

Implementation of DLX Processor on XILINX Nexys-4 FPGA

Project report submitted in partial fulfilment of the requirements for the degree of

Bachelor of Technology


BY

Sujay Pandit (1410110446)

Under Supervision of

Mr. Prateek Sikka (External supervisor)

Prof. Ranendra Narayan Biswas (Internal Supervisor)



DEPARTMENT OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING
SHIV NADAR UNIVERSITY
DECEMBER 2017

Candidate Declaration

I, **Sujay Pandit** , hereby declare that the thesis entitled “**Implementation of DLX Processor on XILINX Nexys-4 FPGA**” submitted for the B Tech Degree program has been written in my own words. I have adequately cited and referenced the original sources.

Sujay Pandit

1410110446

13th December 2017

CERTIFICATE

It is certified that the work contained in the project report titled “**Implementation of DLX Processor on XILINX Nexys-4 FPGA**” by **Sujay Pandit** has been carried out under my/our supervision and that this work has not been submitted elsewhere for a degree.

Signature of Internal Supervisor

Prof. Ranendra Narayan Biswas

Department of Electrical Engineering

School of Engineering

Shiv Nadar University

December, 2017

Project Abstract

The 32-bit load/store DLX processor architecture is a generic RISC processor designed by Hennessy and Patterson for pedagogical purposes. This project aims to develop a Verilog based implementation of DLX and implement that onto an FPGA. The processor has been designed keeping in view 5 individual instruction execution stages. The combined functionality of these stages has been implemented in 5 different Verilog modules. At this stage, the pipelining functionality is not present. However, the modules have been designed in such a way that pipelining and other important features can be added to the design as a next step. An Assembler has also been created in python. The assembler has been implemented to parse assembly programmes and convert it into memory code and load data into ROM block of the design. This report chapterwise covers the introduction, design stages, module descriptions, testing and implementation results in sufficient details.

Key terms- DLX processor, Soft-core processor, Nexys-4 FPGA, Dual-core processor, Verilog, Assembler, Vivado, Implementation

Contents

1.	Introduction.....	1
1.1.	Overview.....	1
1.2.	DLX Processor.....	1
1.2.1.	Introduction.....	1
1.2.2.	Instruction Set.....	2
2.	Literature Review	5
2.1	FPGAs and Soft-Core Processors.....	5
2.2.	Design description and Project implementation Procedure.....	6
3.	Project Methodology	9
3.1.	RTL Design.....	9
3.2.	Assembler, Test Bench and Simulation.....	13
3.3.	Synthesis, Implementation and Post-implementation Simulation.....	17
3.4.	Implementation on FPGA.....	19
3.4.1.	Programming the FPGA.....	19
3.4.2.	Seven Segment Display.....	20
3.5.	Performance Improvement	22
3.5.1.	Dual-Core Design.....	22
4.	Results and Design Analysis.....	25
4.1.	Design Analysis.....	25
4.1.1.	Single-core design analysis.....	25
4.1.2.	Dual-core design analysis.....	26
5.	Conclusion.....	27
5.1.	Future Work.....	27
6.	Appendix.....	28
7.	Bibliography.....	30

Chapter 1 – Introduction

1.1 Overview

This project is aimed at implementation of synthesizable DLX (pronounced “DeLuXe”) processor described in Verilog on XILINX Nexys-4 FPGA. DLX is a RISC architecture based 32-bit processor. Main aim for this project was to understand working of soft-core processors to explore their advantages and disadvantages. There have been some modifications made to the original architecture to meet the requirement of the actual project goal which was to implement small programmes such as “Hello World” to be run on the processor and the string to be displayed onto FPGA board. We will also discuss about the performance of the implemented CPU and steps taken to enhance the performance.

1.2 DLX Processor

1.2.1 Introduction

DLX processor was first developed by John L. Hennessy and David A. Patterson and introduced in their book titled ”Computer Architecture: A Quantitative Approach (1st ed.)”[1] in 1996. The DLX is essentially a cleaned up (and modernized) simplified MIPS CPU. The DLX has a simple 32-bit load/store architecture, somewhat unlike the modern MIPS CPU.

1.2.2 Instruction Set

The DLX RISC instruction set contains the basic load/store, mathematical and branching instructions. The instruction set consists of a comprehensive set of commands. The DLX Processor uses 3 different instruction types, which are R, J and I, to perform meaningful operations. First 6 bits of an instruction are referred to as “Operation Code (op-code)”. Instruction types are defined briefly below:

R-Type Instructions

R type instructions are used to perform purely register based operations. This means there is no memory access needed for these instruction types. The 32 bit structure of an R type instruction is shown in figure.

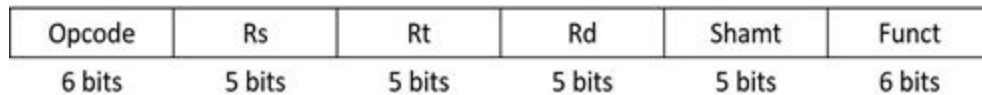


Figure 1.1: R-Type Instruction format

The op-code field for R type instruction is always “110000”. The function field specifies the operation to be performed. Here Rs and Rt represent the 1st and 2nd source registers whereas Rd represents the destination register. The 5 bit shamt field remains unused in our case.

I-Type Instructions

I type instructions are used when an operation is to be performed on a register value and an immediate value. The structure of these instructions is given as follows:



Figure 1.2: I-Type Instruction format

In these instructions, the op-code represents the operation to be performed, Rs represents the source and Rt the destination register. The immediate value can be represented by 16 bits.

J-Type Instructions

J type instructions are specifically used to handle jump statements. These have a simple structure as shown below:

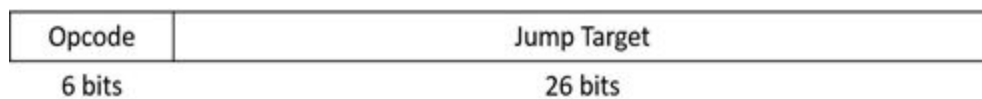


Figure 1.3: J-Type Instruction format

These instructions are used during a programme to jump to an address location in program memory. They can further be divided into conditional and unconditional jump instructions. Unconditional jump instructions such as branch instructions are modelled using I-type instructions.

