ASIA PACIFIC UNIVERSITY OF TECHNOLOGY & INNOVATION

VLSI Design

INDIVIDUAL ASSIGNMENT

NAME & STUDENT ID	Su Xin Hong TP061159
INTAKE	APD3F2305CE
LECTURER	Dr. Jeevan Kanesan
HAND IN DATE	2nd September 2023

Table of Contents

List of Figures

List of Tables

Introduction

In the dynamic landscape of modern computing, digital operation is the core of every advanced system. Digital datapath circuits act as the central nerve of computation system that functions to seamlessly process and manipulate binary data through arithmetic and logical operations. The ability to rapidly execute arithmetic computation and logical evaluation is highly valuable and appreciated. Whether it is a simple addition or a complex bitwise operation, the agility of digital system depends on its underlying datapath design. Register-Transfer Level (RTL) technique plays a vital role as a methodology which offers a structured framework for conceptualizing and implementing datapath operations.

The aim of this assignment is to discover the fundamental principles, operating components, concepts of datapath circuits and also the different types of datapath circuits. Besides that, this assignment also aims to apply the theoretical concepts onto practical designs. By selecting and integrating digital components such as arithmetic logic units (ALU), adders, subtractors, multiplexers, demultiplexers, logical gates and registers, a design of datapath circuit which are capable of performing operations such as arithmetic and logical functions can be designed. In this assignment an ALU that is capable of doing both addition and subtraction according to the control signal will be designed. The ALU that will be designed will only be able to process 1 bit of data at a time. In addition, a suitable datapath will also be designed by integrating the ALU with other memory components such as registers to complete an effective datapath that allows the ALU to process the data bit by bit in a loop without the need of multiple ALU components.

The design of the datapath will be done using RTL technique by utilizing VHDL code. Then the digital datapath circuit will also be simulated and analyzed using Intel Quartus Prime software and simulation results will be displayed.

Objectives

1. Research on concept of datapath circuits.
2. Research on different datapath components.
3. Research on different datapath types.
4. Design datapath circuit by using RTL techniques and utilizing digital components.
5. Simulate and analyze the designed datapath.

Research On Datapath Circuit Designs

Digital Datapath Circuits

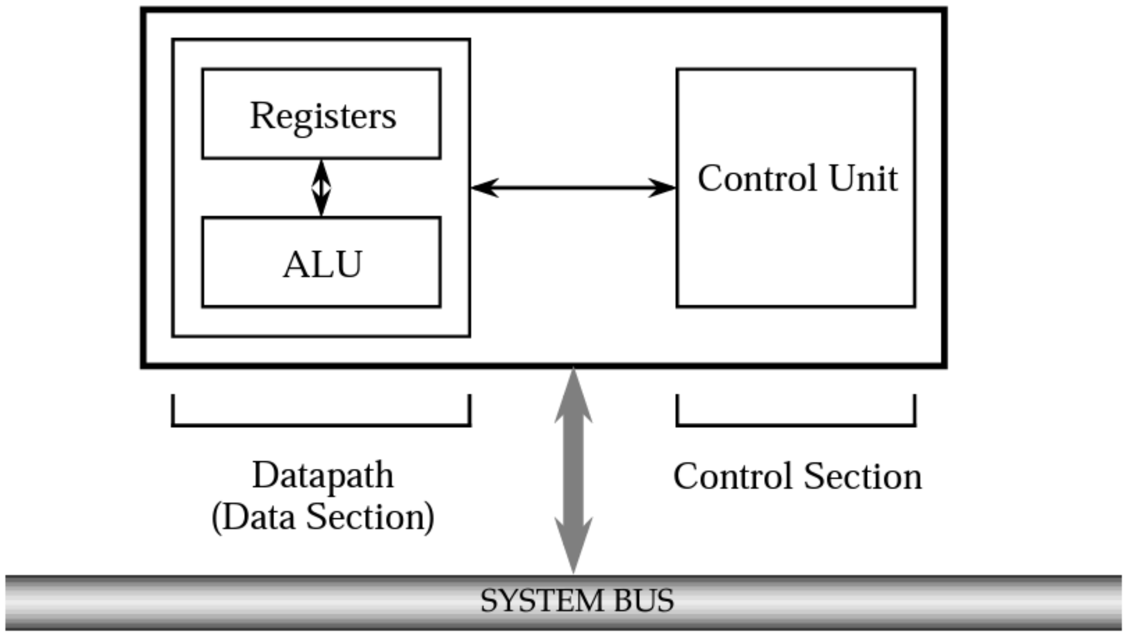


Figure 1: Datapath

Digital datapath circuit is a fundamental component in digital systems. Digital datapath circuit is responsible for performing arithmetic and logical operations on data. Digital datapath circuits are made up of a variety of components such as registers, arithmetic logic units (ALU), multiplexers, demultiplexers, comparators, shifters, rotators and data manipulation units. All of the components are interconnected with each other using buses to process and manipulate digital data according to the instruction provided by the control unit.

Datapath circuits are designed to perform arithmetic operations such as addition, subtraction, multiplication, division and logical operations like AND, OR, NOT and so on. efficiently. This enables digital data to be processed at high speeds and reduces processing time. Besides that, different applications have different computational requirements. Datapath circuits can be customized to optimize performance for specific tasks, ensuring that a digital system meets the requirement of its intended use. Thus, an efficient datapath design can ensure that resources, such as functional units, registers and buses are used efficiently. This also reduces wastage of hardware resources and lowers manufacturing costs.

In essence, digital datapath circuits are responsible for carrying out the core processing tasks that enable digital devices to perform a wide range of functions efficiently and accurately while datapath design is very vital to create an efficient and high-performance digital system.

Registers

Digital datapath circuits make use of registers as a critical component. Registers are storage units that hold data temporarily when processing takes place. Registers can store data like intermediate and final results of operations, as well as operands. Examples of registers include status registers, general-purpose registers and accumulator registers. Moreover, they receive data from different sources in the datapath. ALUs, memory, external inputs and registers are among the sources of data. These registers can also output stored data to be passed on for other processes within the circuit.

In addition, registers are usually synchronized to a clock signal in the datapath. Data transfer to and from registers only occurs at specific times according to the rising or falling edge of the clock signal. This synchronization ensures the stability and reliability of data during data transmission. This synchronization also effectively prevents issues such as data corruption and failures. In some cases, the registers are also controlled by signals from the datapath controller. Control signals from the control unit determine when data should be loaded into registers, when data should be read from registers, and when data should be processed by other components such as the ALU or other data manipulation units.

In some datapath designs, registers have been used to create pipeline stages. A pipeline functions to divide statements into stages. Each stage has its own registers to store the data for that stage. This pipeline allows multiple instructions to be processed at the same time, which means that different instructions can be executed at the same time. This is also commonly known as concurrent processing. This effectively increases throughput and efficiency.

Registers can be read and written. Data in the registers can be read out and sent to other parts for processing. Likewise, new data can be written to registers to replace old data. Registers can store multiple bits of data ranging from a few bits to many bits. It depends on design needs.

The bit width of registers affects the range of values they can store and the precision of calculations they can perform.

Control Unit

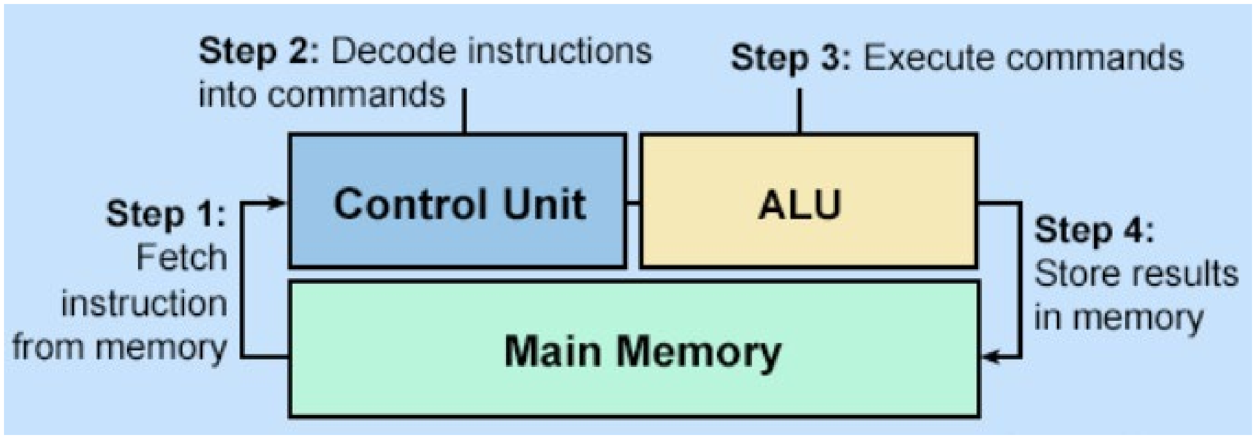

Step 2: Decode instructions into commands
Step 3: Execute commands
Step 1: Fetch instruction from memory
Control Unit
ALU
Step 4: Store results in memory
Main Memory

Figure 2: Control Unit (Computer Hope, 2020)

Figure 2 shows an example of the function of a control unit and its workflow. A control unit (CU) is a component within a computer's central processing unit (CPU) that controls the execution of instructions and coordinates the activities of various different hardware components such as registers, ALUs and so on to carry out the instructions (Computer Hope, 2020). The main function of the control unit is to fetch, decode and execute instructions from the computer's memory. Control unit performs tasks such as instruction fetching, instruction decoding, instruction execution, control signal generation, memory management, instruction pipelining, error handling and so on.

In instruction fetching, the control unit retrieves the next instruction from the computer's memory (RAM). Once the instruction is fetched, the control unit starts to decode the instruction in order to understand what operation needs to be performed and what operands are involved. After decoding the instructions, the control unit coordinates the execution of the instruction. This process involves sending signals to various parts and components in the CPU and other hardware components to perform the necessary calculations and data manipulations. Then the control unit generates control signals that determine the activities of different parts of the CPU, including the arithmetic logic unit (ALU), memory, and input or output devices (Geeks For Geeks, 2023). These signals ensure that the appropriate operations are carried out during the execution of an instruction.

Furthermore, the control unit also handles memory access to ensure that data is correctly read from and written to memory locations as needed by the instructions being executed. Many modern CPUs use a technique called instruction pipelining, where multiple instructions are in various stages of execution simultaneously. Which is also known as concurrent processing. The control unit manages the pipeline stages to maximize CPU throughput and efficiency. Besides, the control unit may detect and handle errors, such as arithmetic overflow, invalid instructions, or other exceptional conditions that arise during program execution.

In short, the control unit is a vital component that plays the role as a brain of a CPU. It coordinates the activity of all subunits in a datapath and ensures that the instructions are executed in a correct manner and sequence.

Arithmetic Logic Unit (ALU)

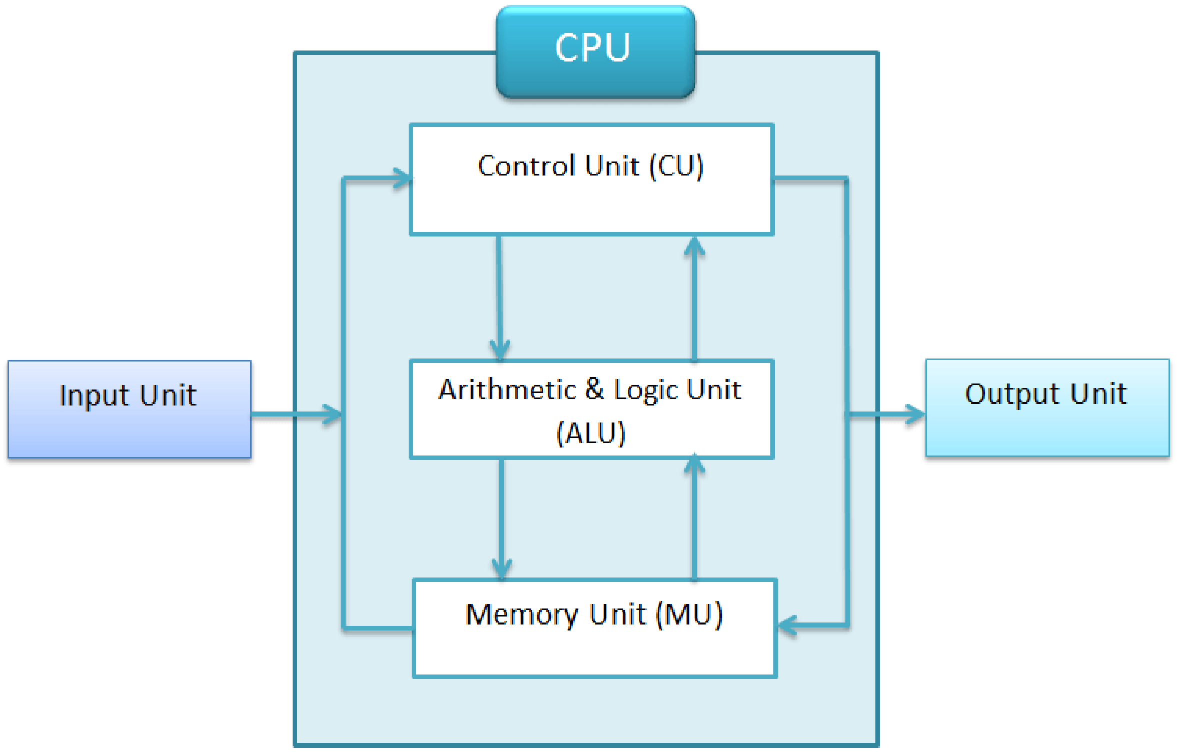


Figure 3: Arithmetic Logic Unit (Mallick, 2023)

The arithmetic logic unit (ALU) is a key component in digital data path circuitry that performs arithmetic and logic operations on binary data. It is often considered the "heart" of the central processing unit (CPU) in a computer, or the processing unit in various digital systems. The ALU takes operands from registers, performs some operations and returns the results to registers.

