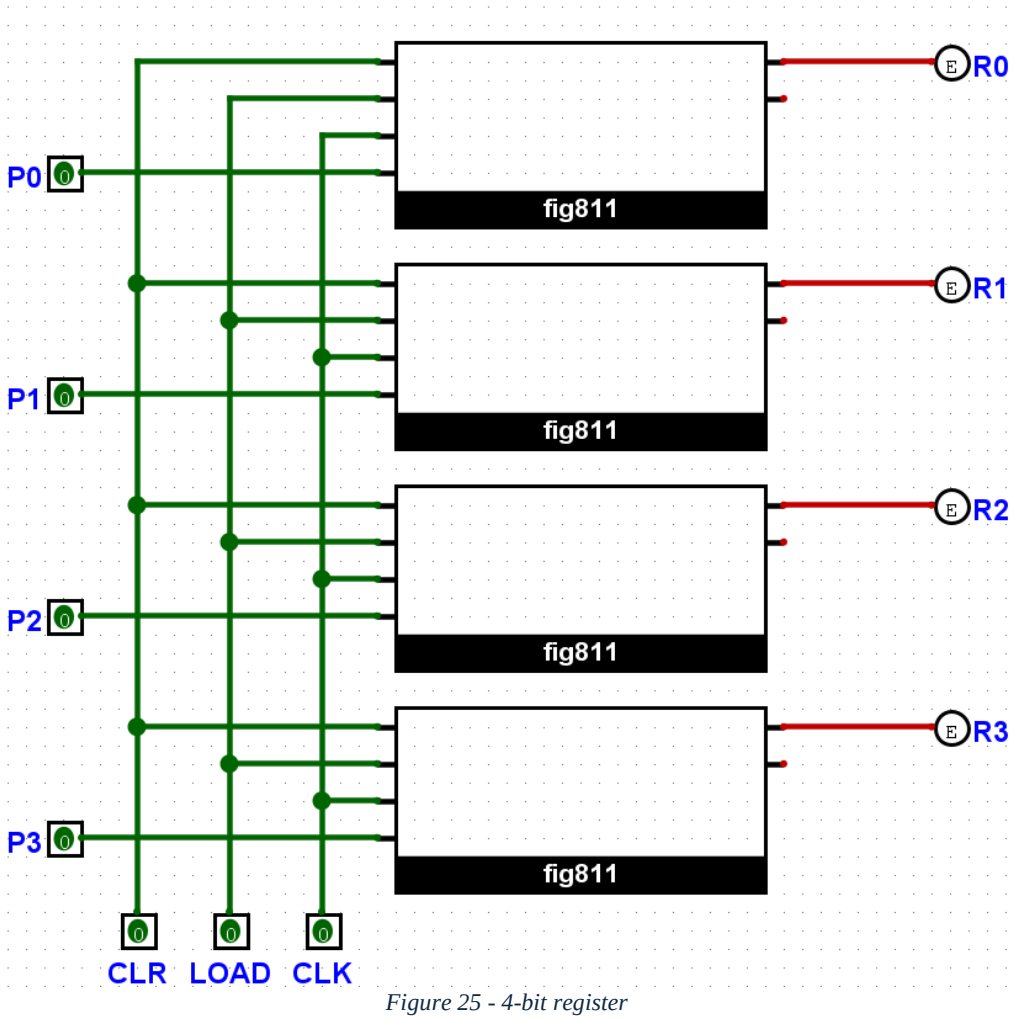


Figure 25 - 4-bit register

Since the design is made for use of buses, there should be a buffer to control the read and write commands for each of these registers as they are commonly connected to one single bus. Following figure 26 shows the schematic of the 4-Bit buffer that is used to control the 4-bit register and the figure 27 shows the schematic of the 4-bit buffered register combining both the buffer and the registers.

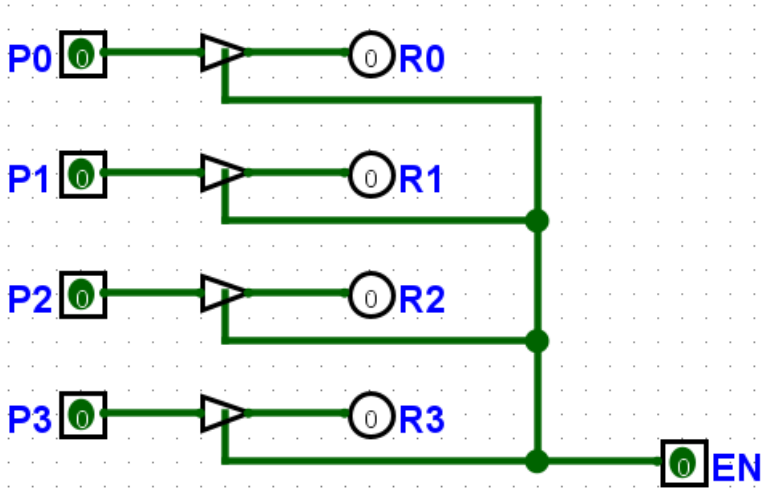


Figure 26 - 4 bit buffer

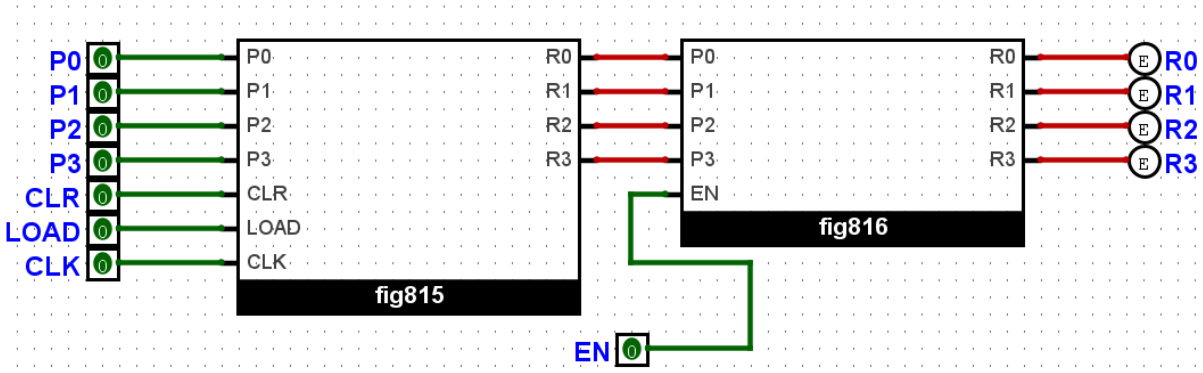


Figure 27 - 4 bit buffered register

Arithmetic Logic Unit (ALU)

All the arithmetic and logical operations of a CPU is performed by the Arithmetic Logic Unit. This unit is responsible for handling all the basic arithmetic operations such as addition, subtraction, multiplication as well as logic operations such as the AND, OR and shift operations. Also, it performs the data manipulation for CPU which can convert the input data and processed information into meaningful results as the system outputs. Also, ALU often uses registers for the data manipulation and performing operations for both data from and to the registers. [13]

ALU primarily consists of two main components called Arithmetic Unit which performs the arithmetic operations and the Logic Unit which performs the bitwise logical operations. Additionally, the ALU consists of Comparator to compare values and determine the equality or the state of inequality between both values, Shifter and Rotator to perform bitwise shifting and rotating of individual bits of multi bit inputs, and the Status flags which state the current status of the outcomes of the computation result.

The arithmetic operations in ALU are performed by a combination of set of adders and several other primary elements. The following sections discuss the development of an AU.

Half Adder

Half adder is the most primitive element of the Arithmetic Unit. The following figure and table shows the schematic overview of the Half adder and the truth table of the half adder, respectively.

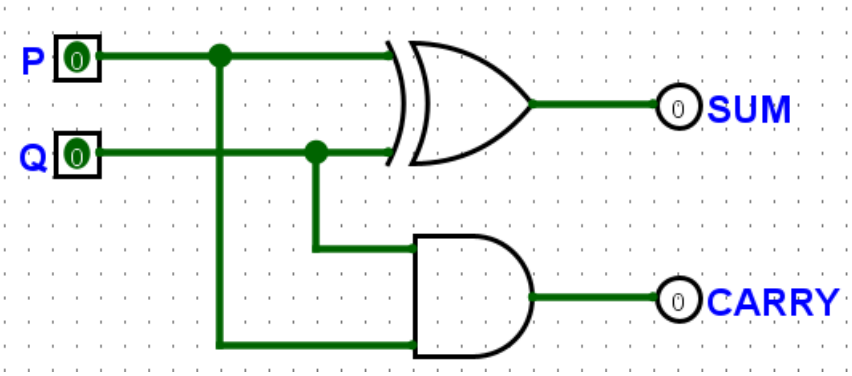


Figure 28 - Half Adder

Table 6 - Half Adder Truth Table

P	Q	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

To perform multi bit subtraction and addition operations a Full Adder can be developed using the concept of Half Adder as shown in the following figure. In Full Adder, the carry bit from previous operation is carried forward to the next (CR)

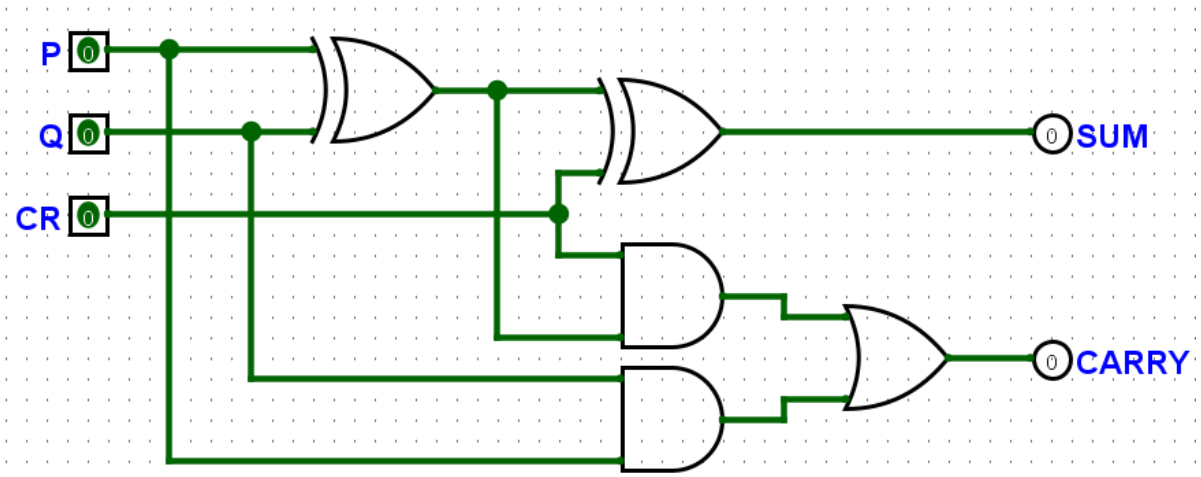


Figure 29 - Full Adder

Following table shows the truth table for the Full Adder.

Table 7 - Full Adder Truth Table

P	Q	CR	SUM	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1

1	1	0	0	1
1	1	1	1	1

Using four of Full adders, the 4-bit full adder can be designed as follows,

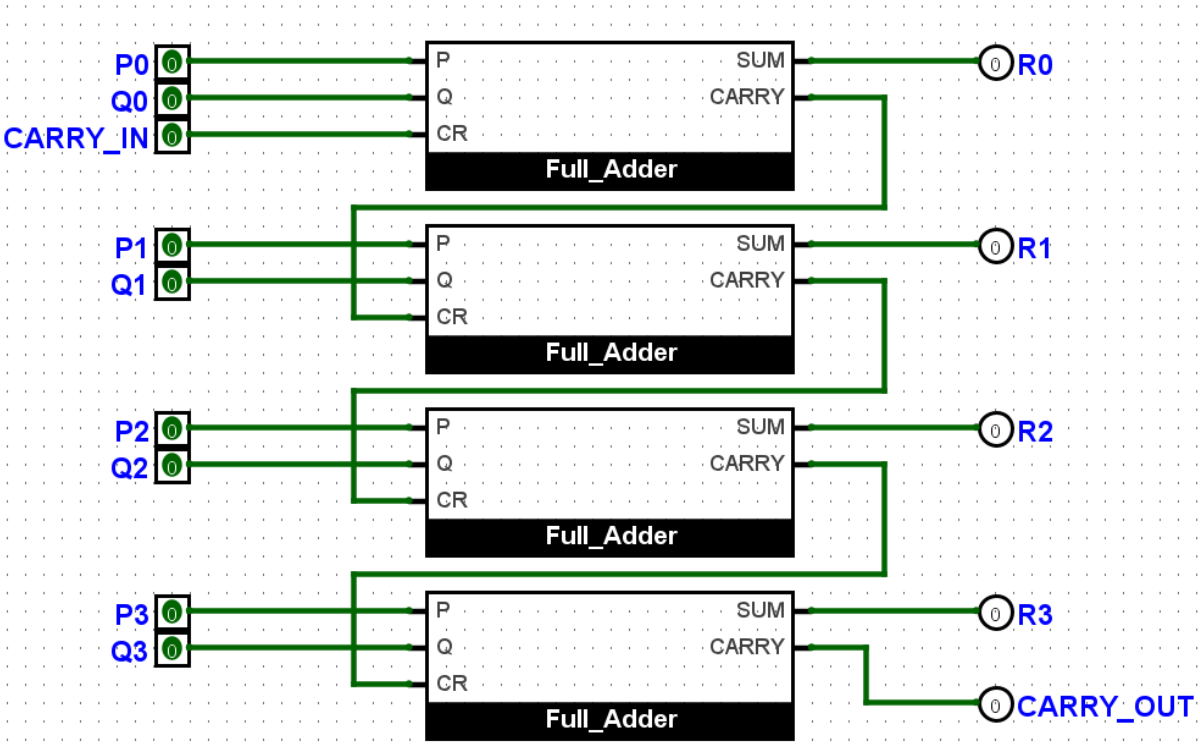


Figure 30 - 4 Bit Full Adder

1-Bit Arithmetic Unit

Following schematic shows the 1-Bit Arithmetic Unit designed using the Logisim software. The SW1 and SW2 switches are set as the selection bits to determine which arithmetic operation should be performed.

