ALU and Dependency Manager Using FPGA

by

Amjad Qusay Hashem

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering (Computer Engineering) in the University of Michigan-Dearborn 2020

Master's Thesis Committee:

Associate Professor Ali El Kateeb, Chair
Professor Selim Awad
Professor Adnan Shaout

Amjad Qusay Hashem

aqhashem@umich.edu

ORCID iD: 0000-0002-7465-6446

© Amjad Qusay Hashem 2020

Dedication

This work is dedicated to the people, who supported me through this work as well as all other challenges through my life. To my family; parents (Qusay & Amina), wife (Israa), and my siblings (Ahmed, Saba, Shams, Nasrin).

Acknowledgements

This work was created with the unparalleled support and dedication of Professor Ali El Kateeb, whose door was always opened for the support and direction I needed throughout the work of this project.

I also would like to thank the Department of Computer Engineering in the University of Michigan-Dearborn for giving me the chance and the resource to make all this work possible.

.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
OP	Operation
OP-Code	Operation Code
INST	Instruction
ns	Nano Second
NDR	New Data Ready
CLK	Clock
VHDL	VHSIC-HDL, Very High-Speed Integrated Circuit Hardware Description Language
REG	Data Register

Abstract

In this project, we have researched the design of high-speed processing units through parallel processing scheme. Our design will be used for Real – Time applications such as Data Acquisition, DSP, Real – Time Controllers, etc.

This project is introducing a new method to handle the parallel processing, this method will process instructions with respect to the dependencies among all instructions. Having a parallel processing capability allows this design to have extremely fast processing speed. In this project we will review every part of the system, and how it contributes to the performance of the system.

Currently FPGAs are used as specialty processing units of systems to boost up the throughput of those systems. Using FPGA to speed up special operations, such as database sorting [1, p. 53], data buffering, and many other function that require either high throughput processing, or low latency processing [2, p. 213], can increase the performance of the system.

In addition, the normal design for a processing unit that can be part of a system is pipelining processing scheme. In this project, we present a new processing scheme capable of parallel processing, as well as dependency management, both built in the internal design of the unit. This design will eliminate the need for a dependency management software, which can take extra processing time or force the processor to have to process instruction sequentially. Instead, this design will allow parallel processing and dependency management built in a single integrated processing unit.

Chapter 1: FPGA

1.1. What is FPGA?

Field Programmable Gate Array is a technology that uses large arrays of logical gates and memory blocks in chip designs. FPGAs can be configured for processing information [3, p. 26], which can be used in a wide range of applications, for low latency or high-speed data processing, as well as for high throughput. The FPGA chip can be programmed to perform any type of functionality, and can be configured by rewiring the connections among logic gates, in order to be able to perform the required functions [3, p. 25].

After the FPGA design is done, which can be accomplished by using several tools, such as schematic design and behavioral programming language VHDL, the design will be synthesized to represent the connections among the logic gates throughout the chip, in order to perform the required functions. The synthesized file will be interpreted to a map of all connections that will later be loaded to the chip [2, p. 11].

After the chip has been programmed, the program that's been written is performed as a circuit of hardwired logic gates, which provide the FPGA the speed and robustness it's known for, as there is no software to drive the execution through the chip, because the chip is a pure hardware that is designed specifically to perform the required functions.

1.2. FPGA Design Methods

To design an FPGA chip, there are two ways or methods to be followed, in order to create a Design that can perform synchronous or asynchronous Function for the application in hand. FPGA design is based on processing input signals, such as Boolean signals, buffer signals that consists of several Boolean inputs, then Propagating those signals throughout logic gates to get the results from the output side. Those design methods are schematic design and behavioral design.

1.1.1. Schematic Design Method

With the schematic design method, we use several available logic gates and pre-designed objects such as multiplexors, decoders, and Flip-Flops. In order to be able to test and download the design, A detailed design will have to be completed with all the inputs, outputs, and connections among all the objects.

Schematic design is normally used to design highly customized parts of the system that require advanced optimization from a functionality and speed standpoint. Otherwise using the existing predefined objects or functions, which were built using the design tool, in order to generate the final schematics with standard objects and standard methods for routing, Inputs, outputs, and configurations.

Schematic design allows to fully customize the design to the lowest level possible, where we can design with the basic logic gates, or we can also use predefined objects that were designed and optimized previously by the system designer. Either ways, schematic design method is the more advanced method available because it allows the highest level of optimization of all aspects of the logic circuits. whether it is a routing, inputs, outputs, or complex objects, Schematic design method allow the designer to fully customize every function used in the design. Even though the design is better to be customized and optimized after it's completed as an integrated unit, it is recommended to use the predefined objects in order to save design time and design effort for larger systems.

1.1.2. VHDL Behavioral Language Design Method

VHDL design is basically using a script based programing language, in order to describe the behavior of the desired circuit, through the common programming tools, like; if then, for Loop, while loops, as the tool to describe the function that is required to be performed by our system. VHDL is a more standardized design tool, it allows the designer to write the expected functionality of the system, and to spend more time in enhancing and expanding the system tasks [4], rather than spending the time dealing with issues that could be addressed by the compiler or the synthesizer [2, p. 106].

What is behavioral design? Behavioral design is describing the task required by the system through a scripting programing language. The compiler will convert the behavioral program code built by the designer to a schematic circuit using standard objects. Using a predesign table assignments to translate the code into schematics, allows the generation of standardized schematics for every possible combination of functions. However, since there is a predesigned VHDL function vs. schematic objects, there is the possibility of optimization of the code that can be done to optimize the schematics, which leads as a result to optimizing the design performance. [1, p. 40]

Chapter 2: Project

2.1. Objective

The initial objectives of this design are to be able to perform specific functions, completely or partially, performed by the processing unit in the existing designs. Titles and descriptions maybe similar to those components used in existing CPU designs, however, the design in this case can be different. Some components maybe handling less or more functions than those in the existing CPU designs. As we started the process of building a parallel processing unit, that can execute several instructions, we started seeing the need to a unit that can distribute the data registers required for instructions as inputs or outputs of the different ALUs. At this point the system consists of a Distributer, Dependency Manager, and four ALUs.

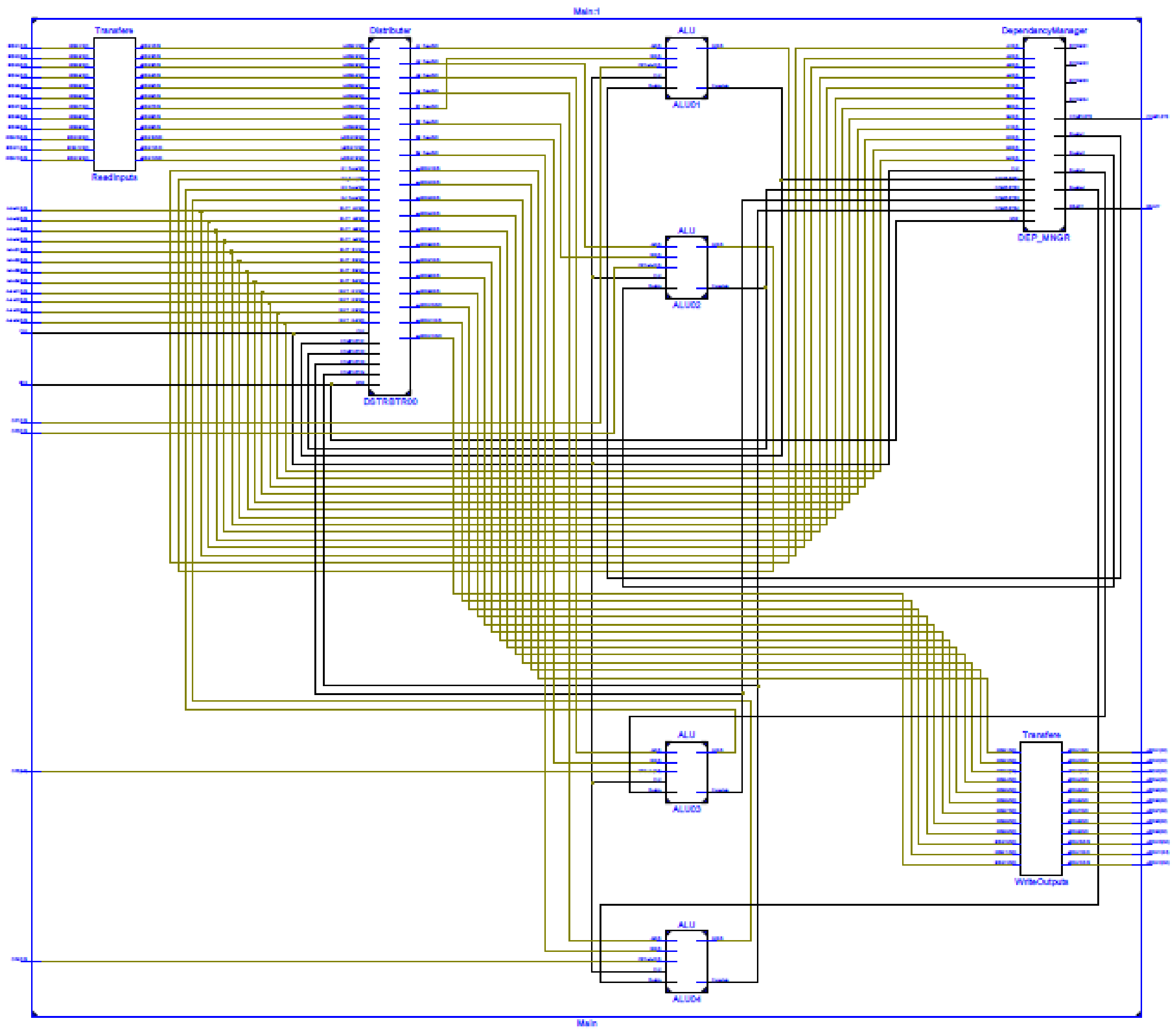

Figure 1. System internal design

2.2. System Interface

As the system is designed to address parallel processing of a predefined number of instructions, the IO interface is designed to handle four instruction sets, as well as twelve input registers that represent the input parameters, and twelve output registers for the output parameters.