The ALU can perform various arithmetic operations, including addition, subtraction, multiplication, and division (Mallick, 2023). These operations are performed on the binary representation of numbers stored in registers. The ALU receives two operands from registers and then performs an operation and finally produces a result. In addition to performing arithme-tic operations, the ALU is responsible for executing logical operations like AND, OR, XOR, and NOT. These operations manipulate individual bits of data and produce Boolean results.

ALU is also capable of operand selection. In datapath, the ALU chooses operands from registers based on control signals given by the control unit. These control signals dictate which registers are used as inputs for each operation. Besides that, the width of the ALU determines how much data it can process in one operation. If an ALU is 32 bits wide, then it can perform arithmetic and logical operations on 32-bit binary numbers.

During arithmetic operations, the ALU will frequently generate carry and overflow flags. Carry flags indicate whether there is a carry-out from the most significant bit during addition or a borrow during subtraction. Overflow flags, on the other hand, indicate if the result of an operation exceeds the capacity to be accurately represented within the given bit width.

In addition to performing calculations, ALUs can also generate flags to provide relevant information (Charles, n.d.). For example, the Zero Flag indicates if the result of an operation is zero, while the Sign Flag indicates whether the sign bit of the result is negative. These flags play a crucial role in guiding decision-making processes within control logic. ALUs also have the ability to perform conditional operations. These operations allow for the setting of a register to a specific value only if a certain condition is met. This feature is commonly utilized in program execution for branching and decision-making purposes.

ALUs can also support conditional operations such as setting a register to a value only if a certain condition is met. This is often used in branching and decision-making during program execution. On top of that, ALUs can manipulate individual bits of data, which is useful for tasks like extracting specific fields from data or setting or clearing individual bits. It is also important to note that Some ALUs offer shift and rotate operations, allowing data to be shifted left or right, as well as rotated through the carry bit. These operations are essential for tasks like multiplication and division.

In short, ALU plays the role of executing arithmetic and logical operations on binary data. While the operands were taken from the registers and performs computations according to control signals. The results can be stored back into registers or for performing further processing.

Buses

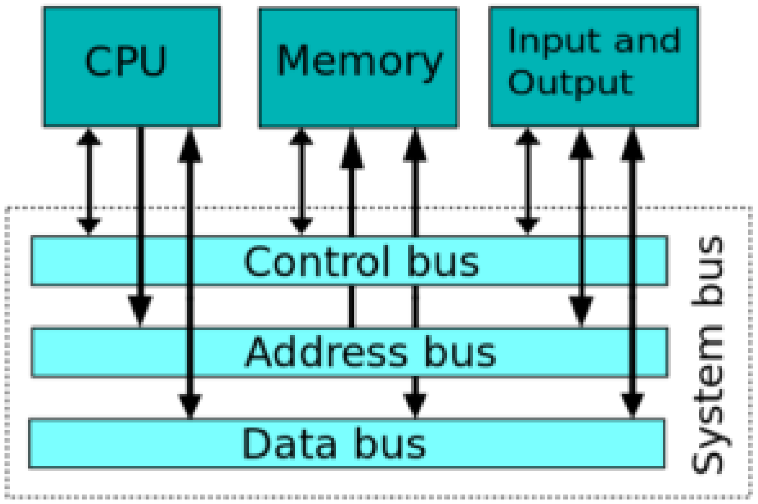


Figure 4: Multiple Bus Structure (Thornton, 2016)

Buses are also a very vital design in digital datapath circuits. Buses' main function is to enable the transfer of data, addresses and control signals between various components in a datapath circuit, computer or other digital devices. Buses play the role of communication pathways that allow different parts of a system to exchange information. Figure 4 shows the structure of a multiple bus. There are a total of 3 different types of internal buses that transfer different types of data which include data bus, address bus and control bus (Thornton, 2016).

First and foremost, data bus is a bidirectional pathway which is utilized to transfer data between various different components in a datapath (Rouse, 2023). Different components include registers, ALUs, memory units and so on. Data bus carries binary data in parallel with each of the lines or wire in the bus representing one single bit of data. The number of bits that can be transferred simultaneously is determined by the width of the data bus. This affects the amount of data that can be processed at one time. Besides that, the address bus is another bidirectional pathway that carries the address of memory location or registers. The addresses are used to specify the source and destination of data for memory read and write operations. The maximum addressable memory space of the system is also determined by the width of the address bus. While the control bus is an either unidirectional or bidirectional pathway that carries the control signal or commands that control the operation of components in a digital datapath circuit. Control signals give commands such as read or write data, register select, clock synchronization, enable or disable operation and so on. Other than communicating with the internal components in a datapath circuit, some buses can also communicate with external circuits. Complex digital systems may have multiple levels of buses, each serving different purposes. For example, a local bus may connect components within a single chip, while a system bus connects different chips or modules in a larger system.

Buses also use multiplexers to select the source of data to be transferred. Multiplexers allow multiple input of data and can select one of the inputs as a single output. The selection will be determined by the control signal. This is very important for directing data from different sources to the bus. While demultiplexers are also used in the bus to distribute data from a single input line to multiple output lines based on control signals. They are used when data from a bus needs to be directed to different destinations.

In systems with multiple components sharing a single bus, bus arbitration is used to manage access to the bus. This ensures that only one component uses the bus at a time and prevents conflicts in data. In addition, to prevent conflict or corruption in data, buses often operate in synchrony with a clock signal. Data transfers occur at specific clock edges, ensuring that data is stable and reliable during transmission.

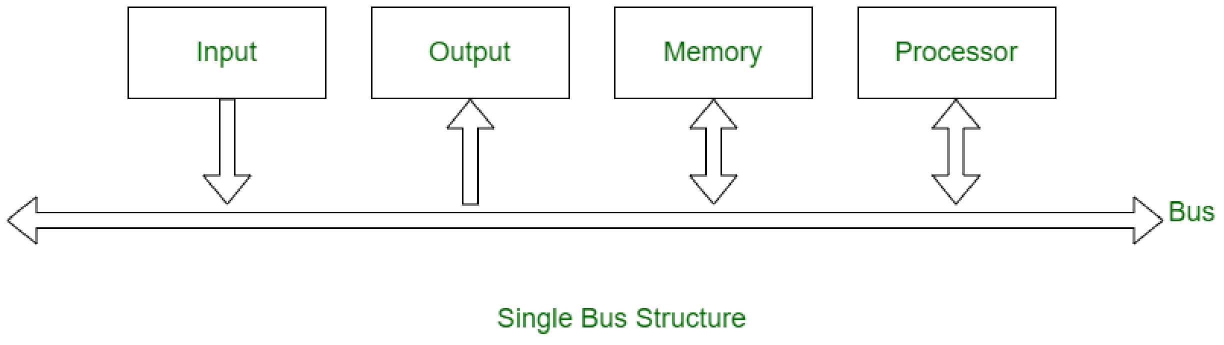


Figure 5: Single Bus Structure (Geeks For Geeks, n.d.)

Figure 5 shows another bus structure which is the single bus structure. A single bus structure is where all components only 1 bus to communicate with each other. In this case, the performance of the datapath circuit will be worse. This is because only one component can use a bus at a time, hence other components have to wait for other components to finish its data transferring process before they can transfer. This will reduce the throughput and increase the waiting time. Nevertheless, a single bus is less expensive and simpler compared to multiple bus structures.

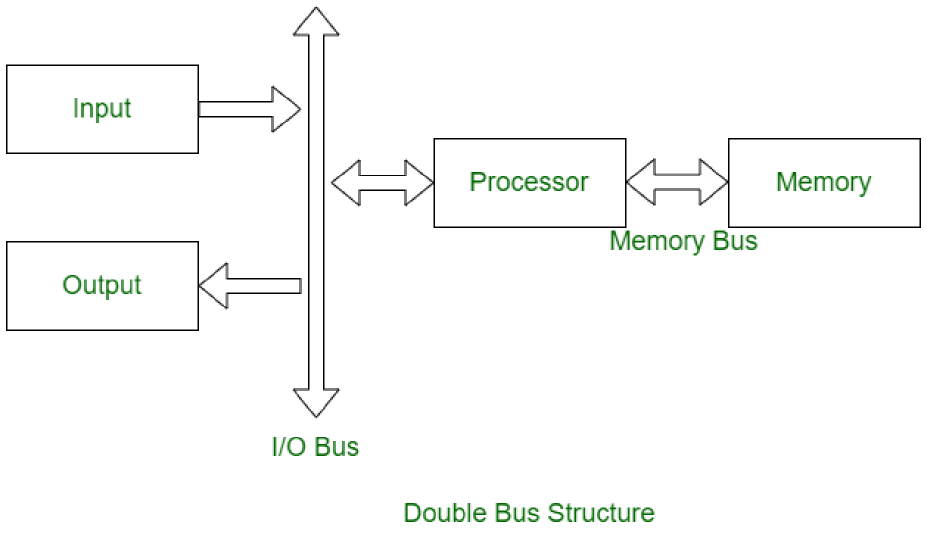


Figure 6: Double Bus Structure (Geeks For Geeks, n.d.)

Figure 6 shows the double bus structure. There are a total of 2 buses which are the input output bus and the memory bus. One of the buses is used to transfer instructions while the other is used to transfer the data required for the execution of instructions. In this case, the performance of a double bus structure will be better than the single bus structure because the load had been separated into 2 buses. However, it still has a lower performance than the multiple bus structure.

As summary, buses in digital datapath circuits act as communication pathways, allowing different components to exchange data and control signals. They enable the coordination and synchronization of operations within the circuit which contributes to the efficient and reliable functioning of digital systems.

Datapath Types

Pipeline Design Datapath

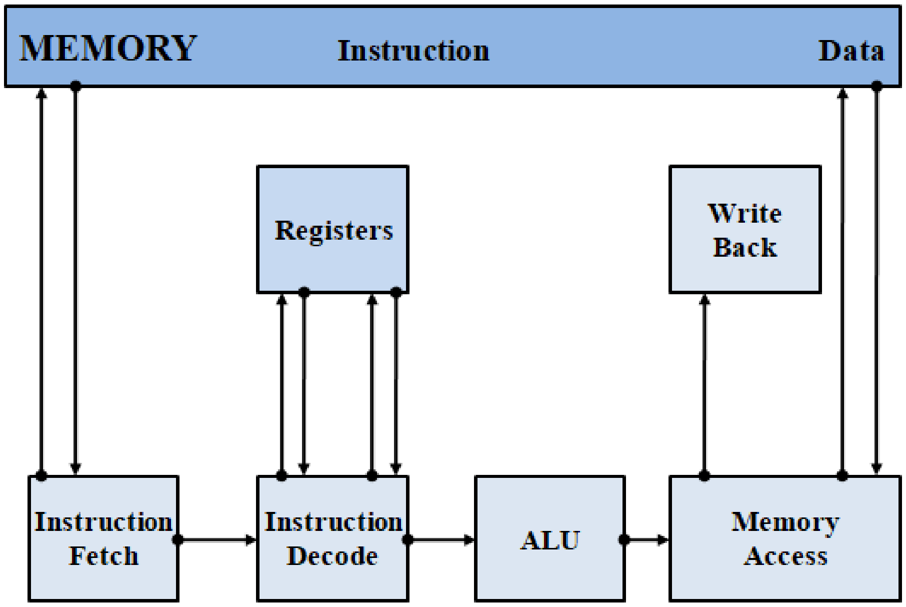


Figure 7: Pipeline Datapath (Indira et al., 2019)

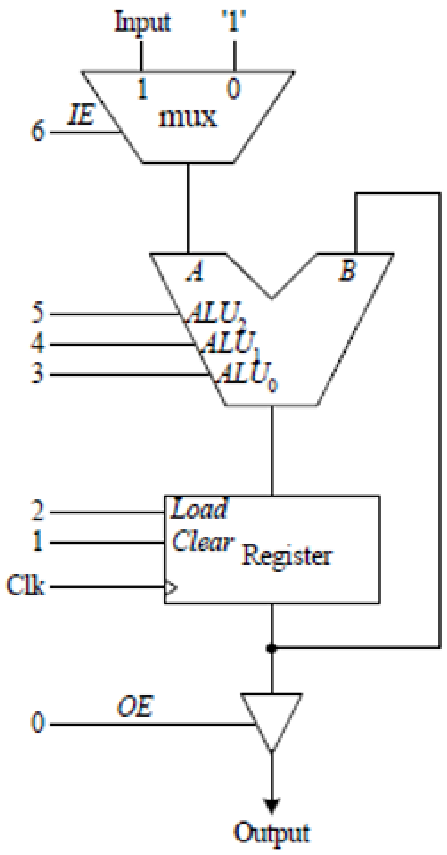
In the context of digital datapath circuits, pipeline design is a method of organizing the execution of commands and instruction to achieve better performance, efficiency and a higher throughput. Pipeline design technique is a technique that overlaps the execution of multiple instructions in order to make better use of the hardware resources and reduce the overall time required to complete a sequence of operations. This is also known as the concurrent processing as the instructions are carried out simultaneously.

In a pipeline design, the execution of an instruction is divided into multiple stages, and each stage corresponds to a specific operation or phase of the instruction's execution. Different instructions are processed in parallel, with each instruction moving through the pipeline stages one after another. As soon as one stage of an instruction is completed, the next instruction can enter that stage. This overlapping of stages allows the processor to be working on multiple instructions at various stages of execution simultaneously. 5 basic standard stages in pipeline design includes the instruction fetch stage, instruction decode stage, execution stage, memory access stage and write back stage (Indira et al., 2019). By breaking down instruction execution into these stages, the pipeline can process multiple instructions that are on different stages simultaneously.