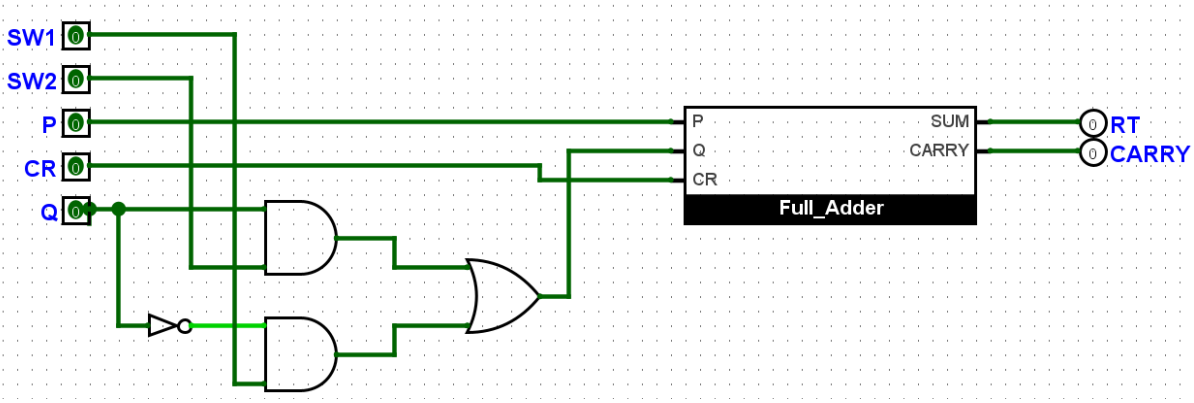


Figure 31 - 1-bit Arithmetic Unit

The following table shows the Operational Truth Table for the 1-Bit Arithmetic Unit

Table 8 - 1-Bit AU Truth Table

Selection		Input	G = P + Input + CR	
SW1	SW2		CR = 0	CR = 1
0	0	0	RT=P	RT = P +1
0	1	B	RT=P+Q	RT = P +Q +1
1	0	$\bar{B}$	RT=P+$\bar{Q}$	RT = P+$\bar{Q}$ +1
1	1	1	RT=P -1	RT = P

4-Bit Arithmetic Unit

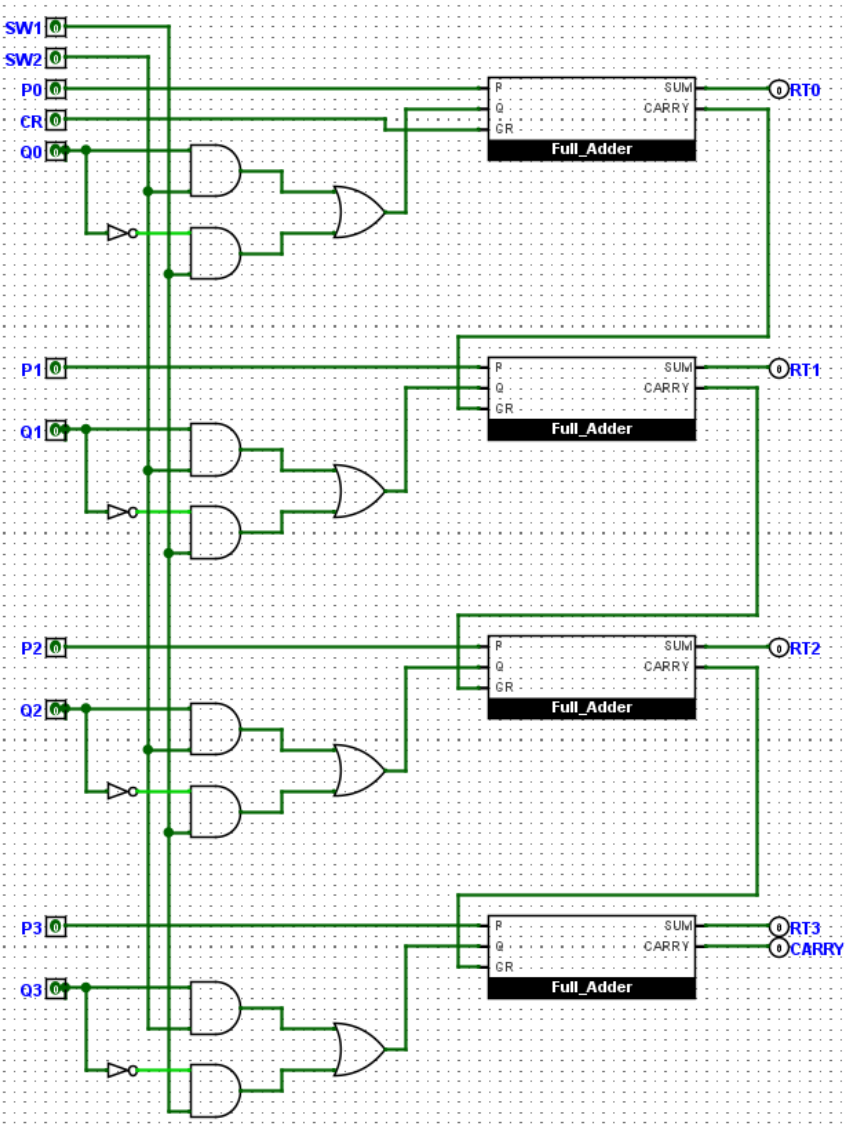


Figure 32 - 4 Bit AU

Logic Unit (LU)

To perform the basic Logic operations of OR, AND, NOT and XOR, a 4 to1 MUX can be developed as in the following figure as the fundamental unit for the Logic Unit. The selection bits SW1 and SW2 decides the operation to be performed.

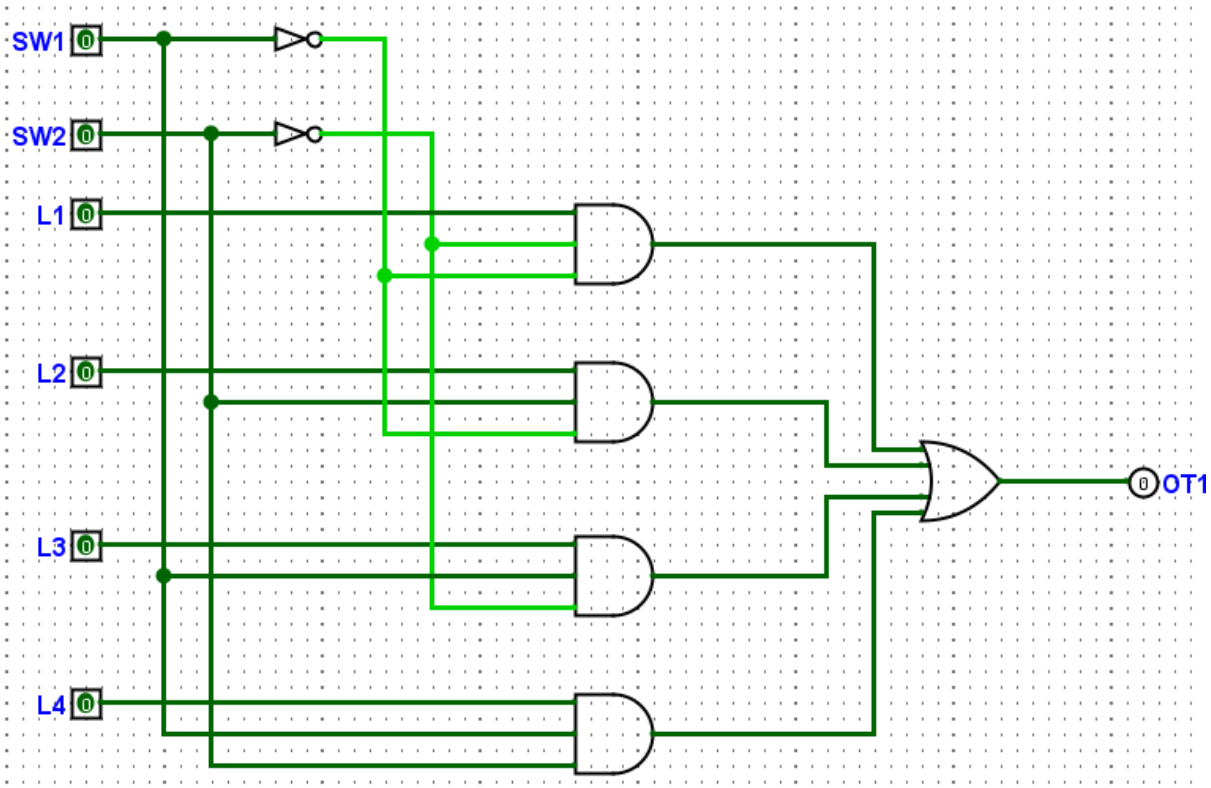


Figure 33 - 4 to 1 MUX

1-Bit Logic Unit

The following figure shows the schematic for 1 bit LU, where the SW1 and SW2 are performing the logic selection as, if both SW1 and SW2 are 0, then the OR operation is performed; if SW1 is 0 and SW2 is 1 then the AND operation is performed; if SW1 is 1 and SW2 is 0 then the XOR operation is performed; and finally if both SW1 and SW2 are set as 1 then the NOT operation is performed.

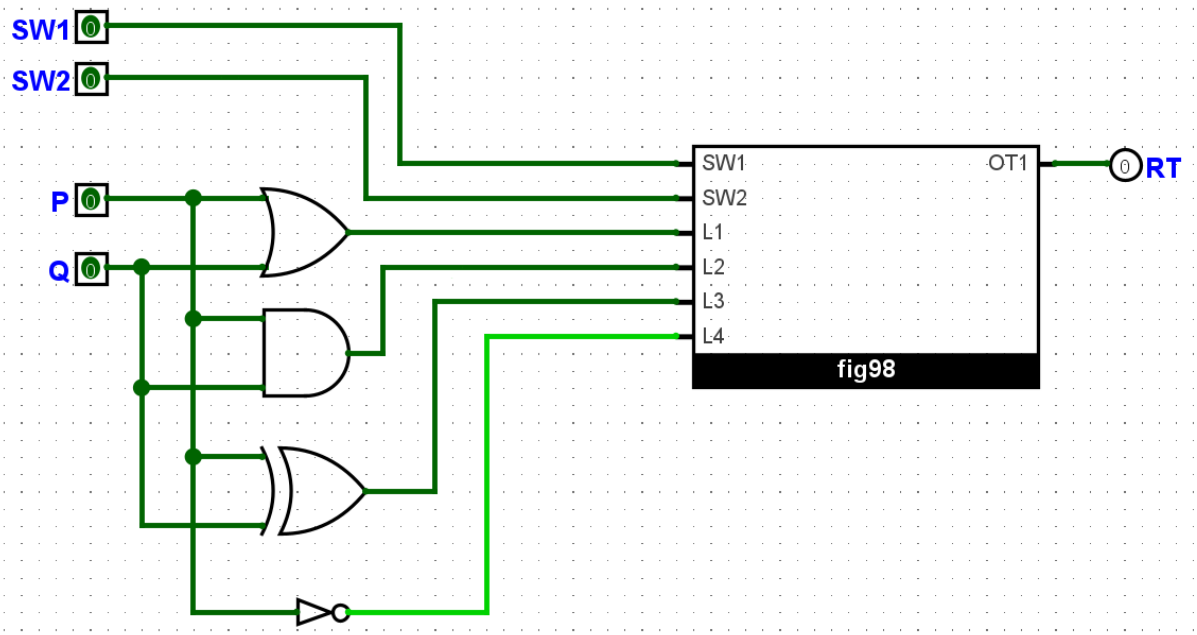


Figure 34 - 1 bit Logic Unit

4-Bit Logic Unit

Using four of 1 bit Logic Units, a 4-Bit Logic Unit can be developed as in the following figure.