- Input parameters

    - 4 X OP Code #: 3 – bits code for the operation
    - 4 X Address A #: 4 – bits for the first input register of the instruction
    - 4 X Address A #: 4 – bits for the Second input register of the instruction
    - 4 X Address A #: 4 – bits for the output register of the instruction
    - NDR: 1 – bit to inform the system that a new set of instructions is ready to process, which will then trigger the execution cycles on the rising edge of the signal.
    - CLK: 1 – bit to control the internal operation sequence of execution. The period of this clock needs to be minimum 40 ns
    - 12 X iREG #: 4 – bits for the data used as inputs for the instructions to be executed
- Output Parameters
    - Ready: 1 – bit to inform the external system that this system is ready for processing a new set of instructions
    - Complete: 1 – bit to inform the external system that this system has completed the execution of the previous set of instructions
    - 12 X oREG #: 4 – bits for the data used as outputs for the instructions to be executed

2.3. Instruction Set

The instruction structure used in this design is based on the following components. Keeping in mind that the system is processing four instructions per Execution Cycle.

[OP Code] [Input Parameter Address 1] [Input Parameter Address 2] [Output Parameter Address]

- OP Code: represents the code of the operation to be executed
- Input Parameter Address 1: register address of the first input parameter. Represents one of twelve input registers
- Input Parameter Address 2: register address of the first input parameter. Represents one of twelve input registers
- Output Parameter Address: register address of the result output parameter. Represents one of twelve output registers

2.4. Modular Design

As we progressed through the design of the ALU and the dependency manager, we started realizing the need to create a modular system, that can be expanded at any time to handle higher level of parallel processing. As the best way to design the system is by designing base modules that can handle several functions and can be duplicated or expanded as required, in order to increase the capabilities of the system [5]. With this approach, we started identifying basic functions that can be later grouped into modules, that we can build as base modules.

2.4.1. Basic Functions

The following are the basic functions performed by the system in different levels and stages of the processing cycle.

- Acquire and store instructions
- Decode instructions

- Acquire data registers called by instructions from 12 Data Registers
- Determine instruction dependencies among the acquired instructions only
- Determine number of execution sub – cycles
- Feed instructions along with data registers to the ALU
- Connect ALU inputs and outputs to the Data registers
- Determine the beginning and the end of the execution cycle
- Output results to the same data register

With this approach we were able to determine the main functions required by the system to be designed, this list allowed us to determine the design requirements of every module and required functionality of the complete system. by breaking the system into functions, we could present a clear picture of the execution cycle of the instructions to be executed simultaneously, which allowed us to design around these requirements.

2.4.2. Data Handling

Data is handled in a way to allow highest flexibility of use for the internal components of the system. This is done by creating input and output data buffers, through the transfer module.

All inputs are read into the input data buffers and used by the distributer module. They are then moved to the internal common data registers, that can be used by the rest of the modules. Common data registers are used as inputs and output of the ALUs and the dependency manager, in order to start and finish the execution cycle. At the end of the process cycle, the values of all common data registers are transferred to the output data buffers by the distributer module, then transferred to the system outputs by another instant of the transfer module.

2.5. Main Components

After we established the basic required functions of the system, then we defined that three basic modules will be the main components of the system to handle all the functions. a distributor, a dependency manager, and ALU, Are the modules that we will build to create the parallel processing system.

2.5.1. ALU

ALU is a processing unit to handle the arithmetic and logical operations to execute the operations of a program. The ALU consist of Execution code of the basic operations such as And, or, add, subtract, and shift operations.

The selection among all these operations must be optimized to eliminate the extra propagation delay from the execution of every cycle. If statements had a small propagation delay of execution [6]. However, nesting if statements created more propagation delay then the switch case statements [7, p. 3]. That is how we picked the switch case statement to design the ALU program, in order to have 15 ns maximum propagation delay for any of the instructions the ALU supports.

Before executing any instructions in the ALU, it is required to read and write information. Information interface is happening through internal registers, where the ALU can read and write by reference from the instruction set. This ALU is designed to handle seven operations as a sample to test functionality and performance

For instruction Example of (OP A B O)

1. OP (001) – O = A OR B
2. OP (010) – O = A AND B

3. OP (011) – O = A + B
4. OP (100) – O = (A – B)
5. OP (101) – O = Logical Shift of A by 1 bit to the Left Direction
6. OP (110) – O = Logical Shift of A by 1 bit to the right direction
7. OP (111) – O = Rotate Parameter A one bit to the right

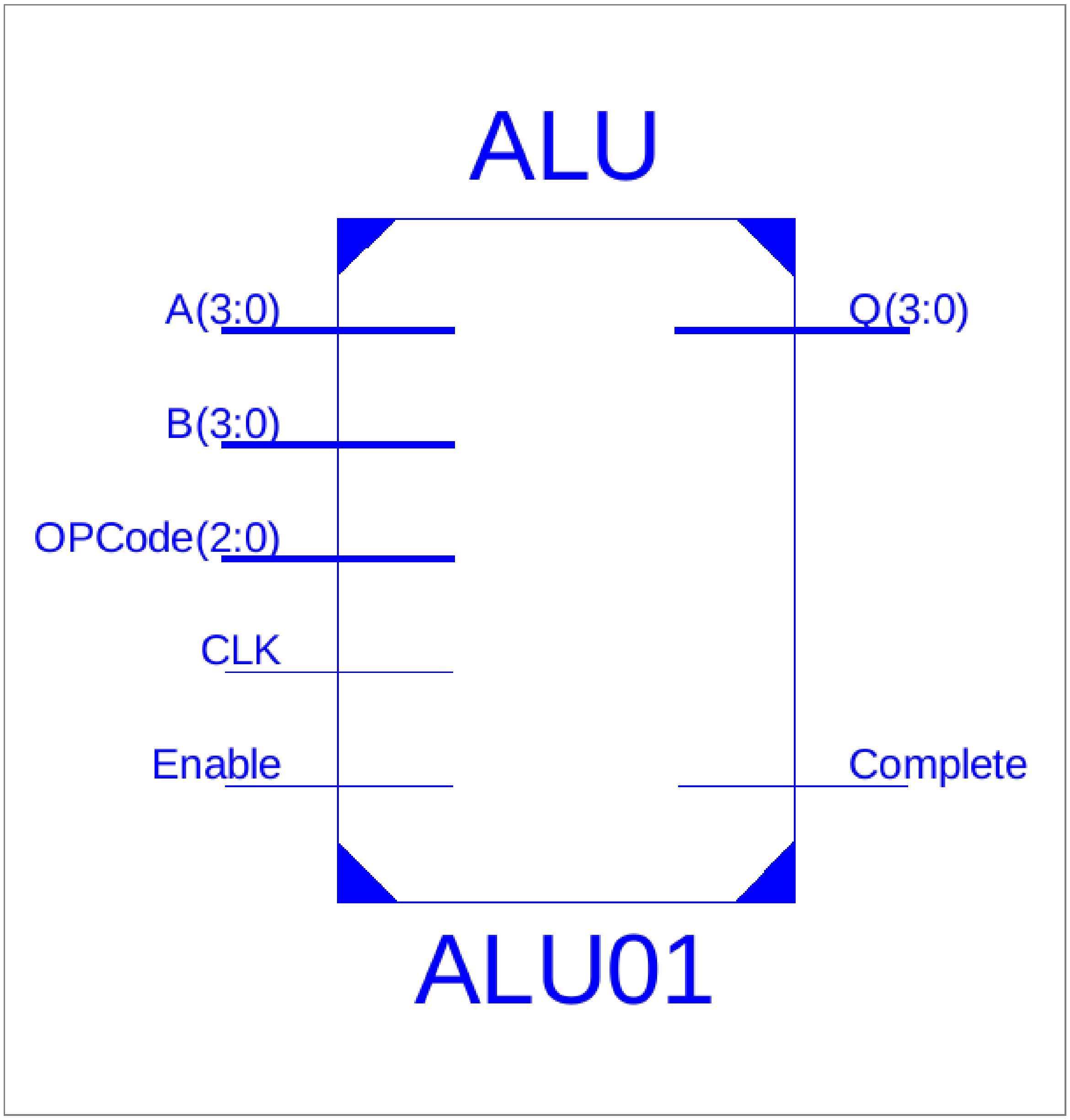

Figure 2. ALU

2.5.1.1. Input Parameters

- OP Code: The Index of the operation to be executed. Assigned directly from the system inputs
- A: The first input register address for the instruction. Assigned through the Distributer Module
- B: The Second input register address for the instruction. Assigned through the Distributer Module
- Enable: allows the ALU to execute the current instruction. Assigned through the Dependency Manager Module
- CLK: The clock signal. Assigned directly from the system inputs

2.5.1.2. Output Parameters

- Q: The output register address for the instruction. Going to the Distributer
- Complete: The complete status of the execution of the instruction. Going to the Dependency Manager and the Distributer

