INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

CSN-221 PROJECT

Designing a 32-bit RISC Processor

TEAM MEMBERS

- *SIDHARTH THOMAS*
  ENRL. NO.: 16112087
  EMAIL: sidh.thomas@gmail.com
  Contact: 8281424921

- *SHUBHANSHU AGARWAL*
  ENRL. NO.: 16118078
  EMAIL: Agarwal.shubhanshu07@gmail.com
  Contact:9897001205

- *VARUN RATHORE*
  ENRL. NO.: 16116074
  EMAIL: throne1032@gmail.com
  Contact: 9818701353

- *KHYATI KIYAWAT*
  ENRL. NO.: 16111018
  EMAIL: khyatikiyawat@gmail.com
  Contact: 8755147398

- *KAUSTUBH NAYYAR*
  ENRL. NO.: 16116030
  EMAIL: kaustubhnayyar@gmail.com
  Contact: 8006483783

- *ATHUL P.*
  ENRL. NO.: 16116012
  EMAIL: athulparakkunnath123@gmail.com
  Contact: 8191839194

ACKNOWLEDGEMENT

In the accomplishment of this project successfully, many people have bestowed upon us their blessings and heart pledged support, this time we are utilizing to thank all the people who have been concerned with project.

Primarily, we would like to thank god for providing us strength to complete this project with success. We would also like to thank **Dr. Vaskar Raychoudhury**, who have given us this opportunity and whose valuable guidance has helped us to complete this project and make it full proof success. His suggestions and instructions have served as the major contributor towards the completion of the project.

We would like to thank our team members and friends who have helped us with their valuable suggestions and guidance that have enabled us to successfully finish this project in time.

CERTIFICATE

This is to certify that the project entitled "**Design of 32-bit RISC Processor**" is the bonafide work of Athul P., Kaustubh Nayyar, Khyati Kiyawat, Sidharth Thomas, Shubhanshu Agarwal and Varun Rathore done in the Autumn semester of the academic year 2017-2018. They have duly completed their project and have fulfilled all the requirements of the course CSN 221, System Development Project, to my satisfaction.

Dr. Vaskar Raychoudhary
Assistant Professor
Department of Computer Science and Engineering
IIT Roorkee

DATE: November 5, 2017

PLACE: IIT Roorkee

ABSTRACT

The report presents a customized 32 bit RISC Processor. The development of processor helps us to know its Instruction Set Architecture (ISA) more clearly, thereby generating the opportunity to do brainstorming on a new ISA which can handle more instructions and is more efficient than the existing ISA. The following customized MIPS processor is superscalar implemented using Verilog without pipelining which can handle 16 different instructions. The corresponding ISA is defined with 6 empty bits to handle 128X32 register file rather than a 32X32 register file. With a flexible instruction set architecture, it can have a larger local memory space whenever needed. Further, an assembler is developed in C++ to convert High-Level Assembly language to machine language.

CONTENT

INTRODUCTION

The microprocessor has wide application and impact in various working procedures of almost all sectors. From a small wristwatch to large computational systems used in Research laboratories, microprocessors are used everywhere, ranging from small microcontrollers to general purpose 64-bit microprocessors. This is a trending and fast-changing technology with a lot of scope of research and development as well as developing and improving the existing technologies.

The high-end (32 and 64-bit) general purpose microprocessor are divided into two distinct design philosophies:

- **CISC (Complex Instruction Set Architecture)**
- **RISC (Reduced Instruction Set Architecture)**

As the processor technology began to develop from a 4-bit processor to 64-bit processor, there always has been an objective to increase the number of instructions handled by the processor. With the increase in number of instructions, the control unit and instruction decoder became more complex and occupied 50-60 percent of the memory. Thus, the research interest shifted to the development of reduced instructions so as to increase the speed of execution. The RISC processors were then developed to simplify hardware design and improve processor performance.
Some features of RISC processor development philosophy:

- Less number of instructions (upto 100)
- Less number of addressing modes
- Memory referencing is reduced
- Support more number of registers (32 to 100) rather than memory
- Simplified instruction format with execution in one cycle
- Support for high level languages

CISC on the other hand performs multiple tasks with a single instruction.

MIPS stands for Microprocessor without Interlocked Pipeline Stage. It has wide application in the development of embedded systems such as video games, digital television, etc. They provide better performance with low power consumption.

TYPES OF COMPUTER ARCHITECTURE

1. VON NEUMANN ARCHITECTURE

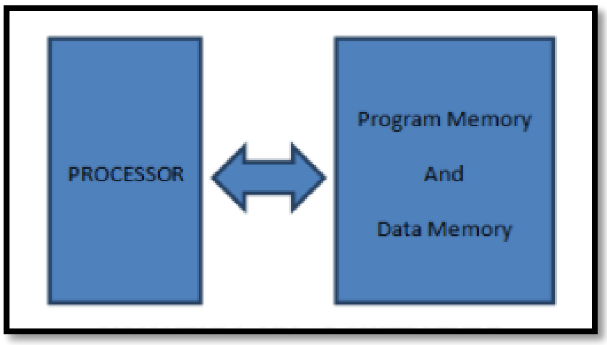


Figure 1: Von Nuemann Structure

The computer has single storage system (memory) for storing data as well as program to be executed. Thus, the processor needs two clock cycles to complete an instruction. It is named after mathematician and computer scientist Von Neumann.

2. HARVARD ARCHITECTURE

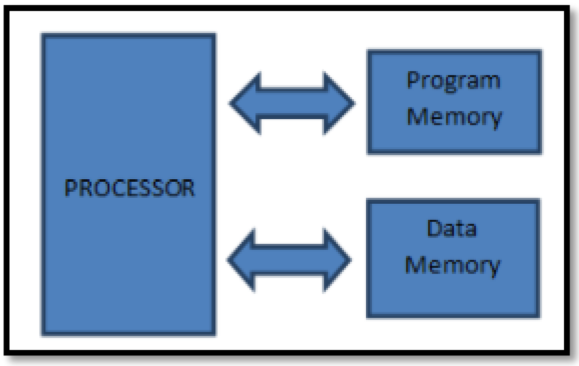


Figure 2: Harvard Structure

The name is originated from "Harvard Mark I" a relay based old computer. The computer has two separate memories for storing data and program. Processor can complete an instruction in one cycle if appropriate pipelining strategies are implemented. Most of the modern computing architectures are based on Harvard architecture. But the number of stages in the pipeline varies from system to system.

The following project shows the implementation of 32-bit RISC processor using Harvard Architecture with a different ISA along with superscaling. Initially, single-cycled datapath is implemented which supports a total of 16 instructions. After that, dual issue superscaling with In-order issue and In-order execution is implemented.

OBJECTIVE

The goal of our study and related research work was to accomplish and complete the following objectives during the period of our learning:

- Designing an ISA which can handle 128 general purpose register(GPR)
- Designing Datapath for a Single-Cycled Customized MIPS Processor
- Implementing Superscalar (Dual issue) concerned with in-order issue and in-order completion
- Developing an Assembler to convert High-Level Assembly code to Machine language
- Implementing and simulating the design in the Verilog

LEARNING OUTCOMES

In the period of three months of working on this project, the team learnt a lot of skills:

- Workplace ethical qualities like teamwork, leadership, patience and punctuality
- Difference between RISC and CISC computing on deeper conceptual level
- Complete understanding the working of a MIPS processor
- The concept of Verilog (Hardware Description Language) got clear with Datapath designing
- The fundamentals of superscaling, pipelining and dependencies
- Some concepts in python and C++ for developing the assembler

INSTRUCTION SET ARCHITECTURE

Instruction Set Architecture is a part of processor which is available to the programmer as well as compiler. It tells the processor what and how to work and execute along with giving the programmer a format to command the processor. Here is the ISA of our Customized 32bit MIPS Processor.