Instruction set used in this project:

Instruction	Opcode	Instruction	Opcode
LW	00010	SW	001010
ADDI	010000	SUBI	010010
ORI	010101	XORI	010110
SLTI	011010	SGTI	011011
SGEI	011100	SEQI	011101
SLEI	011110	SNEI	011111
BEQZ	100000	BNEZ	100001
J	100100	STRING	111111

R-type Op-code: 110000

Function codes for R-type instructions are given below:

Instruction	Function Code	Instruction	Function Code
ADD	000001	SUB	000011
SLT	001011	SGT	001100
SLE	001101	SGE	001110
SEQ	001111	SNE	010000

Chapter 2 – Literature Review

2.1 FPGAs and Soft-Core Processors

The Field Programmable Gate Array or FPGA is a type of device that is widely used in the logic or digital electronic circuits. FPGAs are semiconductor devices that contain programmable logic and interconnections. The great advantage of the FPGA is that the chip is completely programmable and can be re-programmed. In this way it becomes a large logic circuit that can be configured according to a design, but if changes are required it can be re-programmed with an update. Although FPGAs offer many advantages, there are naturally some disadvantages. They are slower than equivalent ASICs (Application Specific Integrated Circuit) or other equivalent ICs, and additionally they are more expensive. (However ASICs are very expensive to develop by comparison). This means that the choice of whether to use an FPGA based design should be made early in the design cycle and will depend on such items as whether the chip will need to be re-programmed, whether equivalent functionality can be obtained elsewhere, and of course the allowable cost. Sometimes manufacturers may opt for an FPGA design for early product when bugs may still be found, and then use an ASIC when the design is fully stable.

Soft-core microprocessors mapped onto field-programmable gate arrays (FPGAs) represent an increasingly common embedded software implementation option. Modern FPGA soft-cores are parameterized to support application-specific customization.

Compared to hard-core microprocessors on some FPGA devices, soft-core processors have the advantages of utilizing standard mass produced and hence lower-cost FPGA parts and of enabling

a custom number of microprocessors per FPGA (subject to size constraints) – over 100 soft-core processors can fit on modern high-end FPGAs. However, soft-core processors have the disadvantages of reduced processor performance, higher power consumption, and larger size.

While any microprocessor soft-core could conceivably be mapped to an FPGA, FPGA vendors have in the past few years introduced soft-core processors specifically targeted for FPGA implementation. Such FPGA soft-cores have instruction sets, arithmetic-logic units, register files, and other features specifically tailored to efficiently use FPGA resources, or perhaps more accurately, to avoid inefficient use of FPGA resources that may occur when synthesizing a general soft-core processor to an FPGA. The performance overhead of such soft-core processors on FPGAs compared to general soft-core processors on ASICs (application-specific integrated circuits) can thus be significantly less than the overheads when comparing FPGA versus ASIC implementations of general circuits.

2.2 Design description and Project implementation procedure

In this section we will discuss about the major steps involved in this project.

2.2.1 RTL Design

First step of the project was to describe the DLX architecture in Verilog HDL using XILINX Vivado. HDL or Hardware Description Languages are used to describe the structure and behaviour of electronic circuits and digital logic circuits. In this project the RTL contains five instruction execution stage modules of the DLX architecture along with the ROM and RAM modules.

2.2.3 Test Bench and Assembler

Once the design is ready, its working is verified using test-bench modules. A test bench basically simulates inputs to the processors and gives us the output waveforms based on which we can debug the design for errors. To feed the inputs to the processor using test bench we have created an Assembler in python. The assembler lets user write assembly in some-what similar fashion to MIPS and then converts it into machine language to be fed into the processor,

2.2.4 Synthesis, Implementation and Post-Layout Simulation

Once the design is working the next step is to make it synthesizable which basically means that there are no elements in the described design which cannot be converted into digital logic elements. Once your design is synthesizable it does not necessarily mean that it will work on the FPGA. Before implementing the design bit-stream onto the FPGA we run a post-layout simulation which helps us understand if the design will work on piece of hardware given the constraints.

2.2.5 Bit-stream Generation and Implementation on the FPGA

If the post-layout simulation is successful and there are no implementation violations then the bit-stream can be generated. Bit-stream is used to programme a FPGA to make it behave like the described design in our case the processor.

2.2.6 Performance Analysis and Enhancement

Once the design is implemented on FPGA we look at the performance reports and analyse it based on a number of factors the limitations of the design, area utilised on board, clock frequency, power consumed, etc.

In coming sections of the report we will talk in detail more about the advantages of soft-core processors with results from our project.

Chapter 3 – Project Methodology

3.1 RTL Design

In digital circuit design, register-transfer level (RTL) is a design abstraction which models digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals.

In this step we create the top DLX module which contains all the five instruction execution stages namely, fetch, decode, execute, memory access and write back. Then we proceed to create the ROM and RAM modules for the design.

3.1.1 DLX Top level module

This is the most challenging part of the whole project as it required the most of time for completion and highest attention to detail. All modules add up to almost twelve hundred lines of code.

DLX processor follows a specific instruction execution procedure. To make the process less complicated, the execution procedure is divided into 5 sub-phases that are identified as instruction execution stages. These 5 stages not only help in clearly defining how an instruction has to be executed but also sets ground work for pipe-lining. These 5 stages are described in the following sections:

Instruction Fetch Stage:

This is the first stage in instruction execution. In this stage, we fetch the 32-bit DLX instruction from memory and place it in the Instruction register. This is useful as registers can be accessed with much more speed as compared to memory access. Once the instruction goes through all the

other stages, a program counter tells this stage to fetch the next instruction from the given address. The program counter is initially set to 0 in our case as the first instruction is placed at this memory address.

