

NYU-6463-RV32I Processor

PROJECT REPORT

ADVANCED HARDWARE DESIGN

Prof. Ramesh Karri

at the

NEW YORK UNIVERSITY

TANDON SCHOOL OF ENGINEERING

by

Anant Singh (as14229)

Fall 2021

Table of Contents

Table of Contents	i
Table of Figures	ii
1. INTRODUCTION	1
1.1. Instruction Set Architecture.....	1
1.2. RISC-V	2
1.3. Instructions Implemented.....	2
1.4. Project Structure	4
2. MODULES.....	6
2.1. Program Counter	6
2.2. Register File	6
2.3. Instruction Memory	7
2.4. Data Memory.....	7
2.5. Immediate Extender	8
2.6. Data Extender	8
2.7. ALU.....	8
2.8. Branch Comparator	9
2.9. Control Unit.....	9
2.10. Processor	11
3. COMPLEX Program Simulation.....	13
3.1. Binary Search	13
3.2. Sum of Cubes	14
4. UTILIZATION REPORT	17
5. TIMING REPORT.....	19
6. OPTIMIZATION SCOPE.....	20
ANNEXURE A - Assembly code of Programs	21
i) Binary Search	21
ii) Sum of Cubes	22
ANNEXURE B	24
i) Video Presentation Link.....	24
ii) How to Run	24
iii) Folder Structure.....	24

Table of Figures

Figure 1:Datapath of the Processor	5
Figure 2: Simulation screenshot of PC_TB.v	6
Figure 3: Simulation screenshot of RegFile_TB.v	6
Figure 4: Simulation screenshot of IMem_TB.v	7
Figure 5: Simulation screenshot of DMem_TB.v	7
Figure 6: Simulation screenshot of ImmExt_TB.v	8
Figure 7: Simulation screenshot of DataExt_TB.v	8
Figure 8: Simulation screenshot of ALU_TB.v	9
Figure 9: Simulation screenshot of BranComp_TB.v	9
Figure 10:Block Diagram of Control Unit	9
Figure 11: Finite State Machine of Control Unit	10
Figure 12: Simulation screenshot of ControlUnit_TB.v	11
Figure 13: Simulation screenshot of Processor_TB.v	11
Figure 14: Simulation screenshot for Binary Search Test Case 1	13
Figure 15: Zoomed In screenshot showing results for Test Case 1	13
Figure 16: Simulation screenshot for Binary Search Test Case 2	14
Figure 17: Zoomed In screenshot showing results for Test Case 2	14
Figure 18: Simulation screenshot for Sum of Cubes Test Case 1	15
Figure 19: Zoomed In screenshot showing results for Test Case 1	15
Figure 20: Simulation screenshot for Sum of Cubes Test Case 2	15
Figure 21: Zoomed In screenshot showing results for Test Case 2	16
Figure 22: Design Run Report	17
Figure 23: Utilization Summary	17
Figure 24: Utilization Report by Hierarchy	17
Figure 25: Slice Register Utilization Report	18
Figure 26: Design Timing Summary	19
Figure 27: Paths sorted by descending Total Delay	19

1. INTRODUCTION

1.1. Instruction Set Architecture

A high-level program written in assembly language which is in turn given to assembler to turn into binary machine language. ISAs abstract away underlying hardware, enabling program environments which can be shared by many users.

Instruction Set Architecture or ISA specifies the language which the machine will understand. A program is specified as instructions and described via an ISA. For This processor, we are using RISC-V ISA. The ISA specifies for each instruction- what the control unit needs to perform, how the data for the operation is provided (via memory or registers or held constant for some amount of time) and how the result should be stored (in memory or registers). Within each ISA there is an opcode which indicates the type of operation being performed. The type of instructions being executed are arithmetic, logical, memory accesses, control transfers, special purpose.

All instructions are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction-address-misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. This exception is reported on the branch or jump instruction, not on the target instruction. No instruction-address-misaligned exception is generated for a conditional branch that is not taken.

For best performance, the effective address for all loads and stores should be naturally aligned for each data type (i.e., on a four-byte boundary for 32-bit accesses, and a two-byte boundary for 16-bit accesses). The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.

Sign-extension is one of the most critical operations on immediates (particularly for $XLEN > 32$), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

Each memory block is organized as a large one-dimensional array with fixed width(32-bits) and length, a memory address is indexed to the array. Each word of memory can be written by some form of byte-to-word address translation before writing data into the memory. The memory then becomes 4 times longer. Each byte is addressed in a specific order which is determined by endianness- our processor implements little endian format. Memory access time is the amount of time required to read/write data from/to memory. The memory map for the system is divided into the Identity registers, I/O registers, Program Memory and Data Memory. To access the memory at different addresses, mask the address bits to determine which address space to use (data or program). For any program that is run on the processor, the control unit sends control signals to the datapath components and reads all the program instructions and status signals from the datapath directly. The datapath components are the register file, memory unit and ALU.

1.2. RISC-V

RISC-V from an architecture standpoint, is both simple and elegant. RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 40 unique instructions, though a simple implementation might cover the ECALL/EBREAK instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE instruction as a NOP, reducing base instruction count to 37 in total. RV32I can emulate almost any other ISA extension.

RISC-V is significant because it will allow smaller device manufacturers to build hardware without paying royalties and allow developers and researchers to design and experiment with a proven and freely available instruction set architecture.

RISC-V has a modular design, consisting of alternative base parts, with added optional extensions. The ISA base and its extensions are developed in a collective effort between industry, the research community and educational institutions. The base specifies instructions (and their encoding), control flow, registers (and their sizes), memory and addressing, logic (i.e., integer) manipulation, and ancillaries. The base alone can implement a simplified general-purpose computer, with full software support, including a general-purpose compiler.

The standard extensions are specified to work with all of the standard bases, and with each other without conflict. Many RISC-V computers might implement compressed instructions extensions to reduce power consumption, code size, and memory use. There are also future plans to support hypervisors and virtualization. Together with a supervisor instruction set extension- S, an RVGC defines all instructions needed to conveniently support a general-purpose operating system.

The NYU-6463-RV32I is based on Harvard Architecture i.e. the data memory and instruction memory architecture are separate. The processor is faster and there is no need to arbitrate between instructions and data memory. The processor performs the tasks of instruction fetch, instruction decode, execution, memory access and write-back in a multi-cycle manner.

RISC or Reduced Instruction Set Computers have all instructions of the same size, there are fewer instructions, boasts simpler hardware but longer program times (as a greater number of instructions are required for certain complex operations).

1.3. Instructions Implemented

- **LUI:** LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros
- **AUIPC:** AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.

