

e-ISSN: 2319-8753 | p-ISSN: 2347-6710

TEDIA

International Journal of Innovative Research in SCIENCE | ENGINEERING | TECHNOLOGY

INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY

Volume 11, Issue 7, July 2022

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

Impact Factor: 8.118

9940 572 462

6381 907 438

 \bigcirc





||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

Design and Verification of 32 bit RISC V Processor using Simulation and FPGA

Sudhir Bussa¹, Reuel Reuben², Rishabh Jain³, Shravan Kumar⁴

Assistant Professor, Department of Electronics & Telecommunication Engineering, Bharati Vidyapeeth (Deemed to be)

University College of Engineering, Pune, India¹

Student, Department of Electronics & Telecommunication Engineering, Bharati Vidyapeeth (Deemed to be) University College of

Engineering, Pune, India²

Student, Department of Electronics & Telecommunication Engineering, Bharati Vidyapeeth (Deemed to be) University College of

Engineering, Pune, India³

Student, Department of Electronics & Telecommunication Engineering, Bharati Vidyapeeth (Deemed to be) University College of

Engineering, Pune, India⁴

ABSTRACT: The recent fast growth of technology has simplified and improved our lives. This progress is mostly owing to the quick development of the silicon industry and the invention of different processors. With the silicon processor serving as the brain of any modern computer, we as humans are able to automate many monotonous tasks and are also able to do many other tasks which are usually very difficult for a human being. So, we thought why not design our own working processor. It used to be difficult to design a processor due to the vast quantities of money and resources necessary, as well as the fact that many corporations make their technology, innovations and research a trade secret. But due to the availability and development of free open-source design tools, open-source ISA like RISC-V and hardware description languages it is possible for undergraduate students to design their own processor just like developing a open-source software. The main objective of this project is to design a basic processor which will be able to execute about 31 different instructions out of the 47 instructions present in the base integer instructions set RV32I of RISC-V. We will be using an FPGA which is highly customizable for prototyping a chip and it's cheaper and very easy to deploy the processor on a FPGA than printing it on a die. The reason for this objective was that if in the near future, once our processor is developed properly, can be used for processing data for a small embedded system, and the hardware descriptions language can be extended easily for some particular applications. These applications can be anywhere from machine learning, vector acceleration to simple signal processing functions.

KEYWORDS - RISC, Pipelining, Verilog, Instruction Set

I. TECHNOLOGY

RISC-V

RISC -V also known as reduced instruction set architecture. It is an open-source Instruction set architecture (ISA). It was developed by the developers at University of California, Berkeley. It Is known as reduced instruction set architecture because it has very few instructions to be micro coded in order to make a functional processor which helps in the development of architecture that perform complex tasks optimally. The number of instructions in RISC is comparatively low compared to 3684 in x86 and 232 in ARM where RISC has just 47 instructions.Open-source ISA not only reduces hardware costs but also allows for scalability by allowing the community to include custom instructions that simplify software.The central processing unit (CPU), or brain of the computational device, is a true technological marvel.It appears to be a fascinating magic that only elite experts understand, but we will unravel this magic by designing our own RISC-V.Our project aims to create a basic RISC-V-based processor that can execute a few instructions from the RV32I base integer instruction set.The project described above can then be developed and extended for any specific application, such as AI, signal processing, and so on.



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

TOOLS USED

GTKWave:

GTKWave is a VCD waveform viewer that is built with the GTK library. This viewer supports VCD, LXT, LXT2, VZT, and GHW files, as well as standard Verilog signal dump formats. It also allows them to visualize. GTKWave was created for Linux, but it can also be used on Windows. It is a free and open-source EDA tool. It aids in wave simulation.

ICARUS VERILOG

The process of designing hardware begins with an abstract description of the intended device. Design entry languages such as schematic drawings and hardware description languages are used by the designer to enter the design.

Quartus prime

Intel Quartus Prime is Intel's programmable logic device design software; prior to Intel's acquisition of Altera, the tool was known as Altera Quartus Prime.Altera Quartus II was released earlier. Quartus Prime supports HDL design analysis and synthesis, allowing developers to compile their designs, analyse timing, examine RTL diagrams, simulate a design's response to various stimuli, and configure the target device with the programmer.Quartus Prime includes VHDL and Verilog implementations for hardware description, visual editing of logic circuits, and vector waveform simulation.