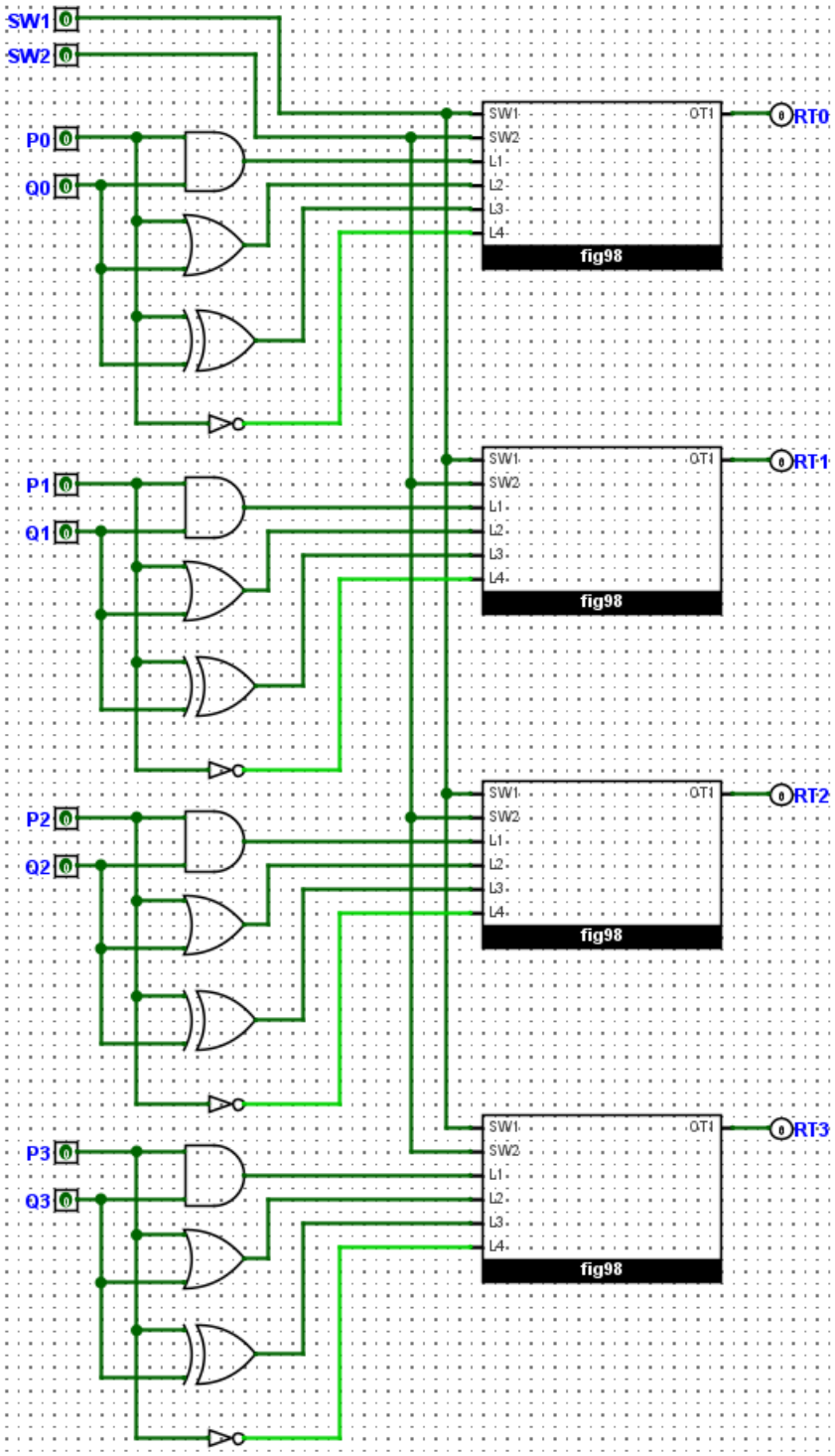


Figure 35 - 4 bit Logic Unit

ALU Development

Using 2 to 1 MUX the AU and LU can be combined together to perform as an ALU as shown in the following figures.

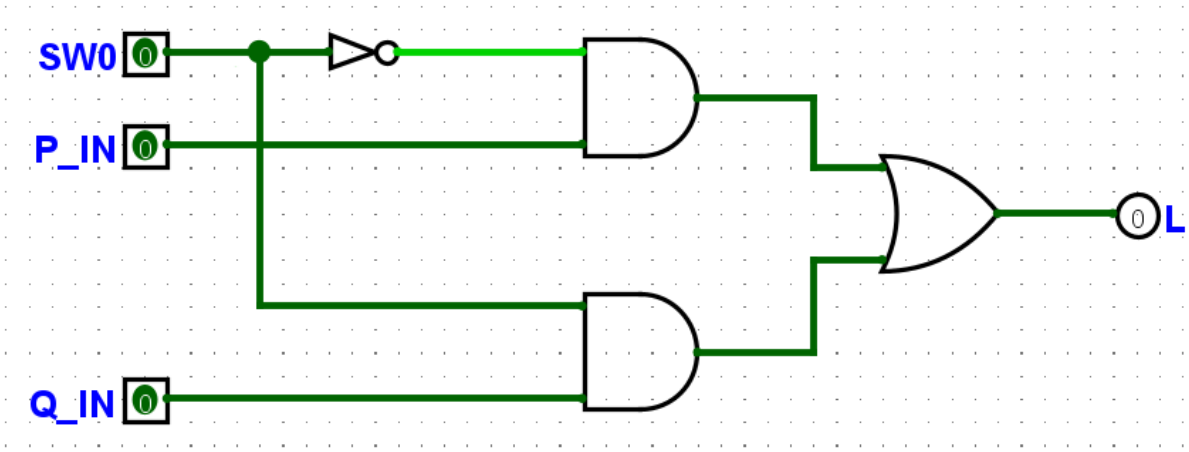


Figure 36 - 2 to 1 MUX

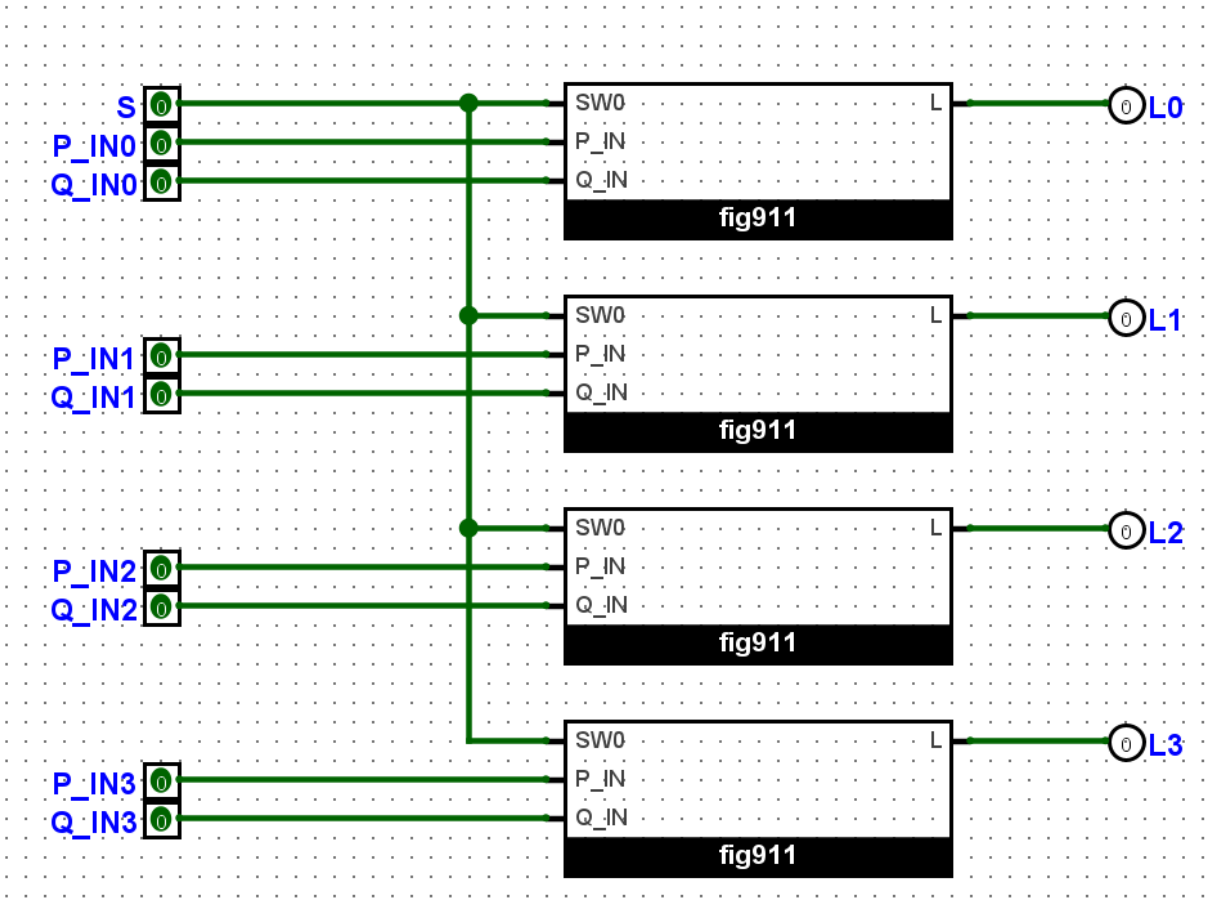


Figure 37 - 4 bit 2 to 1 MUX

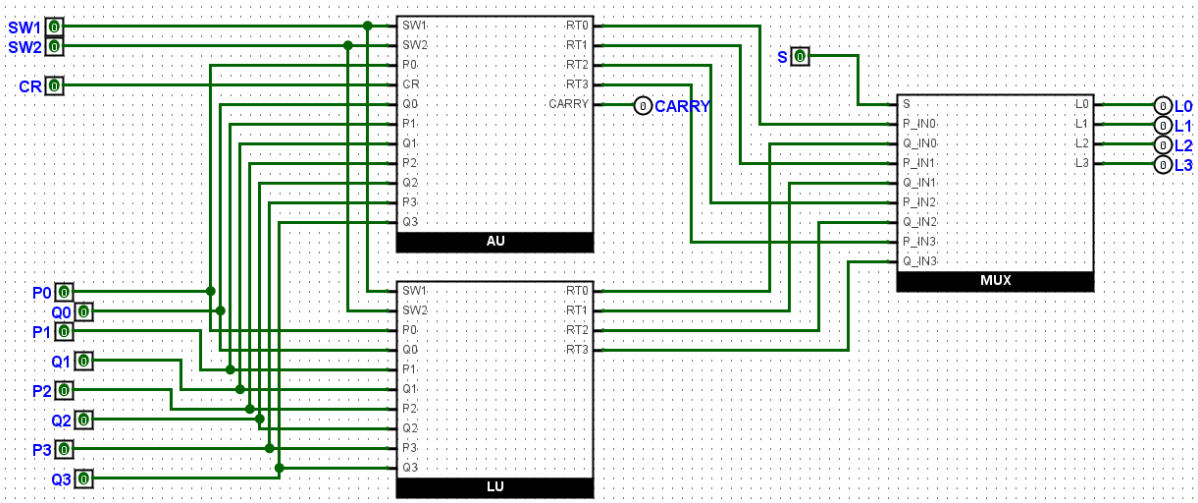


Figure 38 - 4 bit ALU

Using a 2 to 1 MUX as shown in Figure 36 a four bit 2 to 1 MUX can be designed as in Figure 37, then it can be used to combine the AU and LU together to create 4-bit ALU as shown in Figure 38.

Following Figure 39 shows the schematic build of 16-bit ALU using 4 of 4 bit ALUs in Logisim Software.