2.5.1.3. Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_unsigned.all;

entity ALU is
   Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
          B : in  STD_LOGIC_VECTOR (3 downto 0);
          OPCode : in  STD_LOGIC_VECTOR (2 downto 0);
           Enable :in  STD_LOGIC;
          Q : out  STD_LOGIC_VECTOR (3 downto 0) :="0000";
          Complete : out  STD_LOGIC := '0';
          CLK : in STD_LOGIC);

end ALU;
architecture Behavioral of ALU is

signal SQ: STD_LOGIC_VECTOR (3 downto 0);

begin
ALU: process(a, b, OPCode, Enable, CLK)
          Begin
                    if (Enable = '1') then  --Only Execute when this instruction is enabled
                              case (OPCode) is    -- Decoding the OPCode
                                        when "001" =>  -- OP001 = A OR B
                                                  SQ <= A or B;
                                                  if SQ = (A or B) then
                                                            Q <= SQ;
                                                            Q <= SQ;
                                                            Complete <= '1';

                                                  end if;
                                        when "010" =>  -- OP001 = A AND B
                                                  SQ <= A And B;
                                                  if SQ = (A And B) then
                                                            Q <= SQ;
                                                            Q <= SQ;
                                                            Complete <= '1';
                                                  end if;
                                        when "011" =>  -- OP001 = A + B
                                                  SQ <= A + B;
                                                  if SQ = (A + B) then
                                                            Q <= SQ;
                                                            Q <= SQ;
                                                            Complete <= '1';
                                                  end if;
                                        when "100" =>   -- OP001 = A - B
                                                  SQ <= A - B;
                                                  if SQ = (A - B) then
                                                            Q <= SQ;
                                                            Q <= SQ;
                                                            Complete <= '1';
                                                  end if;
                                        when "101" =>   -- OP001 = LOGICAL LEFT SHIFT OF A BY 1
BIT
                                                  SQ <= STD_LOGIC_VECTOR(unsigned(A) sll 1);
                                                  if SQ = (STD_LOGIC_VECTOR(unsigned(A) sll 1)) then
                                        Q <= SQ;
                                        Q <= SQ;
                                        Complete <= '1';
                                end if;
                        when "110" =>   -- OP001 = LOGICAL RIGHT SHIFT OF A BY
1 BIT
                                SQ <= STD_LOGIC_VECTOR(unsigned(A) srl 1);
                                if SQ = (STD_LOGIC_VECTOR(unsigned(A) srl 1)) then
                                        Q <= SQ;
                                        Q <= SQ;
                                        Complete <= '1';
                                end if;
                        when "111" =>  -- OP001 = ROTATE RIGHT OF A BY 1 BIT
                                SQ <= STD_LOGIC_VECTOR(unsigned(A) ror 1);
                                if SQ = (STD_LOGIC_VECTOR(unsigned(A) ror 1)) then
                                        Q <= SQ;
                                        Q <= SQ;
                                        Complete <= '1';
                                end if;
                        when others =>
                                Null;
                end case;
        end if;

    end process;
end Behavioral;
```

2.5.2. Dependency Manager

The dependency manager is responsible for deciding the number of execution sub – cycles and what instruction(s) to perform during every cycle. Dependency manager receives all four instruction sets along with their addresses by reference. Through the address of the used input and output registers, the dependency manager determines what instructions depend on what instructions, and then determines the sequence of execution by setting the enable signals. The enable signals then used by the ALUs to start the execution of the assigned instruction.

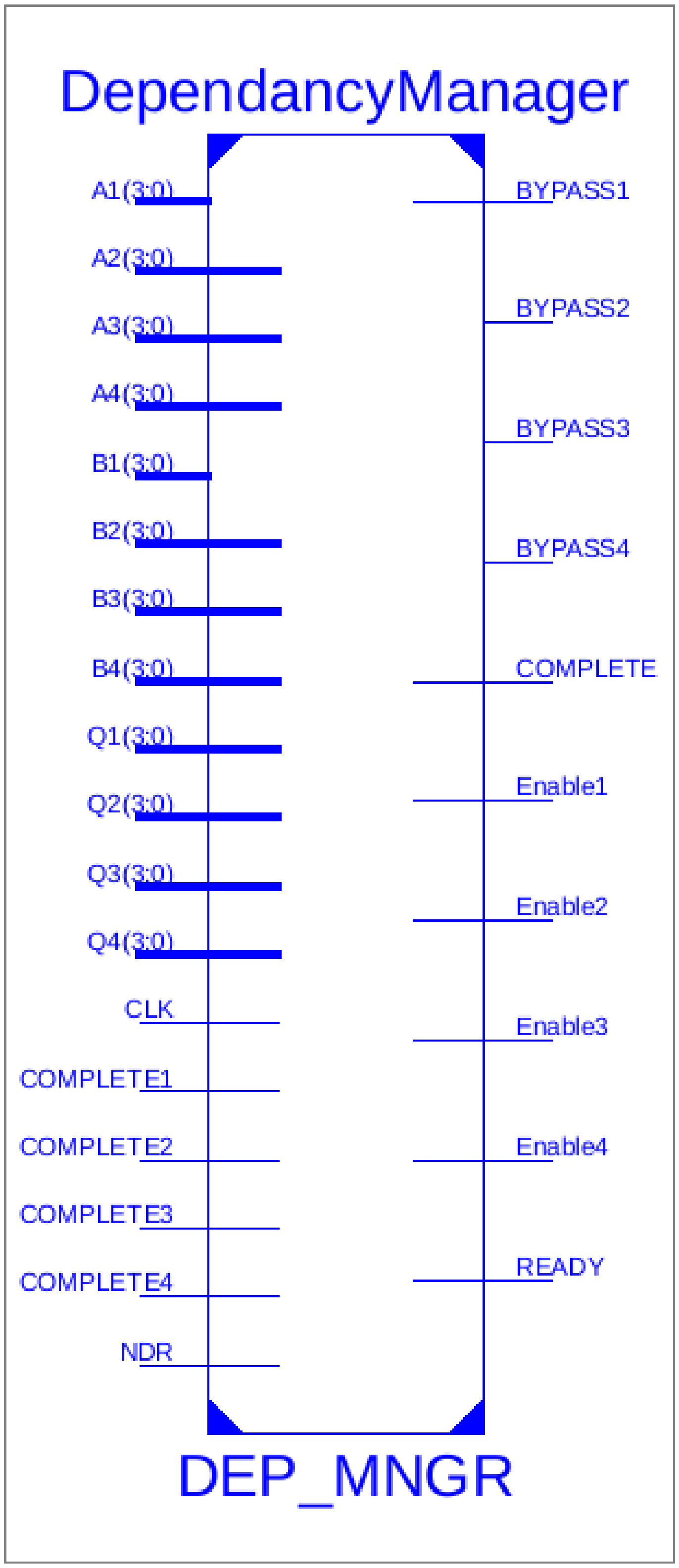

Figure 3. Dependency manager

2.5.2.1. Input Parameters

- NDR (New Data Ready): to inform the dependency manager that a new set of instructions is ready for execution. Assigned directly from the system input.
- A X: The first input register for instruction number X. Four Assigned directly from the system input.
- B X: The Second input register for instruction number X. Four Assigned directly from the system input.
- Q X: The output register for instruction number X. Four Assigned directly from the system input.
- Complete X: The complete status of the execution of instruction number X. Four Coming from the ALUs.
- CLK: The clock signal. Assigned directly from the system input.

2.5.2.2. Output Parameters

- Enable X: The enable signal to inform the ALU to process the instruction number X. Four Going to the ALUs.
- Bypass X: The Signal to inform the ALU NOT to process instruction number X. Four Going to the ALUs.
- Ready: to inform the External system that the Processing Unit is ready to process a new set of instructions. Going directly to the System Outputs.
- Complete: to inform the system that the execution of the previous set of instructions is completed. Going directly to the System Outputs.

2.5.2.3. Code

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

entity DependancyManager is
   Port ( A1 : in  STD_LOGIC_VECTOR (3 downto 0);
          B1 : in  STD_LOGIC_VECTOR (3 downto 0);
          Q1 : in  STD_LOGIC_VECTOR (3 downto 0);
          A2 : in  STD_LOGIC_VECTOR (3 downto 0);
          B2 : in  STD_LOGIC_VECTOR (3 downto 0);
          Q2 : in  STD_LOGIC_VECTOR (3 downto 0);
          A3 : in  STD_LOGIC_VECTOR (3 downto 0);
          B3 : in  STD_LOGIC_VECTOR (3 downto 0);
          Q3 : in  STD_LOGIC_VECTOR (3 downto 0);
          A4 : in  STD_LOGIC_VECTOR (3 downto 0);
          B4 : in  STD_LOGIC_VECTOR (3 downto 0);
          Q4 : in  STD_LOGIC_VECTOR (3 downto 0);
COMPLETE1 : in  STD_LOGIC;
           COMPLETE2 : in  STD_LOGIC;
           COMPLETE3 : in  STD_LOGIC;
           COMPLETE4 : in  STD_LOGIC;
           NDR : in  STD_LOGIC;
Enable1 : out  STD_LOGIC :='0';
Enable2 : out  STD_LOGIC :='0';
Enable3 : out  STD_LOGIC :='0';
          Enable4 : out  STD_LOGIC :='0';
          BYPASS1 : out  STD_LOGIC :='0';
          BYPASS2 : out  STD_LOGIC :='0';
          BYPASS3 : out  STD_LOGIC :='0';
          BYPASS4 : out  STD_LOGIC :='0';
READY: out STD_LOGIC :='0';
COMPLETE: out STD_LOGIC :='0';
CLK : in STD_LOGIC
                                  );
end DependancyManager;

architecture Behavioral of DependancyManager is

SIGNAL STARTCYCLE: STD_LOGIC := '0';
begin
DependancyGenerator : Process (CLK)
           begin
                              IF (NDR = '1') THEN   -- EXECUTE IF NEW DATA IS READY
                                        STARTCYCLE <= '1';
                                        READY <='0';
                                        COMPLETE <= '0';
                                        --Enable ALU1 as it doesn't depend on any other ALU
                                IF (Q1 /= "0000" ) THEN
                                        ENABLE1 <='1';
                                ELSE
                                        BYPASS1 <= '0';
                                END IF;

                                --Enable ALU2 if it doesn't depend on ALU1
                                if (((A2 /= Q1 AND B2 /= Q1) OR COMPLETE1 ='1') AND (Q2
/= "0000"))then
                                        Enable2 <= '1';
                                ELSIF Q2 = "0000" THEN
                                        BYPASS2 <= '1';
                                END IF;

                                --Enable ALU3 if it doesn't depend on ALU2
                                if (((A3 /= Q1 AND B3 /= Q1) OR COMPLETE1 ='1') AND ((A3
/= Q2 AND B3 /= Q2) OR COMPLETE2 ='1')) AND (Q3 /= "0000")then
                                        Enable3 <= '1';
                                ELSIF Q3 = "0000" THEN
                                        BYPASS3 <= '1';
                                end if;

                                --Enable ALU4 if it doesn't depend on ALU2 and ALU3
                                if (((A4 /= Q1 AND B4 /= Q1) OR COMPLETE1 ='1') AND ((A4
/= Q2 AND B4 /= Q2) OR COMPLETE2 ='1') AND ((A4 /= Q3 AND B4 /= Q3) OR
COMPLETE3 ='1')) AND (Q4 /= "0000") then
                                        Enable4 <= '1';
                                ELSIF Q2 = "0000" THEN
                                        BYPASS4 <= '1';
                                end if;
                        ELSE
                                STARTCYCLE <= '0';
                                READY <= '1';
                                        Enable1 <= '0';
                                Enable2 <= '0';
                                Enable3 <= '0';
                                Enable4 <= '0';
                                BYPASS1 <= '0';
                                BYPASS2 <= '0';
                                BYPASS3 <= '0';
                                BYPASS4 <= '0';
                        END IF;
```

```
                    IF (COMPLETE1 = '1' AND COMPLETE2 = '1' AND COMPLETE3 = '1' AND
COMPLETE4 = '1') THEN
                              STARTCYCLE <= '0';
                              COMPLETE <= '1';
                              READY <= '1';
                    END IF ;


          end process;

end Behavioral;
```