The processor supports following Instruction Format:

Table 1: Format of Different Instructions

FORMAT	OPCODE	RS	RT	RD	SHAMT	-
R TYPE	6	5	5	5	5	6
I TYPE	6	5	5	16 BIT(OFFSET)		
J TYPE	6	26 BIT(LABEL)				

R Format

This instruction format is used when all the data is read from registers and written to registers.

OPCODE: The 6 bits of opcode is used to determine the type of instruction format and the operation to be performed on the available data.

RS,RT : The 5 bits are used to locate source registers RS and RT in the register file. This can be increased to 7 bits each with the help of 6 extra bits available in the instruction itself.

RD: These 5 bits are used to address the destination register in the register file, which can be increased to 7 bits as well.

I Format

This instruction format is used when operation is to be carried out on register value and an immediate value (16 bit).

OPCODE: This is similar as in R-type instructions.

RS: This gives the address to the source register (RS) to which the immediate value is operated.

RT: This gives the address to a register which acts as a source register in branch instructions and as a target register in all instructions in which data has to be written.

IMM: This is the 16 bit offset value.

J Format

This instruction format is used when jumps are encountered in the program. It has 26 bit LABEL because the address to which program counter is taken can be large.

INSTRUCTION AND THEIR CORRESPONDING OPCODE

All the 16 instructions were first implemented through a SINGLE CYCLE RISC PROCESSOR. After that dual issue superscalar was developed and the instructions were implemented through it.

Table 2: Instruction of the Processor

INSTRUCTION		OPCODE
R TYPE	ADD	000000
	AND	000001
	OR	000010
	XOR	000011
	SUBTRACT	000100
	SET LESS THAN	000101
I TYPE	ADD	010000
	AND	010001
	OR	010010
	XOR	010011
	SET LESS THAN	010101
	LOAD WORD	011111
	STORE WORD	011101
	BEQ	110000
	BNQ	110001
J TYPE	JUMP	110010

DATAPATH DESIGNING

SINGLE CYCLE DATAPTH DESIGNING

In a single cycled datapath, all the instructions are executed in the same amount of time. Thus, the instruction that consumes most amount of time decides the time period of the clock.

DEVELOPMENT OF CODE FOR INDIVIDUAL COMPONENTS

The Verilog implementation of our datapath and instruction set architecture started with the coding of different modules or components of the processor.

Xilinx Vivado **VERILOG** code was, therefore, developed for –

- Data Memory
- Instruction Memory
- Register file
- ALU and ALU Control Unit
- Control Unit

1. DATA MEMORY

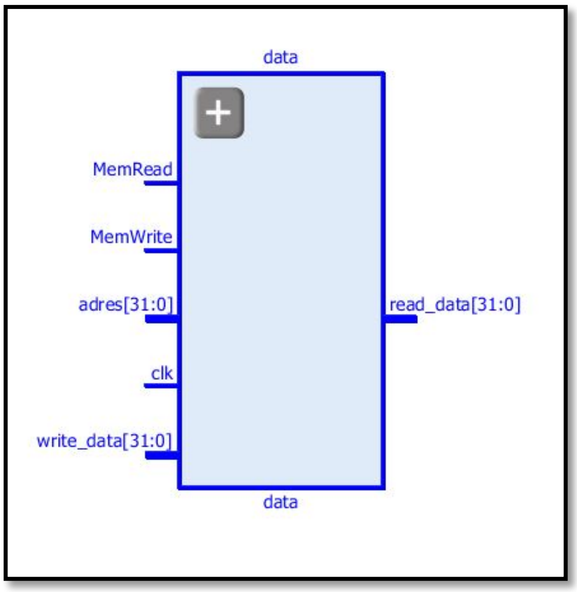


Figure 3: RTL SCHEMATIC

DESCRIPTION OF THE VARIOUS PINS

1. adres: Address to be accessed
2. write_data: data to be written
3. clk: Memory clock
4. MemRead: Read Enable
5. MemWrite: Write enable

6. read_data: data from the memory

2. INSTRUCTION MEMORY

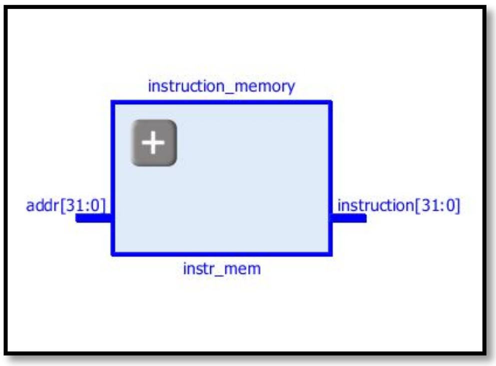


Figure 4: RTL SCHEMATIC

DESCRIPTION OF THE VARIOUS PINS

1. addr: address from which the instruction is to be fetch given by program counter
2. instruction: output of the 32bit instruction from the address given by the program counter

3. REGISTER FILE

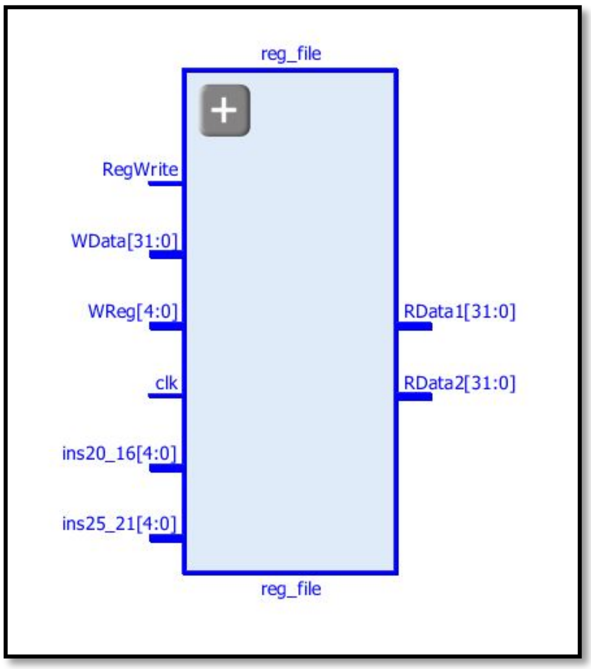


Figure 5: RTL SCHEMATIC

DESCRIPTION OF VARIOUS PINS:

1. RegWrite: Enable writing to the register
2. WData: Data to be written
3. WReg: Address where the data has to be written
4. Clk: Memory Clock
5. Ins20_16: Source register
6. Ins 25_6:   Target register
7. RData1: Value read from the address of the source register
8. RData2: Value read from the address of the target register

4. ALU AND ITS CONTROL

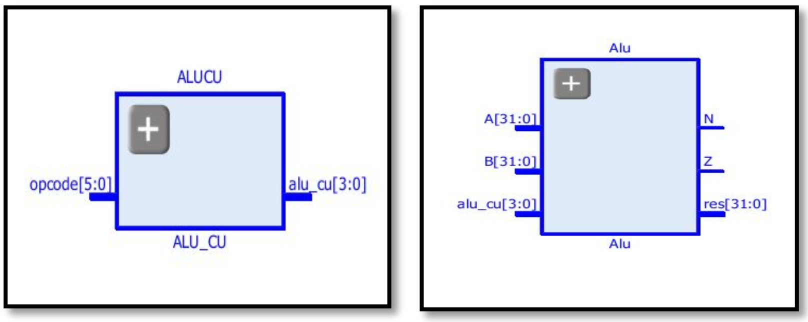


Figure 5: RTL SCHEMATIC

DESCRIPTION OF VARIOUS PINS:

1. opcode: Input to the ALU control unit
2. alu_cu: Corresponding ALU control signal that is given to the ALU unit
3. A: Source data
4. B: Source data
5. Res: Result of the ALU
6. N: Tells whether the output is Negative
7. Z: Tell whether the output is ZERO

The arithmetic logic unit receives an exclusive code from the ALU CONTROL. This code administers the operation of the entity. Provided the operands as inputs to the ALU, it performs an arithmetic or logical operation depending upon the code and provides the output.