- **JAL:** The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a ± 1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd.
- **JALR:** The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not needed. The JAL and JALR instructions will generate a misaligned instruction fetch exception if the target address is not aligned to a four-byte boundary.
- **BEQ:** compares two registers and branches if registers rs1 and rs2 are equal
- **BNE:** compares two registers and branches if registers rs1 and rs2 are unequal
- **BLT:** compares two registers using signed comparison and branches if rs1 is less than rs2
- **BGE:** compares two registers using signed comparison and take the branch if rs1 is greater than or equal to rs2
- **BLTU:** compares two registers using unsigned comparison and branches if rs1 is less than rs2
- **BGEU:** compares two registers using unsigned comparison and take the branch if rs1 is greater than or equal to rs2
- **LB:** LB loads an 8-bit value from memory, then sign-extends to 32-bits before storing in rd.
- **LH:** LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd.
- **LW:** The LW instruction loads a 32-bit value from memory into rd
- **LBU:** LBU loads an 8-bit value from memory, then zero-extends to 32-bits before storing in rd.
- **LHU:** LHU loads a 16-bit value from memory, then zero-extends to 32-bits before storing in rd.
- **SB:** stores lower 8-bits of rs2 to memory
- **SH:** stores lower 16-bits of rs2 to memory
- **SW:** stores 32-bits of rs2 to memory
- **ADDI:** ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored, and the result is simply the low XLEN bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.
- **SLTI:** SLTI (set less than immediate) places the value 1 in register rd if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to rd.
- **SLTIU:** SLTIU is like SLTI but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to XLEN bits then treated as an unsigned number)
- **XORI:** logical operation that perform bitwise XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd. XORI rd, rs1, -1 performs a bitwise logical inversion of register rs1 (assembler pseudo-instruction NOT rd, rs).

- **ORI:** logical operation that perform bitwise OR on register rs1 and the sign-extended 12-bit immediate and place the result in rd. Note, XORI rd, rs1, -1 performs a bitwise logical inversion of register rs1 (assembler pseudo-instruction NOT rd, rs).
- **ANDI:** logical operation that perform bitwise AND on register rs1 and the sign-extended 12-bit immediate and place the result in rd.
- **SLLI:** SLLI is a logical left shift (zeros are shifted into the lower bits)
- **SRLI:** SRLI is a logical right shift (zeros are shifted into the upper bits)
- **SRAI:** SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits)
- **ADD:** ADD performs addition between 2 operands, overflow is ignored and the XLEN bits of results are written to the destination
- **SUB:** SUB performs subtraction between 2 operands, overflow is ignored and the XLEN bits of results are written to the destination
- **SLL:** SLL performs logical left shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.
- **SLT:** sets register rd to 1 if signed register rs1 < signed register rs2, else 0.
- **SLTU:** sets register rd to 1 if unsigned register rs1 < unsigned register rs2, else 0.
- **XOR:** performs rs1 XOR rs2 and stores in rd
- **SRL:** SRL performs logical right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.
- **SRA:** SRA performs arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.
- **OR:** performs rs1 OR rs2 and store in rd
- **AND:** performs rs1 AND rs2 and store in rd
- **FENCE:** The FENCE/NOP instruction does not change any user-visible state, except for advancing the pc. NOP is encoded as ADDI x0, x0, 0.
- **ECALL:** It is implemented as HALT (stops program execution)
- **EBREAK:** It is implemented as HALT (stops program execution)

1.4. Project Structure

The Processor.v is the top module, and it connects and routes all the control signals, status signals and initiates lower modules namely- Program Counter, Instruction Memory, Data Memory, Register File, Branch Comparator, ALU, Data Extender and Immediate Extender. There are 5 control muxes - pc_mux, rfile_mux, alu_mux1, alu_mux2 and op_mux defined to direct the dataflow in execution states of the FSM based on various control and status signals generated by Control Unit.

Processor datapath is shown in Figure 1, in which red boxed signals are input to Control Unit (or FSM) and blue boxed signals are outputs from Control Unit which controls all the other modules. The PC value is used as an address to index the instruction memory which supplies a 32-bit value of the next instruction to be executed.

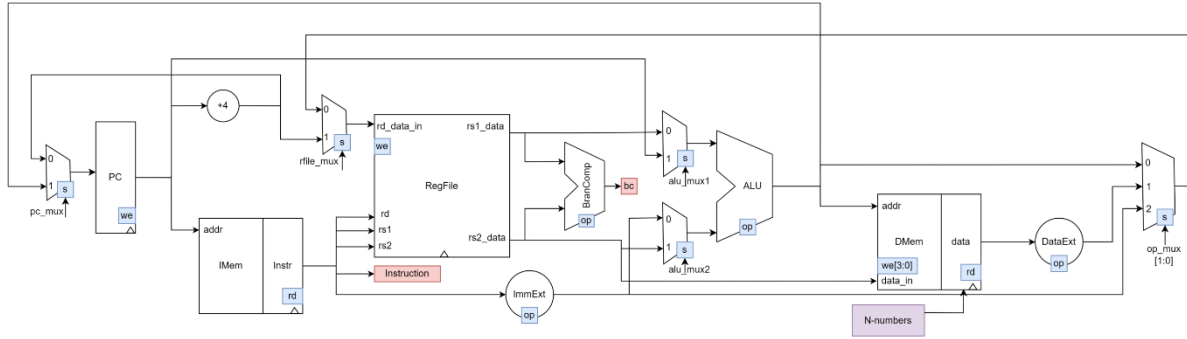


Figure 1:Datapath of the Processor

This instruction is then divided into different fields to specify their types: R-type, I-type, S-type, B-type, U-type, and J-type. The instructions' opcode and function field bits are sent to the decode FSM unit to figure out the type of instruction to execute.

The type of instruction then decides which control signals and opcodes are to be asserted and what function the ALU, and other components will perform. The 5 muxes has select lines to route the data. Each instruction may take a varying number of clock cycles to execute as the FSM invokes the various parts of the datapath.

Depending upon the given instruction, register and memory values may be read or written. Data values can also be worked on by the ALU. Each individual operation is figured out by the control unit to either compute a memory address (e.g., load or store), perform an arithmetic operation (e.g., and, or, xor and add, xor and sub), or compare (e.g., branch). If the instruction decoded is arithmetic, the ALU result must be written to a register. If the instruction decoded is a load or a store, the ALU result is used to address the data memory. The decisive step writes the ALU result or memory value back to the register file.

2. MODULES

The design contains 9 modules with 9 testbenches, 5 Multiplexers and one Processor.v which is the top module of the project and where all the modules are connected. These modules are described below.

2.1. Program Counter

The PCnt.v file contains the 32-bit program counter register for the processor that outputs the address of the next instruction to be loaded from Instruction Memory. Upon invoking active-low reset, it is equal to start of instruction memory (0x01000000).