II. METHODOLOGY

First and foremost, we had to understand the specification of the processor that we were going to make using the RISC-V manual. After noting down the target specification, we started to choose which instruction we will be micro coding from the base integer instruction set. We then chose the Verilog hardware description language to microcode each of our instruction. We first divided the RISC-V processor into five stages of computing. We took each individual instruction and assigned the amount of work that each stage will be handling for each of the chosen instruction.

As a test case we first started with the micro coding of the ADD instruction and divided it into its processing flow for each stage. Then we verified the functionality of the ADD instruction through simulation and once the simulation was functionally correct, we verified if the particular ADD instruction worked on the DE1-SOC FPGA. Some problems arose when we started to test our single ADD Instruction on FPGA. One of the first problems we faced was that we had to reduce the frequency of the clock present on the FPGA from 50MHZ to 1HZ in order to see the results for multiple add instruction one at a time.

To reduce the clock frequency from 50MHZ to 5MHZ we had to use a built in Phased Locked Loop intellectual property present in Quartus Prime. And to further reduce the clock frequency from 5MHZ to 1HZ we designed our own clock divider circuit using Verilog. The second issue was to see the visual output of the particular microcode RISC-V instruction and for that we used 32 LED's connected to the general-purpose input output pins of the FPGA.

Once we had a flow for micro coding the instructions and testing the instructions through both simulations (i.e. software) and using a FPGA (i.e. hardware), our work became a little easier. We had to take individual instructions from the base integer instruction set (RV32I) and then start using the above-described flow for the ADD instruction for every other instruction. We have micro coded 31 instructions.For the verification of individual instructions, we have used an open source online RISC-V assembler which takes the input in the form of assembly language and then dumps out the machine code of the individual instructions which then forced into the memory of our processor using the testbench.

In the project implementation code, we have used two clocks as the input. The reason for using two clocks is to avoid race condition from register to register between two stages. The two clocks are equally space between each other with a 25 % duty cycle.

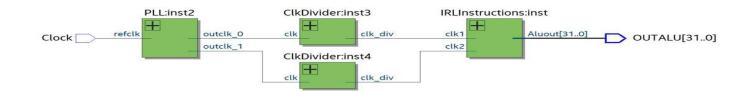


Figure 1- Clock Divider Block Diagram

International Journal of Innovative Research in Science, Engineering and Technology (IJIRSET)

| e-ISSN: 2319-8753, p-ISSN: 2347-6710| <u>www.ijirset.com</u> | Impact Factor: 7.569|



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

We have used a 32-bit program counter for pointing to the next instruction. Note: The program memory starts to execute a new instruction at every memory address which is divisible by 4 i.e., 0x0, 0x4, 0x8, 0x12, 0x16, etc. The reason for doing so is due to the byte addressability of the JAL instruction. The 32 x 32 bits general purpose registers are initialised in our code. We have also initialised 32 x 512 bits program memory for storing the instructions. The ALU has been instantiated with registers a and b directly connected to the ALU's input and the function of the ALU is controlled by the opcode fed into the ALU. A simple initial block is used to initiate all the memory locations to zero and the initial block is also used to force the instructions and values into the memory locations at the start of the code. All the registers and wires are initialised before the start of the main program.

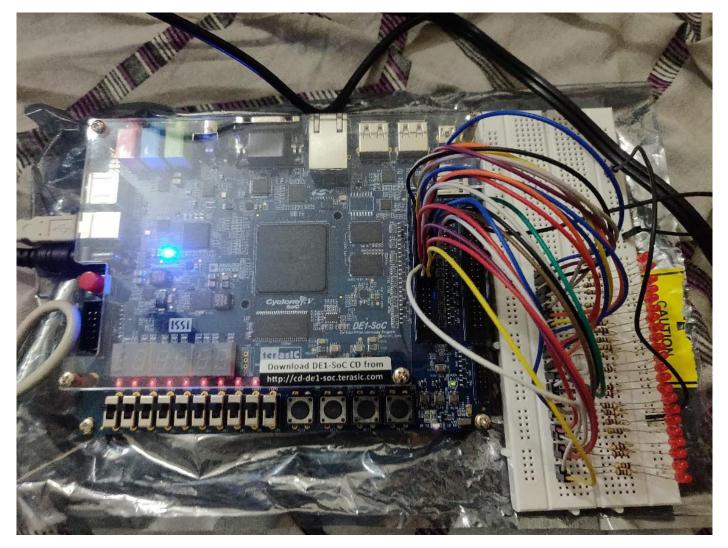


Figure 2- FPGA Testing

INSTRUCTION SET

It is a set of commands that is given to a CPU in machine language.Addressing modes, instructions, native data types, registers, memory architecture, interrupt and exception handling, and external I/O are all part of the instruction set. We've implemented 31 instruction set which are as follows.