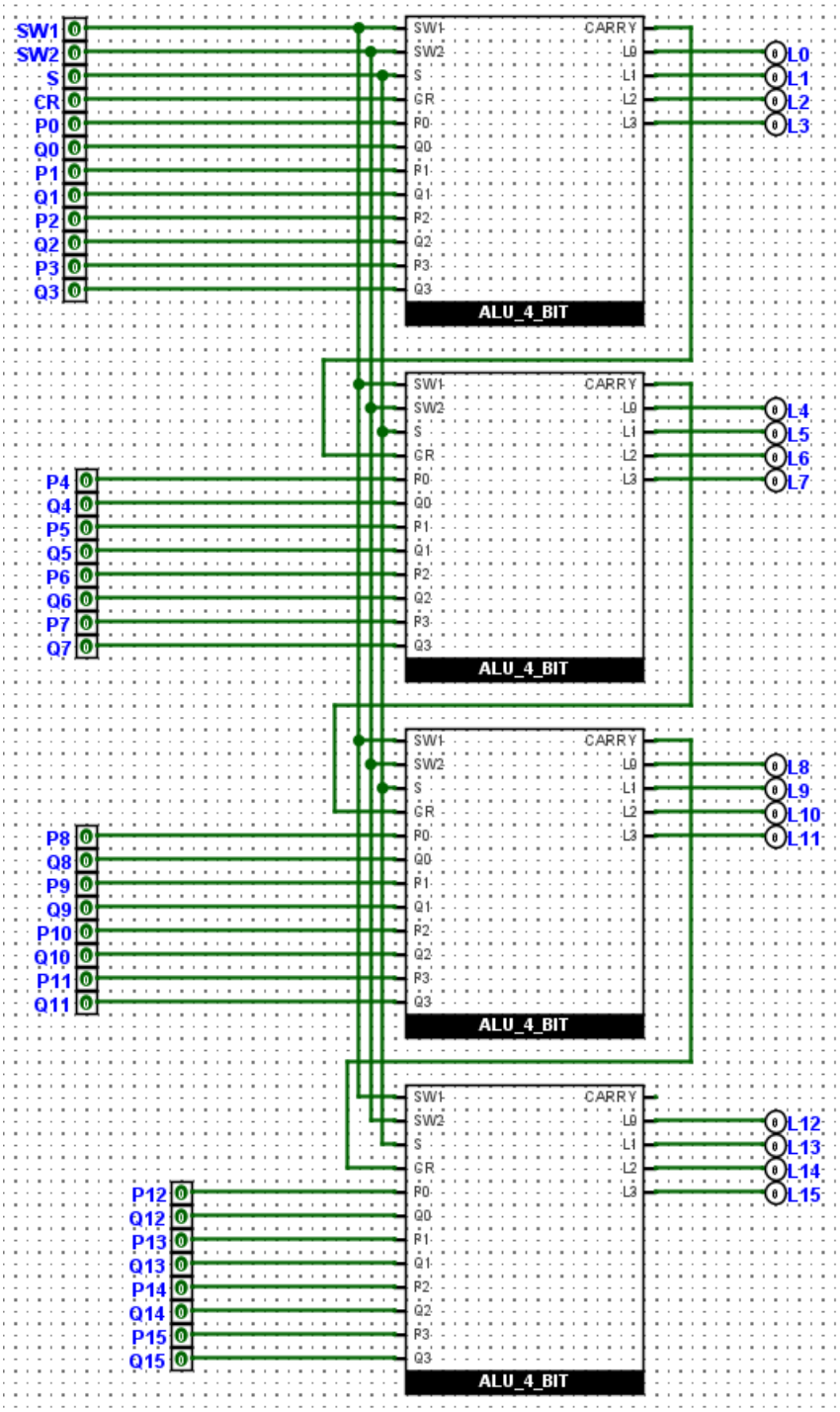


Figure 39 - 16 bit ALU

Random Access Memory (RAM)

Random Access Memory is the primary memory unit that is used by the CPU to store the data and instructions for short periods of time. Among the many purposes of the RAM in CPU environment, facilitating for faster data access, storing the data temporarily for short periods of time, performing the temporary calculations, and facilitating the CPU to perform multiple tasks simultaneously while allocating separate storages for data using for each task, can be identified as the most important deliverables of RAM. Dynamic Accessibility of data is another major function of RAM [14]

There are Different types of RAMs available in the market and following list introduces a few of them.

1. SRAM – Static RAM

   This type RAMs uses flip flops to store the data which would make the stability of the archicture higher than any other RAM. At the same time due to allocating storage for all the data, these are faster than most of other RAM types. SRAMs are mostly used in cache memory in today's computer to store frequently accessible data.

2. DRAM – Dynamic RAM

   These types of RAMs use a combination of transistors and capacitors to store each data bit, which leads to refreshing the memory space over fix periods to maintain the charge and the allocatable space. DRAMs are slower than SRAMs due to this reason, however they are used as main RAM in many devices due to its high density and economical state.

3. SDRAM – Synchronous DRAM

   This is a subdivision of DRAMS where the CPU clock is used to synchronize the refresh rates. Therefore, the performance in these RAMs are better than the classic DRAMs, Thus these types of RAMs are mostly used in personal computers as their high suitability for moderate complexity performance.

4. VRAM – Video RAM

These types of RAMs are specifically designed for Graphical performance to facilitate the high processing demands on graphical computations.

Following sections discusses about the development of RAM in Logisim Software.

2 to 4 Line Decoder

The 2 to 4 Line Decoder is used to take the input information from the input side and decode the information into 4 bits in 'bitwise' manner and send them through different paths to assign the bitwise data to designated registers that is responsible for holding that particular input information. The following figure shows a simple 2 to 4 Line Decoder which takes a two bit binary integer and converts it to the format that the registers can store the information.

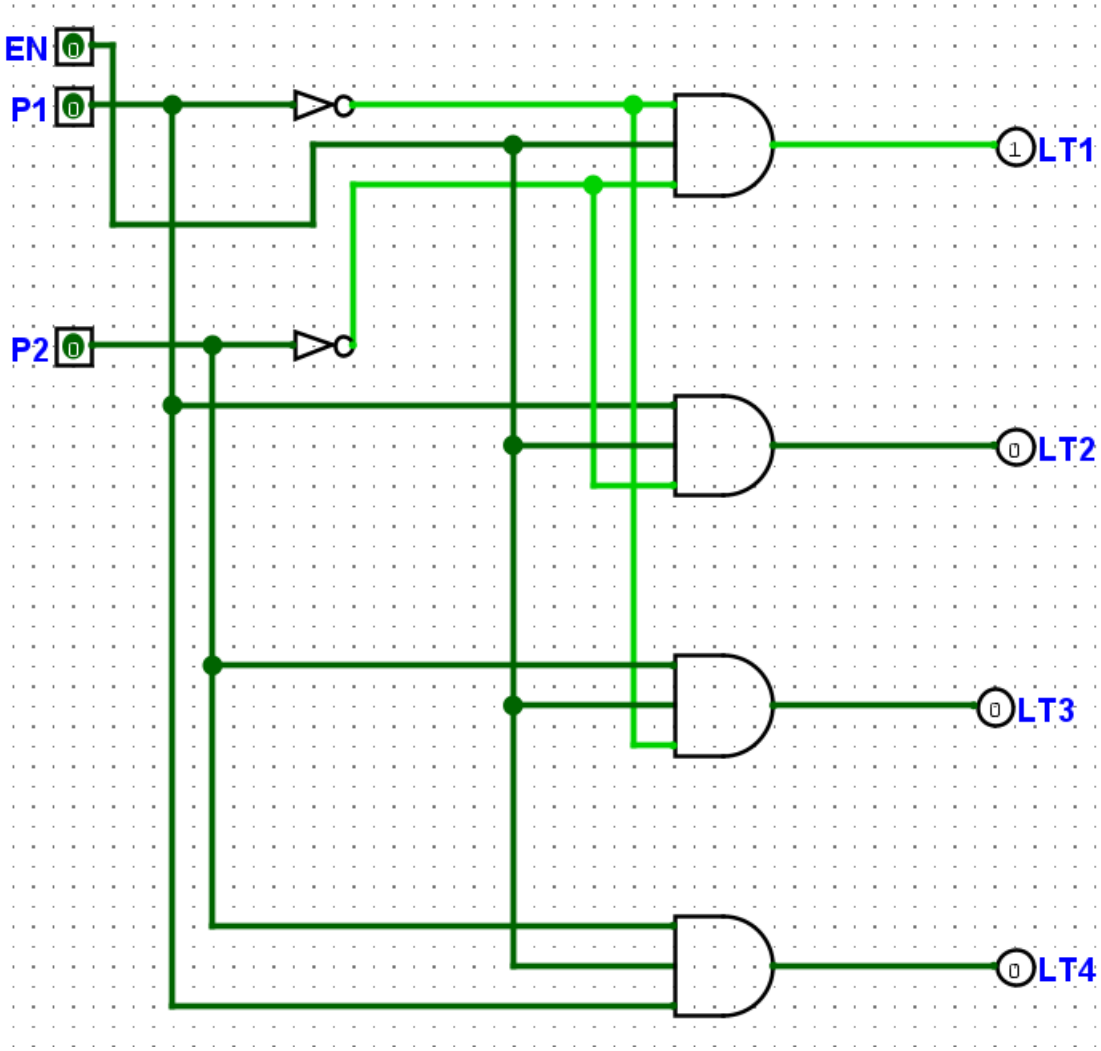


Figure 40 - 2 to 4 Line Decoder

Using four of 2 to 4 decoders, a 4 to 16 decoder can be developed as in the following figure.

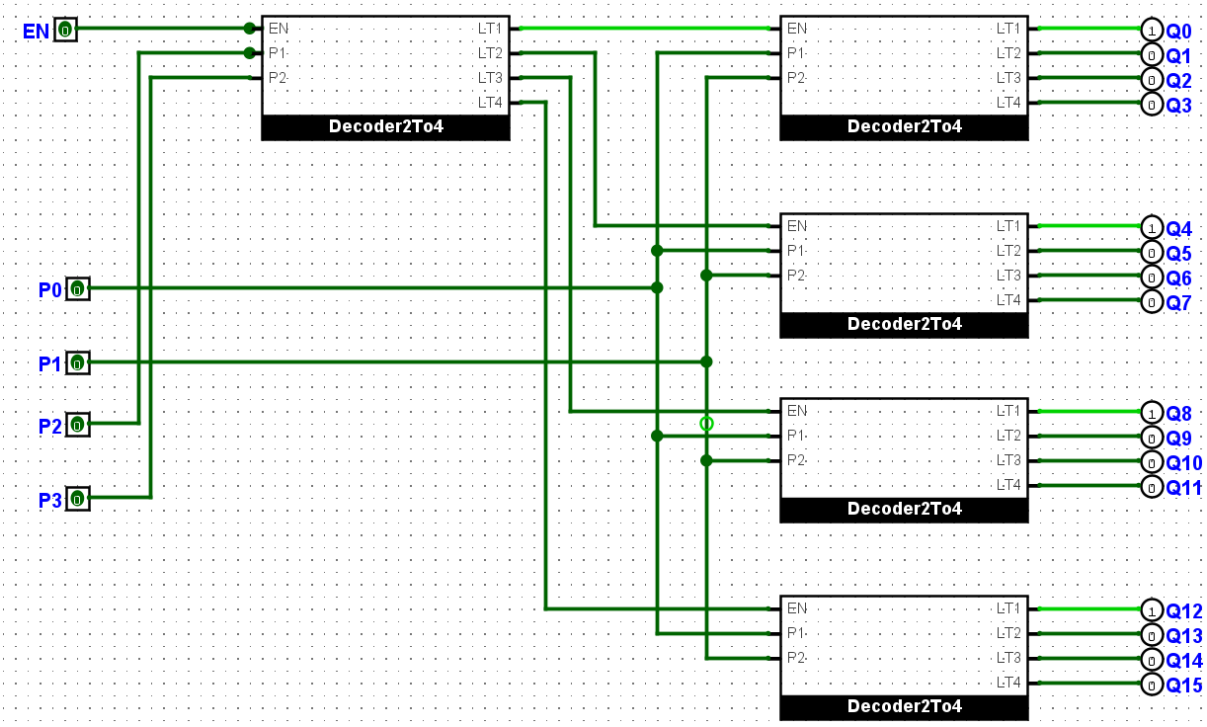


Figure 41 - 4 to 16 Line Decoder

4-Bit RAM

This chapter explains the progress of how to build a 4-bit RAM using the fundamental theories. For this example, design, one address line is taken to make a selection between two memory locations and each memory location can hold up to four bits.

Design explanations:

- The Memory cells are created using four flip flops where 8 flip flops altogether represent the Memory cell segment as the design is set for 2 memory cells.
- Each flip flop represents one bit therefore the memory cell segment have 8 bits for inputs
- Since only one address line is used for the purpose of decoding, this decoder MUX performs the data directing tasks for each respective memory cell based on the address of it.

- Data input line is carrying the four-bit data that have to be written into each memory cell, and WE enable the writing task where the DeMUX uses the relevant address lines to write the data into each memory cell.
- To read the data, a MUX is assigned to each memory cell on the output side where each address line is connected to their memory cell that enables the MUX to read the stored data in each memory cell.