Instruction Decode Stage:

In this stage, the instruction is read from the instruction register and corresponding operands, which may be registers or an immediate value, are fetched into the ALU registers A and B. The program counter is also incremented by 1 at this stage.

Instruction Execute Stage:

This is the third stage of the instruction execution. Since we have already fetched the operands in the previous stage, any operation that is being specified by opcode or function field is performed in this stage. However, it must be remembered that no memory accesses or jump instructions can be performed here. Most operations that are performed in this stage are arithmetic or dealing with comparisons. Results that are generated in this stage are stored in a ALU Output register temporarily before they are written back in the next stages.

Memory Access Stage:

RAM accesses and jumps are performed exclusively in this stage. This means that instructions like load, store and jump will be executed here.

Write Back Stage:

The Write Back stage is the last stage of the instruction execution procedure. In this stage, all results that are generated are updated. This means that destination register values are updated in this section.

The following figure shows the internal module that combine to make a functional top level DLX module:

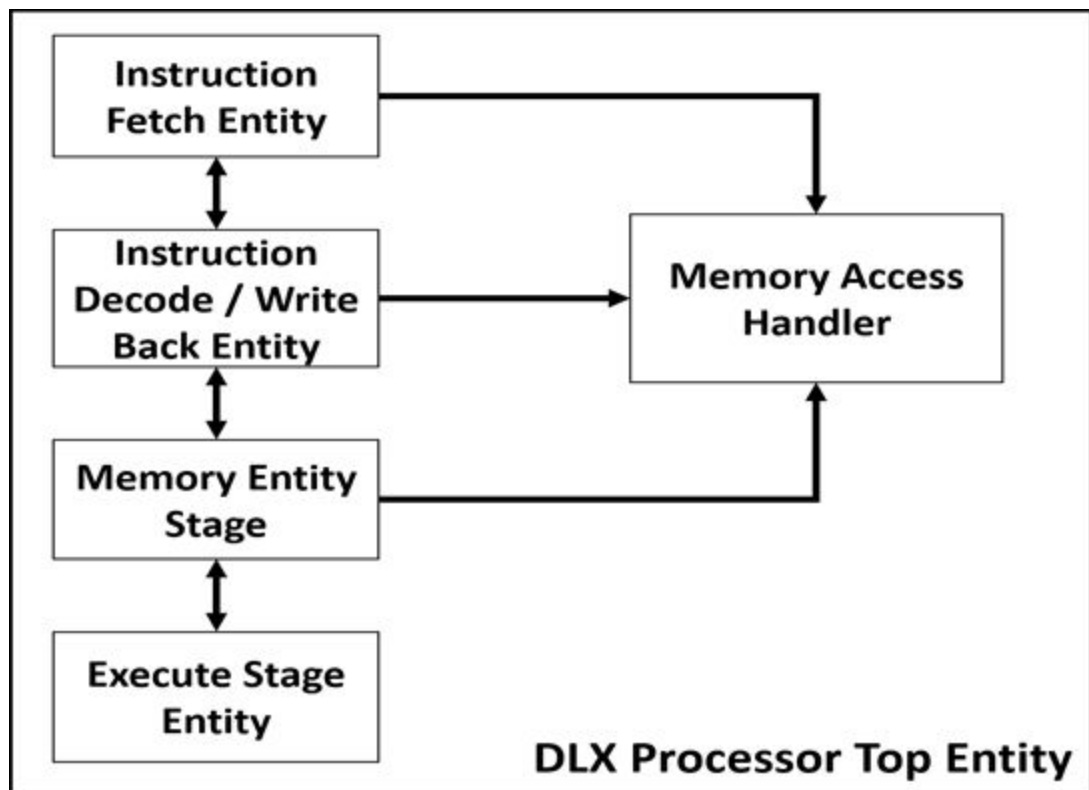


Figure 3.1: DLX Top Level Module Internal Design

ROM Module:

This module contains machine code of the application to be run on the processor. For the simulation step the machine code is read through a text file using the instruction “readmemb” which reads binary machine code and loads it into a 2-Dimensional register array declared in the module.

RAM Module:

This module is mostly used for load /store operations. If some values are not needed for fast access then they can be stored in RAM using store instructions. This frees up registers which are used for faster storing of values.