2.5.3. Distributer

The distributer is the unit responsible to connect the input and output data registers to all internal modules. The distributer provides the ability to read and write to all data registers by the ALUs and the Dependency Manager. Data registers are passed through from inputs to the outputs, unless they are modified by the operations performed by the ALUs, then the results of the ALU operation is applied to the data register addressed in the output field of the instruction set.

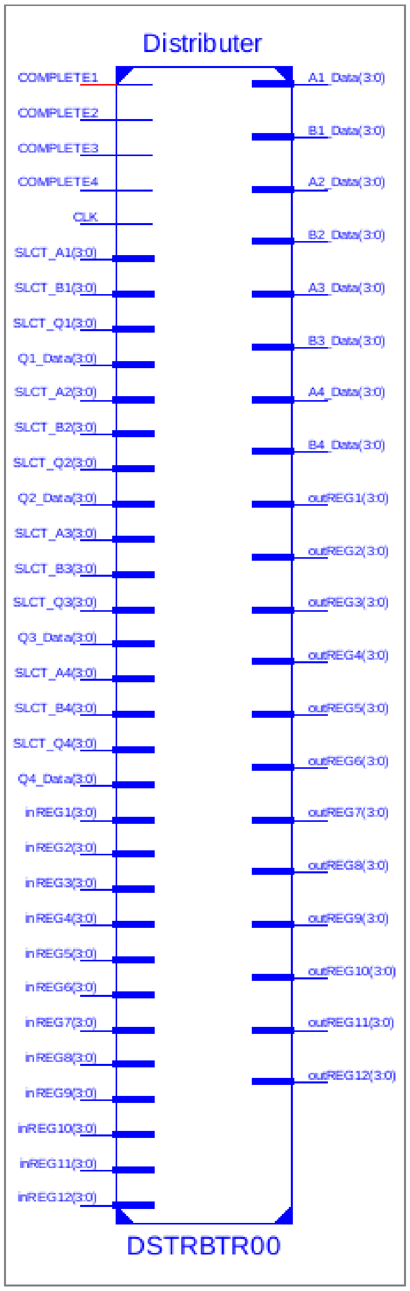

Figure 4. Distributer

2.5.3.1. Input Parameters

- 4 X SLCT_A: address of the register of A value, directly from the instruction set

- 4 X SLCT_B: address of the register of B value, directly from the instruction set
- 4 X SLCT_Q: address of the register of Q value, directly from the instruction set
- 12 X inREG: input registers that hold parameter values referenced in the instruction set
- 4 X Complete: the complete status of the each of the instruction sets. Assigned from the ALUs.
- 4 X Q#_Data: Value for parameter Q for the instruction set number #. Assigned from the ALUs.
- CLK: Clock Signal. from the system inputs

2.5.3.2. Output Parameters

- 4 X A#_Data: Value for parameter A for the instruction set number #. Going to the ALUs
- 4 X B#_Data: Value for parameter B for the instruction set number #. Going to the ALUs
- 12 X outREG: output registers that hold parameter values referenced in the instruction set. Going to the ALUs

2.5.3.3. Code

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

entity Distributer is
   Port ( SLCT_A1 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_B1 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_Q1 : in  STD_LOGIC_VECTOR (3 downto 0);
          A1_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          B1_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          Q1_Data : in  STD_LOGIC_VECTOR (3 downto 0);
                              SLCT_A2 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_B2 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_Q2 : in  STD_LOGIC_VECTOR (3 downto 0);
          A2_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          B2_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          Q2_Data : in  STD_LOGIC_VECTOR (3 downto 0);
                              SLCT_A3 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_B3 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_Q3 : in  STD_LOGIC_VECTOR (3 downto 0);
          A3_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          B3_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          Q3_Data : in  STD_LOGIC_VECTOR (3 downto 0);
                              SLCT_A4 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_B4 : in  STD_LOGIC_VECTOR (3 downto 0);
          SLCT_Q4 : in  STD_LOGIC_VECTOR (3 downto 0);
          A4_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          B4_Data : out  STD_LOGIC_VECTOR (3 downto 0) := "0000";
          Q4_Data : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG1 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG2 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG3 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG4 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG5 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG6 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG7 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG8 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG9 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG10 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG11 : in  STD_LOGIC_VECTOR (3 downto 0);
          inREG12 : in  STD_LOGIC_VECTOR (3 downto 0);
          outREG1 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG2 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG3 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG4 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG5 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG6 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG7 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG8 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG9 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG10 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG11 : out  STD_LOGIC_VECTOR (3 downto 0);
          outREG12 : out  STD_LOGIC_VECTOR (3 downto 0);
          COMPLETE1 : in  STD_LOGIC;
          COMPLETE2 : in  STD_LOGIC;
          COMPLETE3 : in  STD_LOGIC;
          COMPLETE4 : in  STD_LOGIC;
                          CLK : in STD_LOGIC
                          );
end Distributer;

architecture Behavioral of Distributer is

Signal REG1 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG2 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG3 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG4 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG5 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG6 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG7 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG8 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG9 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG10 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG11 : STD_LOGIC_VECTOR (3 downto 0);
Signal REG12 : STD_LOGIC_VECTOR (3 downto 0);

begin
Main : process( COMPLETE1,COMPLETE2,COMPLETE3,COMPLETE4, SLCT_A1,
SLCT_B1, SLCT_Q1, SLCT_A2, SLCT_B2, SLCT_Q2,SLCT_A3, SLCT_B3,
SLCT_Q3,SLCT_A4, SLCT_B4, SLCT_Q4)
          Begin
--IF ASSIGN ALL INPUT REGISTERS TO OUTPUT REGISTERS IN THE BEGINNING OF
THE CYCLE
                    if complete1 /= '1' and complete1 /= '1' and complete1 /= '1' and complete1 /= '1'
then
                              outREG1 <= inREG1;
                              outREG2 <= inREG2;
                              outREG3 <= inREG3;
                              outREG4 <= inREG4;
                              outREG5 <= inREG5;
```

```
                outREG6 <= inREG6;
                outREG7 <= inREG7;
                outREG8 <= inREG8;
                outREG9 <= inREG9;
                outREG10 <= inREG10;
                outREG11 <= inREG11;
                outREG12 <= inREG12;
        end if;