A summarized truth table for the above explained 4-bit RAM is as follows,

Table 9 - Summary of 4-bit RAM

Address	WE	Input	Cell A	Cell B	Output
0	1	1100	1100	Unchanged	NA
1	1	1110	Unchanged	1110	NA
0	0	NA	1100	1110	1100
1	0	NA	1100	1110	1110

Program Counter

The program counter is a register in the CPU that holds the address of the next instruction to execute automatically incrementing after each fetch to sequence through program instructions. It is also known as the instruction pointer that is a fundamental component of a central processing unit (CPU). The program counter can be modified by giving out instructions or events during the execution of the program. [15]

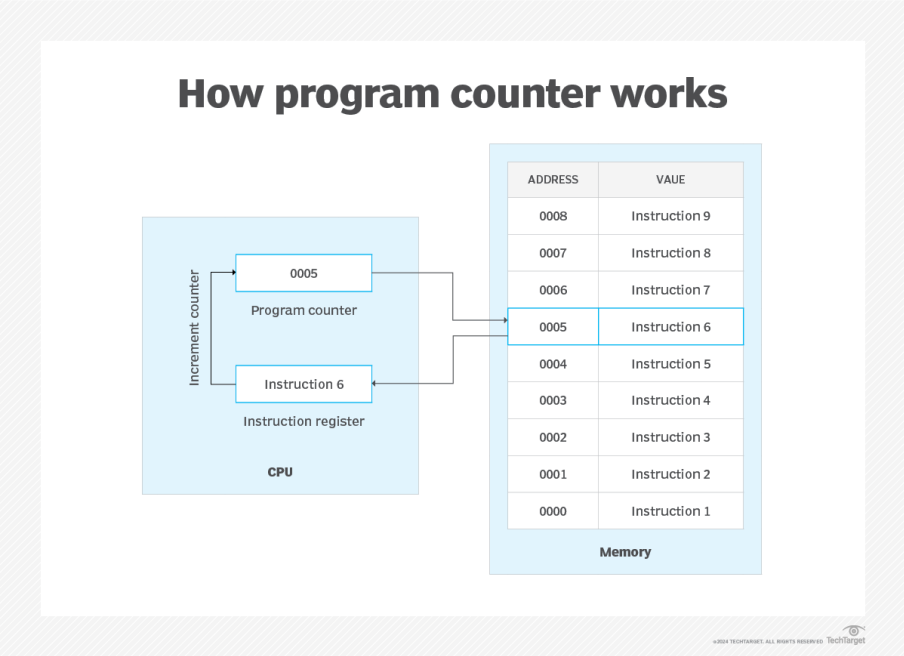


Figure 42 - How Program Counter Works

The CPU determines which instruction to fetch from memory by referencing the address in the Program Counter (PC). Once fetched, the instruction is stored in the Instruction Register, allowing the CPU to decode and execute it. The Instruction Register relies on the PC to provide the correct instruction at each step. [16]

During the instruction cycle, the CPU performs three key steps: fetching, decoding, and executing an instruction. It first retrieves the instruction address from the PC, then fetches the instruction from memory (e.g., if the address is 0851, it retrieves the instruction at memory location 0851). After fetching, the instruction is placed in the Instruction Register, the PC increments by one, and the CPU decodes and executes the instruction, repeating the cycle for the next instruction. [16]

Four-bit Synchronous Up Counter

A counter is a device that tracks and stores the number of occurrences of a specific event, based on a clock signal. In digital logic systems or computers, it is typically a sequential digital circuit with one clock input and multiple outputs, which represent binary or binary-coded decimal values. Each clock pulse increments or decrements the count

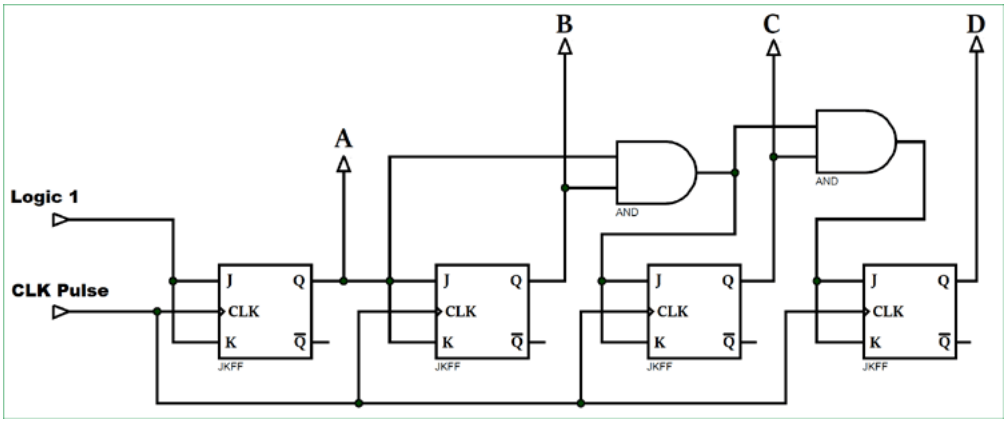


Figure 43 - Synchronous Counter

A four-bit synchronous Up counter has four flip-flops connected synchronously, where all flip-flops are triggered simultaneously by the clock signal. It counts from 0 to 15 in binary (0000 to 1111) with each clock pulse. A 4-bit synchronous up counter begins counting from 0 (0000 in binary) up to 15 (1111 in binary) before resetting and starting a new cycle. Unlike asynchronous counters, it operates at a higher frequency with no propagation delay, as all flip-flops are simultaneously triggered by a shared clock signal. [17]

In this setup, an external clock feeds all J-K flip-flops in parallel. The first flip-flop (FFA), representing the least significant bit, is connected to a constant HIGH signal on its J and K inputs, causing it to toggle with each clock pulse. The second flip-flop (FFB) takes its inputs from the output of FFA, while the subsequent flip-flops (FFC and FFD) use AND gates to determine their state based on preceding flip-flop outputs. This synchronous arrangement avoids the ripple effect present in asynchronous counters, as each flip-flop changes state simultaneously. [17]

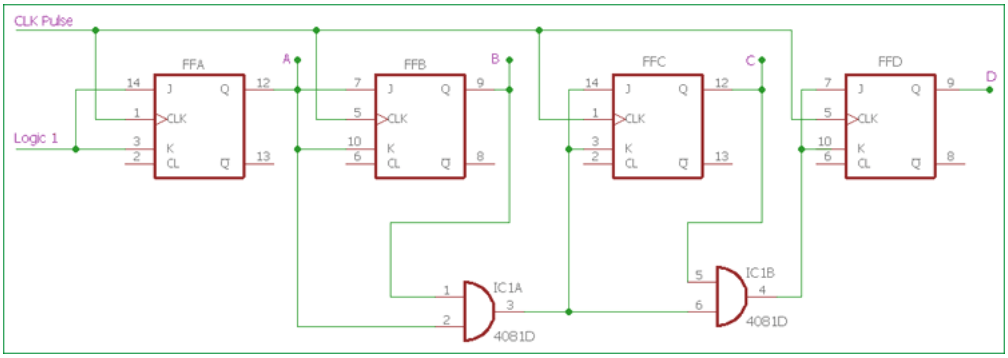


Figure 44 - Synchronous Up Counter

Synchronous counters offer several advantages,

1. They are easier to design than asynchronous counters
2. They operate simultaneously without propagation delay
3. They have a controlled count sequence using logic gates (reducing errors)
4. They work faster.

The only disadvantage these counters have is that they require more complex logic. Applications of synchronous counters include,

1. Machine motion control
2. Motor RPM counting
3. Rotary shaft encoding
4. Digital clocks and pulse generators
5. Digital watches
6. Alarm systems.

Four-bit Synchronous Up Counter with Load

A Four-bit Synchronous Up Counter with Load is a digital circuit that counts from 0 to 15 in binary, with a feature allowing it to load a specific starting value. It increments by one with each clock pulse and resets after reaching 15, operating synchronously (without propagation delay) due to simultaneous clocking of all flip-flops.

It has the following key features:

1. Basic Counting: Counts from 0 to 15, resetting after each cycle.

2. Load Functionality: A parallel load signal enables the counter to start from a preset 4-bit value instead of zero when activated, adding flexibility.
3. Structure: Uses four J-K flip-flops, with logic gates controlling the count sequence.
4. Operation Control: Flip-flops toggle based on preceding flip-flop states, ensuring synchronous and orderly counting.
5. Applications: Useful in digital clocks, pulse generators, frequency dividers, and other timing-related systems needing flexible initialization.

This counter's load feature enhances its versatility, allowing it to meet a variety of application needs that require starting from specific values.

Four-bit Program Counter

This is a four-bit counter used in simple CPUs to track the instruction sequence. It holds the address of the next instruction and counts sequentially, resetting or jumping based on control signals.

Basic Functionality

The program counter holds the address of the next instruction to be fetched from memory. In a four-bit setup, it can represent addresses from 0 (0000 in binary) to 15 (1111 in binary), covering 16 memory locations. It increments after each instruction fetch, ensuring instructions are processed in order. Once it reaches 1111, the PC resets to 0 to restart the counting cycle.

Operation

The PC's operation is integral to the instruction cycle, which includes fetching, decoding, and executing instructions. At the start of the instruction cycle, the CPU retrieves the address stored in the PC, fetches the corresponding instruction from memory, and loads it into the instruction register. After fetching, the PC increments by one, updating the address for the next fetch, and this process repeats as the CPU executes instructions.

Control Signals

The program counter responds to various control signals that influence its behavior, such as:

1. Jump Instructions: These can alter the PC to point to different memory addresses for non-sequential instruction execution.
2. Reset Signals: When activated, the PC can be set to a specific starting address, allowing the CPU to restart execution.

Design and Implementation

Typically implemented using flip-flops (such as J-K or D flip-flops), the four-bit PC represents each bit of the counter. Logic gates may manage the incrementing process and handle control signals for jumps and resets, with an adder circuit often used for automatic incrementing on clock cycles.

Importance in CPU Architecture

The program counter is crucial for ensuring the correct flow of instruction execution. By tracking the current instruction address, it guarantees the CPU processes instructions in the intended sequence. Its responsiveness to control signals allows for flexibility in implementing programming constructs like loops and conditional statements.

Interconnection with Other Components

The PC works alongside other CPU components, including the instruction register, memory unit, and arithmetic logic unit (ALU). For example, after fetching an instruction, the instruction register provides the opcode to the ALU, while the PC prepares the next instruction for fetching.

Applications

Four-bit program counters are commonly used in educational settings and basic microprocessor designs to demonstrate fundamental CPU operations. They help illustrate how more complex CPUs manage instruction execution and control flow.

Control Unit

The Control Unit (CU) is a critical component of a computer's central processing unit (CPU) that manages the operations of the processor. [18][19] It was in John von Neumann's architecture, the control unit instructs the computer's memory, arithmetic/logic unit, and input/output devices on how to respond to the processor's commands. It fetches internal instructions from main memory into the instruction register and generates control signals based on the contents of this register to supervise instruction execution. [18]

The control unit operates by receiving input information, converting it into control signals, and sending these signals to the CPU. This enables the processor to direct the attached hardware to perform the required operations. The functions of the control unit can vary depending on the specific CPU architecture, as designs differ between manufacturers

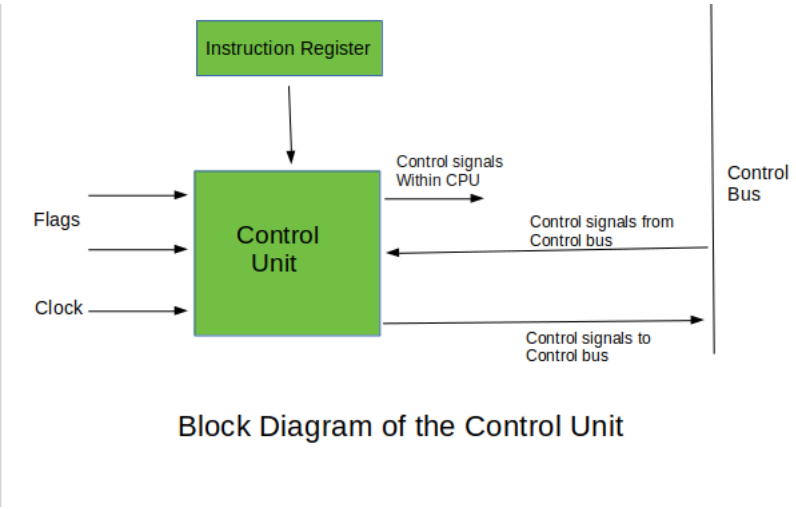


Figure 45 - Block Diagram of Control Unit

There are two types of control units: [19]

1. Hardwired
2. Microprogrammed

A well-designed Control unit has the following advantages [19]

1. Increased Reliability: It improves CPU reliability by detecting and correcting errors, such as memory issues and pipeline stalls.
2. Lower Power Consumption: By optimizing resource usage—like registers and memory and reducing clock cycles, a well-designed control unit can decrease power consumption.
3. Support for Complex Instructions: Such a control unit can handle complex instructions that require multiple operations, which reduces the total number of instructions needed to run a program.
4. Enhanced Performance: It enhances CPU performance by increasing clock speed, reducing latency, and improving overall throughput.
5. Enhanced Scalability: A well-designed control unit improves CPU scalability, enabling it to manage larger and more complex workloads.
6. **Improved Security**: The control unit can implement security features, such as address space layout randomization and data execution prevention, enhancing CPU security.
7. **Cost Efficiency**: By minimizing the number of required components and improving manufacturing efficiency, a well-designed control unit can lower CPU production costs.