Nevertheless, pipeline design can give rise to some potential hazards. Firstly, structural hazards occur when multiple instructions need to use the same hardware resource at the same time. For instance, if one instruction is using the memory unit, another instruction that also requires memory access might have to wait. Besides that, data hazard is also one of the hazards that might arise when one instruction depends on the result of a previous instruction that hasn't been completed yet. These hazards require careful handling to avoid incorrect results. Lastly, control hazard is the branching of instructions that can change the flow of the program. The pipeline needs a mechanism to predict branches and manage any incorrect predictions.

In short, pipeline design is a vital and significant concept in computer architecture. It allows processors to achieve higher throughput and improved performance by parallelizing instruction execution. However, it requires careful consideration of various challenges and trade-offs to ensure correct operation and optimal efficiency.

General Datapath



ALU_2	ALU_1	ALU_0	Operation
0	0	0	Pass through *A*
0	0	1	*A* AND *B*
0	1	0	*A* OR *B*
0	1	1	NOT *A*
1	0	0	$A + B$
1	0	1	$A - B$
1	1	0	$A + 1$
1	1	1	$A - 1$

Figure 8: General Datapath Example (Hwang, 2003)

Figure 8 shows an example of a general datapath. The general datapath contains only 1 ALU and one register for the purpose of storing data. Operand A input of the ALU accepts the input form a 2 to 1 multiplexer with its selector signal IE. While operand B is the output from the register. There are a total of 3 control inputs ALU_0, ALU_1 and ALU_2 which controls the operation of the ALU according to the table. There are also controls available in the register to control whether to load the output of the ALU into the register or clear the register. The register is also clock synchronized. Then the content of the register can be sent as an output by input 1 into the OE which controls the tri-state buffer.

In a general datapath design, the hardware components are designed to handle a broader range of instructions and tasks. This approach provides greater flexibility because the datapath can execute a variety of instructions and support a wide range of algorithms. A general datapath is typically found in general-purpose processors, such as the CPUs in most computers.

A general datapath often includes more complex components, such as a more versatile ALU, flexible register files and additional control logic. This complexity allows it to support a wider variety of instructions and algorithms, but it can also lead to longer execution times for certain operations compared to a dedicated datapath optimized for those specific operations.

There are several advantages of general datapath which includes versatility where the same hardware can handle a wide range of tasks. This makes the processor suitable for general-purpose computing. Besides that, general datapath also has better adaptability where it can support new instructions or algorithms through reconfiguration or microprogramming. General datapath also enables resource sharing. This is because the components are shared across different types of instructions, hence the hardware resources can be used more efficiently.

Dedicated Datapath

When a general datapath does not contain the required or sufficient registers, ALUs or any other functional units, a dedicated datapath must be designed to perform the desired tasks or operations. In dedicated datapath design, the hardware components are specifically optimized to handle a particular set of instructions or tasks. This means that the datapath is tailored to perform a limited range of operations efficiently. Each instruction is implemented with a dedicated circuitry path, which can lead to high performance for the specific set of tasks the datapath is designed for.

For example, in a digital signal processor, it is optimized for signal processing tasks like audio or video processing. Thus, the datapath might have dedicated multiply-accumulate (MAC) units to efficiently perform the types of computations commonly found in signal processing algorithms.

There are also several advantages of dedicated datapath which include high performance on specific tasks. This is because the dedicated datapath is tailor made and its hardware will be optimized for a specific objective or tasks. Hence, high throughput can be achieved easily compared to using a general datapath to carry out the same tasks. In addition, the complexity of a dedicated datapath can have a simpler and more straight forward design since its function is only for carrying out a specific task. Moreover, the dedicated design also usually leads to a more consistent and predictable execution time for the specific task or operation. On top of that, since the components in a dedicated datapath are also tailormade to satisfy a certain requirement to carry out specific task which results in no redundant components. This will increase its efficiency in terms of components and power consumptions.

Research Summary

Table 1: Summary Table

Research	Summary
Registers	• Temporary storage units within a digital system's datapath. • Store data temporarily, holding operands, intermediate results or final outcomes of operations within a digital circuit. • Providing data for computations within components like Arithmetic Logic Units (ALUs) • Receive data from various sources and can send their contents to other parts of the circuit for further processing or output. • Synchronized with a clock signal, ensuring orderly data transfers and avoiding glitches. • Read, write or manipulate data within registers based on the signal from control unit. • Some registers store status flags, like overflow or zero results, influencing control decisions during execution. • In pipelines, registers represent stages where instructions are processed concurrently, enhancing throughput. • Registers can hold memory addresses used for data access or program execution.
Control Unit	• Control unit decodes instructions fetched from memory, determining the necessary operations to perform. • Generates control signals that guide components like registers, ALUs and memory units to carry out specific tasks during instruction execution.

	• Synchronizes operations with the clock signal, ensuring that actions occur at specific clock edges to maintain data integrity. • Manages the interaction between different datapath components, instructing them when to perform read or write operations, arithmetic or logical computations and data transfers.
Arithmetic Logic Unit (ALU)	• Performs arithmetic operations like addition, subtraction, multiplication and division on binary numbers. • Executes logical operations like AND, OR, XOR, and NOT on individual bits. • Receives operands from registers and produces results that can be stored back in registers or used for further processing. • Operates on binary data, using simple electronic circuits to perform complex mathematical and logical operations. • ALU performs operations and handles the results according to signal from control unit. • ALU can produce flags indicating conditions such as overflow, carry, zero and negative results from arithmetic operations. • Multiplexers and bypassing mechanisms can be used to select operands directly from the ALU's outputs or from registers to improve efficiency. • Some ALUs include capabilities for shifting, rotating, and bit-level manipulation to support various data manipulation tasks.

Buses	• Buses transmit binary data and control signals between different hardware components, such as registers, memory, ALUs and input and output devices. • Different types of bus structure which include single, double and multiple. • Multiple bus structure has best performance. • Common types in multiple buses include data buses, address buses, and control buses, each serving distinct purposes in a digital system. • Data Bus: Carries actual data in parallel bit lines, allowing multiple bits to be transferred simultaneously. • Address Bus: Transmits memory or register addresses to specify data sources or destinations. • Control Bus: Carries control signals that guide the timing and execution of operations within the system, such as read or write signals and clock synchronization. • Buses often use multiplexers (MUX) to select data sources and demultiplexers (DEMUX) to distribute data to different destinations.
Pipeline Design Datapath	• Instructions are divided into stages, and each stage performs a specific task. • Different instructions can occupy different stages concurrently. • Common pipeline stages include Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB). • Pipelining enables a continuous flow of instructions, reducing idle time and maximizing hardware utilization.

	• Stages of one instruction can overlap with stages of other instructions, leading to improved throughput and better resource utilization. Advantages • Increased throughput. • Faster execution because different stages of the pipeline can work simultaneously on different instructions. • Better resource utilization because different pipeline stages can be dedicated to specific tasks, ensuring that hardware resources are effectively utilized during each clock cycle. • Pipelining divides the execution of instructions into stages, allowing for a shorter cycle time compared to non-pipelined designs. Disadvantages • Structural hazards, data hazards, and control hazards can slow down the pipeline and require additional logic for resolution. • Increased complexity because pipelining requires careful design to manage data dependencies, control flow, and resource allocation. Handling hazards can introduce additional complexity in the control circuitry.
General Datapath	General datapath is a versatile design approach used in computer architecture to support a wide range of instructions and tasks within a digital system. Advantages • Can execute a wide range of instructions and algorithms, making them suitable for general-purpose computing tasks.

	• Support various software applications without major hardware modifications. • Components are shared across instructions, optimizing hardware utilization and reducing redundancy. • General datapath can accommodate new instructions and technologies through microprogramming or firmware updates. Disadvantages • General datapath may not achieve the same level of performance optimization as dedicated datapath designed for specific tasks. • Instructions may experience varying latencies, affecting overall performance due to complexity and diverse operations.
Dedicated Datapath	Dedicated datapath is a design approach in computer architecture where hardware components are optimized for a specific set of instructions or tasks. Advantages • Dedicated datapath are specifically designed to excel at a particular set of operations, resulting in better performance for those tasks. • The hardware is tailored for specific operations, reducing unnecessary components and minimizing power consumption. • Execution time of instructions can be more predictable and consistent. • A dedicated datapath can lead to simpler and more straightforward hardware design compared to a general-purpose design.

	Disadvantages • Dedicated datapath excels at specific tasks but might perform poorly or inefficiently for other operations. • Adapting the hardware for new instructions or tasks requires significant modifications or even redesign.

Circuit Design

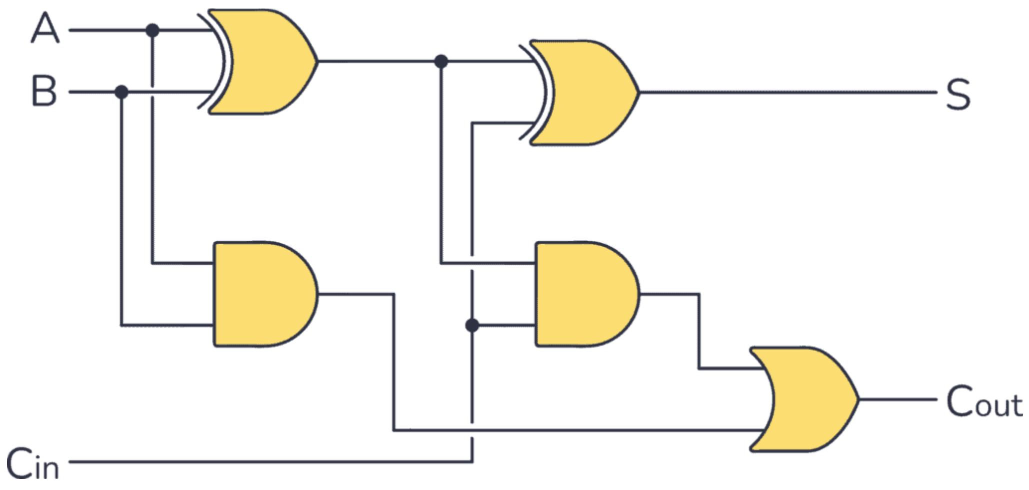


Figure 9: Full Adder Logic Circuit (Urias, 2022)

Figure 9 shows a full adder circuit which consisted of 2 XOR gates, 2 AND gates and 1 OR gate. A full adder is designed to add three binary inputs: two single-digit binary numbers (A and B) and a carry-in (Cin). Its output provides the sum (S) of the inputs and a carry-out (Cout) signal, which indicates whether the addition resulted in a carry-over. The sum is computed using the XOR logic gate, while the carry-out considers both AND and OR gates to manage the carry.

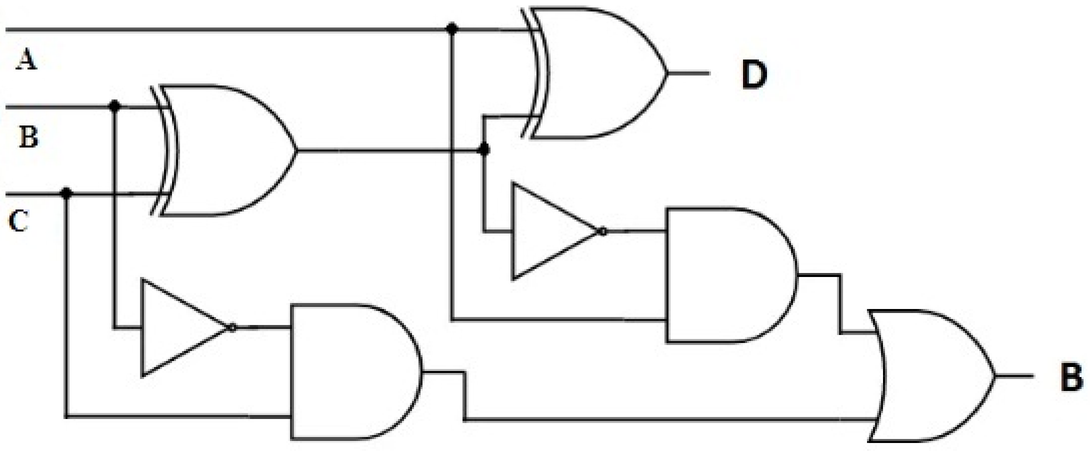


Figure 10: Full Subtractor Circuit (WatElectronics, 2020)

Figure 10 shows a full subtractor circuit which consisted of 2 XOR gates, 2 NOT gates, 2 AND gates and 1 OR gate. A full subtractor performs subtraction of three binary inputs which are the minuend (A), the subtrahend (B) and a borrow-in (Bin) which is C in Figure 10. Its outputs are the difference (D) between A and B, and a borrow-out (B) signal, which indicates whether borrowing needs to occur during the subtraction. Similar to the full adder, the XOR gate computes the difference, while the borrow-out considers AND and NOT gates to manage the borrowing process.

The number of gates that are used in Figure 9 and Figure 10 are very much and it is only for the operation of 1 bit adder and subtractor. To carry out a multiple bit operation, the same circuit will have to be copy pasted which causes more use of components. As an example, to carry out a 4-bit adder operation, 4 of the components in Figure 9 will be required.