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

R-Type	I-Type	Load-Type	Store-Type	U-Type	UJ-Type
SLT	ADDI	LB	SB	LUI	JAL
SLTU	SLTI	LH	SH	AUIPC	JALR
AND	SLTIU	LBU	SW		
OR	ANDI	LHU			
XOR	ORI				
SLL	XORI				
SRL	SLLI				
SUB	SRLI				
SRA	SRAI				
ADD					

PIPELINING

A pipeline in a pipelined processor has two ends, the input end and the output end. There are multiple stages/segments between these ends, with the output of one stage connected to the input of the next, and each stage performing a specific operation. The intermediate output between two stages is stored in interface registers. These interface registers are also referred to as latches or buffers. A common clock controls all stages of the pipeline as well as the interface registers.

Pipelining as 5 stages which help in queuing of multiple instructions in a single clock.

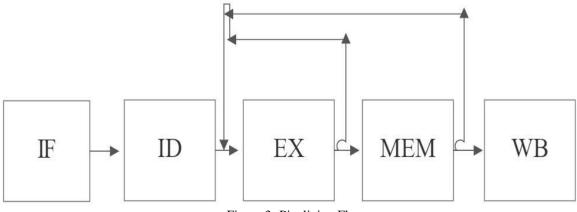


Figure 3- Pipelining Flow

III. **IMPLEMENTATION**

We've pipelined these instructions to ensure improved performance and higher operating frequencies and to reduce the complexity of designing the processor. The flow of pipelining for each type of instruction is shown below.



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

Instruction Fetch:

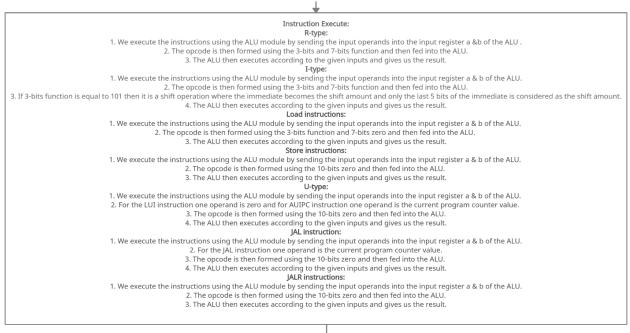
1. The instructions is fetched from the program memory (pc) using the program counter address. 2. The pc incremented by 4 and is stored in a new pc register.

	Instruction Decode:
	R-type:
Usir	ng the R-type opcode from the instruction register, we decode the instruction register of the R-type instructions. We get,
	1. 3-bits function
	2. 7-bits function
	3. destination address
	4. source address 1
	5. source address 2
	I-type:
Usir	ng the I-type opcode from the instruction register, we decode the instruction register of the I - type instructions. We get,
	1. 3-bits function
	2. 7-bits function
	3. destination address
	4. source address 1
	5. immediate data
	Load instructions:
Us	ing the Load opcode from the instruction register, we decode the instruction register of the Load instructions. We get,
	1. 3-bits function
	2. destination address
	3. source address 1
	5. immediate data
-	Store instructions:
Us	ing the Store opcode from the instruction register, we decode the instruction register of the Store instructions. We get,
	1. 3-bits function
	2. source address 1
	3. source address 2
	5. immediate data
	U-type:
Jsin	ng the U-type opcode from the instruction register, we decode the instruction register of the U-type instructions. We get,
	1. destination register
	2. immediate data
	JAL instruction:
	Using the JAL opcode from the instruction register, we decode the instruction register of the JAL instruction. We get,
	1. destination register
	2. immediate data
	JALR instructions:
U	Ising the JALR opcode from the instruction register, we decode the instruction register of the JALR instruction. We get,
	1. destination register
	2. source address 1
	3. immediate data