Table 3: Opcode and their corresponding Signal

INSTRUCTION		OPCODE	ALU Signal(4BIT)
R TYPE	ADD	000000	0000
	AND	000001	0001
	OR	000010	0010
	XOR	000011	0011
	SUBTRACT	000100	1000
	SET LESS THAN	000101	0100
I TYPE	ADD	010000	0000
	AND	010001	0001
	OR	010010	0010
	XOR	010011	0011
	SET LESS THAN	010101	0100
	LOAD WORD	011111	0000
	STORE WORD	011101	0000
	BEQ	110000	1000
	BNQ	110001	1000
J TYPE	JUMP	110010	0100

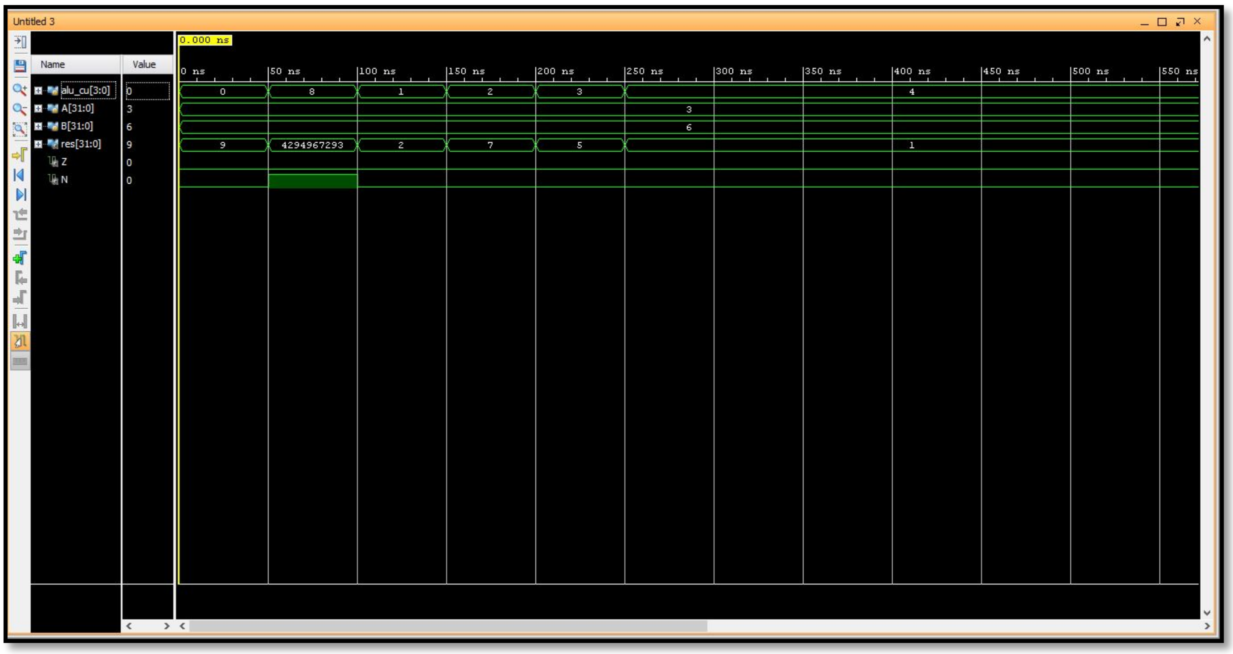


Figure 6: WAVEFORM FOR THE OPCODE AND CORRESPONDING SIGNAL

5. CONTROL UNIT

Once the datapath design was complete, implementing the control unit was just a matter of setting a few control signals high or low based on the type of instruction (as decoded by the control unit).

These signals when applied to the datapath modules would synchronize their working according to the instruction received.

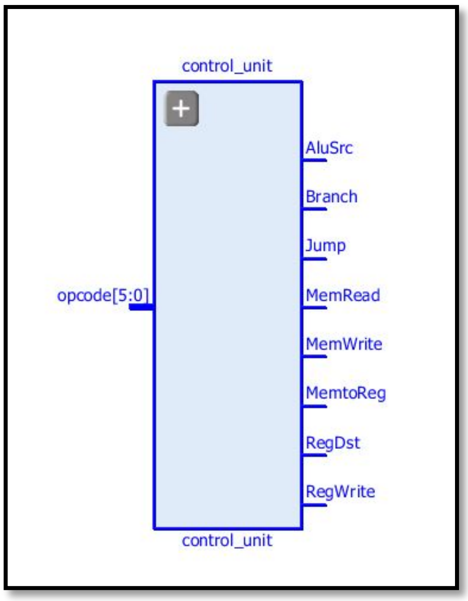


Figure 7: RTL SCHEMATIC

DESCRIPTION OF VARIOUS PINS:

1. AluSrc: Selects the second source operand for the CPU
2. Branch: Combined with a condition test Boolean t enable loading to the branch target address into PC
3. Jump: Enables loading jump address into the PC
4. MemRead: Enables memory read for load type instructions
5. MemWrite: Enables memory write for store type instruction
6. MemtoReg: Determines whether the value comes from ALU or the data memory
7. RegDst: Determines the destination
8. RegWrite: Enable to write in the register

		OPCODE		CONTROL SIGNAL							
	INSTRUCTION	2BIT	4 BIT	REG DST	ALU SRC	MEMTO REG	REG WRITE	MEM READ	MEM WRITE	BRANCH	JUMP
R TYPE	ADD	00	0000	1	0	0	1	0	0	0	0
	AND	00	0001	1	0	0	1	0	0	0	0
	OR	00	0010	1	0	0	1	0	0	0	0
	XOR	00	0011	1	0	0	1	0	0	0	0
	SUBTRACT	00	0100	1	0	0	1	0	0	0	0
	SET LESS THAN	00	0101	1	0	0	1	0	0	0	0
I TYPE	ADD	01	0000	0	1	0	1	0	0	0	0
	AND	01	0001	0	1	0	1	0	0	0	0
	OR	01	0010	0	1	0	1	0	0	0	0
	XOR	01	0011	0	1	0	1	0	0	0	0
	SET LESS THAN	01	0101	0	1	0	1	0	0	0	0
	LOAD WORD	01	1111	0	1	1	1	1	0	0	0
	STORE WORD	01	1101	X	1	X	0	0	1	0	0
	BEQ	11	0000	X	0	X	0	0	0	1	0
	BNQ	11	0001	X	0	X	0	0	0	1	0
J TYPE	JUMP	11	0010	0	0	0	0	0	0	0	1