It is tested by PC_TB.v. The number of clock cycles for all low-level test cases in testbench PC_TB.v takes 16 clock cycles. The module is tested for whether its next value is getting updated in the next clock cycle when the reset signal rstn (active low) is high. It is also tested for when we signal is low, the PC value does not get updated and holds current value. Upon invoking reset or rstn = 'low', PC resets to 0x01000000.

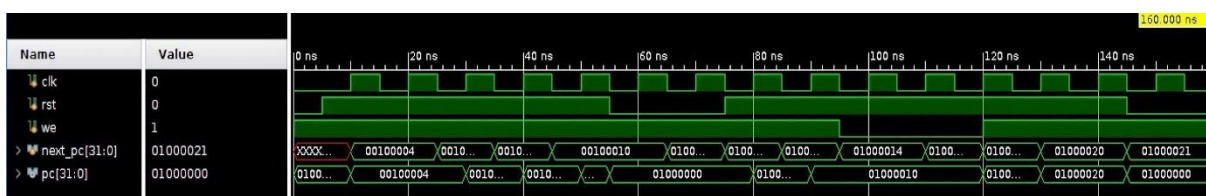


Figure 2: Simulation screenshot of PC_TB.v

2.2. Register File

The RegFile.v module contains the Register File of 32 32-bit registers. The module supports two independent register reads and one register write in one clock cycle. 5 bits are used to address each register. All registers are initialized to 0. If reset signal is invoked, all the registers are set to 0. Register x0 is a special read-only register that is hardwired to 0 always. If write enable signal we is high, input data is written into the register as per the address given. If data is being read from register, then the values at desired locations are loaded into source registers rs1 and rs2.

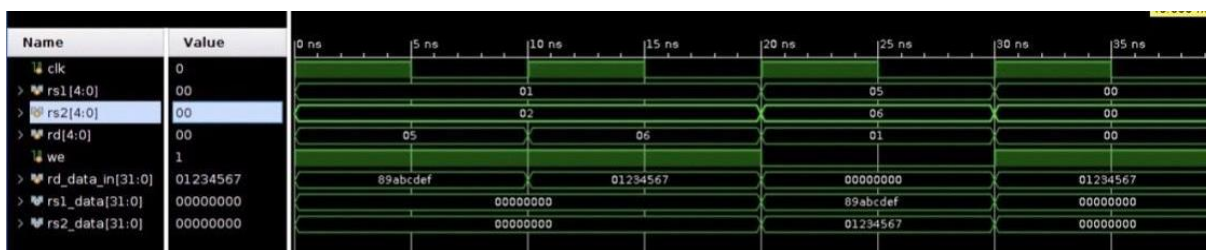


Figure 3: Simulation screenshot of RegFile_TB.v

The module is tested by RegFile_TB.v and the number of clock cycles for all low-level

tests testbench RegFile_TB.v is 4 clock cycles.

Using the testbench, data is being written (write-enable 'we' high) into reg5 and then reg6 in the next clock cycle.

In the subsequent clock cycle, reg5 and reg6 are read ('we' low and read signal 'rd' high) simultaneously. In the next case, we check whether hardwired reg0 can be written into while keeping rd signal low and we signal high.

2.3. Instruction Memory

The IMem.v module is the instruction memory for the processor which is initialized to contain the program to be executed. Instruction memory width is 4 bytes (32-bits), although it is byte-addressed. It is of size 2 KB and addressed by 11 bits. The program is loaded into instruction memory from memory file imem.mem. Instruction and Data Memory accesses are restricted to 4-byte alignment in the NYU-6463-RV32I. It begins at address 0x01000000. It loads the file imem.mem and reads data values at specified addresses

The number of clock cycles taken for all low-level test case testbench by IMem_TB.v is 10 clock cycles. The Testbench checks if the instruction memory or ROM is reading the values that are stored in the desired address locations. After reset it should output instruction at 0x01000000, which is the starting address location for the instruction memory.

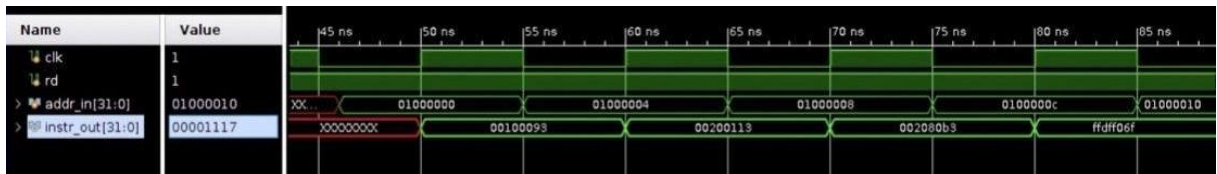


Figure 4: Simulation screenshot of IMem_TB.v

2.4. Data Memory

The DMem.v module contains the data memory for the processor. Data memory is 4 KB size. It begins at the address 0x80000000. The load from and store to the memory takes place in this module. It is tested by DMem_TB.v, the tests include reading the N-numbers of team members at special read-only locations address 0x00100000, 0x00100004, 0x00100008 in the data memory and outputs them one by one.



When the read signal is low and reset (active low) is high, data is written into data memory at specified address byte-wise and the writing of byte-wise data is controlled by 4-bit write enable. The read is set, and stored data is read and verified. At locations 0x00100000, 0x00100004 and 0x00100008, when read signal is high and write enable, ‘we’ is 4'b0, the N-numbers of our team members is output in consecutive clock cycles. DMem_TB.v tests DMem.v in 19 clock cycles.

2.5. Immediate Extender

The ImmExt.v module extends the immediate depending on the type of instruction being executed (J-type, I-type, S-type, B-type, U-type) based on the MSB value of the immediate. It is tested by ImmExt_TB.v.

ImmExt_TB.v checks ImmExt.v depending on the opcode if the data input is of given length is being completely extended to desired 32-bit value.

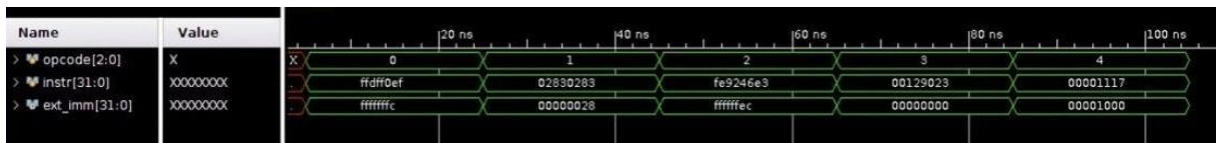


Figure 6: Simulation screenshot of ImmExt_TB.v

2.6. Data Extender

The DataExt.v depending on the type of load instruction being executed sign extends the incoming data bits of given length to 32-bits, using the MSB value of the incoming data bits. It is tested by DataExt_TB.v

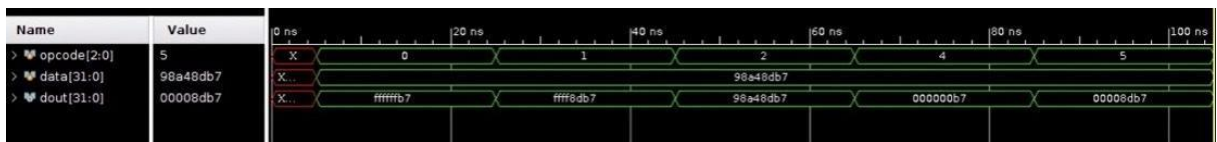


Figure 7: Simulation screenshot of DataExt_TB.v

In the testbench, the data memory is tested for diverse types of load instructions and output is verified against expected result values.