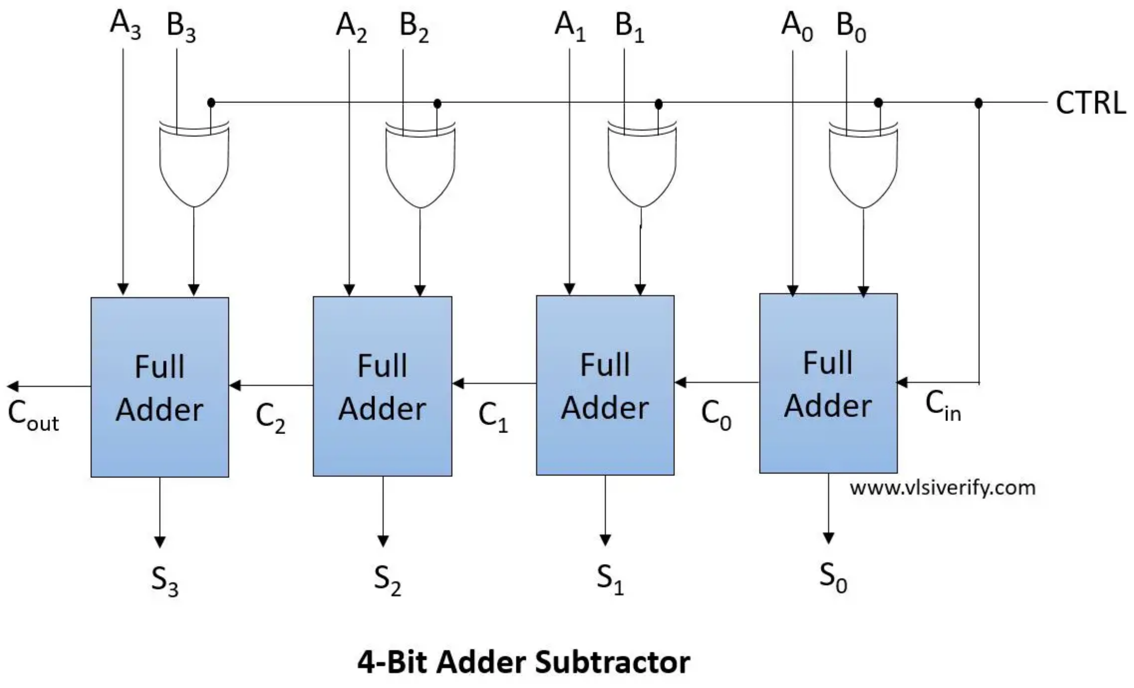


Figure 11: 4-Bit Adder Subtractor Logic Circuit (VLSI Verify, n.d.)

Figure 11 shows a 4-Bit Adder/Subtractor Logic circuit. It has the ability to perform both addition and subtraction operations on 4-bit binary numbers. Two 4-bit inputs, A and B along with a control signal will be input into the circuit. The function of the control signal is to determine whether the operation should be addition or subtraction where 0 will indicate addition while 1 will indicate subtraction. There are only a total of 4 full adders and 4 XOR gates which can carry out both addition and subtraction operation according to the control input. To implement both addition and subtraction, the circuit needs to handle the subtraction operation when the control signal

(CTRL) is 1. When control is 1, subtraction is performed by adding the 2's complement of B to A. The 2's complement of a binary number can be obtained by inverting all its bits and then adding 1 to the result which is performed in the logic circuit by inputting it as carry in (Cin).

Even this is more effective than the circuit in Figure 9 and Figure 10, there are still some improvements can be done to reduce the number of components which is by implementing datapath design. Datapath design can help reduce the amount of logic gates required in a digital circuit, especially in arithmetic and logic operations. A well-optimized datapath design takes account of the arrangement of functional units, data paths and control signals to minimize the overall complexity and improve performance. Hence a design of datapath will be created to carry out multiple bit addition and subtraction more efficiently. Only one full adder is required to create the datapath design where the bits will be input into the adder one by one and with its carry passing to the next stage for the addition of the next bit. This is also the same for the subtraction operation since the subtraction operation is done by adding the 2's complement.

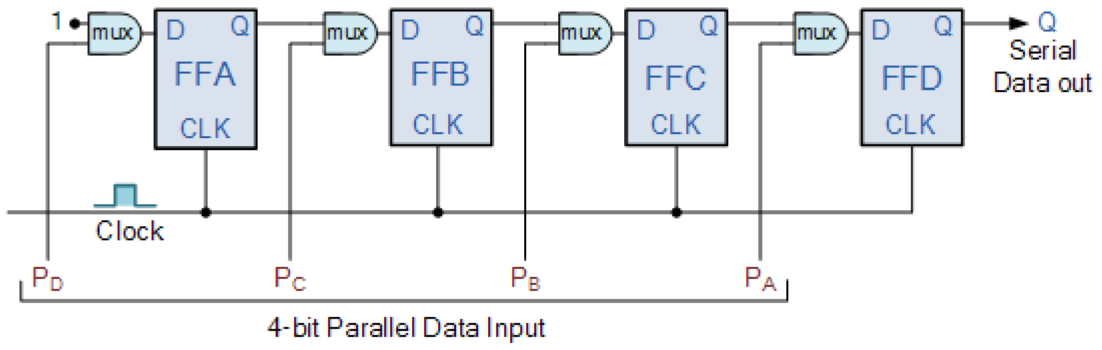


Figure 12: 4-bit Parallel-in to Serial Out Shift Register, PISO (Electronics Tutorials, n.d.)

Figure 12 shows a 4-bit Parallel-in to Serial Out Shift Register, PISO. This shift register will be applied in the design of the datapath circuit to allow the input of 1 bit at a time into the logic circuit that will carry out the addition and subtraction operation. This will effectively reduce the amount of logic gates and components required.

A 4-bit Parallel-in Serial Out (PISO) shift register is a digital circuit that enables the conversion of a parallel input of 4-bits into a serial output bit by bit. There are a total of 4 parallel inputs, 1 clock input and 1 load input. There are 2 different operations that can be done by the PISO shift register which are Parallel Loading Mode (LOAD) and Serial Shifting Mode (SHIFT). In Parallel Loading Mode which is triggered by input 1 into the load input, the 4-bit parallel data is loaded into the shift register and each data is stored in different flip flops. Which means that 4-bit data will require 4 flip flops to store the data. While for the second operation which is the Serial Shifting Mode, the data bits are shifted out serially, one bit at a time, on each clock cycle. When load input is at low and the CLK signal encounters rising or falling edge, the stored bits start shifting to the right toward the Q output.

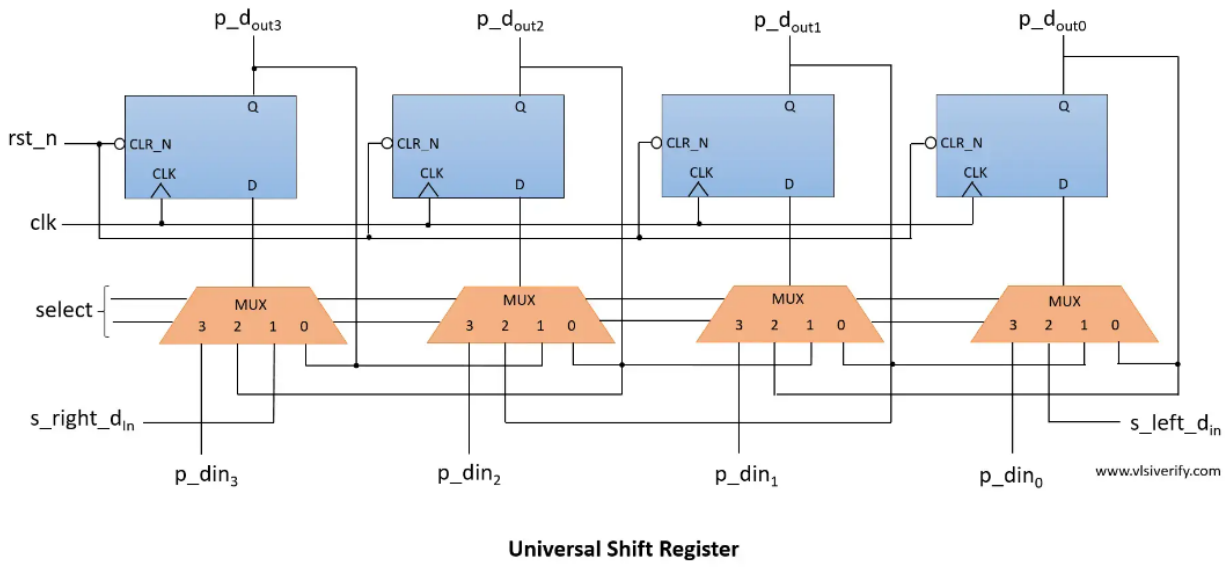


Figure 13: Universal Shift Register (VLSI Verify, n.d.)

In order to have more control in the output of the system, a universal shift register is a better choice of design to be implemented into the datapath. A Universal Shift Register (USR) is a digital circuit that can be configured to perform various types of shifting operations, including both serial and parallel shifts, as well as bidirectional shifts. It is a versatile register that offers flexibility in handling data manipulation tasks. Hence, with appropriate inputs, a universal shift register can be used as a Parallel In Parallel Out shift register, Parallel In Serial Out Shift Register, Serial In Parallel Out shift register and Serial In Serial Out shift register. The concept design of a universal shift register will be implemented in the digital datapath circuit design to give more control on the output. This will enable the output to be constant until the user wants a new output by giving a signal into the ALU.

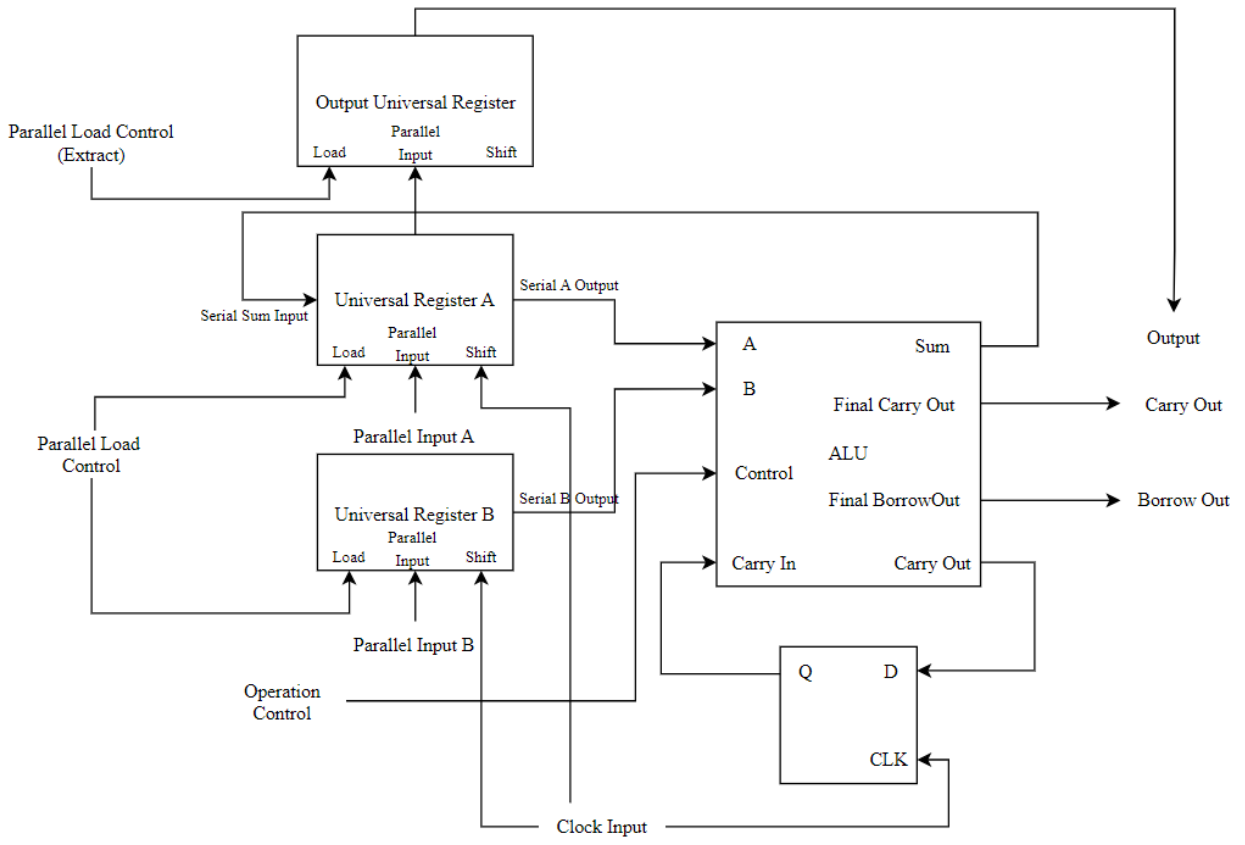


Figure 14: Digital Datapath Design

Figure 14 figure above shows the dedicated datapath design that is created to carry out multiple bit addition and subtraction. First and foremost, the input A and B are input into the universal register A and B. While the register will only load the input A and B according to the Parallel Load Control input when there is a rising edge in the clock input and also a high input of 1 in the Parallel Load Control input. By controlling the input of the registers using Parallel Load Control input, this will prevent the registers from keeping on accepting new loads into the registers before it finishes the operation of addition or subtraction. This prevents interruption in the registers and the whole operation, which ensures correct and stable results.