Table 4: Opcode and their corresponding control Signal

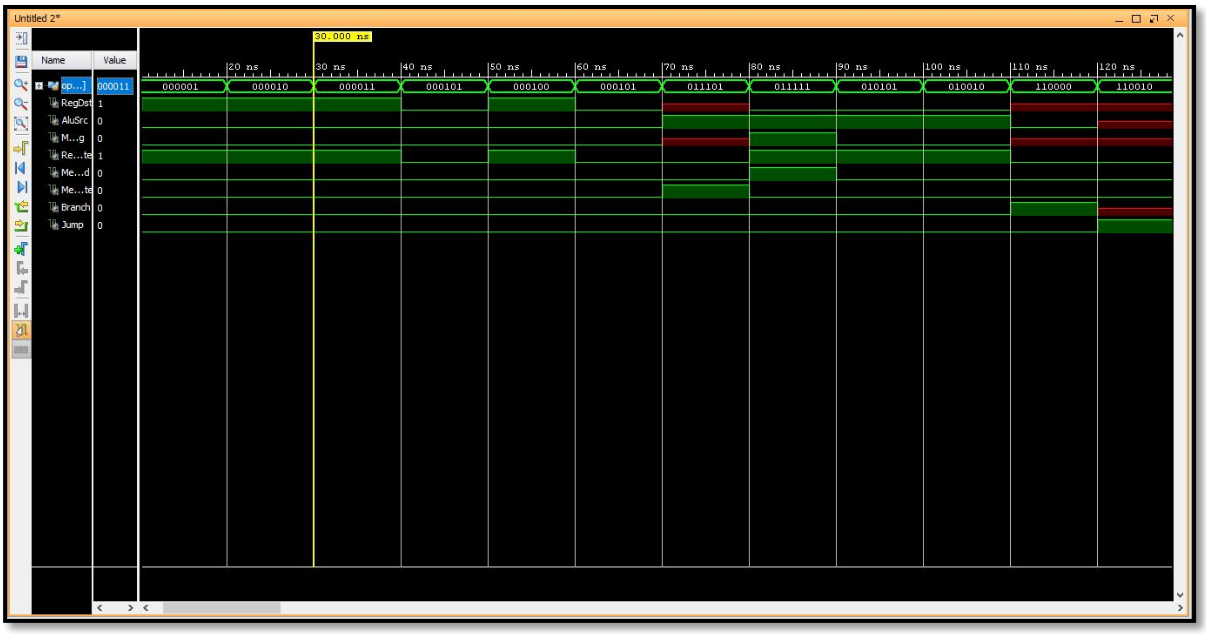


Figure 8: WAVEFORM FOR THE VARIOUS INSTRUCTIONS

VERILOG IMPLEMENTATION OF THE SINGLE CYCLE PROCESSOR

Once the working of each individual component was tested and verified, the next step involved synchronization of each component so that the entire processor worked as a single unit. Here, we present the RTL Schematic and Timing Diagrams generated by testing some sample instruction on the processor.

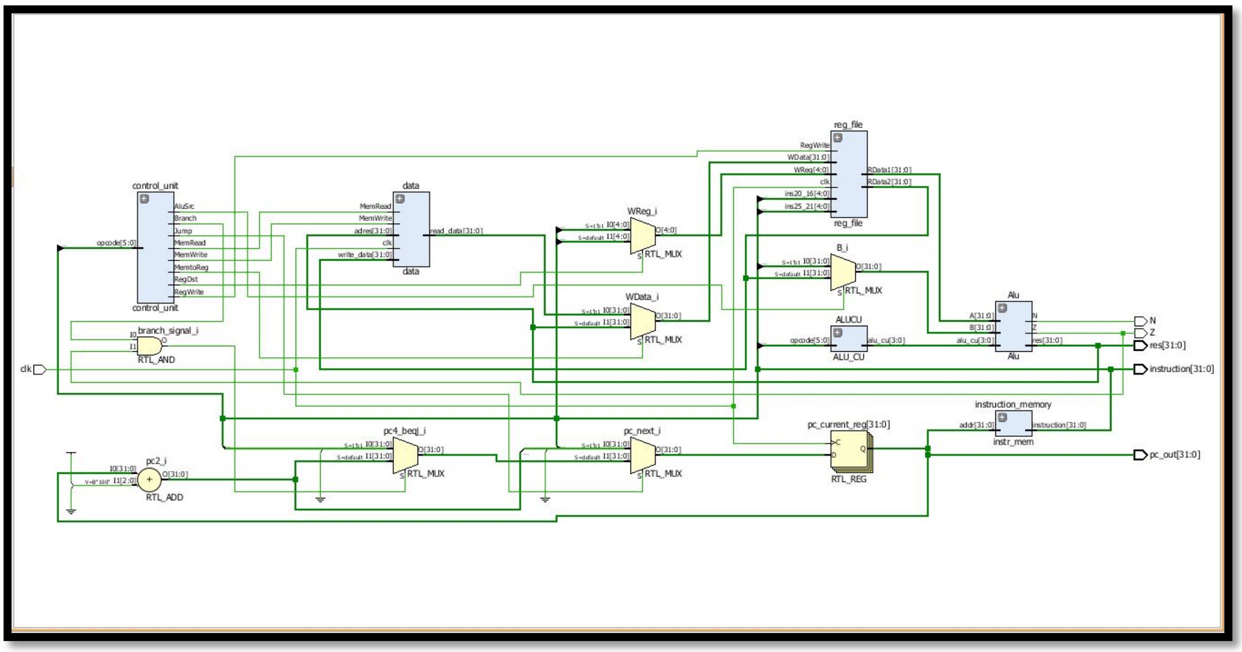


Figure 9: DATAPATH AND RTL SCHEMATIC OF THE TOP MODULE

TEST COMMANDS

```
        mem[0]<=32'b00000000000000000000100000000000;  //add r1 r0 r0
        mem[1]<=32'b00010000000000000001000000000000;   //subtract r2 r0 r0
        mem[2]<=32'b00000000000000010001100000000000;    //add r3 r0 r1
        mem[3]<=32'b01000000000000000000000000000011;   // addi r0 r0 3
        mem[4]<=32'b11111111111111111111111111111111;   // unwanted instruction
        mem[5]<=32'b01111100000000011000000000000000;    //load word from the memory 4 to r3
to base r0
        mem[6]<=32'b01110100000000000000000000001000;  //store word from r0 to mem 12 due to
base value of r0
         end
```