Following is the generated RTL Schematic for the design

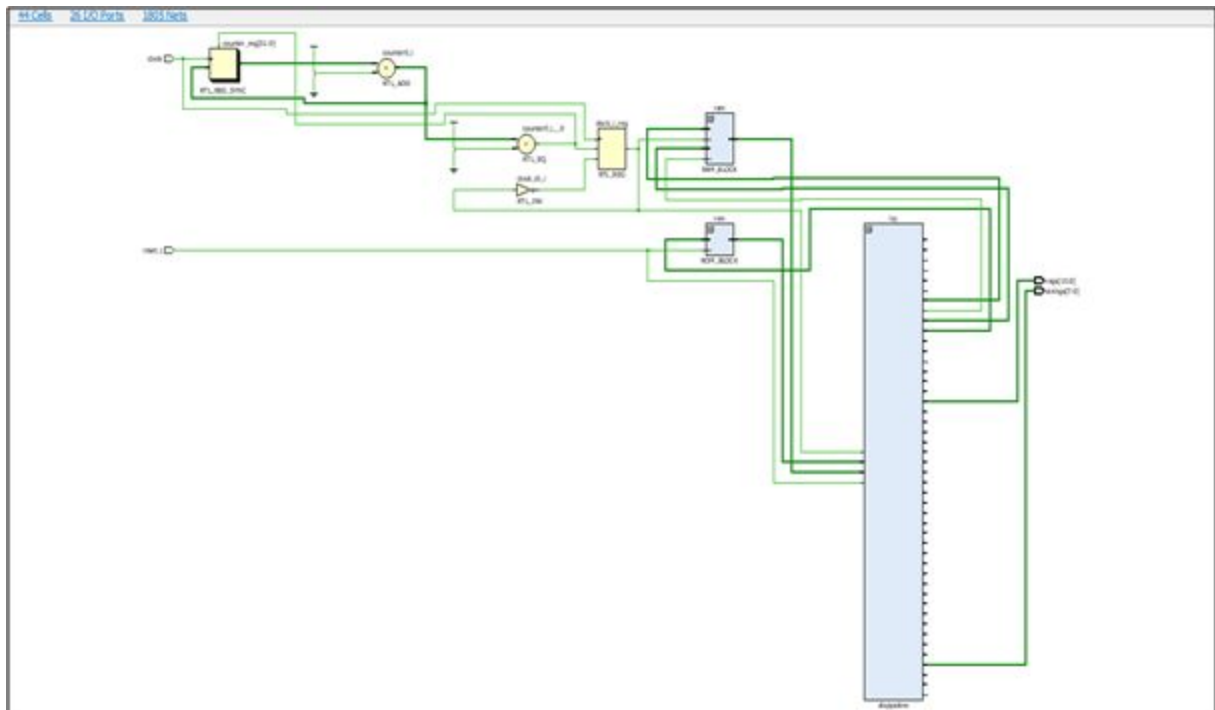


Figure 3.2: RTL Schematic

3.2 Assembler, Test-Bench and Simulation

The next step after creation of top module is to test it's working and debug any errors found. But before we simulate our design using test bench we need to create applications for which we will perform the tests.

In this project for simulation purposes we used “SUM OF FIRST N NATURAL NUMBERS” application. To have a successful simulation run we need to load our ROM with the machine code of this application.

3.2.1 Assembler : An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a processor. An assembler is sometimes referred to as the compiler of assembly language.

In our project the assembler is written in Python 2.6 language. The python code parses through a text document containing assembly language code and converts that code into machine level code line-by-line.

Following is the flow of algorithm for the python script:

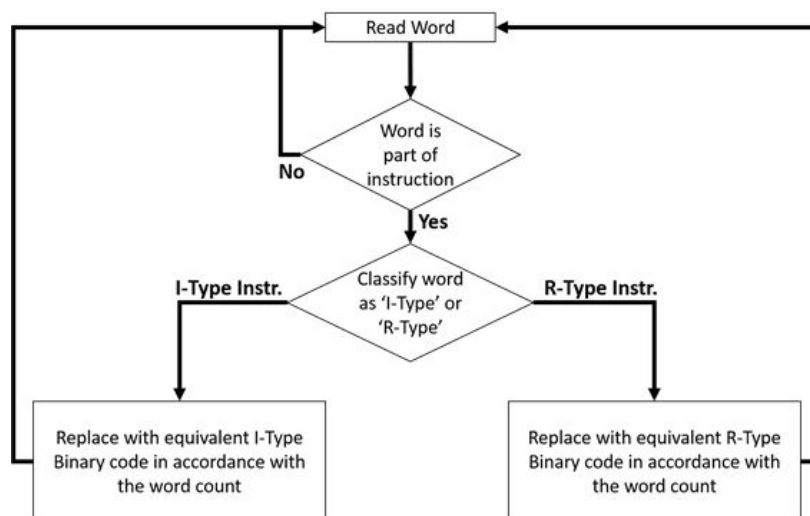


Figure 3.3: Algorithm for Generating Program Binary

Following is an assembly level code followed by generated machine level code to find sum of first 10 Natural numbers.

```

addi r1 r2 0      : 01000000001000100000000000000000
addi r1 r3 0      : 01000000001000110000000000000000
addi r1 r6 0      : 01000000001001100000000000000000
addi r1 r7 0      : 01000000001001110000000000000000
slti r2 r4 11     : 011010000100010000000000000001011
bqez r4 r0 15     : 100000001000000000000000000001111
addi r2 r6 0      : 01000000010001100000000000000000
addi r6 r2 1      : 010000001100001000000000000000001
addi r3 r7 0      : 010000000110011100000000000000000
add r2 r7 r3      : 110000000100011100011000000000001
bnez r4 r0 4      : 100001001000000000000000000000100

```

This machine code is dumped in a text file “Inst_dump.txt” which is read through by ROM Module to load the programme memory to run simulation.

This assembler works on assembly written similar to MIPS assembly but there are a few variations from traditional assembly techniques. One advantage of such an assembler is that it can be updated as the instruction set is updated.

3.2.2 Test Bench

A test bench is a module that instantiates our DLX module and provides input to it. For our processor the inputs are just clock and reset. The test bench is used to generate this clock and assert/de-assert the reset to get the processor running.

In the initial block the clock_i and reset are both set to 1'b1 (High)

To toggle reset:

```
#30;
```

```
reset_i = 1'b0;
```

```
#20;
```

```
reset_i = 1'b1;
```

```
#40;
```

To generate a clock of 100 MHz:

```
always #5 clock_i = ~clock_i;
```

[Values written after '#' are used to add delay in nanoseconds in the code execution]

Code-snippet from ROM module which reads "inst_dump.txt" and loads the data into instruction memory can be found in Appendix I section B.

3.2.3 Simulation

Following is a screenshot from during the simulation run for “Sum of first 10 natural numbers”



Figure 3.4: Sum of first 10 natural numbers

3.3 Synthesis, Implementation and Post-implementation simulation

This three step process helps us determine whether our device is capable of running on the target FPGA device. These steps help us find design violation or constraint violation.

But before we begin with these processes we must first make our synthesizable by removing all non-synthesizable elements such as “readmemb” element in ROM module and hard code all the instructions.

3.3.1 Synthesis:

The synthesizer converts HDL (VHDL/Verilog) code into a gate-level netlist. For our project we use Xilinx Vivado ISE which uses built-in synthesizer XST (Xilinx Synthesis Technology). After a successful synthesis one can run "View RTL Schematic" task to view a gate-level schematic produced by a synthesizer.