--ASSIGN INPUT REGISTER TO THE INPUT PORTS OF THE ALUs

        if complete1 /= '1' then
                if  SLCT_A1 > "0000" then
                        case SLCT_A1 is
                                when "0001" =>
                                        A1_Data<=inREG1;
                                when "0010" =>
                                        A1_Data<=inREG2;
                                when "0011" =>
                                        A1_Data<=inREG3;
                                when "0100" =>
                                        A1_Data<=inREG4;
                                when "0101" =>
                                        A1_Data<=inREG5;
                                when "0110" =>
                                        A1_Data<=inREG6;
                                when "0111" =>
                                        A1_Data<=inREG7;
                                when "1000" =>
                                        A1_Data<= inREG8;
                                when "1001" =>
                                        A1_Data<= inREG9;
                                when "1010" =>
                                        A1_Data<= inREG10;
                                when "1011" =>
                                        A1_Data<= inREG11;
                                when "1100" =>
                                        A1_Data<= inREG12;
                                when others =>
                                        null;
                        end case;
                end if;
                if  SLCT_B1 > "0000"  then
                        case SLCT_B1 is
```

```
                    when "0001" =>
                        B1_Data<= inREG1;
                    when "0010" =>
                        B1_Data<= inREG2;
                    when "0011" =>
                        B1_Data<= inREG3;
                    when "0100" =>
                        B1_Data<= inREG4;
                    when "0101" =>
                        B1_Data<= inREG5;
                    when "0110" =>
                        B1_Data<= inREG6;
                    when "0111" =>
                        B1_Data<= inREG7;
                    when "1000" =>
                        B1_Data<= inREG8;
                    when "1001" =>
                        B1_Data<= inREG9;
                    when "1010" =>
                        B1_Data<= inREG10;
                    when "1011" =>
                        B1_Data<= inREG11;
                    when "1100" =>
                        B1_Data<= inREG12;
                    when others =>
                        null;
                end case;
            end if ;
        end if;
        if complete1 = '1' then
            case SLCT_Q1 is
                when "0001" =>
                    outREG1 <= Q1_Data;
                when "0010" =>
                    outREG2 <= Q1_Data;
                when "0011" =>
                    outREG3 <= Q1_Data;
                when "0100" =>
                    outREG4 <= Q1_Data;
                when "0101" =>
                    outREG5 <= Q1_Data;
                when "0110" =>
                    outREG6 <= Q1_Data;
                when "0111" =>
                    outREG7 <= Q1_Data;
                when "1000" =>
```

```
                    outREG8 <= Q1_Data;
                when "1001" =>
                    outREG9 <= Q1_Data;
                when "1010" =>
                    outREG10 <= Q1_Data;
                when "1011" =>
                    outREG11 <= Q1_Data;
                when "1100" =>
                    outREG12 <= Q1_Data;
                when others =>
                    null;
            end case;
        end if;


--/////////////////////////////////////////////



        if COMPLETE2 /= '1'  then

            if SLCT_A2 > "0000" then
                if SLCT_A2 = SLCT_Q1 then
                    A2_Data <= Q1_Data;
                else
                    case SLCT_A2 is
                    when "0001" =>
                            A2_Data <= inREG1;
                        when "0010" =>
                            A2_Data <= inREG2;
                        when "0011" =>
                            A2_Data <= inREG3;
                        when "0100" =>
                            A2_Data <= inREG4;
                        when "0101" =>
                            A2_Data <= inREG5;
                        when "0110" =>
                            A2_Data <= inREG6;
                        when "0111" =>
                            A2_Data <= inREG7;
                        when "1000" =>
                            A2_Data <= inREG8;
                        when "1001" =>
                            A2_Data <= inREG9;
                        when "1010" =>
                            A2_Data <= inREG10;
```

```
                                        when "1011" =>
                                                A2_Data <= inREG11;
                                        when "1100" =>
                                                A2_Data <= inREG12;
                                        when others =>
                                                null;
                                end case;
                        end if;
                end if;

                if SLCT_B2 > "0000" then
                        if SLCT_B2 = SLCT_Q1 then
                                B2_Data <= Q1_Data;
                        else
                                case SLCT_B2 is
                                        when "0001" =>
                                                B2_Data <= inREG1;
                                        when "0010" =>
                                                B2_Data <= inREG2;
                                        when "0011" =>
                                                B2_Data <= inREG3;
                                        when "0100" =>
                                                B2_Data <= inREG4;
                                        when "0101" =>
                                                B2_Data <= inREG5;
                                        when "0110" =>
                                                B2_Data <= inREG6;
                                        when "0111" =>
                                                B2_Data <= inREG7;
                                        when "1000" =>
                                                B2_Data <= inREG8;
                                        when "1001" =>
                                                B2_Data <= inREG9;
                                        when "1010" =>
                                                B2_Data <= inREG10;
                                        when "1011" =>
                                                B2_Data <= inREG11;
                                        when "1100" =>
                                                B2_Data <= inREG12;
                                        when others =>
                                                null;
                                end case;
                        end if;
                end if;
        end if;
```

```
                if Complete2 = '1' then
                        case SLCT_Q2 is
                                when "0001" =>
                                        outREG1 <= Q2_Data;
                                when "0010" =>
                                        outREG2 <= Q2_Data;
                                when "0011" =>
                                        outREG3 <= Q2_Data;
                                when "0100" =>
                                        outREG4 <= Q2_Data;
                                when "0101" =>
                                        outREG5 <= Q2_Data;
                                when "0110" =>
                                        outREG6 <= Q2_Data;
                                when "0111" =>
                                        outREG7 <= Q2_Data;
                                when "1000" =>
                                        outREG8 <= Q2_Data;
                                when "1001" =>
                                        outREG9 <= Q2_Data;
                                when "1010" =>
                                        outREG10 <= Q2_Data;
                                when "1011" =>
                                        outREG11 <= Q2_Data;
                                when "1100" =>
                                        outREG12 <= Q2_Data;
                                when others =>
                                        null;
                        end case;
                end if;
--////////////////////////////////////////////

                        if COMPLETE3 /= '1'  then
                                if SLCT_A3 > "0000" then
                                        if SLCT_A3 = SLCT_Q1 then
                                                A3_Data <= Q1_Data;
                                        elsif SLCT_A3 = SLCT_Q2 then
                                                A3_Data <= Q2_Data;
                                        else
                                                case SLCT_A3 is
                                                        when "0001" =>
                                                                A3_Data <= inREG1;
                                                        when "0010" =>
                                                                A3_Data <= inREG2;
                                                        when "0011" =>
                                                                A3_Data <= inREG3;
                                                when "0100" =>
                                                        A3_Data <= inREG4;
                                                when "0101" =>
                                                        A3_Data <= inREG5;
                                                when "0110" =>
                                                        A3_Data <= inREG6;
                                                when "0111" =>
                                                        A3_Data <= inREG7;
                                                when "1000" =>
                                                        A3_Data <= inREG8;
                                                when "1001" =>
                                                        A3_Data <= inREG9;
                                                when "1010" =>
                                                        A3_Data <= inREG10;
                                                when "1011" =>
                                                        A3_Data <= inREG11;
                                                when "1100" =>
                                                        A3_Data <= inREG12;
                                                when others =>
                                                        null;
                                        end case;
                                end if;
                        end if;
                        if SLCT_B3 > "0000" then
                                if SLCT_B3 = SLCT_Q1 then
                                        B3_Data <= Q1_Data;
                                elsif SLCT_B3 = SLCT_Q2 then
                                        B3_Data <= Q2_Data;
                                else
                                        case SLCT_B3 is
                                                when "0001" =>
                                                        B3_Data <= inREG1;
                                                when "0010" =>
                                                        B3_Data <= inREG2;
                                                when "0011" =>
                                                        B3_Data <= inREG3;
                                                when "0100" =>
                                                        B3_Data <= inREG4;
                                                when "0101" =>
                                                        B3_Data <= inREG5;
                                                when "0110" =>
                                                        B3_Data <= inREG6;
                                                when "0111" =>
                                                        B3_Data <= inREG7;
                                                when "1000" =>
                                                        B3_Data <= inREG8;
                                                when "1001" =>
                                                        B3_Data <= inREG9;
                                                when "1010" =>
                                                        B3_Data <= inREG10;
                                                when "1011" =>
                                                        B3_Data <= inREG11;
                                                when "1100" =>
                                                        B3_Data <= inREG12;
                                                when others =>
                                                        null;
                                        end case;
                                end if;
                        end if;
                end if;

                if complete3 = '1' then
                        case SLCT_Q3 is
                                when "0001" =>
                                        outREG1 <= Q3_Data;
                                when "0010" =>
                                        outREG2 <= Q3_Data;
                                when "0011" =>
                                        outREG3 <= Q3_Data;
                                when "0100" =>
                                        outREG4 <= Q3_Data;
                                when "0101" =>
                                        outREG5 <= Q3_Data;
                                when "0110" =>
                                        outREG6 <= Q3_Data;
                                when "0111" =>
                                        outREG7 <= Q3_Data;
                                when "1000" =>
                                        outREG8 <= Q3_Data;
                                when "1001" =>
                                        outREG9 <= Q3_Data;
                                when "1010" =>
                                        outREG10 <= Q3_Data;
                                when "1011" =>
                                        outREG11 <= Q3_Data;
                                when "1100" =>
                                        outREG12 <= Q3_Data;
                                when others =>
                                null;
                        end case;
                end if;
        --///////////////////////////////////////////////////////////////
```