A poorly designed control unit has the following disadvantages:

1. Reduced Performance: A poorly designed control unit can degrade CPU performance by causing pipeline stalls, increasing latency, and decreasing throughput.
2. Increased Complexity: Such a design can complicate the CPU, making it more challenging to design, test, and maintain.
3. Higher Power Consumption: Inefficient resource utilization, including registers and memory, can lead to increased power consumption, as more clock cycles may be needed for instruction execution.
4. Decreased Reliability: It can compromise CPU reliability by introducing errors like memory issues and pipeline stalls.
5. Limited Scalability: It may hinder the CPU's ability to scale, making it challenging to handle larger and more complex workloads.

The Control Unit (CU) performs several essential functions within the CPU: [18]

Roles and Functions

1. Coordination of Data Movement: It manages the sequence of data transfers into, out of, and between the various sub-units of the processor.
2. Instruction Interpretation: The CU interprets incoming instructions to determine the required actions.
3. Data Flow Control: It regulates the flow of data within the processor.
4. Conversion of Commands: The CU receives external instructions and converts them into a sequence of control signals.
5. Management of Execution Units: It oversees multiple execution units, such as the Arithmetic Logic Unit (ALU), data buffers, and registers within the CPU.
6. Task Handling: The CU is responsible for several tasks, including fetching instructions, decoding them, managing execution, and storing the results.

VHDL Implementations

Approach 01

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity CPU is
    port (
        clk          : in std_logic;                    -- Clock input
        rst          : in std_logic;                    -- Reset input
        btn_add      : in std_logic;                    -- Button to perform calculation
        switches     : in std_logic_vector(9 downto 0); -- 10 switches for input
        leds         : out std_logic_vector(9 downto 0); -- 10 LEDs for output
        onboard_leds : out std_logic_vector(9 downto 0)  -- Onboard LEDs for output
    );
end entity;

architecture Behavioral of CPU is
    -- Internal signals for registers and results
    signal reg_a       : std_logic_vector(4 downto 0); -- 5 bits for reg_a
    signal reg_b       : std_logic_vector(4 downto 0); -- 5 bits for reg_b
    signal value1      : std_logic_vector(9 downto 0); -- 10 bits for intermediate value (reg_a + reg_b)
    signal value2      : std_logic_vector(9 downto 0); -- 10 bits for intermediate value2 (value1 * reg_a)
    signal value3      : std_logic_vector(9 downto 0); -- 10 bits for intermediate value3 (value2 - reg_a)
    signal alu_result  : std_logic_vector(9 downto 0); -- 10 bits for final result (value3 / reg_a)
    signal calc_state  : integer range 0 to 3 := 0;    -- State to control step-by-step operations
begin

    -- Process for input and ALU operation with multi-step computation
    process(clk, rst)
    begin
        if rst = '1' then
            -- Reset all signals and state
            reg_a <= (others => '0');
            reg_b <= (others => '0');
            value1 <= (others => '0');
            value2 <= (others => '0');
            value3 <= (others => '0');
            alu_result <= (others => '0');
            calc_state <= 0; -- Start at initial state
        elsif rising_edge(clk) then
            if btn_add = '1' then
                case calc_state is
                    -- Step 1: Add reg_a and reg_b (calculate value1)
                    when 0 =>
                        reg_a <= switches(9 downto 5); -- Get reg_a from switches
                        reg_b <= switches(4 downto 0); -- Get reg_b from switches
                        value1 <= std_logic_vector(resize(unsigned(reg_a) + unsigned(reg_b), 10));
                        calc_state <= 1; -- Move to the next step

                    -- Step 2: Multiply value1 by reg_a (calculate value2)
                    when 1 =>
                        value2 <= std_logic_vector(resize(unsigned(value1) * unsigned(reg_a), 10));
                        calc_state <= 2; -- Move to the next step

                    -- Step 3: Subtract reg_a from value2 (calculate value3)
                    when 2 =>
                        value3 <= std_logic_vector(resize(unsigned(value2) - unsigned(reg_a), 10));
                        calc_state <= 3; -- Move to the next step

                    -- Step 4: Divide value3 by reg_a (calculate alu_result)
                    when 3 =>
                        if reg_a /= "00000" then  -- Check for division by zero
                            alu_result <= std_logic_vector(resize(unsigned(value3) / unsigned(reg_b), 10));
                        else
                            alu_result <= (others => '0'); -- Set result to zero in case of division by zero
                        end if;
                        calc_state <= 0; -- Go back to step 0, ready for next operation

                    when others =>
                        calc_state <= 0; -- Default case (shouldn't happen)
                end case;
            end if;
        end if;
    end process;

    -- Output the final ALU result to LEDs
    leds <= alu_result;          -- Display the result on external LEDs
    onboard_leds <= alu_result;  -- Display the result on onboard LEDs

end architecture;
```

Figure 46 - Approach 1 VHDL code

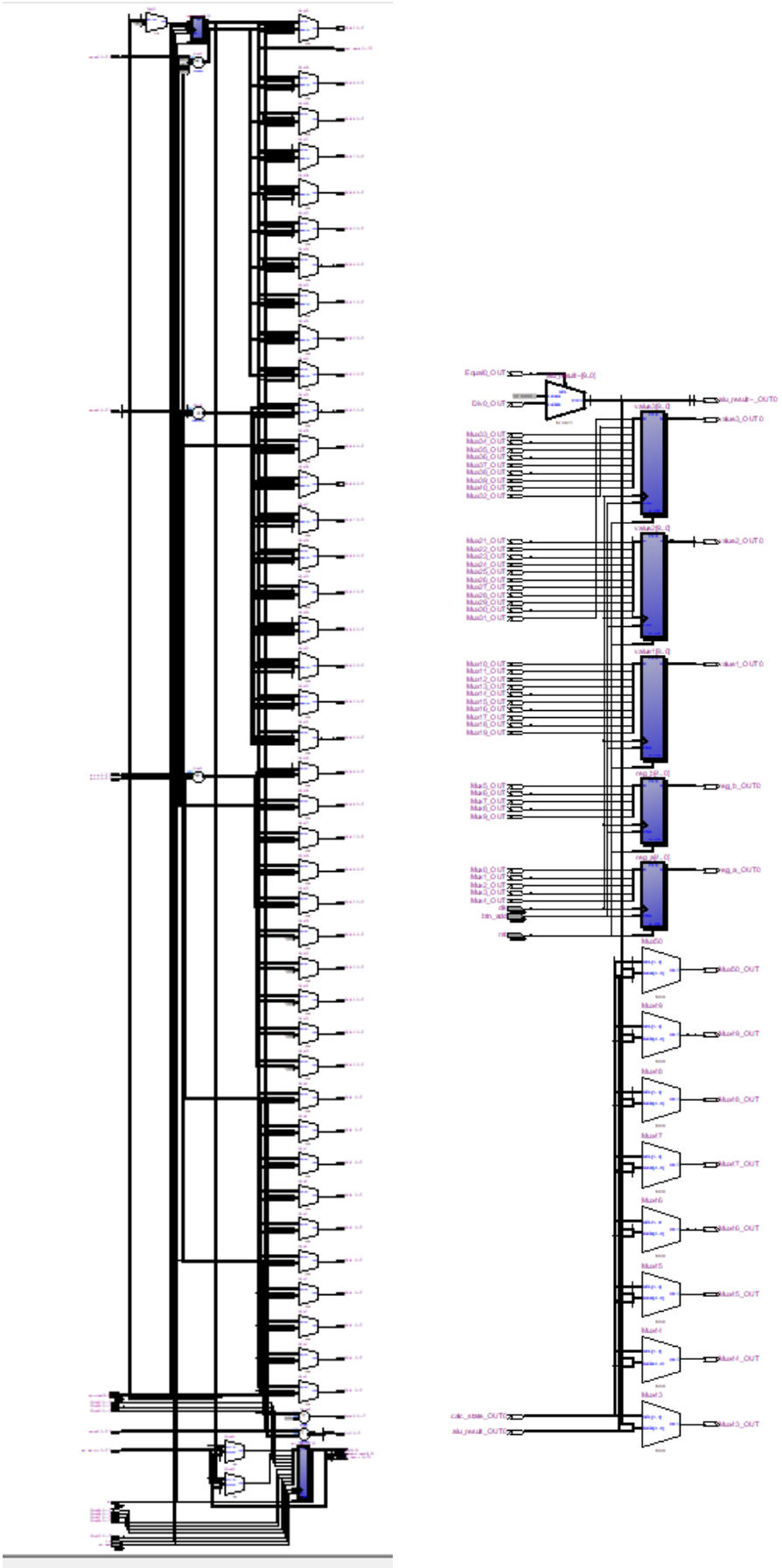


Figure 47 - Approach 01 CPU design

Approach 02

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY ALU16Bit IS
    PORT (
        C_IN  : IN  STD_LOGIC;
        S     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        A     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        B     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        G     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        C_OUT : OUT STD_LOGIC
    );
END ALU16Bit;

ARCHITECTURE Behavioral OF ALU16Bit IS
    SIGNAL A_unsigned : UNSIGNED(15 DOWNTO 0);
    SIGNAL B_unsigned : UNSIGNED(15 DOWNTO 0);
    SIGNAL C_in_bit   : UNSIGNED(0 DOWNTO 0);
    SIGNAL Result     : UNSIGNED(16 DOWNTO 0);
BEGIN
    -- Convert inputs to UNSIGNED
    A_unsigned <= UNSIGNED(A);
    B_unsigned <= UNSIGNED(B);
    C_in_bit(0) <= '1' WHEN C_IN = '1' ELSE '0';

    PROCESS (A_unsigned, B_unsigned, S, C_in_bit)
    BEGIN
        CASE S IS
            WHEN "000" => -- ADD
                Result <= ('0' & A_unsigned) + ('0' & B_unsigned) + C_in_bit;
            WHEN "001" => -- SUBTRACT
                Result <= ('0' & A_unsigned) - ('0' & B_unsigned) - C_in_bit;
            WHEN "010" => -- AND
                Result <= '0' & (A_unsigned AND B_unsigned);
            WHEN "011" => -- OR
                Result <= '0' & (A_unsigned OR B_unsigned);
            WHEN "100" => -- XOR
                Result <= '0' & (A_unsigned XOR B_unsigned);
            WHEN "101" => -- NOT A
                Result <= '0' & (NOT A_unsigned);
            WHEN OTHERS =>
                Result <= (OTHERS => '0');
        END CASE;
        G     <= STD_LOGIC_VECTOR(Result(15 DOWNTO 0));
        C_OUT <= Result(16);
    END PROCESS;
END Behavioral;
```

Figure 48 – ALU_16Bit

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY Buffer16Bit IS
    PORT (
        E : IN  STD_LOGIC;
        A : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        S : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END Buffer16Bit;

ARCHITECTURE Behavioral OF Buffer16Bit IS
BEGIN
    S <= A WHEN E = '1' ELSE (OTHERS => 'Z');
END Behavioral;
```

Figure 49 – Buffer_16Bit

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY BufferedRegister4Bit IS
    PORT (
        CLR      : IN  STD_LOGIC;
        CLK      : IN  STD_LOGIC;
        LOAD     : IN  STD_LOGIC;
        DATA_IN  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
        ENABLE   : IN  STD_LOGIC;
        DATA_OUT : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
    );
END BufferedRegister4Bit;

ARCHITECTURE Behavioral OF BufferedRegister4Bit IS
    SIGNAL Reg_Data : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
    PROCESS (CLK, CLR)
    BEGIN
        IF CLR = '1' THEN
            Reg_Data <= (OTHERS => '0');
        ELSIF RISING_EDGE(CLK) THEN
            IF LOAD = '1' THEN
                Reg_Data <= DATA_IN;
            END IF;
        END IF;
    END PROCESS;

    DATA_OUT <= Reg_Data WHEN ENABLE = '1' ELSE (OTHERS => 'Z');
END Behavioral;