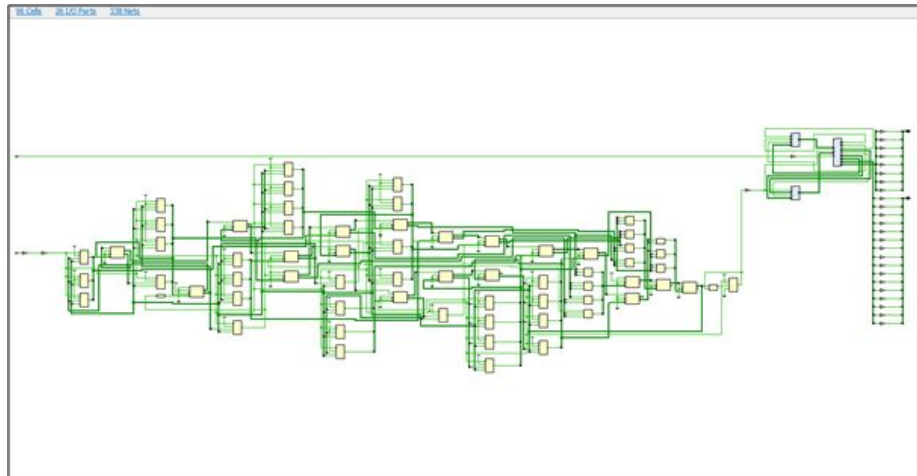


Figure 3.5: Post-Synthesis schematic: Netlist

3.3.2 Implementation:

Implementation stage is intended to translate netlist into the placed and routed FPGA design. Xilinx design flow has three implementation stages: translate, map and place and route. Place and route is the most important and time consuming step of the implementation. Placement and routing is performed by the PAR program. It defines how device resources are located and interconnected inside the target FPGA device , which in our case is XILINX Nexys-4 FPGA.

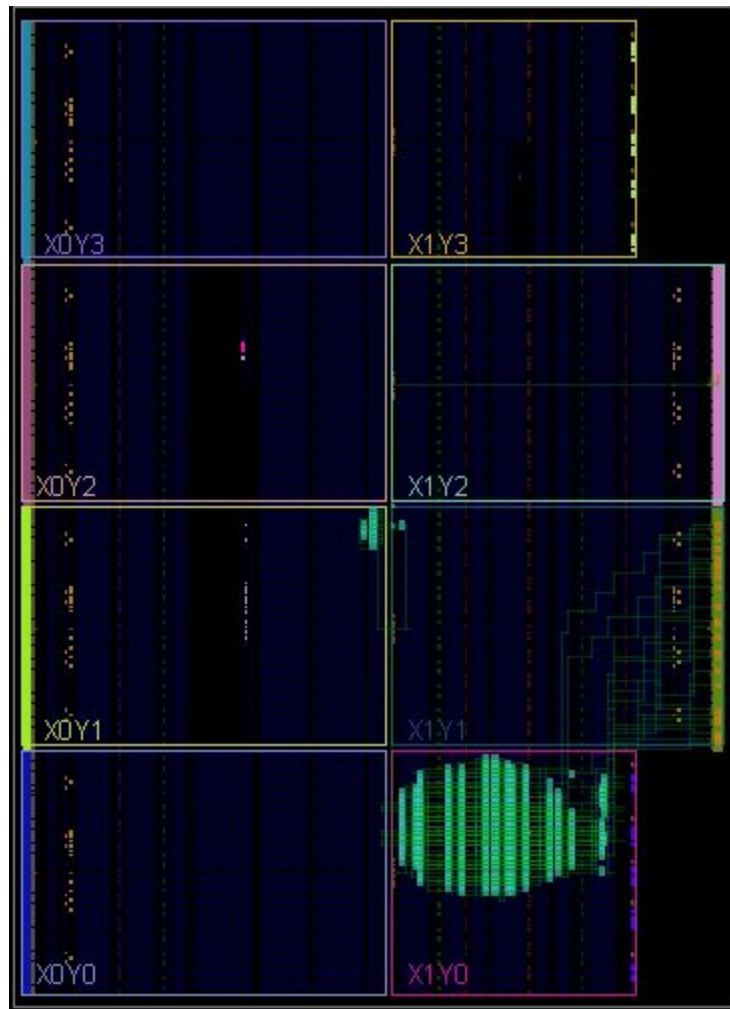


Figure 3.6: Post-Implementation design

3.3.3 Post-Implementation simulation:

One of the most important step in the entire process. It would be almost impossible to debug errors in implementation if not for this step. After the RTL design goes through synthesis and implementation phase there has been a lot of optimization of design with respect to the constraint file of the target device.

3.4 Implementation on FPGA

Once the post-implementation simulation is successful we move on to the next step of generating bit-stream and programming FPGA. But before we do that we need to define outputs. For our project we have connected output ports to seven segment displays on board.

3.4.1 Programming the FPGA:

An FPGA bitstream is a file that contains the programming information for an FPGA. A Xilinx FPGA device must be programmed using a specific bitstream in order for it to behave as an embedded hardware platform. Programming an FPGA is the process of loading a bitstream into the FPGA. During the development phase, the FPGA device is programmed using utilities such as Vivado. These tools transfer the bitstream to the FPGA on board. In production hardware, the bitstream is usually placed in non-volatile memory, and the hardware is configured to program the FPGA when powered on.

3.4.2 Seven Segment Display:

The Nexys4board contains two four-digit common anode seven-segment LED displays, configured to behave like a single eight-digit display. Each of the eight digits is composed of seven segments arranged in a “figure 8” pattern, with an LED embedded in each segment. Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark.

The anodes of the seven LEDs forming each digit are tied together into one “common anode” circuit node, but the LED cathodes remain separate, as shown in Fig. The common anode signals are available as eight “digit enable” input signals to the 8-digit display. The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG. For example, the eight “D” cathodes from the eight digits are grouped together into a single circuit node called “CD.” These seven cathode signals are available as inputs to the 8-digit display. This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.