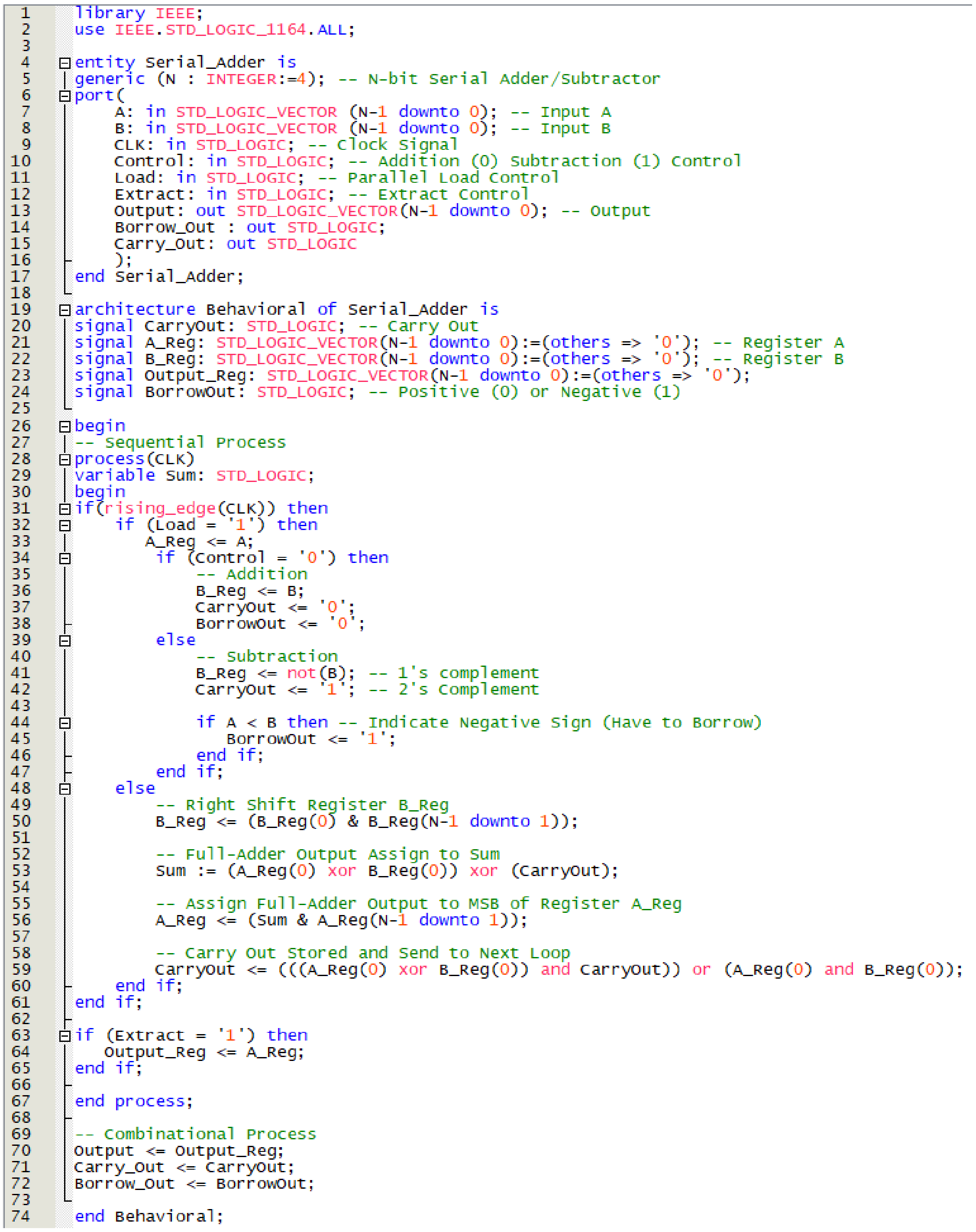
Then the parallel data in the registers will being input into the ALU one by one serially according to the clock signal. This process enables the whole operation of multiple bits to be done by only using 1 set of logical gates. The type of arithmetic operation is controlled by the input of Operation Control to the Control of the ALU. When the Control Input is ‘0’ the operation will be addition while if the input is ‘1’ the operation will be subtraction. The carry will be stored by a D flip flop which is also a basic form of register and then it will be carried on to the operation of the next bit.

The arithmetic operation will only start after the parallel input A and B is loaded into Universal Register A and Universal Register B. The bit-by-bit arithmetic operation follows the clock signal where it is also used to shift the registers A and B to send the next bit into the ALU. 1 cycle of clock signal will send 1 bit if the data in Universal Register A and Universal Register B into the ALU and let the ALU process 1 bit. The output sum of the 1 bit will be used to replace the most significant bit of Universal Register A. Then, the carry out will be stored in the D flip flop and sent to the next stage. This process will continue until all the bits from the input are added and all the correct sum output replaced all the bits in Universal Register A. The waiting time for the ALU to generate the correct valid output depends on the number of bits of the arithmetic operation. 1 bit will require 1 cycle of clock signal. If the operation is 4-bit, it will take 4 cycles of clock signal then only Universal Register A will contain the correct valid result. Then the final carry out and borrow out will be output from the ALU depending on the type of operation addition or subtraction. If the operation is addition only the Carry Out output will be valid while if the operation is subtraction only the Borrow Out output will be valid.

Then the extraction of the data in Universal Register A to the Output Universal Register is also controlled by another Parallel Load Control (Extract) input. This will also prevent the output to be unstable and keep on changing when there is a change in the input to the ALU. This further enhances the stability of data. Hence, only when the output is required, input of ‘1’ will be given into the Output Universal Register to allow it to fetch or extract the data parallelly from the Universal Register A. The timing of fetching or extracting of data has to be correct so that the data fetched from Universal Register A is correct and valid. Hence, the correct amount of clock cycle has to be waited according to the number of bits of the data processed.

VHDL Code

Datapath Design VHDL Code



```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Serial_Adder is
generic (N : INTEGER:=4); -- N-bit Serial Adder/Subtractor
port(
    A: in STD_LOGIC_VECTOR (N-1 downto 0); -- Input A
    B: in STD_LOGIC_VECTOR (N-1 downto 0); -- Input B
    CLK: in STD_LOGIC; -- Clock Signal
    Control: in STD_LOGIC; -- Addition (0) Subtraction (1) Control
    Load: in STD_LOGIC; -- Parallel Load Control
    Extract: in STD_LOGIC; -- Extract Control
    Output: out STD_LOGIC_VECTOR(N-1 downto 0); -- Output
    Borrow_Out : out STD_LOGIC;
    Carry_Out: out STD_LOGIC
    );
end Serial_Adder;

architecture Behavioral of Serial_Adder is
signal CarryOut: STD_LOGIC; -- Carry Out
signal A_Reg: STD_LOGIC_VECTOR(N-1 downto 0):=(others => '0'); -- Register A
signal B_Reg: STD_LOGIC_VECTOR(N-1 downto 0):=(others => '0'); -- Register B
signal Output_Reg: STD_LOGIC_VECTOR(N-1 downto 0):=(others => '0');
signal BorrowOut: STD_LOGIC; -- Positive (0) or Negative (1)

begin
-- Sequential Process
process(CLK)
variable Sum: STD_LOGIC;
begin
if(rising_edge(CLK)) then
    if (Load = '1') then
       A_Reg <= A;
        if (Control = '0') then
            -- Addition
            B_Reg <= B;
            CarryOut <= '0';
            BorrowOut <= '0';
        else
            -- Subtraction
            B_Reg <= not(B); -- 1's complement
            CarryOut <= '1'; -- 2's Complement

            if A < B then -- Indicate Negative Sign (Have to Borrow)
               BorrowOut <= '1';
            end if;
        end if;
    else
        -- Right Shift Register B_Reg
        B_Reg <= (B_Reg(0) & B_Reg(N-1 downto 1));

        -- Full-Adder Output Assign to Sum
        Sum := (A_Reg(0) xor B_Reg(0)) xor (CarryOut);

        -- Assign Full-Adder Output to MSB of Register A_Reg
        A_Reg <= (Sum & A_Reg(N-1 downto 1));

        -- Carry Out Stored and Send to Next Loop
        CarryOut <= (((A_Reg(0) xor B_Reg(0)) and CarryOut)) or (A_Reg(0) and B_Reg(0));
    end if;
end if;

if (Extract = '1') then
   Output_Reg <= A_Reg;
end if;

end process;

-- Combinational Process
Output <= Output_Reg;
Carry_Out <= CarryOut;
Borrow_Out <= BorrowOut;

end Behavioral;
```

Figure 15: Datapath VHDL Code

Figure 15 shows the VHDL code of the datapath designed based on the block diagram in Figure 14. The VHDL code defines a digital datapath circuit called “Serial_Adder” which implements a serial adder subtractor capable of performing arithmetic operations on N-bit binary numbers. The circuit takes inputs A and B, which are N-bit binary numbers to be added or subtracted, a clock signal (CLK) for synchronization, a control signal (Control) to switch between addition and subtraction, a parallel load control signal (Load), an extract control signal (Extract), and produces outputs for the result of the operation (Output), a carry-out signal (Carry_Out), and a borrow-out signal (Borrow_Out) indicating negative results in subtraction. The N-bit binary numbers indicated that the datapath design is capable of doing any number of bit operation just that the number of clock cycle needed to be waited to allow the production of correct and valid result is also N-cycles.

Inside the architecture section, the process driven by the rising edge of the clock signal (CLK) represents the core behavior of the serial adder/subtractor. In parallel loading mode where Load = 1, the input values A and B are loaded into registers A_Reg and B_Reg, respectively. The “Control” signal determines whether the operation is addition or subtraction. In the case of subtraction, 2’s complement logic is applied to the B_Reg. Firstly, to obtain the 1’s complement of the B input the B_Reg is assigned with the inverse of B and then CarryOut signal is set as 1 to make the input to be 2’s complement. Then by applying comparator, if A is less than B, BorrowOut is set to ‘1’ to indicate the negative result and also indicate there is a borrow out.

In non-parallel loading mode where Load = 0 the operation started. The B_Reg is right-shifted by one position according to the clock signal (CLK) which simulated a shift-right operation. The sum of the LSBs of A_Reg and B_Reg, along with the CarryOut signal, is calculated using XOR gates and assigned to the variable “Sum”. This is the process of a full adder by adding 1 bit by one bit. The Sum bit is then written at the MSB of A_Reg while the rest of the bits were right shifted, removing the LSB that had done its operation. The carry-out for the next iteration is calculated based on the full-adder logic. This process effectively simulates a serial addition operation. While this process should be repeated according to the number of bits in the input so that all the bits will be processed and produce a correct and valid output.

When Extract = 1, the Output_Reg fetches the value of A_Reg, which represents the final result after the addition or subtraction operation. While the outputs “Output,” “Carry_Out,” and “Borrow_Out” are assigned the values of Output_Reg, CarryOut, and BorrowOut, respectively.

In summary, this code is a datapath of a sequential circuit that performs serial addition and subtraction of N-bit binary numbers. It utilizes shift registers and full-adder logic to achieve the desired arithmetic operations, handling both positive and negative results, and providing outputs indicating the outcome of the operation.

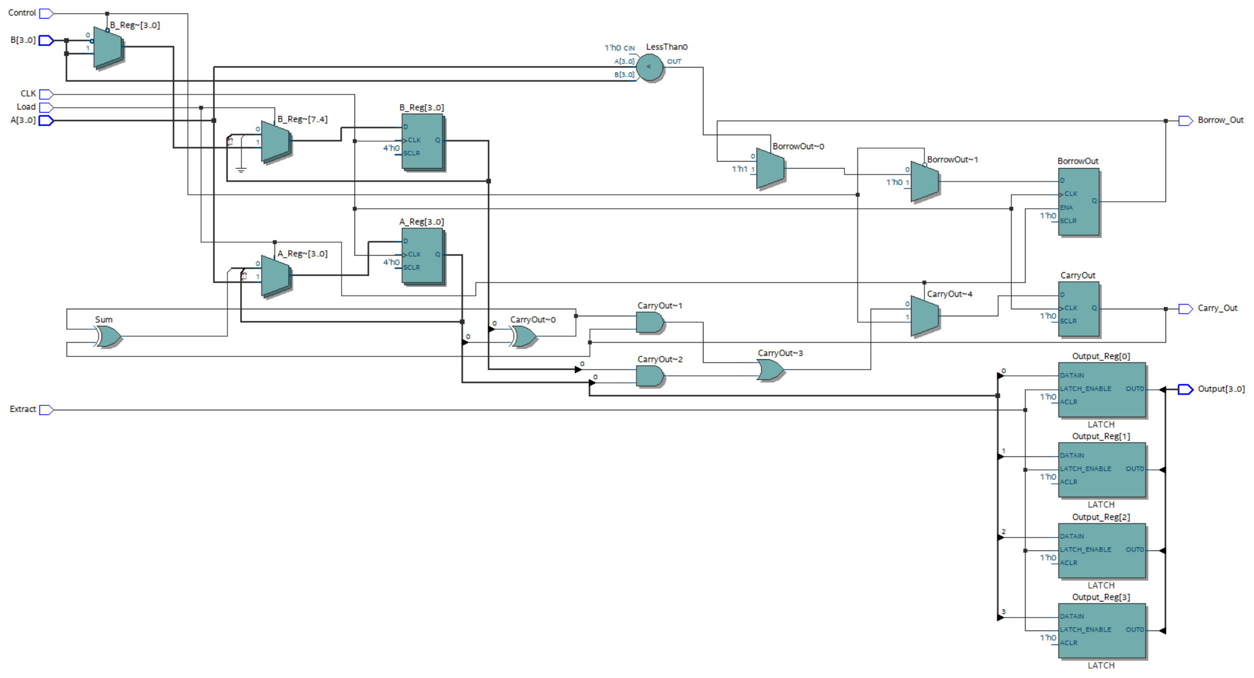


Figure 16: Datapath Design RTL Circuit

Figure 16 shows the RTL circuit generated from the VHDL code in Figure 15. Register Transfer Level (RTL) is a hardware description abstraction used in digital design and computer engineering to model the behavior of digital circuits at a higher level of abstraction than gate-level logic. RTL focuses on the flow of data between registers and the operations performed on that data. Its an intermediate representation between high -level behavioral descriptions such as algorithmic descriptions and gate-level descriptions such as individual logic gates. RTL describes how data moves between registers and how operations are performed using a series of register transfer operations. A register transfer operation represents the movement of data from one register to another or the manipulation of data within a register using logical or arithmetic operations.