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |



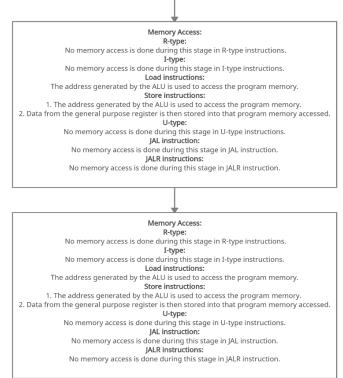


Figure 4- Instruction Set Flow

||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

IV. RESULTS AND SIMULATIONS

Each and every instruction was verified and tested with multiple test cases which involved multiple register level state changes which showed accurate functioning of the instruction which we had micro coded before it became a part of a larger system. Using an online RISC-V assembler we convert the assembly language test code we write as the stimulus for our processor into machine code which is understandable by our processor. Using the testbench we force machine code into the program memory of the RISC-V processor. This is done using the first initial block present in the testbench. We also initiate our program counter to zero using another initial block. We also configure our two clocks to run at the correct intervals of time so that we can avoid any timing violations in our processor. The last initial block is used to dump out the. VCD file used for viewing the simulations in GTKWave. It dumps out all the variables i.e. all the registers, wires and nets used in our design.

Simulation of R Type Instructions

Assembly Code Used for Testing and Simulations:

addi x1 , x0 , 1000 /* x1 = 1000 0x3E8 addi x2 , x0 , 2000 /* x2 = 2000 0x7D0 add x3 , x1 , x2 /* x3 = 3000 0xBB8 sub x7 , x3 , x1 /* x7 = 2000 0x7D0 addi x6 , x0 , -500 /* x6 = -500 -0x1F4 slt x4 , x6 , x3 /* x4 = 1 0x1 sltu x20 , x6 , x3 /* x20 = 0 0x0 and x15 , x3 , x7 /* x15 = 912 0x390 or x14 , x3 , x7 /* x15 = 912 0x390 or x14 , x3 , x7 /* x16 = 3176 0xC68 addi x5 , x0 , 12 /* x5 = 12 0xC sll x21 , x4 , x5 /* x21 = 4096 0x1000 addi x5 , x0 , 6 /* x5 = 6 0x6 srl x20 , x21 , x5 /* x20 = 64 0x40 addi x5 , x0 , 7 /* x5 = 16 0x10 sra x22 , x6 , x5 /* x22 = -4 0xFFFC

Time		6550	0 ps	1	31 ns	1	196600) ps	2621	00 ps		3276	00 ps		39320	ð ps		458700 ps	
clk1=																			th i
clk2=																			
										40					24				
pc[31:0] =)	+ 0	XXX		4 XXX		8 444		12 XXX		16	XXX	4	0 XX		24	XXX	28	XXX	
IF_ID_IR[31:0] =;	XXXXXXXX	3E800093	XXXXXXXX	7D000113 x	OOXXXXX	002081B3 xx	xxxxx	x 4011838	3 xxxxxxxx		E0C00313	XXXXXXXXX	003	32233 xxxxxxxx		00333A33	XXXXXXXXX	0071F7	B3 xxxxxxx
ID_EX_A[31:0] =0	XXX	0				1000		36	00		0			-500					000
ID_EX_B[31:0] =>	XXX					2000		16	00		XXX			3000)2	000
a[31:0] =0	XXX		0			10	00		3000			0		- 500					000
b[31:0] =]	XXX		1000	2	000				1000			-500		3000					2000
opcode[9:0] =0	ххх		000						100			000		002			003		007
ID_EX_IMM[31:0] =]	XXX	1000		2000		XXX					-500			XXX					
EX_MEM_ALUOUT[31:0] =:	XXX		1000		2000	3	3000		2000			-500		1			0		912
<pre>MEM_WB_ALUOUT[31:0] =</pre>	XXX		1000		2000		30	00	200	0		- 500		1			0		91