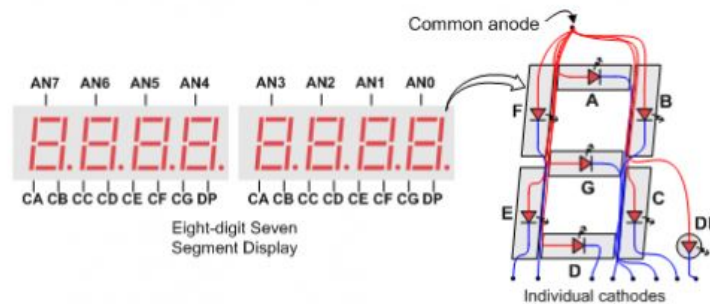


Figure 3.7: Connection to Seven Segment Display on board

To illuminate a segment, the anode should be driven high while the cathode is driven low. However, since the Nexys 4 uses transistors to drive enough current into the common anode point, the anode enables are inverted. Therefore, both the AN0..7 and the CA..G/DP signals are driven low when active.

One limitation of the on board seven segment displays on Nexys-4 is that although the displays have separate anodes to turn the displays on and off they all have common cathodes implies that all seven segment displays on board are programmed as the same character at a time.

Thus, in our project the string “Hello Sujay” comes on output as a series of letters one after another. Following are the output screenshots of two different programmes implemented on the single-core design:

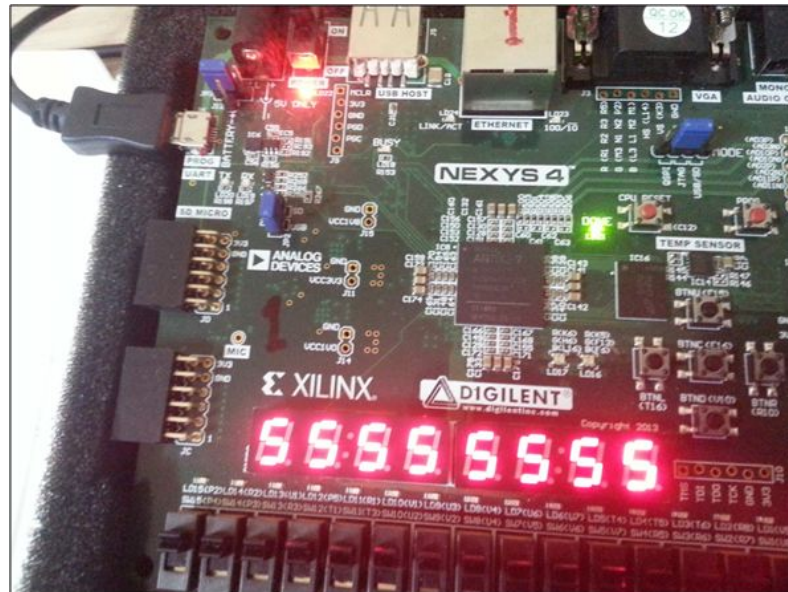


Figure 3.8: Single-core output: String “Hello Sujay”

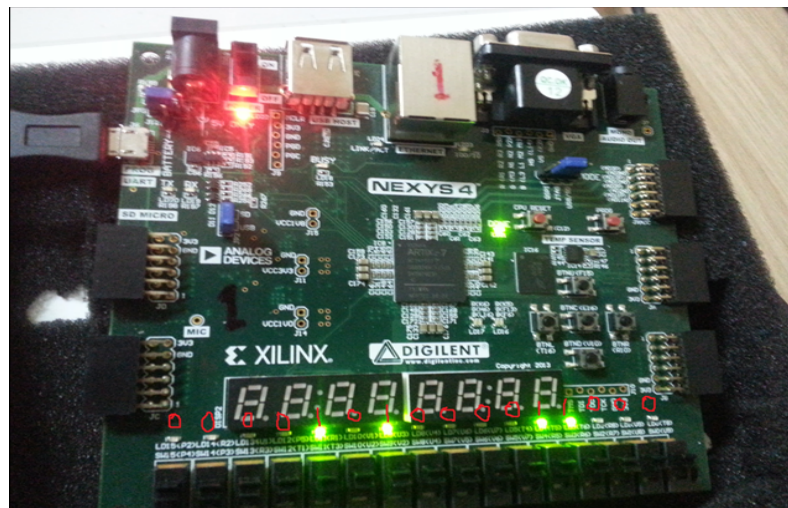


Figure 3.9: Single-core output: 19th Fibonacci numbers = 2584
[Every glowing LED is a ‘1’ and off led is a ‘0’]

3.5 Performance Enhancement

Once the final basic single core design was complete, we took to improve the performance of the design. Since there was not much time left to fully pipeline the design we improved the performance of the design by exploiting the parallelism offered by FPGA and put it use by creating a Dual-core implementation.

3.5.1 Dual-Core Design

The major advantage of a dual core implementation over single core implementation is that in a single core processor the programs are dumped in the Program memory(ROM block) one after another thus they all work on the same clock and run one after another in a sequential fashion. In a Dual-core processor you have two ROM blocks and both can have different programmes thus you can have to separate programmes running parallelly and if desired at different clock speeds.

One disadvantage of a Dual-core implementation is that the design will occupy more area.

To try to minimize this extra area use-age we implement the two cores in such a way that they both share the same RAM block.