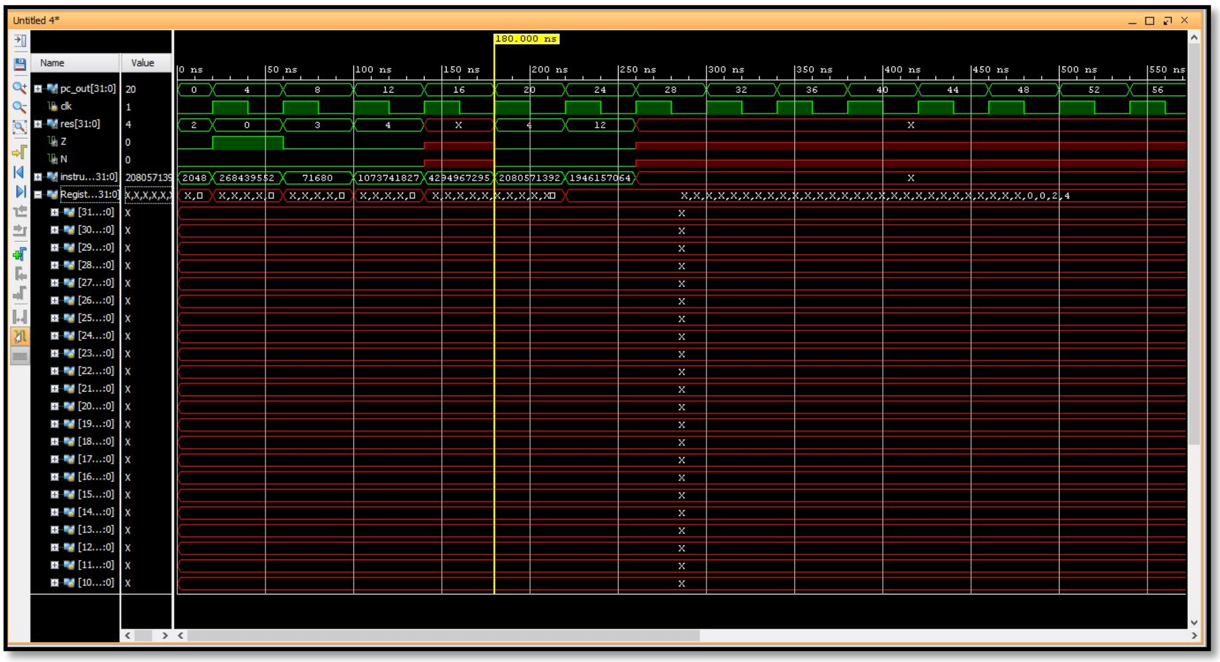


Figure 10(a): WAVEFORM OF THE PROCESSOR

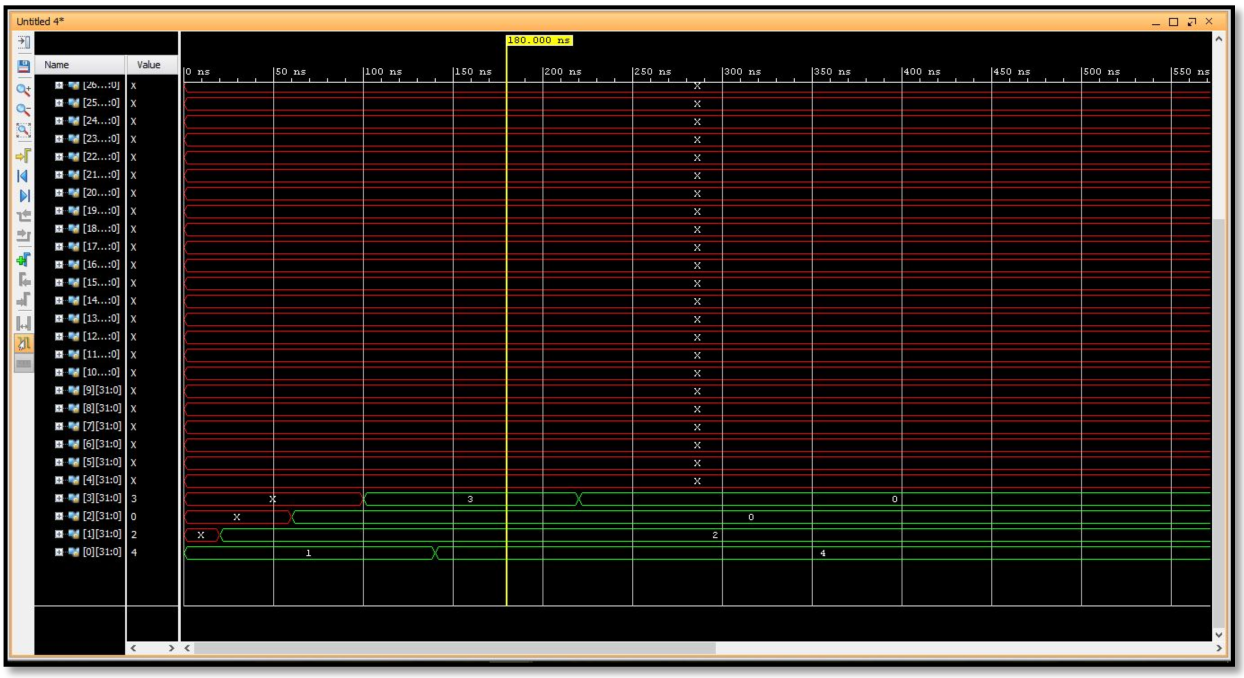


Figure 10(b): WAVEFORM OF THE PROCESSOR

SUPERSCALAR DATAPATH DESIGNING

"Parallelism improves performance". Using this concept, the team has designed an **In-order issue and In-order execution** superscalar. Two instructions are fetched from instruction memory in every clock cycle, then true data dependencies are checked between those instruction, based on that either both instructions are executed in parallel or the one issued later and creating dependency is stalled for a clock cycle.

The Verilog Implementation of superscalar datapath and instruction set architecture started with the coding of module that are different from the single cycle processor and those modules are:

1. Instruction Memory
2. Data Memory
3. Register File

Therefore, new modules have been made for the above components.

1. INSTRUCTION MEMORY

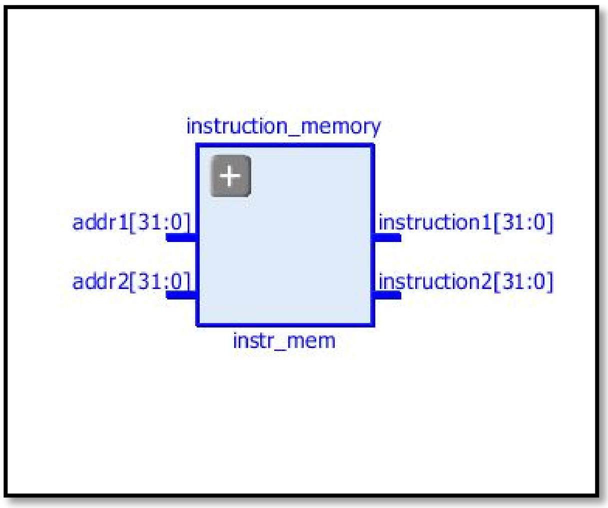


Figure 11: RTL SCHEMATIC

In the instruction memory, now two addresses are given so that they can fetch two instructions in the one clock cycle and thus, help in achieving the concept of dual stage super scalar processor.

2. DATA MEMORY

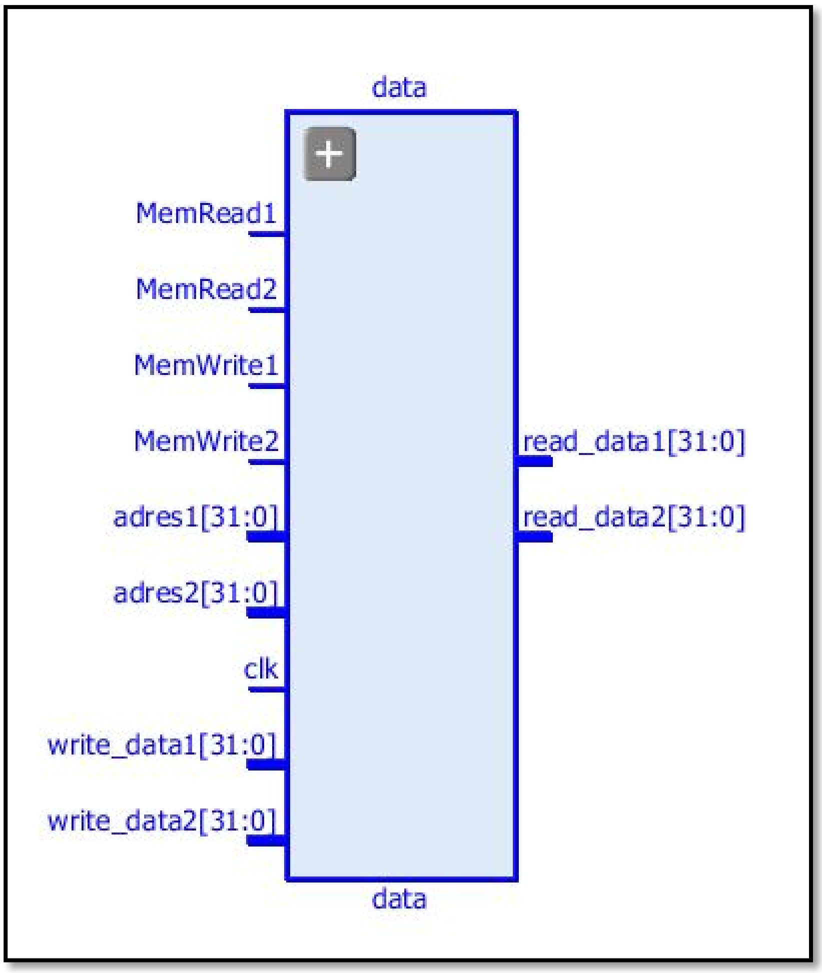


Figure 12: RTL SCHEMATIC

In the data memory also, the inputs are replicated so that it can be used by both the instructions or both processor simultaneously.