```

Figure 50 – Buffered_Register_4Bit

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ALU16BitBuffered IS
    PORT (
        -- Input Operands
        A        : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        B        : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        -- Control Signals
        CLK      : IN  STD_LOGIC;
        FLAG_CLR : IN  STD_LOGIC;
        FI       : IN  STD_LOGIC;
        C_IN     : IN  STD_LOGIC;
        S        : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        EO       : IN  STD_LOGIC;
        -- Output Signals
        G        : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
        CF       : OUT STD_LOGIC;
        ZF       : OUT STD_LOGIC
    );
END ALU16BitBuffered;

ARCHITECTURE Behavioral OF ALU16BitBuffered IS

    COMPONENT ALU16Bit
        PORT (
            C_IN  : IN  STD_LOGIC;
            S     : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
            A     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            B     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            G     : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
            C_OUT : OUT STD_LOGIC
        );
    END COMPONENT;

    COMPONENT Buffer16Bit
        PORT (
            E : IN  STD_LOGIC;
            A : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
            S : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
        );
    END COMPONENT;

    COMPONENT BufferedRegister4Bit
        PORT (
            CLR      : IN  STD_LOGIC;
            CLK      : IN  STD_LOGIC;
            LOAD     : IN  STD_LOGIC;
            DATA_IN  : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
            ENABLE   : IN  STD_LOGIC;
            DATA_OUT : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
        );
    END COMPONENT;

    SIGNAL ALU_Output      : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL Carry_Out       : STD_LOGIC;
    SIGNAL Zero_Flag_Input : STD_LOGIC;
    SIGNAL Flag_Reg_Output : STD_LOGIC_VECTOR(3 DOWNTO 0);

BEGIN

    -- Instantiate the ALU
    U_ALU: ALU16Bit
        PORT MAP (
            C_IN  => C_IN,
            S     => S,
            A     => A,
            B     => B,
            G     => ALU_Output,
            C_OUT => Carry_Out
        );

    -- Zero Flag Logic
    Zero_Flag_Input <= NOT (ALU_Output(0) OR ALU_Output(1) OR ALU_Output(2) OR ALU_Output(3) OR
                            ALU_Output(4) OR ALU_Output(5) OR ALU_Output(6) OR ALU_Output(7) OR
                            ALU_Output(8) OR ALU_Output(9) OR ALU_Output(10) OR ALU_Output(11) OR
                            ALU_Output(12) OR ALU_Output(13) OR ALU_Output(14) OR ALU_Output(15));
```

```
    -- Zero Flag Logic
    Zero_Flag_Input <= NOT (ALU_Output(0) OR ALU_Output(1) OR ALU_Output(2) OR ALU_Output(3) OR
                            ALU_Output(4) OR ALU_Output(5) OR ALU_Output(6) OR ALU_Output(7) OR
                            ALU_Output(8) OR ALU_Output(9) OR ALU_Output(10) OR ALU_Output(11) OR
                            ALU_Output(12) OR ALU_Output(13) OR ALU_Output(14) OR ALU_Output(15));

    -- Instantiate the Flag Register
    U_Flag_Reg: BufferedRegister4Bit
        PORT MAP (
            CLR      => FLAG_CLR,
            CLK      => CLK,
            LOAD     => FI,
            DATA_IN  => Carry_Out & Zero_Flag_Input & '0' & '0',
            ENABLE   => '1',
            DATA_OUT => Flag_Reg_Output
        );

    -- Assign CF and ZF from Flag Register Output
    CF <= Flag_Reg_Output(3);
    ZF <= Flag_Reg_Output(2);

    -- Instantiate the Output Buffer
    U_Buffer: Buffer16Bit
        PORT MAP (
            E => EO,
            A => ALU_Output,
            S => G
        );

END Behavioral;
```

Figure 51 – ALU_16Bit_Buffered

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ControlUnit IS
    PORT (
        Step    : IN  STD_LOGIC_VECTOR(2 DOWNTO 0);
        Opcode  : IN  STD_LOGIC_VECTOR(15 DOWNTO 12);
        -- Control Signals
        MI, RI, RO, IO, II, AI, AO, EO, C_IN, BI, OI, CE, CO : OUT STD_LOGIC;
        S       : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
    );
END ControlUnit;

ARCHITECTURE Behavioral OF ControlUnit IS

    -- Define Opcodes
    CONSTANT LDA     : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0001";
    CONSTANT ADD     : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0010";
    CONSTANT SUB     : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0011";
    CONSTANT AND_OP  : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0100";
    CONSTANT OR_OP   : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0101";
    CONSTANT XOR_OP  : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0110";
    CONSTANT NOT_OP  : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0111";
    CONSTANT OUT_OP  : STD_LOGIC_VECTOR(3 DOWNTO 0) := "1110";
    CONSTANT HLT     : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";

BEGIN

    PROCESS (Opcode, Step)
    BEGIN
        -- Default Control Signal Values
        MI <= '0'; RI <= '0'; RO <= '0'; IO <= '0'; II <= '0';
        AI <= '0'; AO <= '0'; EO <= '0'; C_IN <= '0';
        S <= "000"; BI <= '0'; OI <= '0'; CE <= '0'; CO <= '0';

        -- Decode the Opcode
        CASE Opcode IS
            WHEN LDA =>
                CASE Step IS
                    WHEN "000" =>
                        MI <= '1'; CO <= '1';
                    WHEN "001" =>
                        RO <= '1'; II <= '1'; CE <= '1';
                    WHEN "010" =>
```

```
                WHEN "010" =>
                    MI <= '1'; IO <= '1';
                WHEN "011" =>
                    RO <= '1'; AI <= '1';
                WHEN OTHERS =>
                    NULL;
            END CASE;

        WHEN ADD =>
            CASE Step IS
                WHEN "000" =>
                    MI <= '1'; CO <= '1';
                WHEN "001" =>
                    RO <= '1'; II <= '1'; CE <= '1';
                WHEN "010" =>
                    MI <= '1'; IO <= '1';
                WHEN "011" =>
                    RO <= '1'; BI <= '1';
                WHEN "100" =>
                    AI <= '1'; EO <= '1'; S <= "000";
                WHEN OTHERS =>
                    NULL;
            END CASE;

        WHEN SUB =>
            CASE Step IS
                WHEN "000" =>
                    MI <= '1'; CO <= '1';
                WHEN "001" =>
                    RO <= '1'; II <= '1'; CE <= '1';
                WHEN "010" =>
                    MI <= '1'; IO <= '1';
                WHEN "011" =>
                    RO <= '1'; BI <= '1';
                WHEN "100" =>
                    AI <= '1'; EO <= '1'; S <= "001"; -- Subtract operation
                WHEN OTHERS =>
                    NULL;
            END CASE;
```

```
            END CASE;

        WHEN AND_OP =>
            CASE Step IS
                WHEN "000" =>
                    MI <= '1'; CO <= '1';
                WHEN "001" =>
                    RO <= '1'; II <= '1'; CE <= '1';
                WHEN "010" =>
                    MI <= '1'; IO <= '1';
                WHEN "011" =>
                    RO <= '1'; BI <= '1';
                WHEN "100" =>
                    AI <= '1'; EO <= '1'; S <= "010"; -- AND operation
                WHEN OTHERS =>
                    NULL;
            END CASE;

        WHEN OR_OP =>
            CASE Step IS
                WHEN "000" =>
                    MI <= '1'; CO <= '1';
                WHEN "001" =>
                    RO <= '1'; II <= '1'; CE <= '1';
                WHEN "010" =>
                    MI <= '1'; IO <= '1';
                WHEN "011" =>
                    RO <= '1'; BI <= '1';
                WHEN "100" =>
                    AI <= '1'; EO <= '1'; S <= "011"; -- OR operation
                WHEN OTHERS =>
                    NULL;
            END CASE;
```

```
                END CASE;

            WHEN XOR_OP =>
                CASE Step IS
                    WHEN "000" =>
                        MI <= '1'; CO <= '1';
                    WHEN "001" =>
                        RO <= '1'; II <= '1'; CE <= '1';
                    WHEN "010" =>
                        MI <= '1'; IO <= '1';
                    WHEN "011" =>
                        RO <= '1'; BI <= '1';
                    WHEN "100" =>
                        AI <= '1'; EO <= '1'; S <= "100"; -- XOR operation
                    WHEN OTHERS =>
                        NULL;
                END CASE;

            WHEN NOT_OP =>
                CASE Step IS
                    WHEN "000" =>
                        AI <= '1'; EO <= '1'; S <= "101"; -- NOT operation
                    WHEN OTHERS =>
                        NULL;
                END CASE;

            WHEN OUT_OP =>
                CASE Step IS
                    WHEN "000" =>
                        AO <= '1'; OI <= '1';
                    WHEN OTHERS =>
                        NULL;
                END CASE;

            WHEN HLT =>
                CASE Step IS
                    WHEN "000" =>
                        -- No operation, halt
                        NULL;
                    WHEN OTHERS =>
                        NULL;
                END CASE;

            WHEN OTHERS =>
                NULL;
        END CASE;
    END PROCESS;

END Behavioral;
```

Figure 52 – Control_Unit

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY AccumulatorRegister IS
    PORT (
        CLK       : IN  STD_LOGIC;
        CLR       : IN  STD_LOGIC;
        LOAD      : IN  STD_LOGIC;
        DATA_IN   : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        DATA_OUT  : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END AccumulatorRegister;

ARCHITECTURE Behavioral OF AccumulatorRegister IS
    SIGNAL RegData : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    PROCESS (CLK, CLR)
    BEGIN
        IF CLR = '1' THEN
            RegData <= (OTHERS => '0');
        ELSIF RISING_EDGE(CLK) THEN
            IF LOAD = '1' THEN
                RegData <= DATA_IN;
            END IF;
        END IF;
    END PROCESS;
    DATA_OUT <= RegData;
END Behavioral;

```

Figure 53 - Accumulator_Register

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY BRegister IS
    PORT (
        CLK       : IN  STD_LOGIC;
        CLR       : IN  STD_LOGIC;
        LOAD      : IN  STD_LOGIC;
        DATA_IN   : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        DATA_OUT  : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END BRegister;

ARCHITECTURE Behavioral OF BRegister IS
    SIGNAL RegData : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    PROCESS (CLK, CLR)
    BEGIN
        IF CLR = '1' THEN
            RegData <= (OTHERS => '0');
        ELSIF RISING_EDGE(CLK) THEN
            IF LOAD = '1' THEN
                RegData <= DATA_IN;
            END IF;
        END IF;
    END PROCESS;
    DATA_OUT <= RegData;
END Behavioral;
```

Figure 54 – B_Register

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY InstructionRegister IS
    PORT (
        CLK       : IN  STD_LOGIC;
        CLR       : IN  STD_LOGIC;
        LOAD      : IN  STD_LOGIC;
        DATA_IN   : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        DATA_OUT  : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END InstructionRegister;

ARCHITECTURE Behavioral OF InstructionRegister IS
    SIGNAL RegData : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    PROCESS (CLK, CLR)
    BEGIN
        IF CLR = '1' THEN
            RegData <= (OTHERS => '0');
        ELSIF RISING_EDGE(CLK) THEN
            IF LOAD = '1' THEN
                RegData <= DATA_IN;
            END IF;
        END IF;
    END PROCESS;
    DATA_OUT <= RegData;
END Behavioral;
```

Figure 55 - Instruction_Register

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  ENTITY MemoryAddressRegister IS
6      PORT (
7          CLK       : IN  STD_LOGIC;
8          CLR       : IN  STD_LOGIC;
9          LOAD      : IN  STD_LOGIC;
10         ADDR_IN   : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
11         ADDR_OUT  : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
12     );
13 END MemoryAddressRegister;
14
15 ARCHITECTURE Behavioral OF MemoryAddressRegister IS
16     SIGNAL RegAddr : STD_LOGIC_VECTOR(15 DOWNTO 0);
17 BEGIN
18     PROCESS (CLK, CLR)
19     BEGIN
20         IF CLR = '1' THEN
21             RegAddr <= (OTHERS => '0');
22         ELSIF RISING_EDGE(CLK) THEN
23             IF LOAD = '1' THEN
24                 RegAddr <= ADDR_IN;
25             END IF;
26         END IF;
27     END PROCESS;
28     ADDR_OUT <= RegAddr;
29 END Behavioral;
```