Figure 4.1 R-Type Result



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

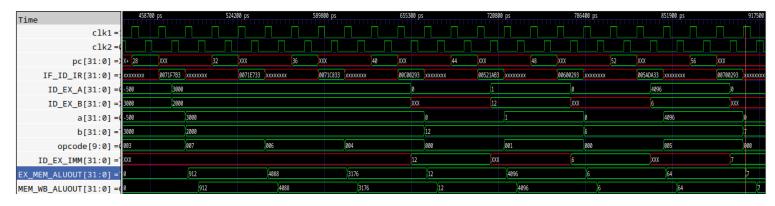
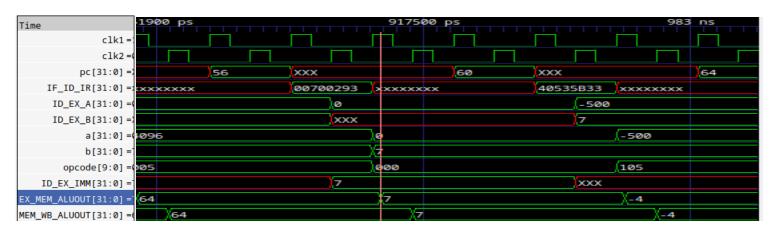


Figure 4.2 R-Type Result





Note: Zoom in for clarity

Simulation of I Type Instructions

Assembly Code Used for Testing and Simulations:

addi x1 , x0 , 1000 /* x1 = 1000 0x3E8 addi x2 , x0 , 2000 /* x2 = 2000 0x7D0 add x3 , x1 , x2 /* x3 = 3000 0xBB8 addi x6 , x0 , -500 /* x6 = -500 -0x1F4 slti x4 , x6 , 50 /* x4 = 1 0x1 sltiu x20 , x6 , 300 /* x20 = 0 0x0 andi x15 , x3 , 2029 /* x15 = 936 0x3a8 ori x14 , x3 , 965 /* x14 = 3069 0xBFD xori x16 , x3 , 1442 /* x16 = 3610 0xE1A slli x21 , x4 , 12 /* x21 = 4096 0x1000 srli x20 , x21 , 6 /* x20 = 64 0x40



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

srai x22 , x6 , 7 /* x22 = -4 0xFFFC

Time			100	ns			200	ns			300	ns			40) ns			
clk1=0				<u></u>			Л										<u></u>		<u>F</u>
clk2=0																			
pc[31:0] =)	+ 0	XXX	4	XXX	8	XXX		12 XXX		16	XXX		20	XXX	24	XXX		28	XXX
IF_ID_IR[31:0] =0	XXXXXXXX	3E800093 xxxx	CXXXX	7D000113	XXXXXXXX	00208183	XXXXXXXX	EØC	00313	XXXXXXXX	0323221	3 xxxxxxxx		12C33A13	XXXXXXXXX	7ED1F793	XXXXXXXX		3C51E713
ID_EX_A[31:0] =-	XXX	0				1000			0			00				3000			
ID_EX_B[31:0] =>	XXX					2000			XXX										
ID_EX_IMM[31:0] =:	XXX	1000		2000		XXX			-500		56			300		2029			965
a[31:0] =-	XXX	0					1000			0		-500					3000		
b[31:0] =	XXX	1000)		2000					-500		50			300		2029		
opcode[9:0] =:	XXX	000										002			003		807		
EX_MEM_ALUOUT[31:0] =	XXX	100	00		2000		3000			-500)1			0		936		
<pre>MEM_WB_ALUOUT[31:0] =</pre>	XXX		1000		2000		300	0		-500)1			0		936		

Figure 5.1 – I-Tpe Result

Time			500	ns		600	ns		700 ns	5		
clk1=												
clk2=												
pc[31:0] =	XXX	28 XXX		32 XXX	36	XXX	40	XXX	44	XXX	48	XXX
IF_ID_IR[31:0] =	7ED+ x	XXXXXXXX 3C51E7:	.3 xxxxxxx	x 5A21C813	XXXXXXXX	00C21A93	XXXXXXX	006ADA13 xxxxxxx	x	40735B13 x	XXXXXXX	0000
ID_EX_A[31:0] =	3000					1		4096		- 500		
ID_EX_B[31:0] =	XXX											
ID_EX_IMM[31:0] =	2029)9	55	1442		12		6		1031		
a[31:0] =	-500 3	8000					1	4096)!	500	
b[31:0] =	300 2	2029	965		1442		12	6		7		
opcode[9:0] =	003 0	007	006		004		001	005		10	ð5	
<pre>EX_MEM_ALUOUT[31:0] =</pre>	0	936	3069		3610		4096	64		X	-4	
<pre>MEM_WB_ALUOUT[31:0] =</pre>	0	936	30	69	3610		4096	64	ļ		-4	
				Figure 5.2	– I-Type Res	sult						