3. REGISTER FILE

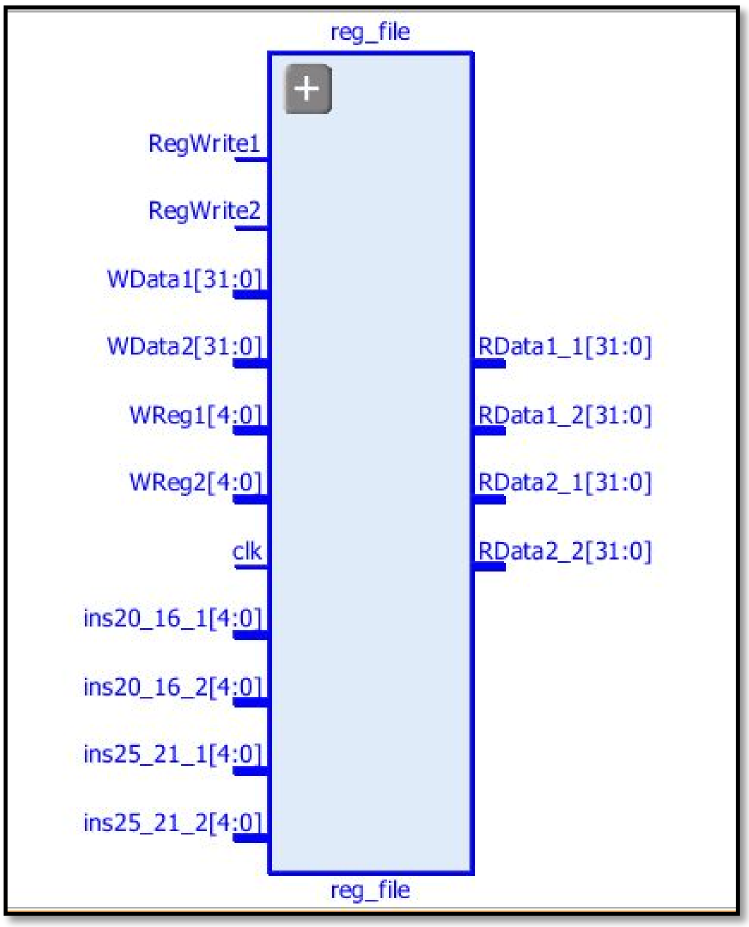


Figure 13: RTL SCHEMATIC

VERILOG IMPLEMENTATION OF SUPERSCALAR DATAPATH

After implementing and synchronizing datapath for single cycled datapath. To increase the performance of processor, dual issue superscaling was implemented. Here, we present the RTL Schematic and Timing Diagrams generated by testing some sample programs on the processor.

The superscalar as mentioned earlier fetches two instructions from the instruction memory with replicated program counter, then the program checks for dependencies based on which execution takes place. If there is any JUMP or BRANCH instruction, then it has to be the first instruction of dual issue.

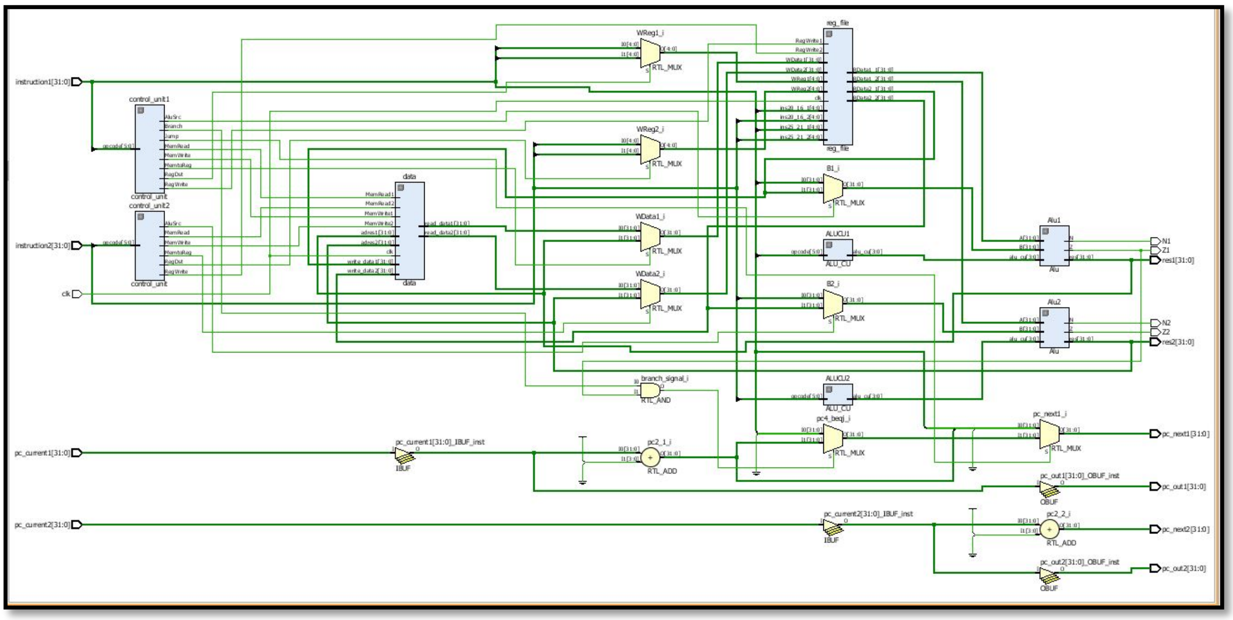


Figure 14: DATAPATH AND RTL SCHEMATIC OF THE PROCESSOR MODULE

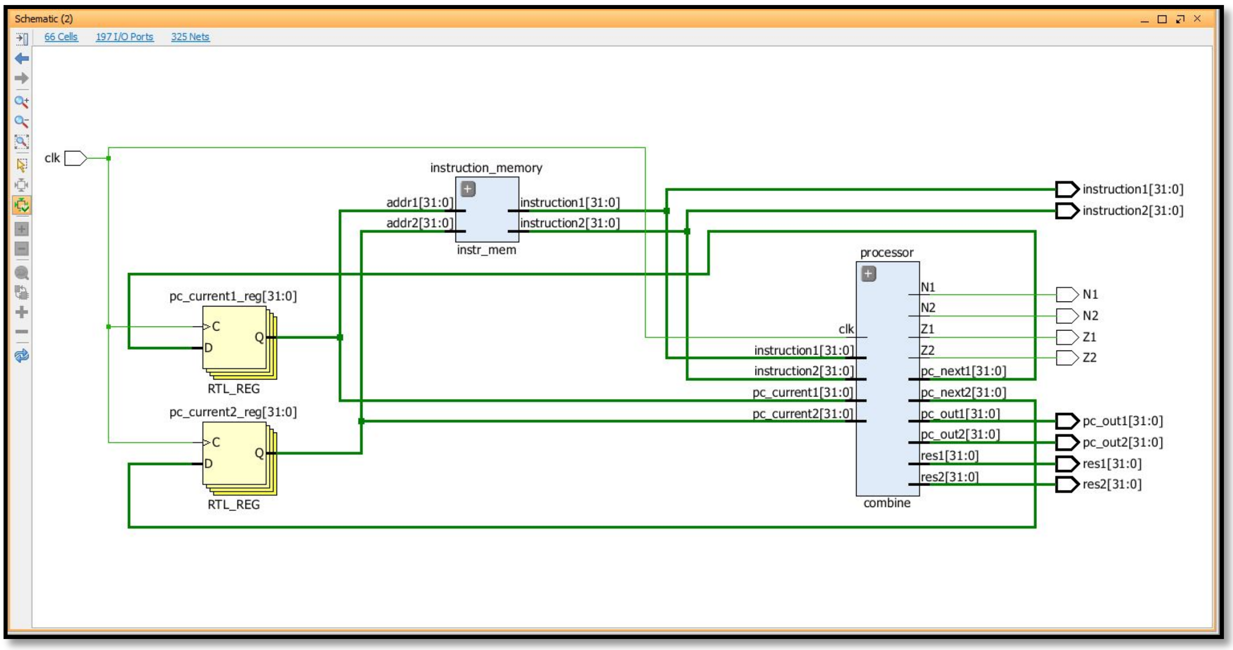


Figure 15: DATAPATH AND RTL SCHEMATIC OF THE SUPERSCALAR MODULE (without dependencies)