Figure 56 - Memory_Address_Register

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.numeric_std.ALL;
4
5  ENTITY ProgramCounter IS
6      PORT (
7          CLK       : IN  STD_LOGIC;
8          CLR       : IN  STD_LOGIC;
9          INC       : IN  STD_LOGIC;
10         LOAD      : IN  STD_LOGIC;
11         ADDR_IN   : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
12         ADDR_OUT  : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
13     );
14 END ProgramCounter;
15
16 ARCHITECTURE Behavioral OF ProgramCounter IS
17     SIGNAL PC_Reg : UNSIGNED(15 DOWNTO 0);
18 BEGIN
19     PROCESS (CLK, CLR)
20     BEGIN
21         IF CLR = '1' THEN
22             PC_Reg <= (OTHERS => '0');
23         ELSIF RISING_EDGE(CLK) THEN
24             IF LOAD = '1' THEN
25                 PC_Reg <= UNSIGNED(ADDR_IN);
26             ELSIF INC = '1' THEN
27                 PC_Reg <= PC_Reg + 1;
28             END IF;
29         END IF;
30     END PROCESS;
31     ADDR_OUT <= STD_LOGIC_VECTOR(PC_Reg);
32 END Behavioral;
33
```

Figure 57 - Program_Counter

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY OutputRegister IS
    PORT (
        CLK        : IN  STD_LOGIC;
        CLR        : IN  STD_LOGIC;
        LOAD       : IN  STD_LOGIC;
        DATA_IN    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        DATA_OUT   : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END OutputRegister;

ARCHITECTURE Behavioral OF OutputRegister IS
    SIGNAL RegData : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    PROCESS (CLK, CLR)
    BEGIN
        IF CLR = '1' THEN
            RegData <= (OTHERS => '0');
        ELSIF RISING_EDGE(CLK) THEN
            IF LOAD = '1' THEN
                RegData <= DATA_IN;
            END IF;
        END IF;
    END PROCESS;
    DATA_OUT <= RegData;
END Behavioral;

```

Figure 58 - Output_Register

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY RAM IS
    PORT (
        CLK        : IN  STD_LOGIC;
        WE         : IN  STD_LOGIC;
        OE         : IN  STD_LOGIC;
        ADDR       : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        DATA_IN    : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        DATA_OUT   : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
END RAM;

ARCHITECTURE Behavioral OF RAM IS
    TYPE memory_array IS ARRAY (0 TO 65535) OF STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL RAM_Data : memory_array := (OTHERS => (OTHERS => '0'));
BEGIN
    PROCESS (CLK)
    BEGIN
        IF RISING_EDGE(CLK) THEN
            IF WE = '1' THEN
                RAM_Data(to_integer(unsigned(ADDR))) <= DATA_IN;
            END IF;
        END IF;
    END PROCESS;

    DATA_OUT <= RAM_Data(to_integer(unsigned(ADDR))) WHEN OE = '1' ELSE (OTHERS => 'Z');
END Behavioral;

```

Figure 59 - RAM

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY Top_Lvl_CPU IS
    PORT (
        CLK         : IN  STD_LOGIC;
        RESET       : IN  STD_LOGIC;
        DATA_IN     : IN  STD_LOGIC_VECTOR(15 DOWNTO 0);
        DATA_OUT    : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
        ADDR_OUT    : BUFFER STD_LOGIC_VECTOR(15 DOWNTO 0);
        HLT         : OUT STD_LOGIC
    );
END Top_Lvl_CPU;

ARCHITECTURE Behavioral OF Top_Lvl_CPU IS

    -- Signals for connecting components
    SIGNAL PC_Out, MAR_Out, IR_Out, ACC_Out, B_Out : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL ALU_Out                                 : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL ALU_CF, ALU_ZF                          : STD_LOGIC;
    SIGNAL Control_Signals                         : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL Step                                    : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL Opcode                                  : STD_LOGIC_VECTOR(15 DOWNTO 12);

    -- Control signals
    SIGNAL MI, RI, RO, IO, II, AI, AO, EO, C_IN, BI, OI, CE, CO : STD_LOGIC;

BEGIN

    -- Program Counter Instance
    U_PC: ENTITY WORK.ProgramCounter
        PORT MAP (
            CLK         => CLK,
            CLR         => RESET,
            INC         => CE,
            LOAD        => CO,
            ADDR_IN     => MAR_Out,
            ADDR_OUT    => PC_Out
        );

    -- Memory Address Register Instance
    U_MAR: ENTITY WORK.MemoryAddressRegister
        PORT MAP (
            CLK         => CLK,
            CLR         => RESET,
            LOAD        => MI,
            ADDR_IN     => PC_Out,
            ADDR_OUT    => ADDR_OUT
        );
```

```
52      -- RAM Instance
53      U_RAM: ENTITY WORK.RAM
54          PORT MAP (
55              CLK         => CLK,
56              WE          => RI,
57              OE          => RO,
58              ADDR        => ADDR_OUT,
59              DATA_IN     => ACC_Out,
60              DATA_OUT    => DATA_OUT
61          );
62
63      -- Instruction Register Instance
64      U_IR: ENTITY WORK.InstructionRegister
65          PORT MAP (
66              CLK         => CLK,
67              CLR         => RESET,
68              LOAD        => II,
69              DATA_IN     => DATA_OUT,
70              DATA_OUT    => IR_Out
71          );
72
73      -- Output Register Instance
74      U_OR: ENTITY WORK.OutputRegister
75          PORT MAP (
76              CLK         => CLK,
77              CLR         => RESET,
78              LOAD        => OI,
79              DATA_IN     => ACC_Out,
80              DATA_OUT    => DATA_OUT
81          );
82
83      -- Accumulator Register Instance
84      U_ACC: ENTITY WORK.AccumulatorRegister
85          PORT MAP (
86              CLK         => CLK,
87              CLR         => RESET,
88              LOAD        => AI,
89              DATA_IN     => ALU_Out,
90              DATA_OUT    => ACC_Out
91          );
92
93      -- B Register Instance
94      U_BReg: ENTITY WORK.BRegister
95          PORT MAP (
96              CLK         => CLK,
97              CLR         => RESET,
98              LOAD        => BI,
99              DATA_IN     => DATA_OUT,
100             DATA_OUT    => B_Out
101         );
102
```

```
        );

    -- ALU Instance
    U_ALU: ENTITY WORK.ALU16BitBuffered
        PORT MAP (
            A         => ACC_Out,
            B         => B_Out,
            CLK       => CLK,
            FLAG_CLR  => RESET,
            FI        => '1',
            C_IN      => C_IN,
            S         => Control_Signals,
            EO        => EO,
            G         => ALU_Out,
            CF        => ALU_CF,
            ZF        => ALU_ZF
        );

    -- Control Unit Instance
    U_Control: ENTITY WORK.ControlUnit
        PORT MAP (
            Step    => Step,
            Opcode  => Opcode,
            MI      => MI,
            RI      => RI,
            RO      => RO,
            IO      => IO,
            II      => II,
            AI      => AI,
            AO      => AO,
            EO      => EO,
            C_IN    => C_IN,
            BI      => BI,
            OI      => OI,
            CE      => CE,
            CO      => CO,
            S       => Control_Signals
        );

    -- Signal Assignments
    Opcode <= IR_Out(15 DOWNTO 12);
    Step <= "000";

    HLT <= '1' WHEN Opcode = "0000" ELSE '0';

END Behavioral;
```

Figure 60 - Top Level VHDL for Approach 02

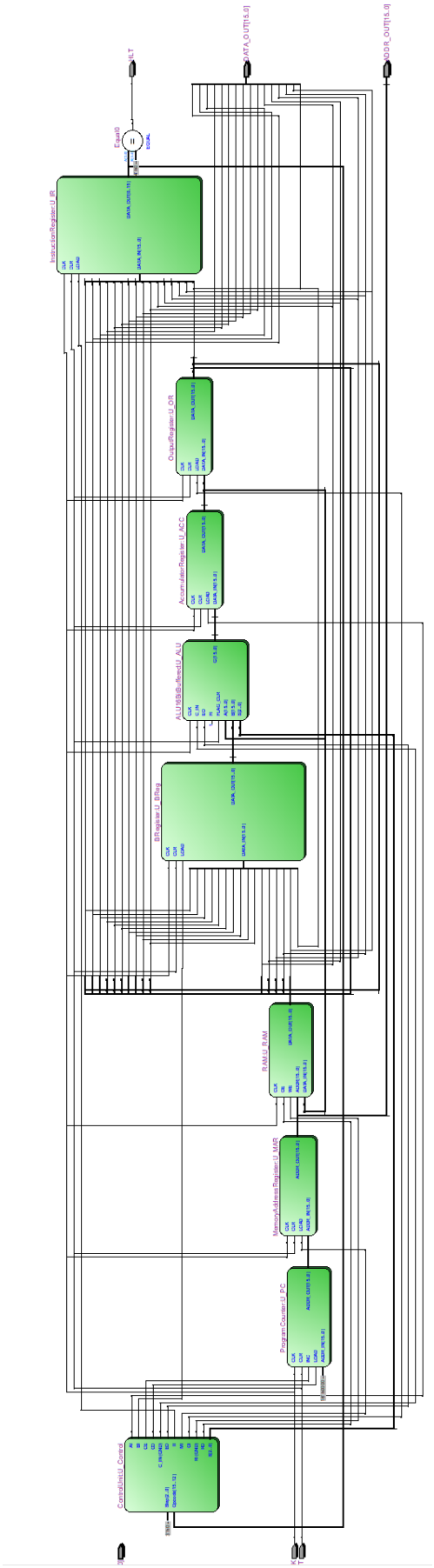


Figure 61 - Approach 02 CPU design

Appendices – Video Link

https://youtu.be/qW-HjAeE-Ps?feature=shared

References

[1]

Gillis, “What is Computer Memory and What are Different Types?,” *WhatIs.com*, Oct. 2020. https://www.techtarget.com/whatis/definition/memory

[2]

K. Gallo, “What is I/O (Input/Output) in Computing? | Built In,” *builtin.com*, Mar. 02, 2023. https://builtin.com/hardware/i-o-input-output

[3]

“What is data structure? Definition, types, examples,” *AltexSoft*. https://www.altexsoft.com/blog/data-structure/

[4]

“Computer Architecture: Components, Types, Examples | Spiceworks,” *Spiceworks*. https://www.spiceworks.com/tech/tech-general/articles/what-is-computer-architecture/#_001

[5]

“VarTech Systems Articles | Computer Processing Unit (CPU) History,” *www.vartechsystems.com*. https://www.vartechsystems.com/articles/cpu-history

[6]

University of Sunderland, “What is computer architecture?,” *University of Sunderland*, Aug. 26, 2021. https://online.sunderland.ac.uk/what-is-computer-architecture/

[7]

R. Groves, “Brief History of Computer Architecture Evolution and Future Trends.” Available: https://cds.cern.ch/record/399391/files/p147.pdf?version=1

[8]

“Von Neumann Architecture - an overview | ScienceDirect Topics,” *Sciencedirect.com*, 2018. https://www.sciencedirect.com/topics/computer-science/von-neumann-architecture

[9]

GeeksforGeeks, “Harvard Architecture,” *GeeksforGeeks*, Apr. 30, 2020. https://www.geeksforgeeks.org/harvard-architecture/

[10]

“Designing and Implementing a SAP-1 Computer,” *SAP-1-Computer*. https://karenok.github.io/SAP-1-Computer/

[11]

“What is RISC architecture?,” *Educative: Interactive Courses for Software Developers*. https://www.educative.io/answers/what-is-risc-architecture

[12]

F. Iozzi, S. Saponara, A. J. Morello and L. Fanucci, "8051 CPU core optimization for low power at register transfer level," Research in Microelectronics and Electronics, 2005 PhD, Lausanne, Switzerland, 2005, pp. 178-181, doi: 10.1109/RME.2005.1542966.

[13]

M. H. Rahmad, Soo Saw Meng, E. K. Karuppiah and Hong Ong, "Comparison of CPU and GPU implementation of computing absolute difference," 2011 IEEE International Conference on Control System, Computing and Engineering, Penang, 2011, pp. 132-137, doi: 10.1109/ICCSCE.2011.6190510.

[14]

D. Chen, "An Overview Method of Viable Dependability for RAMS in Life Cycle," 2018 Annual Reliability and Maintainability Symposium (RAMS), Reno, NV, USA, 2018, pp. 1-7, doi: 10.1109/RAM.2018.8463055.

[15]

Lenovo, [Online]. Available: https://www.lenovo.com/us/en/glossary/program-counter/?orgRef=https%253A%252F%252Fwww.google.com%252F.

[16]

R. Sheldon, “program counter,” Tech Target, [Online]. Available: https://www.techtarget.com/whatis/definition/program-counter.

[17]

S. Gupta, “Synchronous Counter,” Circuit Digest, 21 August 2018. [Online]. Available: https://circuitdigest.com/tutorial/synchronous-counter.

[18]

“Introduction of Control Unit and its Design,” Geeks for Geeks, 3 October 2023. [Online]. Available: https://www.geeksforgeeks.org/what-is-program-counter/.

[19]

A. Volle, “Control Unit,” Britannica, [Online]. Available: https://www.britannica.com/technology/control-unit.