2.7. ALU

The ALU.v module contains the ALU for the processor. It handles arithmetic, logical and memory offset operations based on the control signals. It uses the control signals generated by the Control Unit, and data from the registers or from the instruction directly. It computes data that can be written into one of the registers (including PC).

It is tested by ALU_TB.v and takes 225 ns total run time. The testbench checks if given two operand inputs and operation type, depending on type of operation (arithmetic and logical type)

to be performed, a case statement is invoked which returns the desired output from given data values after the operation has been performed on them. ALU_TB.v checks for all the operations.

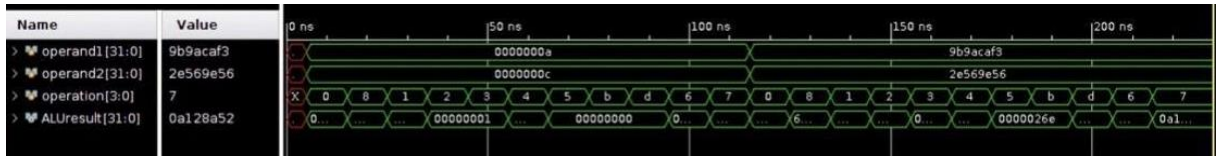


Figure 8: Simulation screenshot of ALU_TB.v

2.8. Branch Comparator

The BranComp.v module has the logic to check if the branch will be taken for the given B-type instructions or Branch instructions. It is tested by BranComp_TB.v.

The testbench BranComp_TB.v verifies for all the branch operations supported by the processor whether the signed or unsigned branch is occurring properly. BranComp_TB.v takes 390 ns total run time.

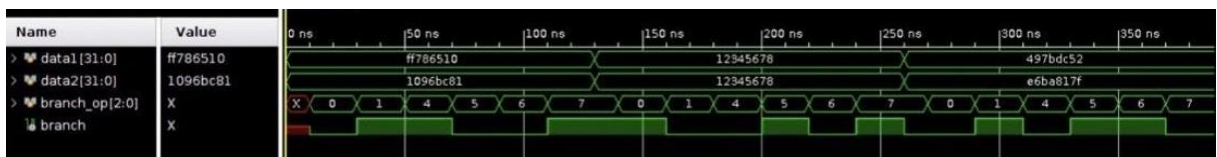


Figure 9: Simulation screenshot of BranComp_TB.v

2.9. Control Unit

The ControlUnit.v module contains the Control unit of the processor. It generates the control signals for coordinating all the modules according to the instruction being executed and takes care of the multicycle state machine for current execution cycle (Instruction Fetch, Instruction decode and execution, Memory, Write back and PC update, and Halt). It is tested by ControlUnit_TB.v.

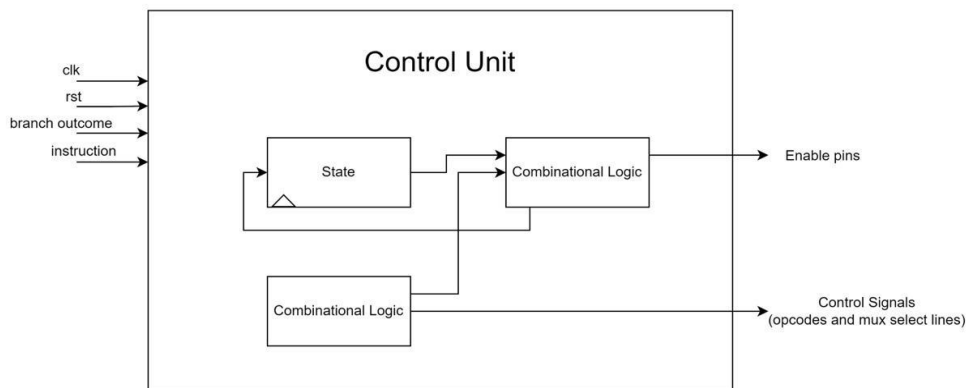


Figure 10: Block Diagram of Control Unit

The behavior of the system is governed by an FSM that lists 5 states: fetch-access the memory to get the next instruction (activate the read signal, place the right memory address on address bus, read the content of memory pointed by the address), decoding (interpret the data bits of instruction word to identify the operation and its data which can be taken from memory or registers), execution (perform specific operation using processor resources and write the result into memory and/or registers). Our Processor implements a Mealy machine where current output is dependent on current input and current state. The FSM states -

- State 0: Instruction Fetch
- State 1: Instruction Decode and Execute
- State 2: Memory state
- State 3: Write back and Update PC
- State 4: HALT

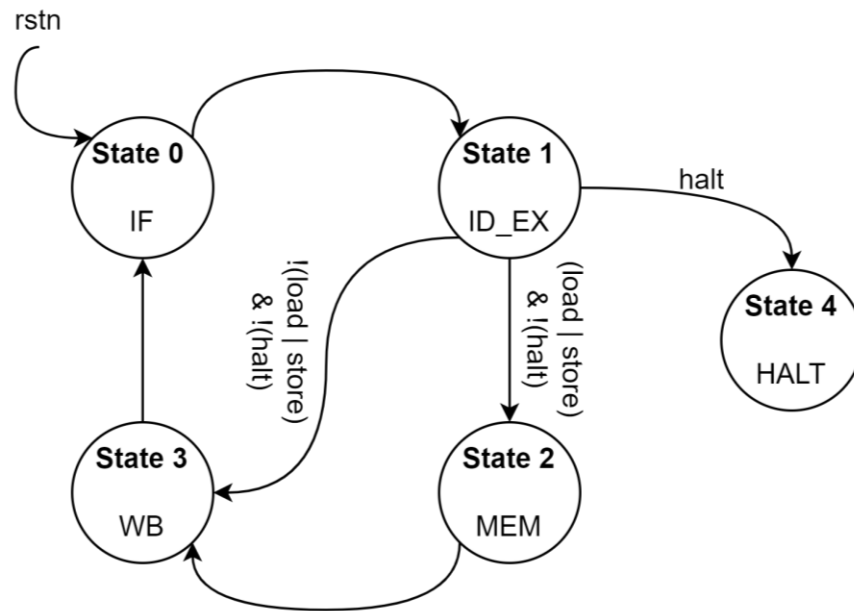


Figure 11: Finite State Machine of Control Unit

ControlUnit_TB.v tests the ControlUnit.v module. The control unit testbench comprises test cases for LUI, AUIPC, JAL, JALR, BEQ, BNE, BLT, BGE, BLTU, BGEU, LB, LH, LW, LBU, LHU, SB, SH, SW, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI, ADD, SUB, SLTU, XOR, SRL, SRA, OR, AND, FENCE, ECALL and EBREAK. It checks if the proper control signals for the current instruction being executed are generated for all the instructions. The number of clock cycles taken for all low-level test cases executed in the testbench ControlUnit_TB.v is 100 clock cycles. There are 4 opcodes, 5 select lines for multiplexers and 4 enable signals, each of these are checked for each instruction thoroughly so that other modules perform in desired fashion once connected through top modules Processor.v.