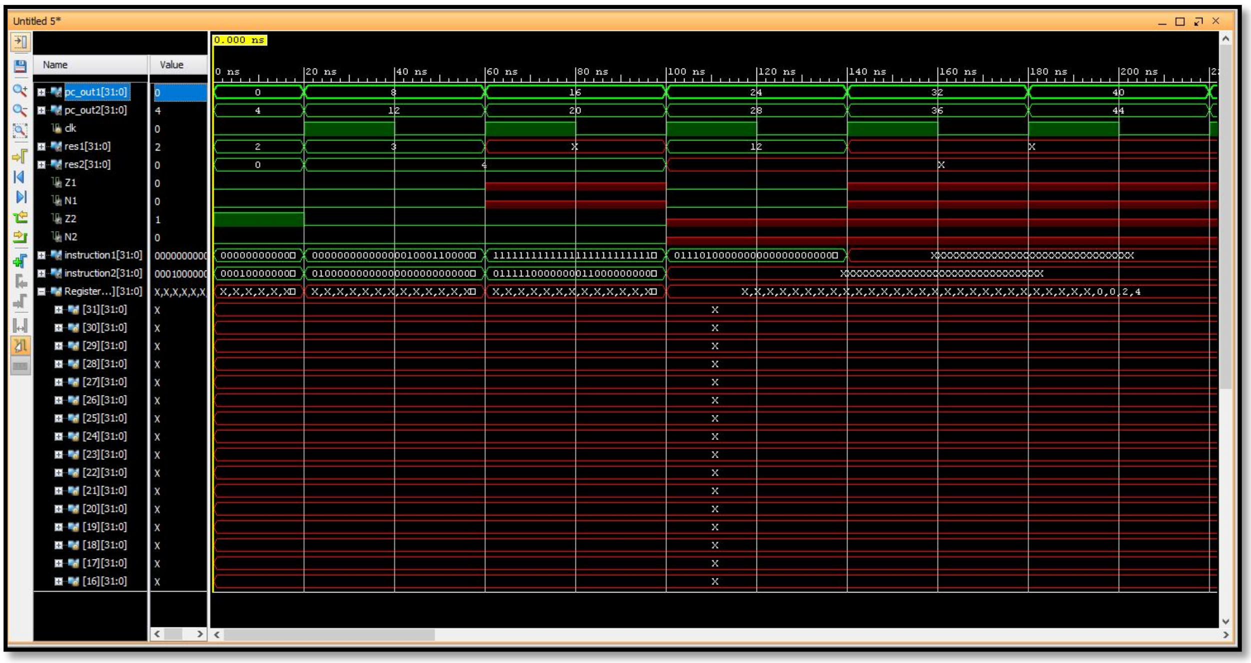


Figure 16(a): WAVEFORM OF THE SUPERSCALAR PROCESSOR

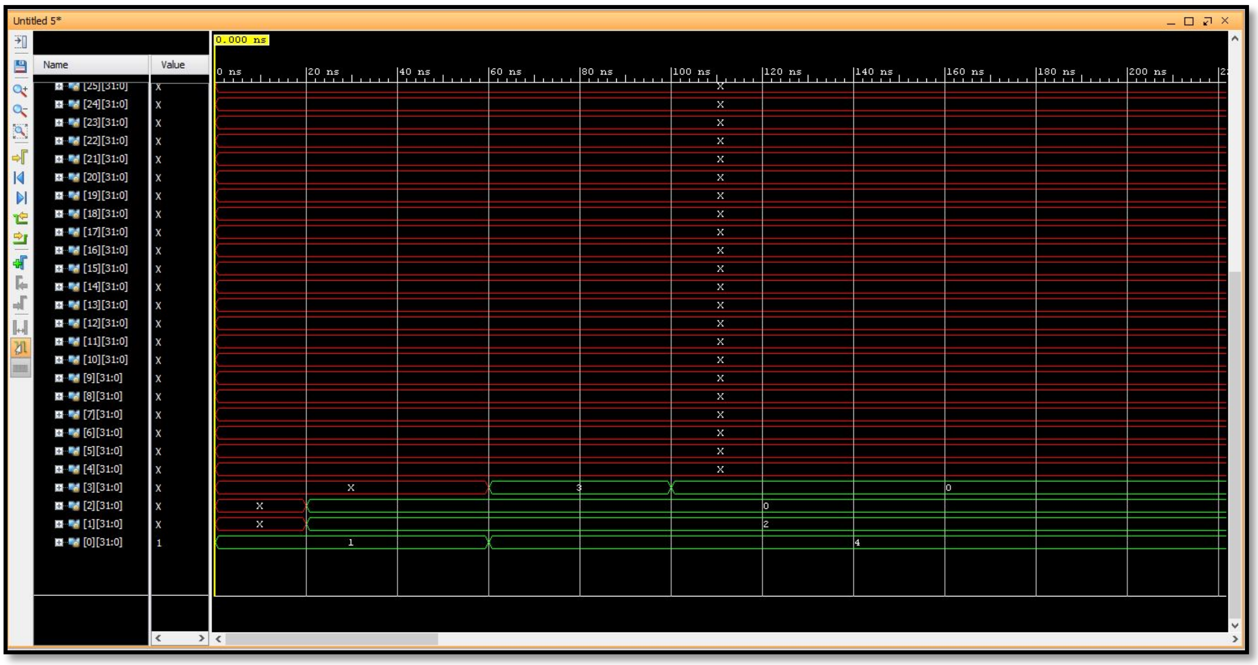


Figure 16(b): WAVEFORM OF THE SUPERSCALAR PROCESSOR (having no dependencies)

Working of the Superscalar processor has been shown for the same instructions used for the single cycle RISC Processor and total time taken to execute all the instructions in single cycle datapath processor is **260ns** while that of dual issue superscalar processor is **140ns** which almost half of the single cycle datapath excluding any type of the dependencies.

DEVELOPMENT OF ASSEMBLER

The RISC processor that we designed takes 32 bit input instructions from the instruction memory. But for executing a set of instructions, this memory has to be hard-coded each time, that too in machine language. This task is quite tedious. This has been our motivation for making an assembler.

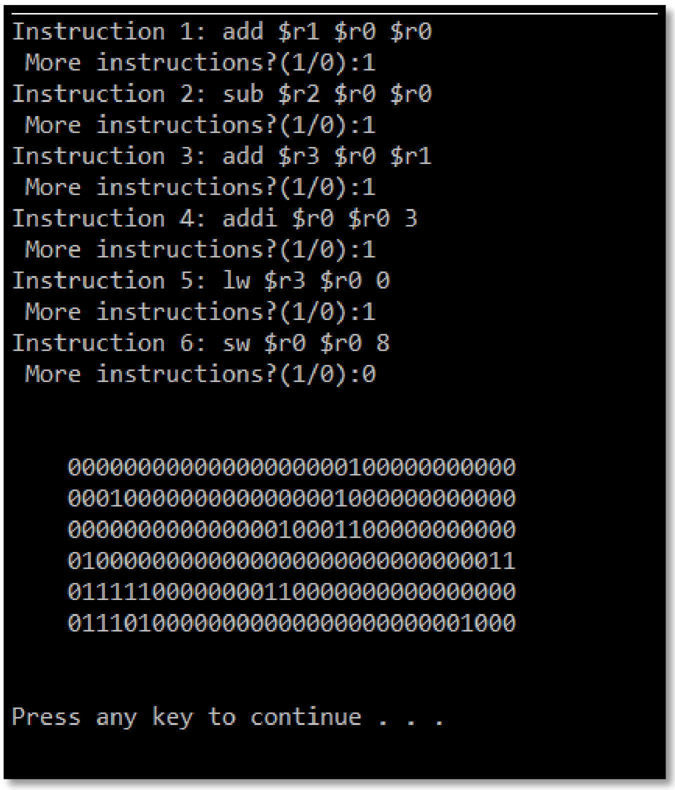
The assembler has been written in C++. It converts the high level instructions entered by the user to machine language. The basic steps involved are the following:

- User gives the input instructions in their high level form.
- Assembler converts the instructions to their machine language form, i.e., binary.
- The binary data is written to a text file.
- The instruction memory module of the processor gets initialized from the text file.