```
if complete4 /= '1' then
        if SLCT_A4 > "0000" then
                if SLCT_A4 = SLCT_Q1 then
                        A4_Data <= Q1_Data;
                elsif SLCT_A4 = SLCT_Q2 then
                        A4_Data <= Q2_Data;
                elsif SLCT_A4 = SLCT_Q3 then
                        A4_Data <= Q3_Data;
                else
                        case SLCT_A4 is
                                when "0001" =>
                                        A4_Data <= inREG1;
                                when "0010" =>
                                        A4_Data <= inREG2;
                                when "0011" =>
                                        A4_Data <= inREG3;
                                when "0100" =>
                                        A4_Data <= inREG4;
                                when "0101" =>
                                        A4_Data <= inREG5;
                                when "0110" =>
                                        A4_Data <= inREG6;
                                when "0111" =>
                                        A4_Data <= inREG7;
                                when "1000" =>
                                        A4_Data <= inREG8;
                                when "1001" =>
                                        A4_Data <= inREG9;
                                when "1010" =>
                                        A4_Data <= inREG10;
                                when "1011" =>
                                        A4_Data <= inREG11;
                                when "1100" =>
                                        A4_Data <= inREG12;
                                when others =>
                                        null;
                        end case;
                end if;
        end if;

        if SLCT_B4 > "0000"  then
                if SLCT_B4 = SLCT_Q1 then
                        B4_Data <= Q1_Data;
                elsif SLCT_B4 = SLCT_Q2 then
                        B4_Data <= Q2_Data;
                elsif SLCT_B4 = SLCT_Q3 then
```

```
                    B4_Data <= Q3_Data;
                else
                    case SLCT_B4 is
                        when "0001" =>
                            B4_Data <= inREG1;
                        when "0010" =>
                            B4_Data <= inREG2;
                        when "0011" =>
                            B4_Data <= inREG3;
                        when "0100" =>
                            B4_Data <= inREG4;
                        when "0101" =>
                            B4_Data <= inREG5;
                        when "0110" =>
                            B4_Data <= inREG6;
                        when "0111" =>
                            B4_Data <= inREG7;
                        when "1000" =>
                            B4_Data <= inREG8;
                        when "1001" =>
                            B4_Data <= inREG9;
                        when "1010" =>
                            B4_Data <= inREG10;
                        when "1011" =>
                            B4_Data <= inREG11;
                        when "1100" =>
                            B4_Data <= inREG12;
                        when others =>
                            null;
                    end case;
                end if;
            end if;
        end if;
        if complete4 = '1' then
            case SLCT_Q4 is
                when "0001" =>
                    outREG1 <= Q4_Data;
                when "0010" =>
                    outREG2 <= Q4_Data;
                when "0011" =>
                    outREG3 <= Q4_Data;
                when "0100" =>
                    outREG4 <= Q4_Data;
                when "0101" =>
                    outREG5 <= Q4_Data;
                when "0110" =>
```