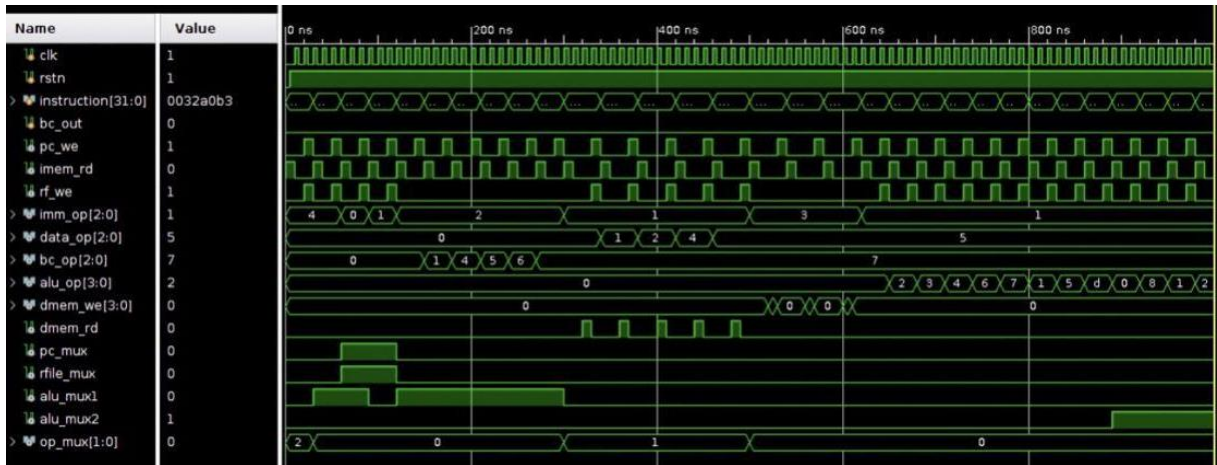


Figure 12: Simulation screenshot of ControlUnit_TB.v

2.10. Processor

Processor.v is the top module for the project, and it combines all the modules into one single module that initiates all the other modules and works to execute instructions. This module also has 5 multiplexers to control the data flow as per the execution state of the FSM based on control signals generated by ControlUnit.v.

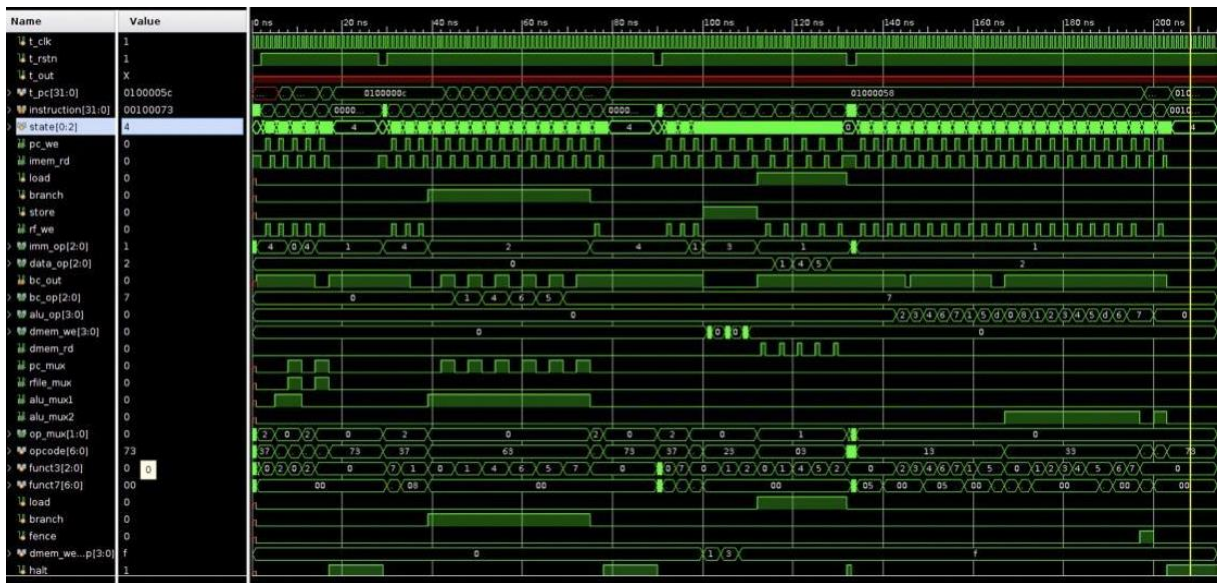


Figure 13: Simulation screenshot of Processor_TB.v

Processor_TB.v tests for all the instructions and checks the top module with the help of 4 small Assembly programs. Each test loads a memory file and loads the program to the instruction memory which are then executed, and the output is verified depending upon the current instruction. The 4 memory files are named P_TB_test in series from 1 to 4 for each test. It contains the following test cases -

- Test 1 - Testing for LUI, AUIPC, JAL, JALR, ECALL.
- Test 2 - Testing for Branch instructions BEQ, BNE, BLT, BGE, BLTU, BGEU
- Test 3 - Testing for Load Store Instructions LB, LH, LW, LBU, LHU, SB, SH,
- Test 4 - Testing for R-type and I-type Arithmetic Instructions

Testbench takes 190 clock cycles to run the 4 tests. The testbench reads the hex values from files P_TB_test1.mem, P_TB_test2.mem, P_TB_test3.mem and P_TB_test4.mem. It checks the various status and control signals and verifies proper flow, operation, and coordination between all the modules by testing the functionalities of all the instructions.

3. COMPLEX PROGRAM SIMULATION

The processor is also tested on 2 high level programs which ensures that the processor is working as expected.