Thus, the processor receives the instructions required for running the task.

The logic involved in creating the assembler is as follows.

Consider the example: **add $r0 $r1 $r2**
If we take this high level instruction, it contains the mnemonic 'add', followed by two registers whose names start with '$'. The assembler first uniquely identifies the mnemonic 'add'. Using a look-up-table, it decides whether the instruction is R format, I format or J format. Depending on the format, it executes a separate function. This function then converts the mnemonic 'add' to its binary op-code value. It also uniquely identifies each of the registers and maps them to their corresponding values. The values are then converted to binary and are appended to the op-code. Thus we get the 26 bit instruction. The last six bits are filled with zeros to make it 32 bit. In case of I type instructions, it identifies the constant/address and converts it to its binary.



```
Instruction 1: add $r1 $r0 $r0
 More instructions?(1/0):1
Instruction 2: sub $r2 $r0 $r0
 More instructions?(1/0):1
Instruction 3: add $r3 $r0 $r1
 More instructions?(1/0):1
Instruction 4: addi $r0 $r0 3
 More instructions?(1/0):1
Instruction 5: lw $r3 $r0 0
 More instructions?(1/0):1
Instruction 6: sw $r0 $r0 8
 More instructions?(1/0):0


    00000000000000000000100000000000
    00010000000000000001000000000000
    00000000000000010001100000000000
    01000000000000000000000000000011
    01111100000000110000000000000000
    01110100000000000000000000001000


Press any key to continue . . .
```

Figure 17: Conversion of assembly code used in test commands to binary instructions

INSPECTION MODULE

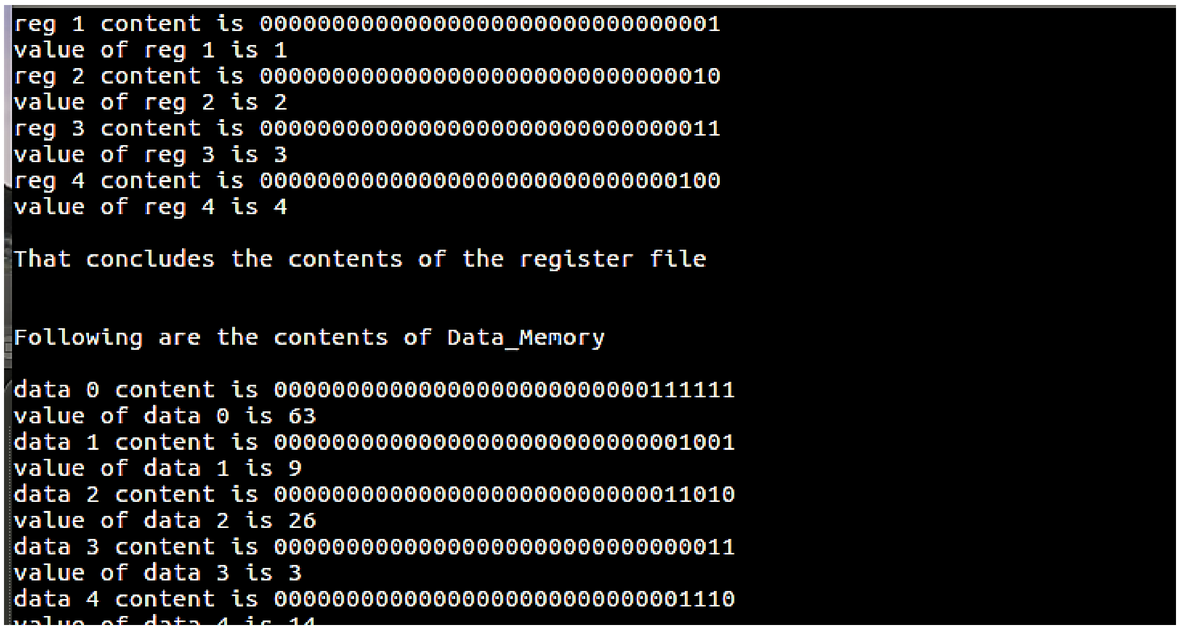
INSPECTION MODULE as the name suggests is used to inspect the contents of any data location at any point of time. We are all curious to see the contents of Registers and other memory location, but while working in VERILOG our curiosity seems to fade when we see continuous arrays of 0s and 1s in data storing locations. So we made an inspection module which helps us to inspect the contents in an organized manner.

FEATURES

- Total module include a code in C++ and another in VERILOG
  - The VERILOG code has been integrated to Data_Memory and Register file.
  - Data is written in a text file during instruction execution.
  - C++ code converts these binary arrays into a ordered list of contents
- We can inspect any data locations by integrating the code to respective modules.

This module not only helped to shed light on the uncanny portions of data memory, it also helped to make error corrections more efficient and simple.

<u>MODEL</u>



```
reg 1 content is 00000000000000000000000000000001
value of reg 1 is 1
reg 2 content is 00000000000000000000000000000010
value of reg 2 is 2
reg 3 content is 00000000000000000000000000000011
value of reg 3 is 3
reg 4 content is 00000000000000000000000000000100
value of reg 4 is 4

That concludes the contents of the register file


Following are the contents of Data_Memory

data 0 content is 00000000000000000000000000111111
value of data 0 is 63
data 1 content is 00000000000000000000000000001001
value of data 1 is 9
data 2 content is 00000000000000000000000000011010
value of data 2 is 26
data 3 content is 00000000000000000000000000000011
value of data 3 is 3
data 4 content is 00000000000000000000000000001110
value of data 4 is 14
```

<u>Figure 18: Testing the Inspection Module and reading all the register and data memory values</u>

CONCLUSION

Our processor is able to handle 16 different instructions with in-order dual issue and execution. The synchronization using single cycle datapath is able to execute all R-type, I-type, J-type formats, which are extended to Superscalar datapath. Moreover, the processor can work on negative values. An assembler is developed so that one does not need to enter machine coded instruction in the Verilog program. The concepts of Datapath, Superscalar implementation, different issue policies and dependencies were understood. Customized 32-bit MIPS processor has increased our understanding of the field computer architecture significantly. It has taught us work ethics and teamwork.

SCOPE FOR FURTHER WORK

We have outlined the strengths and unique features of customized MIPS in the report. This project paves the way to develop the following features to a larger extended.

- Module generation to check the dependencies in the various instructions
- Introduction of Pipeline and pipelining register
- Functionality of combining registers to allow floating point operations
- Including In-Order Issue - Out of Order Execution and Out of Order Issue - Out of Order Execution policies
- Increasing the number of instructions handled by the processor
- Implementing flexibility of the instruction format based on the requirement of the general purpose registers
- FPGA implementation of the processor

REFERENCES

1. Computer Organization and Architecture, 8th edition-*William Stallings*
2. Digital Design and Computer Architecture- D.M. Harris, S.L. Harris
3. Computation Structures- MIT Course EECS 6.004
4. A Report on 16- bit RISC Processor by R.K.S. Parihar, S.Reddy (BITS Pilani)
5. Computer Organization and Design, 4th edition- D.A. Patterson, J.L. Hennessey
6. www.ocf.berkeley.edu
7. https://www.eg.bucknell.edu/~csci320/mips_web/