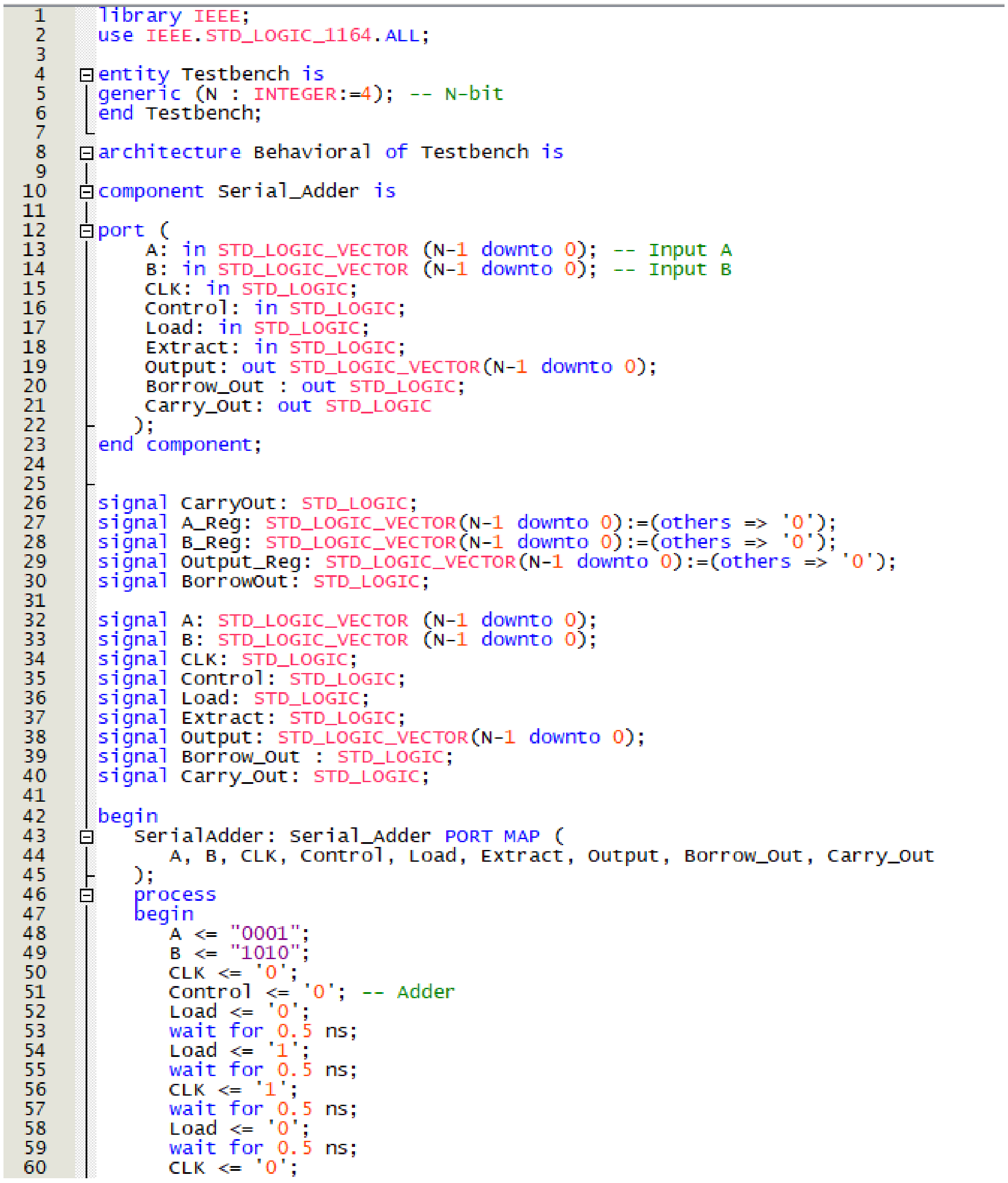
The diagram consisted of 6 inputs which include A operand, B operand, clock signal (CLK), operation control (Control), output register fetching control (Extract) and operand register loading control (Load). On top of that, there are also 3 outputs which are borrow out of subtraction operation (Borrow_Out) and carry out of addition operation (Carry_Out).

There are 4 latches which control the output of the operation which are Output_Reg[0], Output_Reg[1], Output_Reg[2] and Output_Reg[3]. These latches function to fetch the output from A_Reg after all bits had been processed. These latches ensure that the output can be controlled and will only be updated once the arithmetic operation is completed which is indicated by when Extract input is '1'. This ensures the data stability and reliability of the output. The outputs of the latches are in parallel and they are connected with a bus to combine them and produce a serial output.

There are also a total of 4 registers which are A_Reg, B_Reg, BorrowOut and CarryOut. A_Reg and B_Reg are used to load the parallel input A and B and then convert them into a serial output to be sent into the ALU bit by bit to be processed. Referring to the figure, the ALU are represented by different logic gates that are used to carry out addition and subtraction process. While the BorrowOut and CarryOut registers are used to store the borrow out and the carry out of the arithmetic operation. In addition, the CarryOut register also plays a vital role in performing the bit-by-bit operation. It has the function of sending the carry of the operation to the next stage when the input bits are not all processed yet. This is because all the operations are done using adder just that when the operation is a subtraction, the B input will be converted into its 2's complement and add with input A by inverting B input and then assign '1' into the carry out. Hence, the CarryOut register is very important in this datapath design.

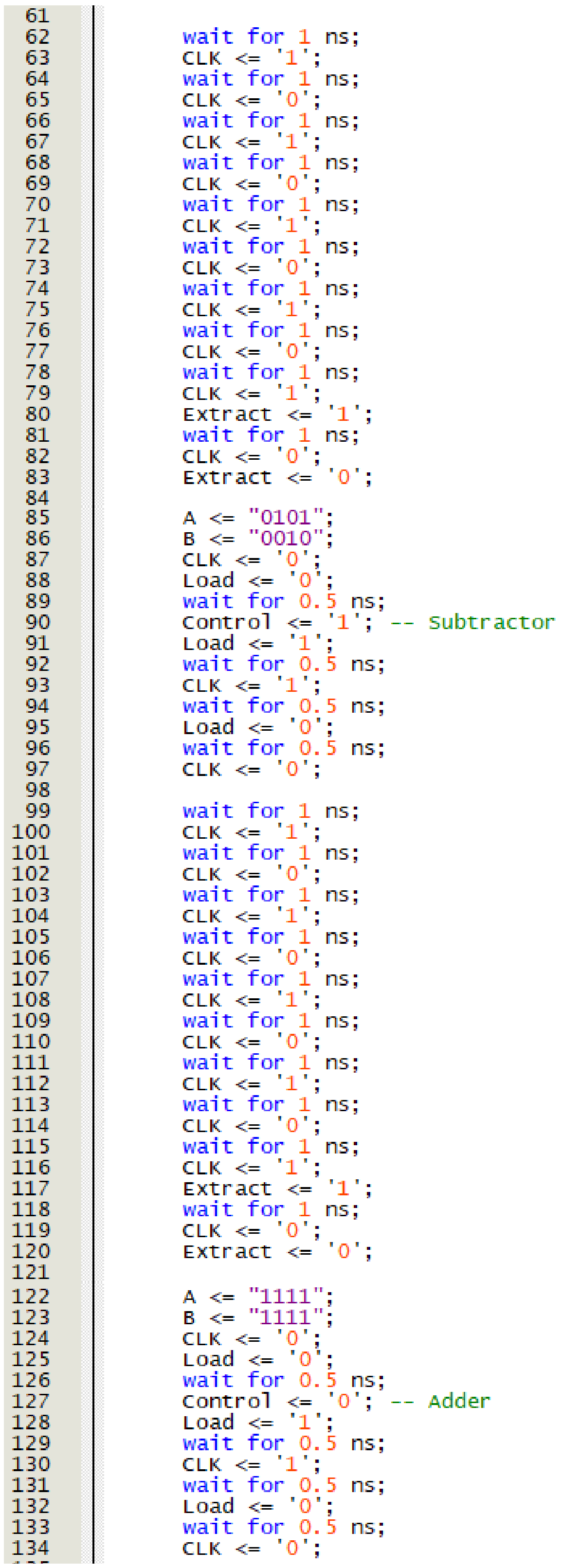
There are also several logic gates such as AND gates and XOR gates to carry out the addition and subtraction operation so does the generation of 2's complement of B operand when the operation chosen is subtraction. Moreover, there is also a comparator which is used to compare operand A and operand B to identify which one is larger to indicate if borrow out will exist or not in the case of operand B is larger than operand A. All of them made up the ALU and are used to perform arithmetic operations which are addition and subtraction.

Testbench VHDL Code



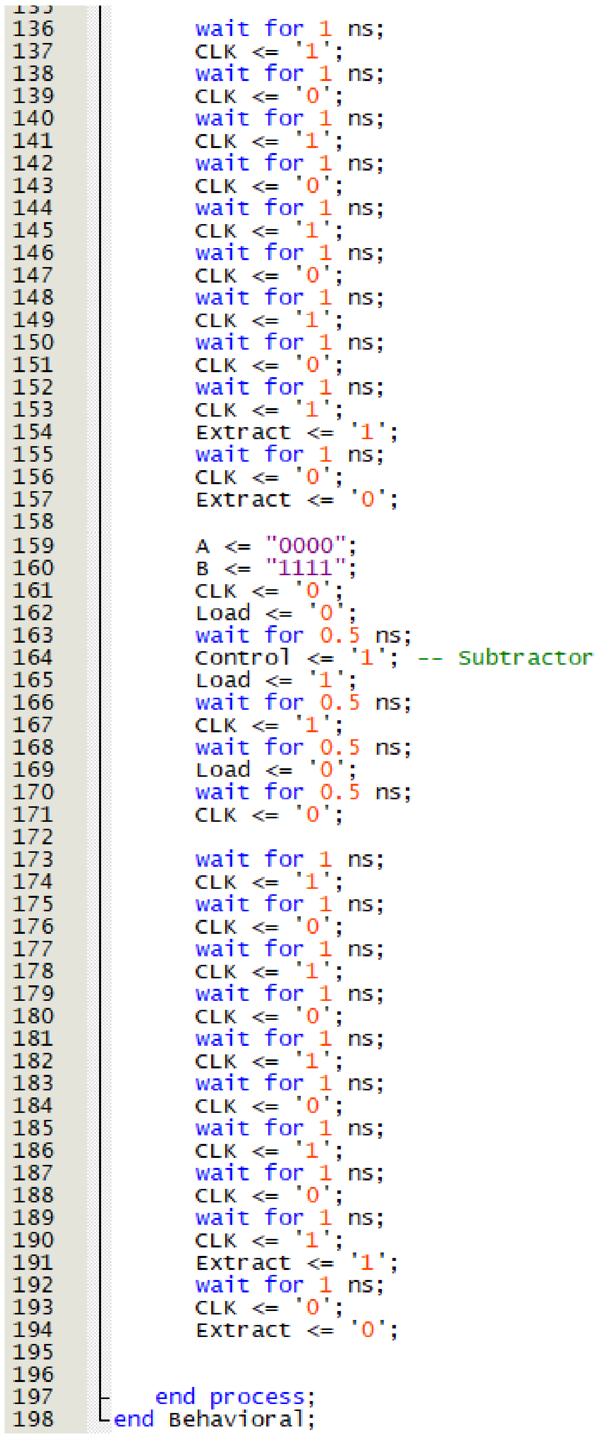
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Testbench is
generic (N : INTEGER:=4); -- N-bit
end Testbench;

architecture Behavioral of Testbench is

component Serial_Adder is

port (
     A: in STD_LOGIC_VECTOR (N-1 downto 0); -- Input A
     B: in STD_LOGIC_VECTOR (N-1 downto 0); -- Input B
     CLK: in STD_LOGIC;
     Control: in STD_LOGIC;
     Load: in STD_LOGIC;
     Extract: in STD_LOGIC;
     Output: out STD_LOGIC_VECTOR(N-1 downto 0);
     Borrow_Out : out STD_LOGIC;
     Carry_Out: out STD_LOGIC
    );
end component;


signal CarryOut: STD_LOGIC;
signal A_Reg: STD_LOGIC_VECTOR(N-1 downto 0):=(others => '0');
signal B_Reg: STD_LOGIC_VECTOR(N-1 downto 0):=(others => '0');
signal Output_Reg: STD_LOGIC_VECTOR(N-1 downto 0):=(others => '0');
signal BorrowOut: STD_LOGIC;

signal A: STD_LOGIC_VECTOR (N-1 downto 0);
signal B: STD_LOGIC_VECTOR (N-1 downto 0);
signal CLK: STD_LOGIC;
signal Control: STD_LOGIC;
signal Load: STD_LOGIC;
signal Extract: STD_LOGIC;
signal Output: STD_LOGIC_VECTOR(N-1 downto 0);
signal Borrow_Out : STD_LOGIC;
signal Carry_Out: STD_LOGIC;

begin
   SerialAdder: Serial_Adder PORT MAP (
      A, B, CLK, Control, Load, Extract, Output, Borrow_Out, Carry_Out
   );
   process
   begin
      A <= "0001";
      B <= "1010";
      CLK <= '0';
      Control <= '0'; -- Adder
      Load <= '0';
      wait for 0.5 ns;
      Load <= '1';
      wait for 0.5 ns;
      CLK <= '1';
      wait for 0.5 ns;
      Load <= '0';
      wait for 0.5 ns;
      CLK <= '0';
```

Figure 17: Testbench Code 1



```vhdl

        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        Extract <= '1';
        wait for 1 ns;
        CLK <= '0';
        Extract <= '0';