3.1. Binary Search

An array of size 12 is defined in data memory and a key value is defined for searching. The mid value is found from the sorted array and the range of key search is adjusted accordingly with each iteration. The key value and its position w.r.t the starting address of the array defined are stored in r1 and r2, respectively. The array elements of size 12 are stored in data memory with corresponding offset values.

- **Test Case 1:**

Input Array in Data Memory:	{ 1, 3, 7, 9, 11, 13, 15, 17, 19, 21, 23 }
Memory locations:	0x80000000, 0x80000004, ...
Key to be searched:	21 (stored in r11)
Output:	
Obtained value:	21 (stored in r1)
Memory offset:	36 (stored in r2)
Number of cycles :	186 (4 for all load, store instructions; 3 for all other)

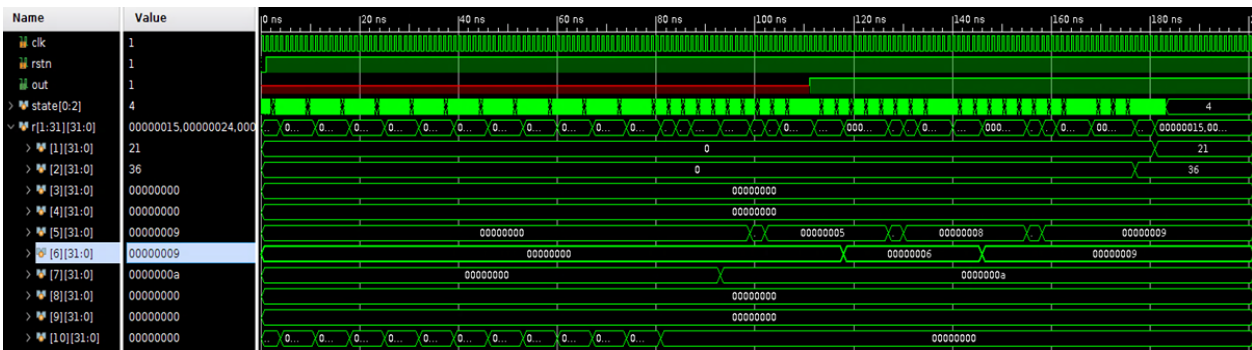


Figure 14: Simulation screenshot for Binary Search Test Case 1

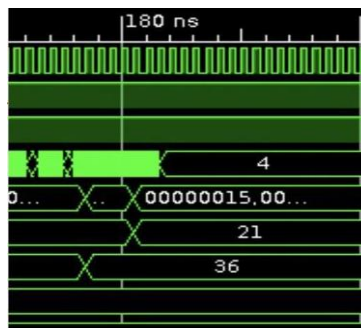


Figure 15: Zoomed In screenshot showing results for Test Case 1

- **Test Case 2:**

Input Array in Data Memory: {1, 3, 7, 9, 11, 13, 15, 17, 19, 21, 23}

Memory locations: 0x80000000, 0x80000004, ...

Key to be searched: 8 (stored in r11)

Output:

Obtained value: 0 (stored in r1)

Memory offset: 0 (stored in r2)

Number of cycles: 192 (4 for all load, store instructions; 3 for all other)

These values for output are obtained because the key value 8 does not exist in the array

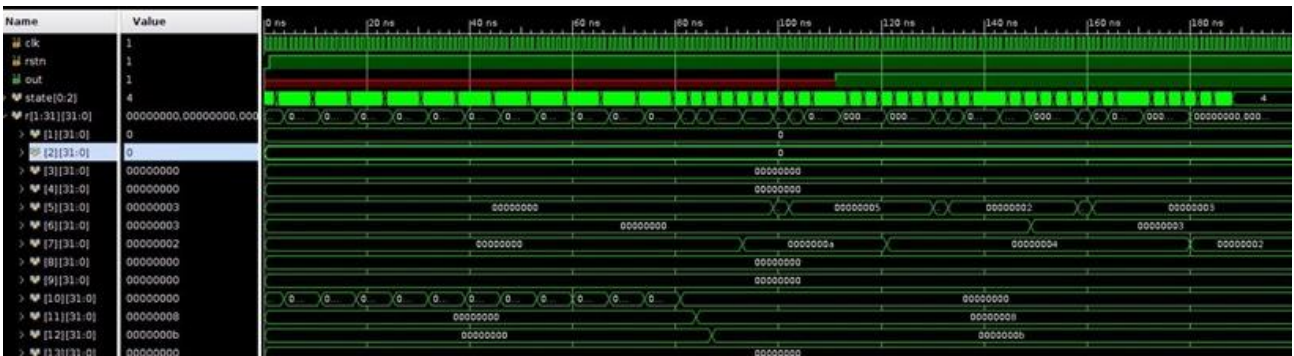


Figure 16: Simulation screenshot for Binary Search Test Case 2



Figure 17: Zoomed In screenshot showing results for Test Case 2

2. Sum of Cubes

The program sums up the cubes of odd numbers between m and n. The values of m and n (positive or negative integers) are stored in r5 and r6 registers. The cube for each individual odd number is calculated by repeated addition and added to the output register r7. The final value in r7 after the HALT instruction gives the output.

- **Test case 1:**

Inputs: r5 = m = 1; r6 = n = 8

Output: r7 = 496 ($1^3 + 3^3 + 5^3 + 7^3$)

Number of cycles: 744 (4 for all load, store instructions; 3 for all other)



Figure 18: Simulation screenshot for Sum of Cubes Test Case 1

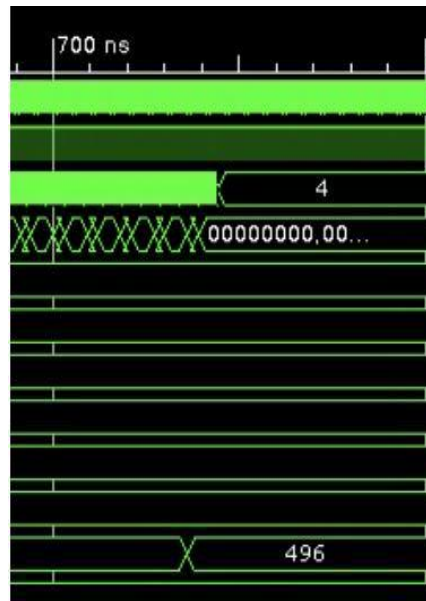


Figure 19: Zoomed In screenshot showing results for Test Case 1

- **Test case 2:**

Inputs: $r5 = m = -3$; $r6 = 20$

Output: $r7 = 19872$

Number of cycles: 3687 (4 for all load, store instructions; 3 for all other)