```
                                        outREG6 <= Q4_Data;
                                when "0111" =>
                                        outREG7 <= Q4_Data;
                                when "1000" =>
                                        outREG8 <= Q4_Data;
                                when "1001" =>
                                        outREG9 <= Q4_Data;
                                when "1010" =>
                                        outREG10 <= Q4_Data;
                                when "1011" =>
                                        outREG11 <= Q4_Data;
                                when "1100" =>
                                        outREG12 <= Q4_Data;
                                when others =>
                                        null;
                        end case;
                end if;
        end process;
end Behavioral;
```

2.6. Supporting Components

2.6.1. Transfer

A module that is instantiated twice, once for transferring the input data registers to input data buffers, and then instantiated for transferring the output data buffers to the output data registers. Input and output data buffers are consumed and produced by all the main components of the system, and only connect to the outside of the system through the transfer modules.

2.6.1.1. Input Parameters

- 12 X iREG#: Value for input data register #

2.6.1.2. Output Parameters

12 X oREG#: Value for output data register #

Chapter 3: Comparative Performance

3.1. Comparison with Existing Work

There are many methods used for designing processing units, the most common method is called Pipelining. Pipelining is the method where the processing units are designed around the concept of dividing the execution cycle to several sequential stages. By using processing units as many as the number of stages, then transfer the executed instruction among the different stages. Below is the comparison between the Pipelining and parallel processing scheme done in this project. Assuming the same hardware is used. [8, p. 370]

	Pipelining	Parallel Processing
Stages	Fetch ➔ Decode ➔ Execute ➔ Write	Data Distribution ➔ Execute
Timing	Assume every stage takes 100 ps, and we have four processing units. Executing 1000 instructions will take $(4+99) \times 100 = 103\ ns$	$\frac{1000}{4} \times (100 * 2) = 50\ ns$
Dependency Delay	One Dependency will take four stages delay (400 ps)	One Dependency will take one stage delay (100 ps)
Dependency Management	Done through the compiler	Done through the Dependency manager module (Internal to the processing unit)
Scalability	Can only scale to more than number of stages	More flexible scalability to any number of ALUs

Table 1. Pipelining vs. parallel processing table

3.2. Advantages Introduced Over Sequential Design

Sequential processing requires a single processing unit, and all instruction must be processed through the same processing unit. Parallel processing design has many advantages over the sequential processing method. If we assume using the same hardware, as well as the same stages in designing the processing unit, we can get the following comparison

	Sequential	Parallel
Stages	Data Distribution ➔ Execute	Data Distribution ➔ Execute
Timing	$1000 \times (100 * 2) = 200.00\, ns$	$\frac{1000}{4} \times (100 * 2) = 50\, ns$
Dependency Delay	0	One Dependency will take one stage delay (100 ps)
Dependency Management	Not needed, as it is handled by the sequential processing of instructions	Done through the Dependency manager module (Internal to the processing unit)
Scalability	Not possible as it requires processing all instructions through the same processing unit	More flexible scalability to any number of ALUs

Table 2. Sequential vs. parallel table

Chapter 4: Challenges

4.1. Data Handling

Data handling represents major parts from the Execution Cycle Time. data handling starts as soon as the system receives the NDR signal and only ends when the system sends out the complete signal. Choosing to read the data registers, when and how to make them available to the ALUs and the other parts of the system, was on the challenges in this project.

There are a few different ways used in the industry, some involve fixed number of data registers, others involve memory registers references. The route we chose to take to have a fixed number of data registers to interface with the system to simplify the data interface design with the external system. Where the system reads and writes to the same set of data registers in every execution cycle [9]. However, internally, the values of the data registers will be assigned to the buffers registers by the Transfer modules, then the buffers themselves are assigned to the input and output ports of the ALUs by the Distributer module, where the last will work to introduce the input values to the ALUs before they start processing instructions, and sends the results output data to the buffers after the ALUs finish processing the instructions. By having a single entity managing the data ins and outs of the ALUs, this method allowed me to asynchronously manage the data handling while processing the instructions, and independent from the dependencies that can occur among the instructions to be processed.

This data management method has some advantages, where it works asynchronously with the ALUs. And has some disadvantages, where it takes a large part of the execution total cycle time.

4.2. Synchronous and Asynchronous Execution

Choosing between the synchronous and asynchronous execution scheme was one of the challenges as most of the processing units designs have synchronous cycles, where every operation happens following the clock signal.

In this case and taking an advantage of the FPGA capabilities, we chose to have synchronous and asynchronous interlocks among the different modules of the system, and these modules will react and execute their cycles in response to the interlocks from other modules or the clock. For example, the ALU will react to the change in value of any of its inputs or the clock signal, where the dependency manager is executing at every clock cycle, and the distributer is processing only when it reads a change in any of its input values.

This approach gave the flexibility to the system to react to the right events only and allows for future optimizations.

4.3. Interlocks (Handshakes)

Interlock among the internal components of the system is a major aspect for the system to perform the full execution cycle as a unit. Choosing when to trigger signals, how to respond to signals, and what signals to respond to, was a vital part of making the system function correctly during the execution cycle. This part required a lot of test and simulations to have every component function in the right time, every signal to be generated in the right time, and every result to be put on the output in the right time.

Chapter 5: Execution Cycle

The execution cycle starts when the system reads the rising edge of the NDR signal. The system will read and store OP-CODEs and data values required for the set of instructions to be processed, the dependency manager will determine the number of execution by setting the Enable signals to the ALUs, Bypass signals, and receiving the Complete signals from the ALUs. At every completing of an ALU cycle, the Distributer will update the effected data registers, and the dependency manager will enable the next execution of instruction. As all instruction are executed, the system sets the COMPLETE output to inform the external system that the four instructions are executed, and the results are stored in the addressed registers. Below are more details of the execution cycle.

5.1. Data Preparation

First step in the execution cycle is when the system receives the NDR (New Data Ready) signal. The rising edge of the NDR signal will trigger the Distributer module to transfer the data stored in the input data buffers to the common internal data registers. The common data registers referred to as

- AX_Data receives the data from the first input register buffer addressed in the instruction set to be executed by ALU number X
- BX_Data receives the data from the Second input register buffer addressed in the instruction set to be executed by ALU number X

- QX_Data used to output the data from the output data register buffer addressed in the instruction set to be executed by ALU number X
- Where X is the number of the ALU unit that uses these registers. At this point all output data buffers are copies of the input data buffers.

The ALU specific data registers are used and updated as needed every execution cycle by the ALUs, then used by the distributer to update the input and output data buffers, to be used in the next execution cycle in the case of any instruction dependencies. The handshake among all ALUs and the distributer every execution cycle, allows the distributer module to keep the input and output data buffers up to date triggered by the complete of an execution cycle, and ready for the next execution cycle to start using the new values needed as inputs.

5.2. Data Dependency

Instructions are said to be dependent if a later instruction is using a data input that was generated or modified by an earlier instruction, for example

14 Add A B C

15

16

17

18 Sub C D E

In the example above, the instruction in line 18 is using data register C that was modified by the instruction earlier in line 14. In this case we say the instruction in line 18 is dependent on the execution of the instruction in line 14.

In the case of our system design, the Dependency Manager module determines the dependencies among all instructions that are loaded in the current cycle, which happens to be four instruction only for the capacity of the system and for the sake of simpler testing.

5.3. Instruction Execution Cycle

The start and end events of the execution of every instruction in the ALUs are independently interlocked with dependency manager. The Dependency Manager is driving the execution flow through out all ALUs, driven by the dependencies among all instruction in the loaded set of instructions in the current execution cycle. In this case, the size of the set of instructions is four instructions to be processed simultaneously unless they have dependencies.

Every instruction in this step will be assigned to an ALU in the sequence of appearance in the source code. For example, if this execution cycle is processing instructions 21 – 24, then the assignments will be as follows

- INST 21 ➔ ALU1
- INST 22 ➔ ALU2
- INST 23 ➔ ALU3
- INST 24 ➔ ALU4

This will allow pre-assignments of the data registers to all ALUs before the execution cycle starts, in the same time that the Dependency Manager is determining what ALUs to be enabled for the next execution. By the time the dependency manager sets the enable signal of the required ALUs, the data processed by the assigned instruction is ready at the input ports of the assigned ALU.

After the ALU receives the Enable signal, it will start executing the instruction using the data on the ports right at that moment. The ALU will use the OP-CODE to select what operation to perform on the data. In the case of this design, we only have seven operations to choose from, and that can be expanded as much as required for the field of usage of this processing unit. After the ALU chooses the operation, it performs the operation using the data from the input registers, then write the results of the operation to the output register. At this point and after the operation is completed, the ALU will set the Complete signal output, in order to inform the system that the execution cycle of the current instruction is completed.