        A <= "0101";
        B <= "0010";
        CLK <= '0';
        Load <= '0';
        wait for 0.5 ns;
        Control <= '1'; -- Subtractor
        Load <= '1';
        wait for 0.5 ns;
        CLK <= '1';
        wait for 0.5 ns;
        Load <= '0';
        wait for 0.5 ns;
        CLK <= '0';

        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        Extract <= '1';
        wait for 1 ns;
        CLK <= '0';
        Extract <= '0';

        A <= "1111";
        B <= "1111";
        CLK <= '0';
        Load <= '0';
        wait for 0.5 ns;
        Control <= '0'; -- Adder
        Load <= '1';
        wait for 0.5 ns;
        CLK <= '1';
        wait for 0.5 ns;
        Load <= '0';
        wait for 0.5 ns;
        CLK <= '0';
```

Figure 18: Testbench Code 2



```
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        Extract <= '1';
        wait for 1 ns;
        CLK <= '0';
        Extract <= '0';

        A <= "0000";
        B <= "1111";
        CLK <= '0';
        Load <= '0';
        wait for 0.5 ns;
        Control <= '1'; -- Subtractor
        Load <= '1';
        wait for 0.5 ns;
        CLK <= '1';
        wait for 0.5 ns;
        Load <= '0';
        wait for 0.5 ns;
        CLK <= '0';

        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        wait for 1 ns;
        CLK <= '0';
        wait for 1 ns;
        CLK <= '1';
        Extract <= '1';
        wait for 1 ns;
        CLK <= '0';
        Extract <= '0';


    end process;
end Behavioral;
```

Figure 19: Testbench Code 3

Figure 17, Figure 18 and Figure 19 shows the test bench VHDL code to simulate the datapath design. There is a total of 4 cases simulated in the testbench. The first is an addition without carry out. The input values are A = “0001” (decimal 1) and B = “1010” (decimal 10). The

Control signal is set to ‘0’ to indicate addition mode, and Load is first set to ‘0’ and then to ‘1’ after a delay of 0.5 ns to enable parallel loading of inputs where the input A and B will be loaded into the registers A_Reg and B_Reg. The clock signal (CLK) toggles, simulating clock cycles. Then there will be 4 clock cycles which allows all of the bit to be processed since the input of A and B is a 4-bit input. Then after the 4 clock cycles, the Extract input is set to ‘1’ and then ‘0’ after 1 ns. This allows the complete result in A_Reg to be fetched into the output register.

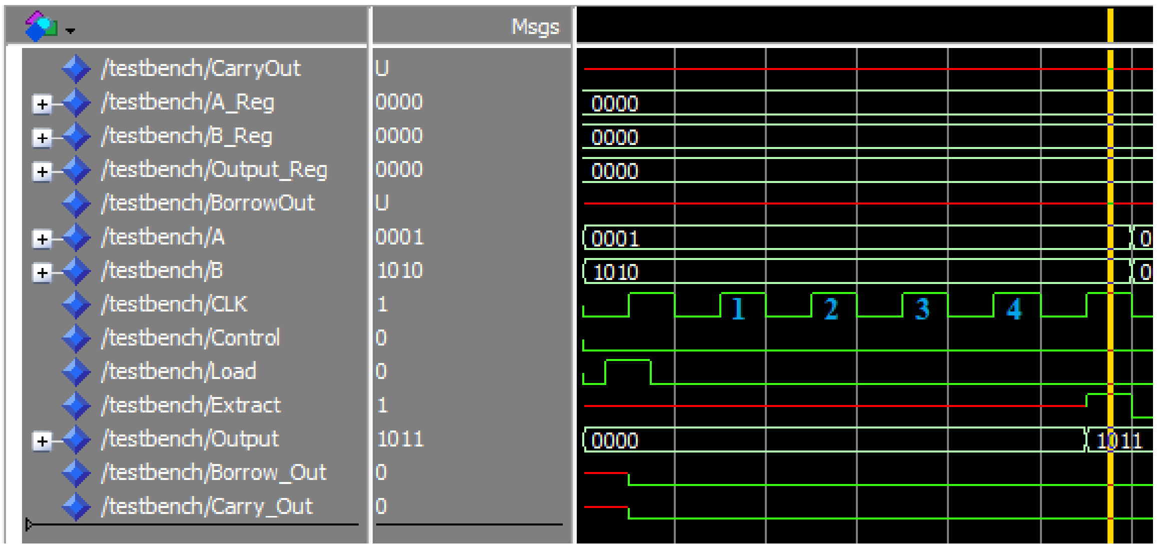
The second case is subtraction without borrow out. In this test case, subtraction is simulated using A = “0101” (decimal 5) and B = “0010” (decimal 2). The Control signal is set to ‘1’ for subtraction mode, and Load is managed similarly as in the previous case.

While the third case is an addition that will produce a carry. The addition is with A = “1111” (decimal 15) and B = “1111” (decimal 15). The Control signal is set to ‘0’, and loading follows the pattern established in the previous cases.

The last case is a subtraction that will produce a borrow out. The subtraction is performed with A = “0000” (decimal 0) and B = “1111” (decimal 15). The Control signal is set to ‘1’ for subtraction, and loading is handled as before.

All these 4 cases had successfully tested all possible outcomes of adder and subtractor. Hence the testbench is suitable to be used to test the designed datapath. Due to the lack of a control unit design, the inputs to the datapath are coded perfectly to simulate all possible conditions that will be encountered by the designed datapath which included addition with no carry out, subtraction with no borrow out, addition with carry out and subtraction with borrow out. This ensures that the designed datapath have the capability to handle all situations and possible inputs.

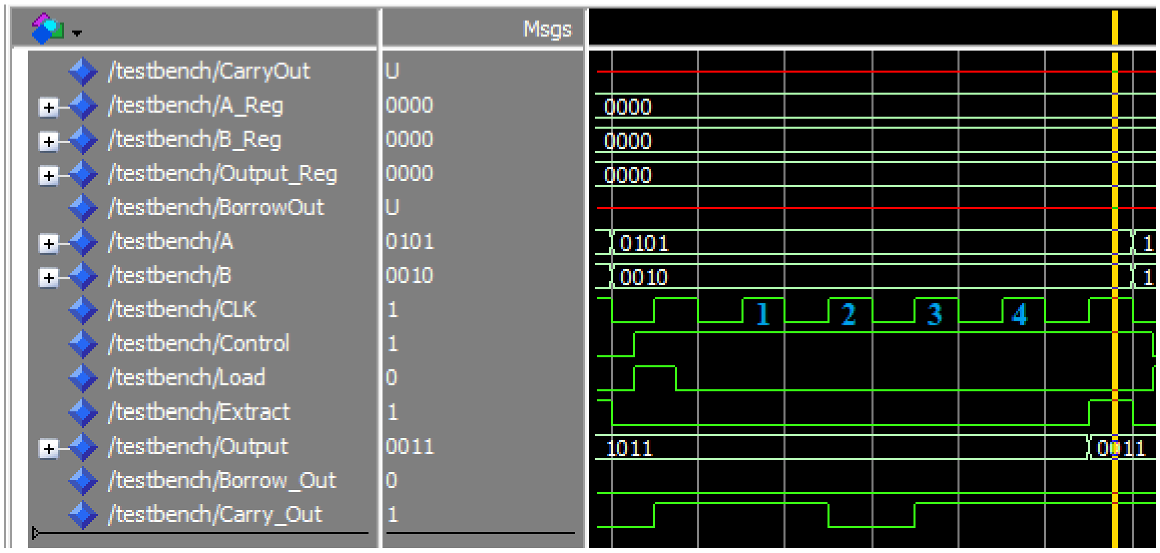
RTL Simulation Results



	Msgs
/testbench/CarryOut	U
/testbench/A_Reg	0000
/testbench/B_Reg	0000
/testbench/Output_Reg	0000
/testbench/BorrowOut	U
/testbench/A	0001
/testbench/B	1010
/testbench/CLK	1
/testbench/Control	0
/testbench/Load	0
/testbench/Extract	1
/testbench/Output	1011
/testbench/Borrow_Out	0
/testbench/Carry_Out	0

Figure 20: Case 1 Simulation Result (Addition No Carry Out)

Figure 20 shows the case 1 input mentioned in the testbench code section where the input of A is “0001” and B is “1010”. The operation is addition because the Control input is ‘0’. After 4 clock cycles, when the Load input is ‘1’, the output obtained is “1011”. The Carry_Out output is ‘0’.

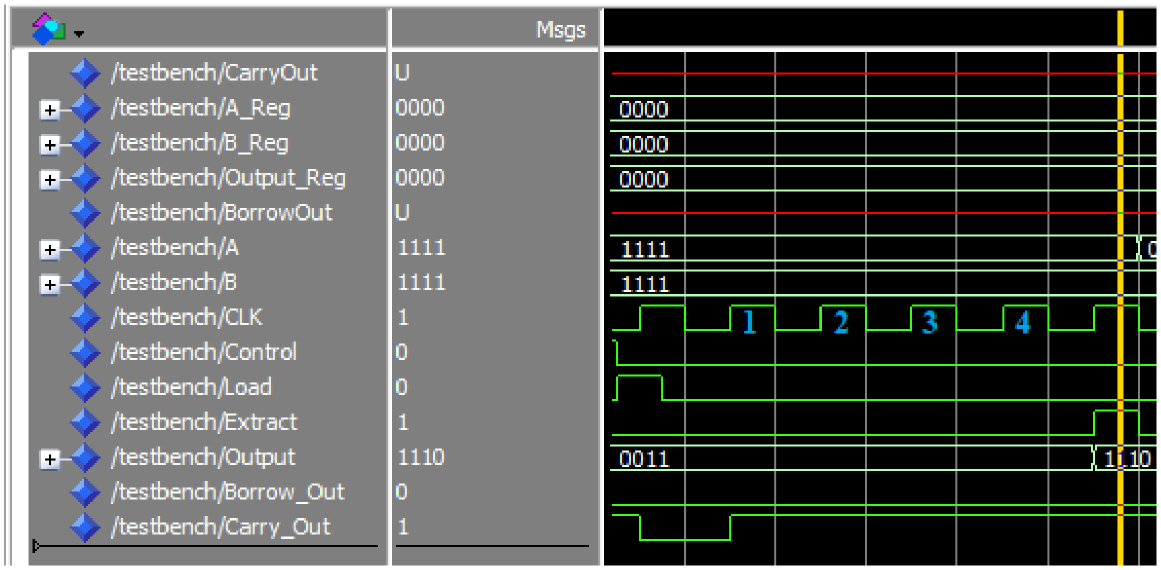


	Msgs
/testbench/CarryOut	U
/testbench/A_Reg	0000
/testbench/B_Reg	0000
/testbench/Output_Reg	0000
/testbench/BorrowOut	U
/testbench/A	0101
/testbench/B	0010
/testbench/CLK	1
/testbench/Control	1
/testbench/Load	0
/testbench/Extract	1
/testbench/Output	0011
/testbench/Borrow_Out	0
/testbench/Carry_Out	1

Figure 21: Case 2 Simulation Result (Subtraction No Borrow Out)

Figure 21 shows the case 2 input mentioned in the testbench code section where the input of A is “0101” and B is “0010”. The operation is subtraction because the Control input is ‘1’. After

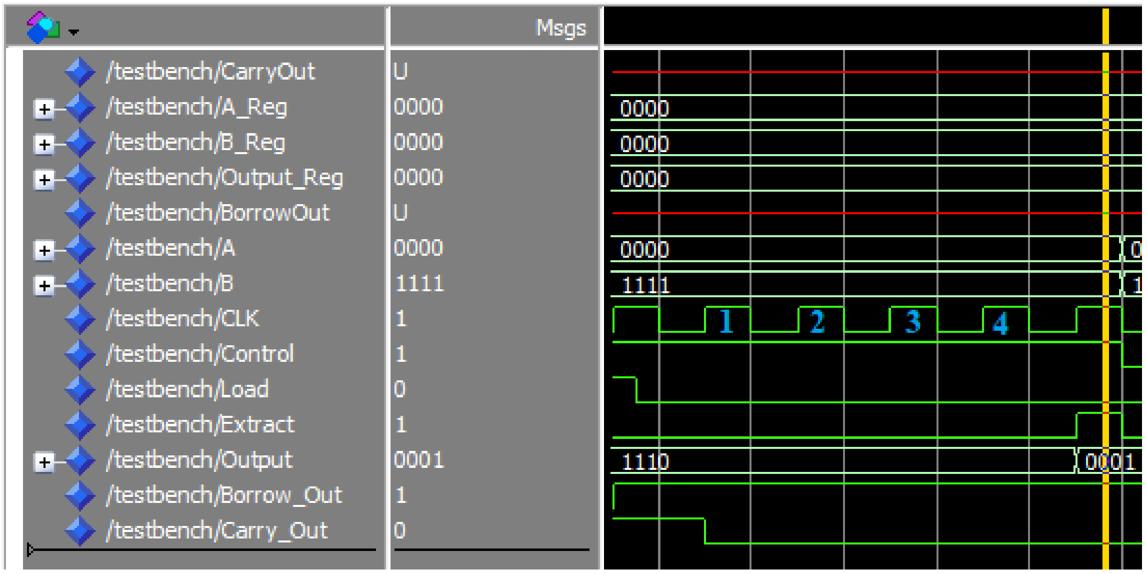
4 clock cycles, when the Load input is ‘1’, the output obtained is “0011”. The Borrow_Out output is ‘0’.



```
                              Msgs
/testbench/CarryOut           U
/testbench/A_Reg              0000
/testbench/B_Reg              0000
/testbench/Output_Reg         0000
/testbench/BorrowOut          U
/testbench/A                  1111
/testbench/B                  1111
/testbench/CLK                1
/testbench/Control            0
/testbench/Load               0
/testbench/Extract            1
/testbench/Output             1110
/testbench/Borrow_Out         0
/testbench/Carry_Out          1
```

Figure 22: Case 3 Simulation Result (Addition With Carry Out)

Figure 22 shows the case 3 input mentioned in the testbench code section where the input of A is “1111” and B is “1111”. The operation is addition because the Control input is ‘0’. After 4 clock cycles, when the Load input is ‘1’, the output obtained is “1110”. The Carry_Out output is ‘1’.