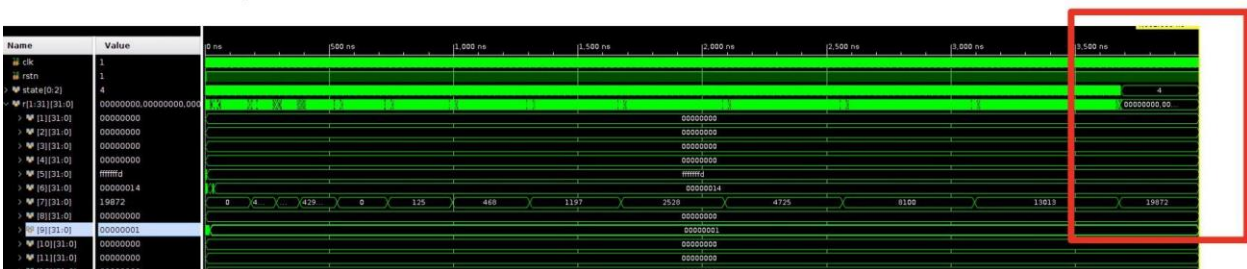


Figure 20: Simulation screenshot for Sum of Cubes Test Case 2



Figure 21: Zoomed In screenshot showing results for Test Case 2

4. UTILIZATION REPORT

The design is synthesizable and after implementation the resource utilization is below than 3% in all the categories for Nexys 4 DDR FPGA board. A constraint file is used for Nexys 4 DDR FPGA board.

Tcl Console	Messages	Log	Reports	Design Runs	x	DRC	Methodology	Power	Timing	Utilization				
Q	≡	≡	≡	≡	≡	≡	≡	≡	≡	≡				
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!								1704	606	0.00	0	0
✓ impl_1	constrs_1	route_design Complete!	3.266	0.000	0.202	0.000	0.000	0.101	0	1704	606	0.00	0	0

Figure 22: Design Run Report

Our Design uses 1704 LUTs as logic and 512 LUTs as distributed RAM. 742 flipflops and only 3 IO ports are being utilized as shown in the screenshot.

Resource	Utilization	Available	Utilization %
LUT	1704	63400	2.69
LUTRAM	512	19000	2.69
FF	742	126800	0.59
IO	3	210	1.43

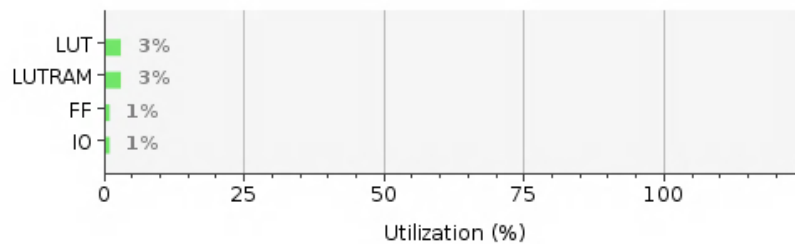


Figure 23: Utilization Summary

Memory elements are utilizing the most number of resources, which is expected. Utilization by individual modules is shown below.

Name	^1	Slice LUTs (63400)	Bonded IOB (210)	BUFGCTRL (32)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)
Processor		1704	3	3	742	317	131	527	1192	512
ArithmeticLogicUnit (ALU)		64	0	0		23	0	79	64	0
BranchComparator (BranComp)		0	0	0		0	0	1	0	0
DataExtender (DataExt)		0	0	0		0	0	15	0	0
DataMemory (DMem)		576	0	0		256	128	179	64	512
ImmediateExtender (ImmExt)		0	0	0		0	0	10	0	0
InstructionMemory (IMem)		104	0	0		0	0	58	104	0
MainController (ControlUnit)		61	0	0		0	0	55	61	0
ProgramCounter (PCnt)		29	0	0		0	0	32	29	0
RegisterFile (RegFile)		870	0	0		38	3	297	870	0

Figure 24: Utilization Report by Hierarchy

There are 742 slice Registers are being utilized by different modules as reported by Vivado.

Name	Used
▼ N Processor	742
<i>I</i> RegisterFile (RegFile)	480
<i>I</i> DataMemory (DMem)	64
<i>I</i> ArithmeticLogicUnit (ALU)	56
<i>I</i> DataExtender (DataExt)	32
<i>I</i> ProgramCounter (PCnt)	32
<i>I</i> ImmediateExtender (ImmExt)	26
<i>I</i> MainController (ControlUnit)	26
<i>I</i> InstructionMemory (IMem)	25
<i>I</i> BranchComparator (BranComp)	1

Figure 25: Slice Register Utilization Report

5. TIMING REPORT

The system clock period is set to 10 ns and the operating frequency of the processor is 100 MHz. All timing constraints are met by our design and worst negative slack reported is 3.266 ns, which leaves room to run the processor on even higher clock frequency.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 3.266 ns	Worst Hold Slack (WHS): 0.202 ns	Worst Pulse Width Slack (WPWS): 3.750 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 2194	Total Number of Endpoints: 2194	Total Number of Endpoints: 1119

All user specified timing constraints are met.

Figure 26: Design Timing Summary

The max path will be for the 4-cycle path for the Load instructions. This is because the load instruction covers the maximum possible stages and components. But in our implementation, since write back and PC update takes place in the same cycle; therefore, the store instructions take the same time. In implementation flow the Vivado tool optimizes the memory elements and memory elements are implemented in a distributed manner, which effects the time taken by paths.

The most time taken by a path in our design is from Instruction Memory to Data Memory. The longest path reported by implemented design sorted by total delay is shown below.









Name	Slack	Levels	High Fanout	From	To	Total ...  1	Logic Delay	Net Delay	Requirement	Source Clock
 Path 1	3.266	3	230	InstructionM...t_reg[20]/C	DataMemory...AMS64E_A/I	5.938	0.966	4.972	10.000	sys_clk_pin
 Path 2	3.327	2	230	InstructionM...t_reg[20]/C	DataMemory...AMS64E_A/I	5.876	0.842	5.034	10.000	sys_clk_pin
 Path 3	3.333	3	230	InstructionM...t_reg[20]/C	DataMemory...AMS64E_A/I	5.870	0.966	4.904	10.000	sys_clk_pin
 Path 4	3.382	2	133	InstructionM...t_reg[22]/C	DataMemory...AMS64E_A/I	5.820	0.704	5.116	10.000	sys_clk_pin
 Path 6	3.397	3	230	InstructionM...t_reg[20]/C	DataMemory...AMS64E_A/I	5.806	0.966	4.840	10.000	sys_clk_pin
 Path 5	3.394	3	230	InstructionM...t_reg[20]/C	DataMemory...AMS64E_A/I	5.802	0.966	4.836	10.000	sys_clk_pin
 Path 7	3.413	3	230	InstructionM...t_reg[20]/C	DataMemory...AMS64E_A/I	5.792	0.966	4.826	10.000	sys_clk_pin

Figure 27: Paths sorted by descending Total Delay

6. OPTIMIZATION SCOPE

An idea for future improvement would be to implement a pipelined design for the processor. To maximize the resources and minimize latency and operation idle times, it makes sense to implement a pipelined design wherein multiple operations and read and write access to different memory locations can be implemented. This ensures faster computation speed and significantly improved response and wait times.