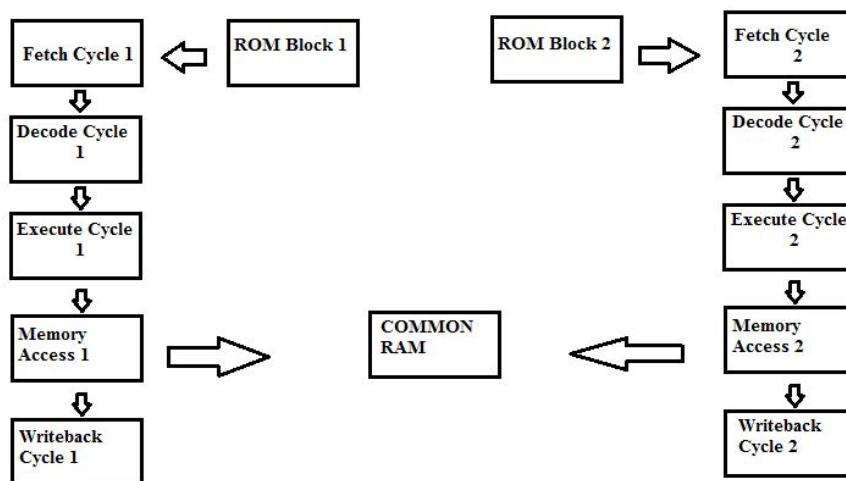


Figure 3.10: Dual-core design flow

Implementation:

The increase in area utilized as compared to single-core is clearly visible in the device implementation output below.

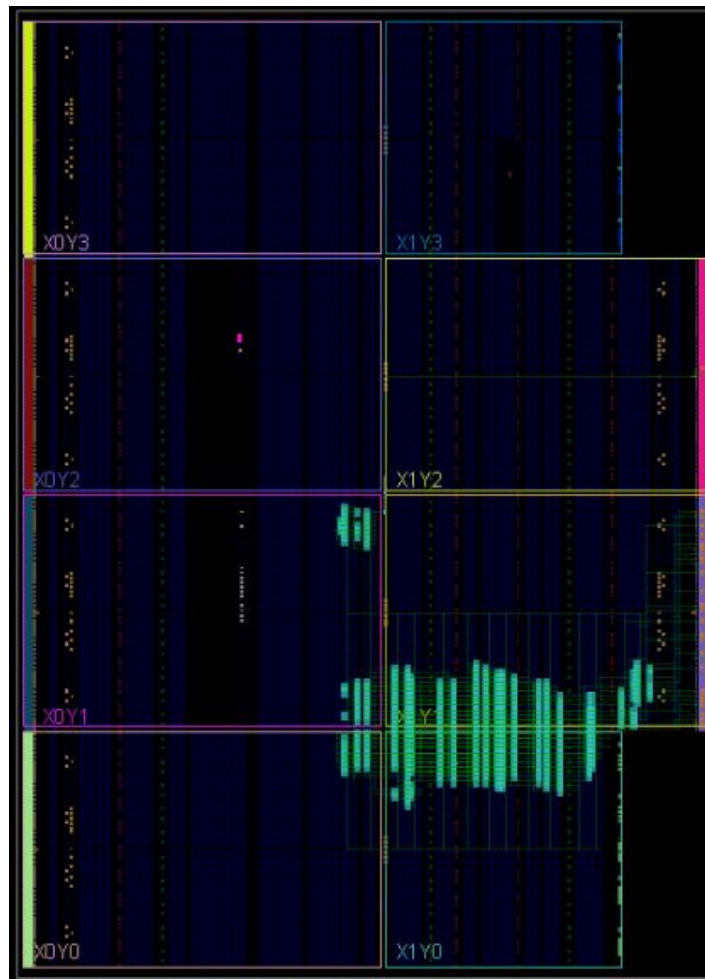


Figure 3.11: Device Implementation of Dual-core

Following is the output screenshot of the dual-core design running two different programmes parallelly:

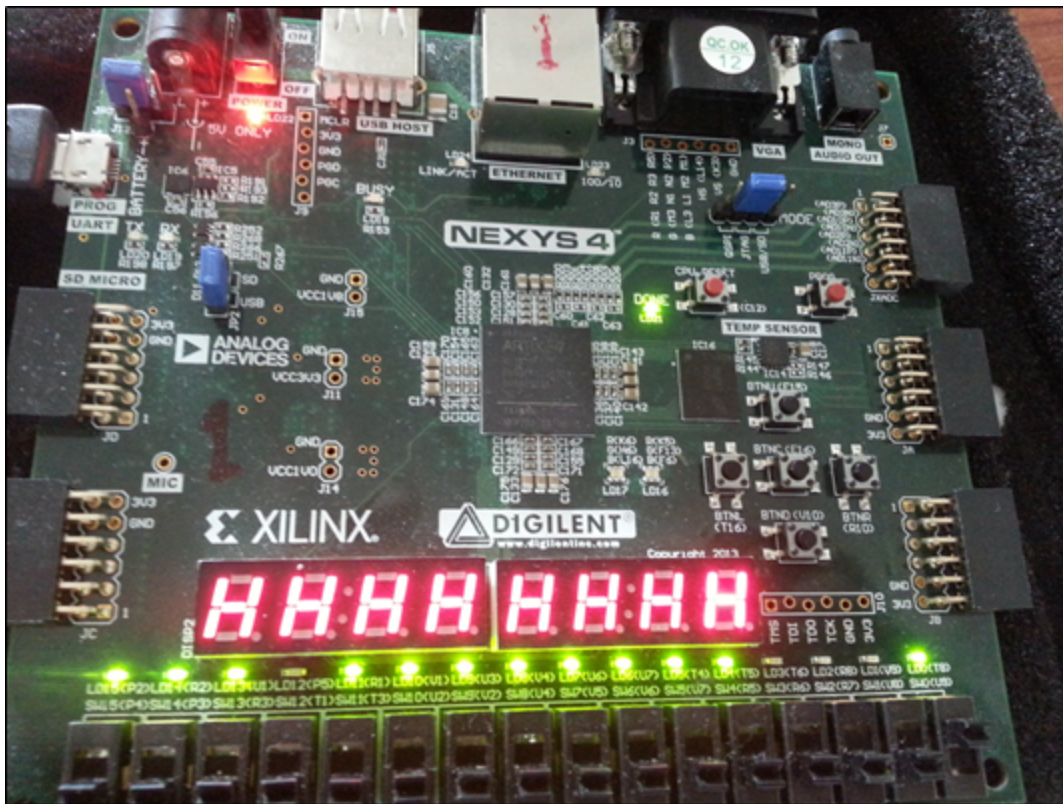


Figure 3.12: Running two programs parallelly at two different clock speeds

(a) “Hello Sujay” String Programme

(b) Sum of first 350 Natural numbers: $1110_1111_1111_0001 = 61425$

Chapter 4 – Results and Design Analysis

4.1 Design Analysis

In this section of the report we look at some key performance parameters for both the single-core and the dual-core design. Analysis of the designs is broken down into three major sections:

- Device utilization
- Timing analysis
- Power consumed

4.1.1 Single-Core design analysis:

As is clear from the performance report that the theoretical limit for the design is around 400 MHz after which further increasing the clock frequency causes timing violations in the design.

Design utilization Report:			
	Utilised	Available	% Utilised
Slice LUTs	1119	63400	1.76
• LUTs as logic	1114		
• LUTs as memory	5		
Slice Registers	1452	126800	1.15
• FlipFlops	1434		
• Latches	18		
Muxes	129	31700	0.41
I/O	16	210	7.62

Timing Analysis:			
Clock Frequency	Setup worst slack	Hold worst slack	Pulse Width worst slack
100MHz	8.429ns	0.460ns	
250MHz	2.439ns	0.454ns	1.500ns
400MHz	0.939ns	0.454ns	0.345ns
500MHz	0.547ns	0.427ns	-0.155ns
800MHz	0.069ns	0.311ns	-0.905ns
1GHz	-0.187ns	0.339ns	

Power Analysis:		
Clock Frequency	Power (Static+Dynamic)	Device Junction Temp (C)
100MHz	0.134W (0.097+0.037)	25.6
250MHz	0.189W (0.097+0.092)	25.9
400MHz	0.245W (0.098+0.147)	26.1

Figure 4.1: Design analysis: Single core

4.1.2 Dual-Core design analysis:

One major point to notice in the report below is that power used in the dual-core design is lesser than the single-core design which can be attributed to the fact that in dual-core both cores aren't working all the time as is the case in single core.

One disadvantage of this design is that it can run at a theoretical maximum clock frequency of 250 MHz as compared to single-core which can reach upto 400 MHz.

Design utilization Report:

	Utilised	Available	% Utilised
Slice LUTs	837	63400	1.32
• LUTs as logic	833		
• LUTs as memory	4		
Slice Registers	815	126800	0.64
• FlipFlops	793		
• Latches	22		
Muxes	2	31700	<0.01
I/O	16	210	7.6

Timing Analysis:

Clock Frequency	Setup worst slack	Hold worst slack	Pulse Width worst slack
100MHz	8.429ns	0.460ns	
250MHz	-0.591ns	0.206ns	2.000ns

Power Analysis:

Clock Frequency	Power (Static+Dynamic)	Device Junction Temp (C)
100MHz	0.099W (0.097+0.002)	25.4

Figure 4.2: Design Analysis: Dual-core

Chapter 5 – Summary and Conclusion

A single-core RISC architecture based DLX processor was described using Verilog and implemented onto XILINX Nexys-4 FPGA board. An assembler was also created for the processor which was coded in python. The design was then tested by multiple programmes such as finding sum of first N natural number, finding Nth fibonacci number, displaying string on the board's seven segment display. The theoretical maximum clock frequency limit for this design was 400 MHz. Once the single-core design was fully functional, the design was improved by designing a dual-core implementation of the same processor which utilized lesser power and could run two programs parallelly at a maximum clock of 250 MHz.

The project was completed well within semester duration and sufficient time was spent on improving the design. The design is described using Verilog HDL in an ordered manner such that future improvements can be made to the design.

5.1 Future Work

The following features were studied but not implemented.

- a. Fully pipelining the design
- b. Compiler for the processor
- c. Multi-core implementation (More than 2 cores)
- d. Hyper-threading (To decrease area utilization of dual core but to keep the parallelism)

Chapter 6 - Appendix I

A. Assembly codes for the assembler

I. Sum of first N natural number

```
addi r1 r2 0
addi r1 r3 0
addi r1 r6 0
addi r1 r7 0
slti r2 r4 'N'
bqez r4 r0 15
addi r2 r6 0
addi r6 r2 1
addi r3 r7 0
add r2 r7 r3
bnez r4 r0 4
```

II. Finding Nth Fibonacci Number

```
addi r0 r1 0
addi r0 r2 1
addi r0 r3 0
addi r0 r4 0
slti r4 r5 'N'
```

```

beqz r5 r0 15
add r2 r1 r3
addi r2 r1 0
addi r3 r2 0
addi r4 r6 0
addi r6 r4 1
bnez r5 r0 4

```

B. Code to read assembler generated machine code in Vivado for simulation

```

reg [31:0] inst_memory [0:63];
$readmemb("testp.txt",inst_memory);
$display("data:");
for (i=0; i <20; i=i+1)
    $display("%d:%b",i,inst_memory[i]);
end
always@(data_i)
begin
    data_o<=inst_memory[data_i];
end

```

Chapter 7 – Bibliography

- [1] David Goldberg David A. Patterson John L. Hennessy. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 1996.
- [2] John L. Hennessy David A. Patterson. Computer Organization and Design: The Hardware- Software Interface. Morgan Kaufmann Publishers, 2014.
- [3] SP Ritpurkar, MN Thakare, and GD Korde. “Synthesis and Simulation of a 32Bit MIPS RISC Processor using VHDL”. In: Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on. IEEE. 2014, pp. 1–6.
- [4] Safaa Omran and Ali J Ibada: “FPGA Implementation of MIPS RISC Processor for Educational Purposes” Research Gate 2016