```
                              Msgs
/testbench/CarryOut           U
/testbench/A_Reg              0000
/testbench/B_Reg              0000
/testbench/Output_Reg         0000
/testbench/BorrowOut          U
/testbench/A                  0000
/testbench/B                  1111
/testbench/CLK                1
/testbench/Control            1
/testbench/Load               0
/testbench/Extract            1
/testbench/Output             0001
/testbench/Borrow_Out         1
/testbench/Carry_Out          0
```

Figure 23: Case 4 Simulation Result (Subtraction With Borrow Out)

Figure 23 shows the case 4 input mentioned in the testbench code section where the input of A is “0000” and B is “1111”. The operation is subtraction because the Control input is ‘1’. After 4 clock cycles, when the Load input is ‘1’, the output obtained is “0001”. The Borrow_Out output is ‘1’.

Table 2: Simulation Results

Case	Operation	A	B	Output	Carry Out	Borrow Out
1	Addition	0001	1010	1011	0	X
2	Subtraction	0101	0010	0011	X	0
3	Addition	1111	1111	1110	1	X
4	Subtraction	0000	1111	0001	X	1

Discussion

Table 2 shows all the RTL simulation results where all the inputs and outputs are listed. It is important to note that when the process is addition, only the Carry_Out output is considered while the Borrow_Out output is not considered. This is because addition will not require borrow. While for subtraction operation, only Borrow_Out output is considered because subtraction will not produce a carry. The Borrow_Out output is also used to indicate that the output is negative. This is because only a smaller number that is minus with a bigger number will require a borrow. Hence the output of a negative number has to be converted to its 2's complement to obtain its true value.

Figure 20 shows the RTL simulation result of addition of "0001" (decimal 1) and "1010" (decimal 10) which are operand A and operand B. As shown in Figure 20, the Control input is kept as '0' the whole time to control the ALU to perform addition. The input operand A and B are only loaded into the registers A_Reg and B_Reg when the Load input is '1'. After the registers are loaded with the input A and B, the ALU will require 4 cycles of clock which is also equivalent to 4 rising edges since that the ALU is designed to be positive triggered. Within each cycle, 1 bit of the input A and B are added in the ALU using a full adder logic. The carry that is produced during the process is stored in a D flip-flop and then is transferred to the operation of the next bit. To send the next bit into the ALU, the universal register A_Reg and B_Reg are right shifted. The B_Reg is shifted right and then the most significant bit is replaced with the bit that had gone through the ALU, which is similar to a rotator. While the A_Reg is not only right shifted but its most significant bit is also written with the sum result previously. After 4 clock cycles, all the bits in the registers had gone through the ALU and all the results had completely replaced all bits in A_Reg which results in the A_Reg now containing the sum of input operand A and B. The result of 1 + 10 is 11 which is "1011" in binary. Then when the Extract input is '1' after the 4th clock cycle, the data in A_Reg is extracted to Output_Reg and is displayed which also shows "1011". Hence, it is verified correct. Besides that, the Carry_Out output is also '0' which indicated no carry which is also valid. The Borrow_Out output is ignored since the operation is addition.

Figure 21 shows the RTL simulation result of subtraction of "0101" (decimal 5) and "0010" (decimal 2) which are operand A and operand B. As shown in Figure 21, the Control input is kept as '1' the whole time to control the ALU to perform subtraction. The input operand A and B are only loaded into the registers A_Reg and B_Reg when the Load input is '1'. Since the operation is subtraction, input B is converted to its 1's complement before loaded into B_Reg. While to make it 2's complement, CarryOut signal which is used in the operation is assigned with '1'. Hence, B input will be converted into 2's complement. After the registers are loaded with the input A and 2's complement of B, the ALU will also require 4 cycles of clock which is also equivalent to 4 rising edges since that the ALU is designed to be positive triggered. Within each cycle, 1 bit of the input A is added with the 2's complement of B in the ALU using a full adder logic. The carry that is produced during the process are stored in a D flip-flop and then are transferred to the operation of the next bit. To send the next bit into the ALU, the universal register A_Reg and B_Reg are right shifted. The B_Reg is shifted right and then the most significant bit is replaced with the bit that had gone through the ALU, which is similar to a rotator. While the A_Reg is not only right shifted but its most significant bit is also written with the sum result previously. After 4 clock cycles, all the bits in the registers had gone through the ALU and all the results had completely replaced all bits in A_Reg which results in the A_Reg now containing the sum of input operand A and B. The result of 5 – 2 is 3 which is "0011" in binary. Then when the Extract input is '1' after the 4th clock cycle, the data in A_Reg is extracted to Output_Reg and is displayed which also shows "0011". Hence, it is verified correct. Besides that, the Borrow_Out output is also '0' which indicated no borrow is needed and the result is a positive number which is also valid. The Carry_Out output is ignored since the operation is subtraction.

Figure 22 shows the RTL simulation result of addition of "1111" (decimal 15) and "1111" (decimal 15) which are operand A and operand B. As shown in Figure 22, the Control input is kept as '0' the whole time to control the ALU to perform addition. The input operand A and B are only loaded into the registers A_Reg and B_Reg when the Load input is '1'. After the registers are loaded with the input A and B, the ALU will require 4 cycles of clock which is also equivalent to 4 rising edges since that the ALU is designed to be positive triggered. Within each cycle, 1 bit of the input A and B are added in the ALU using a full adder logic. The carry that is produced during the process is stored in a D flip-flop and then is transferred to the operation of the next bit. To send the next bit into the ALU, the universal register A_Reg and B_Reg are right shifted. The B_Reg is shifted right and then the most significant bit is replaced with the bit that had gone through the ALU, which is similar to a rotator. While the A_Reg is not only right shifted but its most significant bit is also written with the sum result previously. After 4 clock cycles, all the bits in the registers had gone through the ALU and all the results had completely replaced all bits in A_Reg which results in the A_Reg now containing the sum of input operand A and B. The result of 15 + 15 is 30 which is "11110" in binary. Then when the Extract input is '1' after the 4th clock cycle, the data in A_Reg is extracted to Output_Reg and is displayed which also shows "1110" and with the Carry_Out output of '1'. Hence, it is indicated that the output is '11110' which is also verified correct. The Borrow_Out output is ignored since the operation is addition.

Figure 23 shows the RTL simulation result of subtraction of "0000" (decimal 0) and "1111" (decimal 15) which are operand A and operand B. As shown in Figure 23, the Control input is kept as '1' the whole time to control the ALU to perform subtraction. The input operand A and B are only loaded into the registers A_Reg and B_Reg when the Load input is '1'. Since the operation is subtraction, input B is converted to its 1's complement before loaded into B_Reg. While to make it 2's complement, CarryOut signal which is used in the operation is assigned with '1'. Hence, B input will be converted into 2's complement. After the registers are loaded with the input A and 2's complement of B, the ALU will also require 4 cycles of clock which is also equivalent to 4 rising edges since that the ALU is designed to be positive triggered. Within each cycle, 1 bit of the input A is added with the 2's complement of B in the ALU using a full adder logic. The carry that is produced during the process are stored in a D flip-flop and then are transferred to the operation of the next bit. To send the next bit into the ALU, the universal register A_Reg and B_Reg are right shifted. The B_Reg is shifted right and then the most significant bit is replaced with the bit that had gone through the ALU, which is similar to a rotator. While the A_Reg is not only right shifted but its most significant bit is also written with the sum result previously. After 4 clock cycles, all the bits in the registers had gone through the ALU and all the results had completely replaced all bits in A_Reg which results in the A_Reg now containing the sum of input operand A and B. The result of 0 – 15 is -15 which is the 2's complement of "1111" (decimal 15) and is "0001" (decimal -15) in binary. Then when the Extract input is '1' after the 4th clock cycle, the data in A_Reg is extracted to Output_Reg and is displayed which also shows "0001". Hence, it is verified correct. Besides that, the Borrow_Out output is also '1' which indicated borrow is needed and the result is a negative number which is valid. The Carry_Out output is ignored since the operation is subtraction.

All the possible outcomes of the datapath designed were simulated and all were verified correct. All the operations were controlled by its dedicated control signal such as CLK, Control, Load and Extract to ensure that the process is organized. CLK is the clock signal that controls bit-by-bit operation in the ALU and registers shifting operation. Control is the signal that is used to indicate the type of arithmetic operation addition or subtraction. Load is the control of the input registers to load parallel input whenever the Load input is '1'. Only when new inputs are loaded into A_Reg and B_Reg, then only the ALU will carry out a new operation of different inputs. Extract is the control of the output register to load parallel output from the ALU to extract the result of the arithmetic operation. Only when the Extract input is '1', the Output_Reg will then load the result data stored in A_Reg. This control enables the registers to hold data more stable as it is not prone to changes when there are changes happening in the A_Reg and ALU during the middle of arithmetic operations. The registers will only be able to load new data when they are allowed to, according to their corresponding control signal. This enhanced the data stability and reliability of the datapath.

In addition, the designed datapath is also capable of carry out operation of any number of bits input just that the corresponding cycles of clock signal have to be waited to allow all the bits gone through the ALU and produced the correct result before extracting it. However, from the physical circuit's perspective, the registers also should be changed to be able to store the amount of bit. No matter how many bits the inputs are, only 1 set of ALU is needed to complete the whole operation.

Conclusion

In a nutshell, the concept of a digital datapath circuit and its fundamental components and their importance had been investigated in this assignment. On top of that, a simple ALU and digital datapath circuit that is capable of carrying out addition and subtraction is designed. All the objectives of the assignment had been achieved. The design is done using VHDL code and its Register Transfer Level (RTL) diagram is generated, displayed and analyzed. RTL technique is also applied in the process of designing the datapath by using Hardware Description Language which is VHDL in this assignment.

The ALU is design using a 1-bit adder subtractor and then the inputs are loaded into universal register to allow the parallel input to be converted to serial and input 1 bit by 1 bit into the 1-but adder subtractor logic circuit. Then the carry will also be brought to the next bit operation using a register to store it temporarily. The type of arithmetic operation addition or subtraction is also controlled by a control signal. All the registers in the designed datapath are also controlled with different control signals which will only allow them to load new data at the correct time. This will ensure the stability of the output of the datapath because the registers will not be prone to changes constantly. They will only be changed whenever it is needed according to the control signal.

On top of that, the designed datapath is also capable of carrying out any number of bit operation since all the operation is done by looping all the bits into an ALU that can carry out addition and subtraction. Just that the number of clock cycles that have to be waited for the result to be produced is also same with the number of bits of the input. As an example, if the input is 6-bit, then 6 clock cycles have to be waited for the datapath to produce the correct result then only the output register will be able to fetch the correct result. This is because 1 clock cycle will only allow the operation of 1-bit which includes the shifting and arithmetic operations. In addition, from the physical circuit point of view, the registers should also be changed to be able to fit the input bits.

Further improvement can be applied to the designed datapath by adding more choice of arithmetic operations such as multiply and division. This will make the datapath capable of carrying out more types of arithmetic operations. Besides that, the control signals that are applied in the simulation are coded using the testbench VHDL code, a complete control unit can be designed to make the datapath to be simpler to be used where the user will only have to input operand A and B to carry out the operation instead of having to control the control signals.

References

Computer, Hope. (2020). *Control Unit.* https://www.computerhope.com/jargon/c/contunit.htm

Geeks For Geeks. (2023). *Introduction of Control Unit and Its Design.* https://www.geeksforgeeks.org/introduction-of-control-unit-and-its-design/

Mallick, C. B. (2023). *What Is an Arithmetic Logic Unit (ALU)? Meaning, Components, and Functions.* https://www.spiceworks.com/tech/hardware/articles/what-is-alu/

Charles, W. K. (n.d.). *The ALU.* https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Electronics/Implementing_a_One_Address_CPU_in_Logisim_(Kann)/

Rouse, M. (2023). *Data Bus.* https://www.techopedia.com/definition/6733/data-bus

Thornton, S. (2016). *The Internal Processor Bus: Data, Address, and Control Bus.* https://www.microcontrollertips.com/internal-processor-bus-data-address-control-bus-faq/

Geeks For Geeks. (n.d.). *Difference between Single Bus Structure and Double Bus Structure.* https://www.geeksforgeeks.org/difference-between-single-bus-structure-and-double-bus-structure/

Indira, P. et al. (2019). Design and Analysis of A 32-bit Pipelined MIPS Risc Processor. *International Journal of VLSI Design & Communication Systems*. 10. 1-18. 10.5121/vlsic.2019.10501.

Hwang, E. (2003). *Principle of Digital Logic Design.* http://www.cs.ucr.edu/~ehwang/courses/cs120a/datapaths.pdf

Urias, O. M. (2022). *Full Adder Circuit – How it Works.* https://www.build-electronic-circuits.com/full-adder/

WatElectronics. (2020). *Full Subtractor Circuit Diagram Using Basic Gates and Applications.* https://www.watelectronics.com/full-subtractor-circuit-diagram-using-basic-gates-and-applications/

VLSI Verify. (n.d.) *4-bit Adder Subtractor.* https://vlsiverify.com/verilog/verilog-codes/4-bit-adder-subtractor

Electronics Tutorials. (n.d.). *The Shift Register.* https://www.electronics-tutorials.ws/sequential/seq_5.html

VLSI Verify. (n.d.). *Universal Shift Register.* https://vlsiverify.com/verilog/verilog-codes/universal-shift-register