Pipelining sequences the execution of multiple instructions like cars on an assembly line. The execution of each instruction is divided into several steps which are performed by dedicated hardware units. Pipelining is like an assembly line, in which each stage focuses on one unit of work. The result of each stage passes to the next stage until the final stage. To apply the pipelining strategy to an application that will run on a multi-core CPU, the algorithm is divided into steps that require the same amount of work and runs each step on a separate core. The algorithm can process multiple sets of data or the data that streams continuously.

We can also use HLS to synthesize several pipelined implementations, each characterized by a particular trade-off point between area and performance. The instruction set architecture and format can be tweaked to include a bit-manipulation subset which can aid cryptographic, graphic, and mathematical operations.

ANNEXURE A - ASSEMBLY CODE OF PROGRAMS

i) Binary Search

```
addi x10,x0,1
sw x10,0(x0)
addi x10,x0,3
sw x10,4(x0)
addi x10,x0,7
sw x10,8(x0)
addi x10,x0,9
sw x10,12(x0)
addi x10,x0,11
sw x10,16(x0)
addi x10,x0,13
sw x10,20(x0)
addi x10,x0,15
sw x10,24(x0)
addi x10,x0,17
sw x10,28(x0)
addi x10,x0,19
sw x10,32(x0)
addi x10,x0,21
sw x10,36(x0)
addi x10,x0,23
sw x10,40(x0)
addi x10,x0,0
addi x11,x0,21      #change this to 8 for testcase 1
addi x12,x0,12

# x10 = int arr[]
# x11 = int key
# x12 = int size
# x5 = mid
# x6 = left
# x7 = right
    addi x6, zero, 0      # left = 0
    addi x7, a2, -1      # right = size - 1
LOOP1:
# while loop
    blt x6, x7, ABSENT   # left > right, break
    add x5, x6, x7       # mid = left + right
```



```

srai x5, x5, 1      # mid = (left + right) / 2
# Get the element at the midpoint
slli x29, x10, 2    # Scale the midpoint by 4
add x29, x5, x29    # Get the memory address of arr[mid]
lw x29, 0(x29)      # Dereference arr[mid]
# See if the key (a1) > arr[mid] (t3)
# if key <= t4, we need to check the next condition
bge x29, x11, LOOP2
# If we get here, then the key is > arr[mid]
addi x6, x5, 1      # left = mid + 1
jal zero, LOOP1

```

LOOP2:

```

beq x11, x29, FOUND # skip if key == arr[mid]
# If we get here, then key < arr[mid]
addi x7, x5, -1     # right = mid - 1
jal zero, LOOP1

```

FOUND:

```

# If we get here, then key == arr[mid]
slli x2, x5, 2      # Scale the midpoint by 4
lw x1, 0(x2)
jal zero, halt

```

ABSENT:

```

addi x1, x0, -1

```

halt:

```

ecall

```

ii) Sum of Cubes

```

add x7, x7, x0      # Output x7 = 0,
addi x5, x0, 1
addi x6, x0, 8
add x18, x0, x5      # x18 = x5
# We want to store lower value in x18 and higher in x6
blt x5, x6, continue
add x18, x0, x6      # If x5 >= x6, x18 = x6
add x6, x0, x5        # x6 = x5

```

Continue:

```

andi x9, x18, 1     # If N1 is even x9 = 0, else if odd x9 = 1

```

```

bne x9, x0, oddn1
addi x18, x18, 1    # If N is even, x18 (N1) = N+1 (First odd number)

oddn1:,
andi x10, x6, 1    # If N2 is even x10 = 0, else if odd x10 = 1
bne x10, x0, oddn2
addi x6, x6, -1    # If N2 is even, x6 (N2) = N-1 (Last odd number)

oddn2:
addi x6, x6, 1     # x6 = x6 + 1 to have iterator (i) max = N

loopi:             # Loop to iterate from N1 to N2
addi x20, x0, 1    # x20 = 1
add x21, x18, x0   # Temporary register to store i
bge x18, x0, pos   # If i is positive, skip to next steps
sub x21, x0, x21   # If i is negative, x21 = -i
addi x20, x0, -1   # x20 = -1 if i is negative

pos:
addi x31, x0, 3    # Iterator j = 3
add x30, x0, x0    # x30 = 0, to have iterator (j) min = 1

loopj:             # Loop to calculate i^3
andi x29, x29, 0   # Iterator k = 0
andi x28, x28, 0   # Temporary register to store sum x28 = 0

loopk:             # Loop to add i iteratively to perform multiplication
# If i is positive, we iteratively add 1, else add -1
add x28, x28, x20
addi x29, x29, 1    # k = k + 1
blt x29, x21, loopk # If k < i, run loop again
addi x20, x28, 0    # x20 = i^(4 - j)
addi x31, x31, -1   # j = j - 1
bne x31, x30, loopj # If j != 0, run loop again
add x7, x7, x20     # output x7 = (-N1)^3 + (-(N1+1))^3 ...+ i^3
addi x18, x18, 2    # i = i + 2, increment to next odd number
blt x18, x6, loopi  # If iterator i <= N2, run loop i again

endi:              # end loopi
ecall

```

ANNEXURE B

i) Video Presentation Link

ii) How to Run

The various modules and the top Processor module can be run and tested using Simulation on Xilinx Vivado. Apart from the testbenches, the TCL Simulation Commands are present at the end of each module file to stress test the components.

For the main Processor module, the assembly codes for the instructions are converted to the hex values and stored in the memory files. The instructions are read in the Instruction Memory from this memory file and the simulation is run for those instructions.

iii) Folder Structure

The project is hosted on Github and anyone can clone from-

<https://github.com/95anantsingh/ECE-GY-6463-NYU-6463-RV32I-Processor>

The Project Structure is unique to Vivado and has not been changed, after cloning or forking one can directly start the project by opening \Project_Vivado\Project_Vivado.xpr in Xilinx Vivado 2018.3 or higher.

All the vivado files are present in the Project_Vivado folder. The text files for the Complex tests with the assembly codes and hex conversions are present in these files. The .mem files for these tests are also present in the simulation sources.

\Project_Vivado\Project_Vivado.srcs\sim_1\new folder contains the design files for all modules and the defines.vh file for the `define values. This is used as a header file in the modules.

\Project_Vivado\Project_Vivado.srcs\sources_1\new The sim_1 folder contains the testbenches for all the modules, .mem files for the Processor testbench and .mem files for the Complex programs.

\Project_Vivado\Project_Vivado.srcs\constrs_1\imports\Project_Vivado folder contains the constraints file Nexys4DDR_Master.xdc used for timing simulation.