When ALUs finish their operations and generate their complete signals, the complete signal of every ALU is used by the Distributer and the dependency manager for two different purposes. The Distributor uses the ALU Complete signal to update the values stored in the common registers as well as the input registers to all ALUs, in order to have all data ready to start performing the next instruction execution cycle. However, the dependency manager will use the ALU complete signal to trigger the enable signal of the next dependent instruction execution, by setting the enable signal to trigger the start of operation of the assigned ALU.

In the case where the OP-CODE of an instruction happens to be 000, then the Dependency Manager will set a Bypass signal to the assigned ALU to inform the ALU that this instruction execution is bypassed and not to be processed. In this case the ALU will not perform any operation, and the Dependency Manager will not be looking for the completion of that instruction.

After the dependency manager has all the Complete and Bypass signals for all the instructions in this execution cycle, the Dependency Manager will set the Complete output signal, in order to inform the external system that the execution of all instruction, in the currently loaded set of instructions is completed, and the output results are to be stored in the output data buffers, where the output instant of the Transfer module puts that data on the data output ports.

Chapter 6: Test Cases and Simulation

6.1. Test Cases

The test cases required to measure the system performance and predict best case and worst-case scenarios are defined to test the restrains of the system design. As the system is designed to:

- Fetch and Decode the instruction set: by assigning every portion of the instruction set to the corresponding internal register to be used by the modules that need it.
- Read and assign data registers: by reading the addresses of the required read and write data registers from each of the four instruction sets, the system will assign the values to the corresponding ALUs
- Processing Data: after ALUs receive all required data, OP-CODE and the enable signal, they will decode and perform the OP-CODE, put the results out to the assigned internal data register, then trigger the complete signal
- Report Results: after all instruction sets are executed, and data was assigned and stored in the internal registers, the system will report all data to the external ports to be collected by the external system.

To be able to test all the above stages of the execution cycle, and what is the range of the execution cycle time for the best case scenario, where all instructions are executed simultaneously in a single execution cycle, and the worst case where every one of the instruction needs the results of the previous instruction. With this approach, we believe the results will show the importance of every stage of the system, and how it effects the performance of the whole system. With this feedback, we will have a good idea on what is the best part of the system to improve for the next step of development.

6.2. Simulation

Two simulation programs were written to test the functionality of the system in behavioral simulation a software environment. Then we wrote a "Post Routing" simulation to predict the actual performance of the FPGA chip including delays, overlaps, and physical properties that may affect the performance of the design. The Clock signal in all simulations was kept to 6 ns period with 50% Duty Cycle.

6.2.1. Behavioral Simulation

As shown in the Figure below. This simulation shows the complete process of the system from the rising edge of the NDR signal until the rising edge of the Complete signal. this simulation shows that the system is theoretically functional in the level of the VHDL code. In the simulation graph below, we can conclude the following

- The system has a clock of 6 ns and 50% duty cycle
- 3 ns: NDR (New Data Ready) signal started at
- 9 ns: The Distributer finishes the data distribution
- 9 ns: The Dependency Manager detects the NDR and resets the Ready Signal
- 9 ns: Dependency Manager sets the Enable signals to all ALUs[1]
- 15 ns: ALUs set their Complete Signals
- 21 ns: ALUs results are populated to the output registers
- 21 ns: Dependency Manager Sets the Complete Signal and the Ready Signal to accept a new set of instructions

[1] All ALUs are enabled because instructions are not dependent on each other

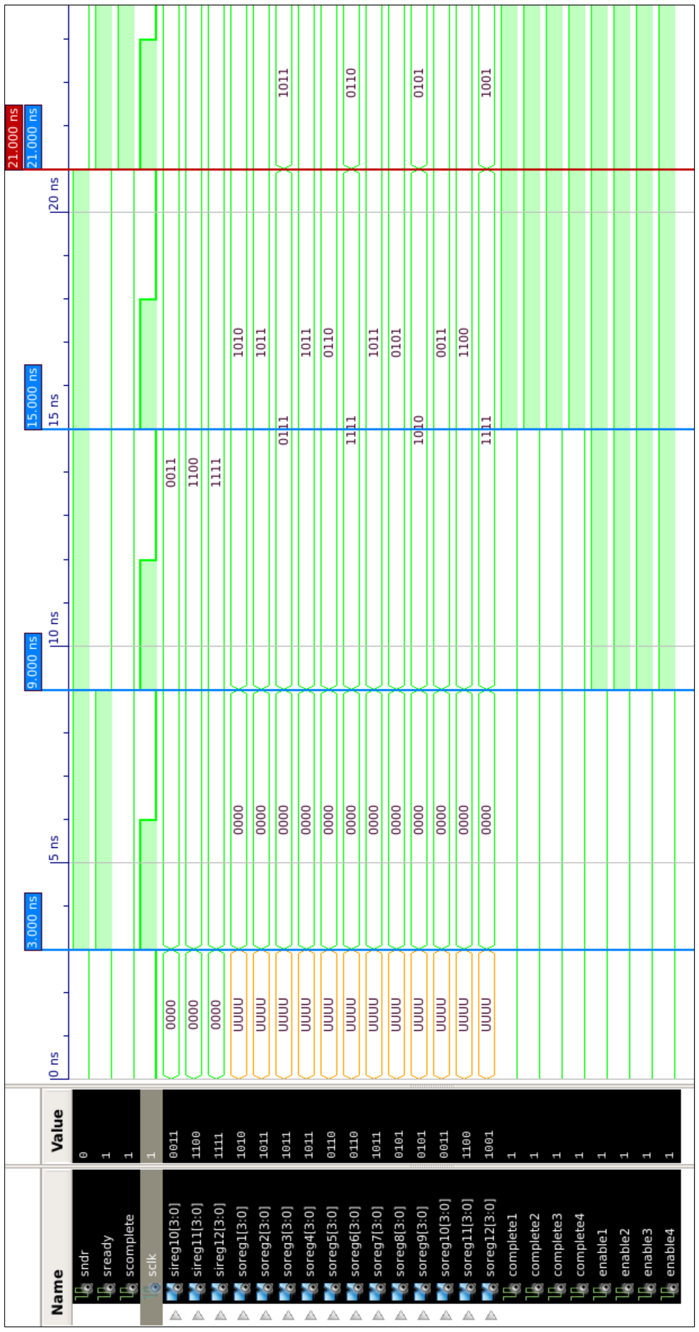

Figure 5. Behavioral simulation

6.2.2. Post Routing Simulation

Post Routing simulation is used to simulate the functionality of the VHDL code, with respect to the used hardware, in order to have a closer approximation of the functionality, speed, and most of the expected issues that can be caused by hardware special characteristics, like delays, capacity, capability, or support to some instructions.

As shown in the figures below, going from no dependencies, where all instructions are loaded and processed simultaneously, the total processing time going around 24 ns for all 4 instructions. Then as shown in the second simulation that represents the worst case scenario, where the second instruction depends on the output of the first instruction, the third instruction depends on the output of the second instruction, and fourth instruction depends on the results of the third instruction. In this case we can experience the maximum execution cycle time to process all instruction individually. And in this case the total execution time increased to around 78 ns.

6.2.3. Calculations

From the simulation graphs below, the execution cycle calculations are as follows.

- Clock:
  - Period = 6 ns
  - Duty Cycle = 50%
- Load input data = 3 ns
- Instruction execution cycle = 18 ns
- Write results to the output = 3 ns

- With these numbers, we can calculate the processing speed of this processing unit. If we consider that we are processing:
    - 4 instructions
    - 3 parameters per instruction
    - 4 bits per parameter
    - 24 ns / cycle

$$processing\ rate = \frac{4\ \times 3\ \times 4}{24\ \times 10^{-9}} = \frac{48}{24\ \times 10^{-9}} = 2\ \times 10^{9}\,\frac{bit}{sec}$$

$$Speed = \frac{1}{Clock\ period} = \frac{1}{6\ \times 10^{-9}} = 166\ MHz$$

$$at\,\frac{4}{3}\ (Cycles/instruction)$$

$$every\ dependancy\ adds\ 18\ ns\ (3\ Cycles)\ delay$$

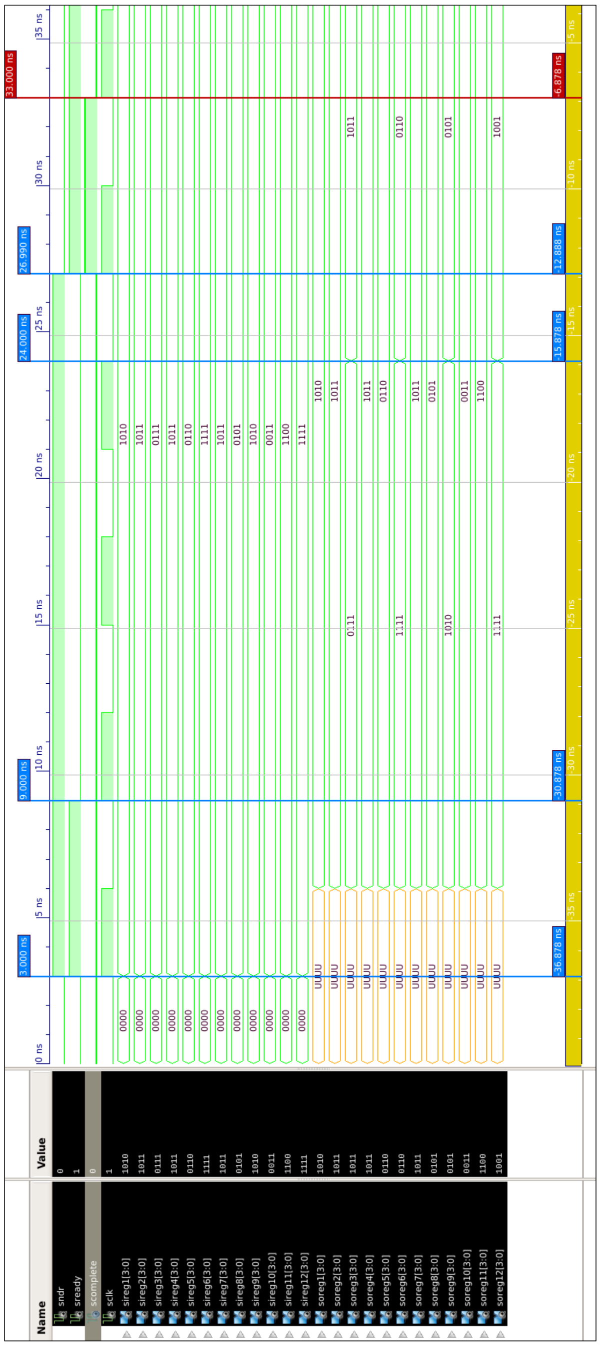

Figure 6. Post routing simulation - no dependencies

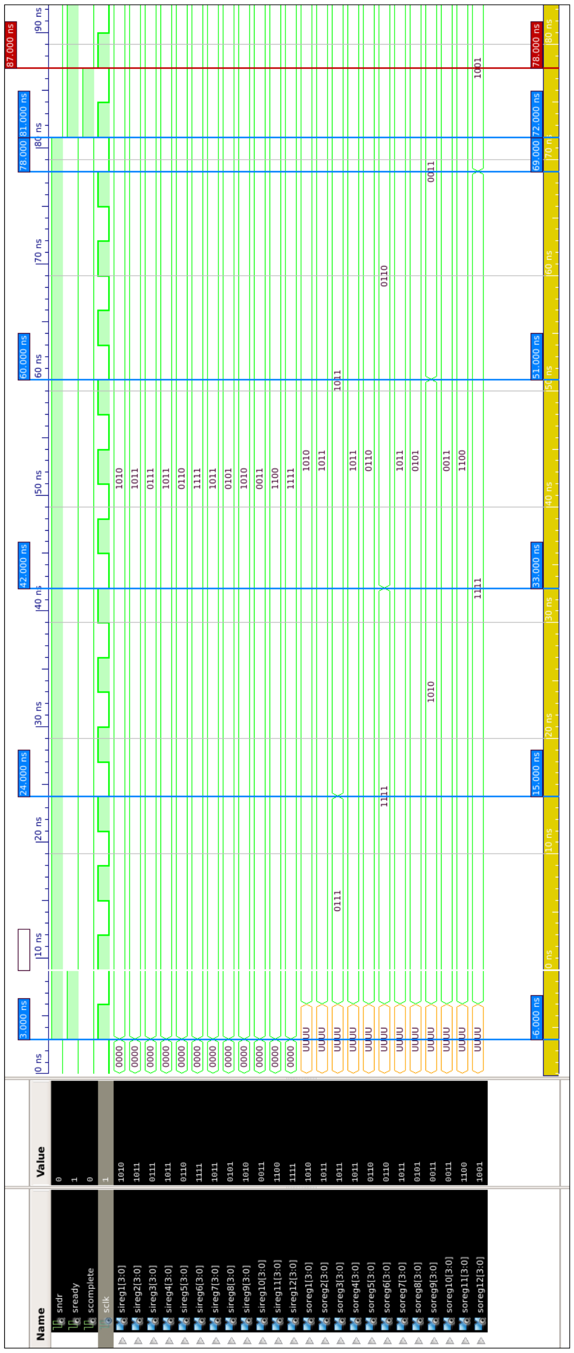

Figure 7. Post routing simulation - 3 dependencies (worst case)

Chapter 7: Conclusion and Next Steps

7.1. Conclusion

In this project, the main focus was to create a system design, which can combine multiple functions of a processing unit as well as the dependency management, in order to be able to address parallel processing at the hardware level, and without any software functionality from the compiler or any other modules in the processor. In this case we could design a system of multiple integrated hardware modules, the system can execute multiple instruction that are programmed in a sequential programing language, also can determine if those instructions have any dependencies, then process them, in order to maximize the parallel processing of the program.

Designing this system came with multiple challenges that required attention and design time, like.

- The interlocks among internal modules
- Interlocks with the external systems
- Synchronous or Asynchronous internal and external operation
- Data transfer schemes

Other challenges came after starting the Post Routing Simulation, where the system needs to interact with properties of the configured hardware, like

- What instructions to use to result in faster propagation
- Programing methods to have less components in the resulting design

- Find balanced decision between speed and capabilities to add to the system

Some of these challenges were resolved by making design decisions and determine a method to address them. Other challenges were addressed by rewriting the specific modules and run post routing simulation, to determine the approach for the best balance. Where we addressed some challenges by building the schematics and inspect the propagation delay to determine the shortest path of the data.

7.2. Next Steps

As many of the challenges were resolved, some of them were identified as future research and development opportunities, in order to improve the system performance and usability. These improvements will get the system to a closer status to a marketable design, and as further from a concept as possible. Below are some of the opportunities we identified to be in the next level of development.

Data handling scheme, an optimized approach to get the best data collection speed for the system, in order to speed up the data transfer among the internal modules, which is expected to eliminate a large portion of the execution cycle

Increase the number of instructions that can be processed simultaneously. In order to make the performance of the system as optimized as possible, a balanced design needs to be optimized to determine the best Number of instructions to be processed simultaneously. This part of the system is limited by the size of the design, the amount of data generated and consumed by the design, and the speed of the execution cycle. if we can find an optimized number of simultaneously instructions to process, we will be able to produce the highest performance processor possible for this design approach.

Increase number of operations to cover more instructions. By this part of the system, we will allow the system to process more types of instructions, and that will be limited by the time of execution, as well as the flexibility of the processor, and if it's better to handle some operations in the compiler level or the processor level.

References

[1] I. Skliarova and V. Sklyarov, FPGA-BASED Hardware Accelerators, vol. 566, Cham, Switzerland: Springer Nature Switzerland AG, 2019.

[2] S. Kilts, Advanced FPGA Design Architecture, Implementation, and Optimization, Minneapolis, Minnesota: John Wiley & Sons, Inc., 2007.

[3] D. P. Antonik, Application of FPGA to Real-Time Machine Learning, Brussels, Belgium: Springer International Publishing AG, 2018.

[4] T. Kumar, B. Pandey, T. Das and B. S. Chowdhry, "Mobile DDR IO standard based high performance energy efficient portable ALU design on FPGA," *Wireless Personal Communications,* vol. 76, p. 569–578, 2014.

[5] T. N. Sasamal, A. K. Singh and A. Mohan, "Efficient design of reversible alu in quantum-dot cellularautomata," *Optik - International Journal for Light and Electron Optics,* vol. 127, no. 15, pp. 6172-6182, 2016.

[6] T. Kumar, B. Pandey, S. H. A. Mussavi and N. Zaman, "CTHS based energy efficient thermal aware image ALU design on FPGA," *Wireless Personal Communications,* vol. 85, p. 671–696, 2015.

[7] R. Chen and V. Prasanna, "Accelerating equi-join on a CPU-FPGA heterogeneous platform," in *Proceedings of the 24th IEEE annual international symposium on field-programmable*, Washington, DC, USA, 2016.

[8] D. A. Patterson and J. L. Hennessy, Computer Organization and Design, San Francisco, CA: Elsevier Inc, 2005.

[9] N. Telagam and N. Kandasamy, "Low power delay product 8-bit ALU design using decoder and data selector," *Majlesi Journal of Electrical Engineering,* vol. 12, pp. 103-108, 2018.