Simulation of Load, Store and U Type Instructions

Assembly Code Used for Testing and Simulations:

```
addi x3 , x0 , 19 /* x1 = 19
lui x5 , 0x34aff /* regbank[5] = 34AFF000
addi x5 , x5 , 0xa3 /* regbank[5] = 34AFF0A3
sb x5 , 3(x3) /* rom[22] = 000000A3
lui x10 , 0x8765F /* regbank[10] = 8765F000
sh x10 , 6(x3) /* rom[25] = 0000F000
auipc x24 , 0xffc32 /* regbank[24] = FFC32018
sw x24 , 16(x3) /* rom[35] = FFC32018
lb x20 , 16(x3) /* regbank[20] = 18
lbu x21 , 16(x3) /* regbank[21] = 18
lh x18 , 6(x3) /* regbank[18] = FFFFF000
lhu x19 , 6(x3) /* regbank[19] = 0000F000
lw x30 , 16(x3) /* regbank[30] = FFC32018
```



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

Time		100	ns	200	ns	300 ns	40	ns	500
clk1=0									
clk2 =:									
pc[31:0] =)	0	XXX 4	XXX 8	XXX	12 XXX	16 XXX 20	XXX 24	XXX	8 XXX
IF_ID_IR[31:0] =>	XXXXXXXX	01300193 xxxxxxxx	34AFF2B7 xxxxxxxx	ØA328293 xxxxxxx	o 005181A3 xxxxxxxx	8765F537 xxxxxxxx	00A19323 xxxxxxxx	FFC32C17 xxxxxxxxx	0181A823 xxxxxxx
ID_EX_A[31:0] =0	XXXXXXXX	0000000		34AFF000	00000013	0000000	00000013	00000000	00000013
ID_EX_IMM[31:0] =	XXXXXXXX	00000013	34AFF000	000000A3	0000003	8765F000	0000006	FFC32000	00000010
ID_EX_RS2[31:0] =:	XXXXXXXX				34AFF0A3	XXXXXXXX	8765F000	XXXXXXXX	FFC32018
a[31:0]=	XXXXXXXX	0000000		34AFF000	00000013	0000000	00000013	FFC32000	00000013
b[31:0]=0	XXXXXXXX	00000013	34AFF000	000000A3	8 0000003	8765F000	00000006	00000018	0000010
opcode[9:0] =0	XXX	000							
EX_MEM_ALUOUT[31:0] =	XXXXXXXX	00000013	34AFF000	34AFF0A	43 00000016	8765F000	00000019	FFC32018	0000002
MEM_WB_ALUOUT[31:0] =	XXXXXXXX	00000013	34AFF000	<u></u> 34A	AFF0A3	8765F000		FFC320	18

Figure 6..1- Load Store -Type Result

Time	40	ns		500	ns			(600 n	15		7	80 ns				800	ns	
clk1=0			ĹĹ				ſĹĹſ				ſĹ.					\square	Л		
clk2=:									Γ										
pc[31:0] =)	XXX 24	XXX	28	XXX	32	XXX	36	XXX		40	XXX	44	XXX		48	XXX		52	XXX
IF_ID_IR[31:0] =:	XXXXXXXX	FFC32C17 xxxx	oxxx	0181A823 xxxxxxx	(01018A03	XXXXXXXX	01010	A83	XXXXXXXX	00619903	XXXXXXXX	0061D983	XXXXXXXX		0101AF03	XXXXXXXX		00000000 xxxxxxx
ID_EX_A[31:0] =	00000013	0000000		00000013															
ID_EX_IMM[31:0] =	00000006	FFC32000		00000010							000	00006				0000	0010		
ID_EX_RS2[31:0] =>	8765F000	XXXXXXXX		FFC32018		XXXXX	XXXX												
a[31:0] =	00000013	FFC3	000	0000001	}														
b[31:0]=0	00000006	0000	018	0000001)							00000006					00000010		
opcode[9:0] =0	000																		
EX_MEM_ALUOUT[31:0] =	00000019	,FFC	2018	000000	23							00000019					0000002	3	
<pre>MEM_WB_ALUOUT[31:0] =F</pre>	8765F000		FFC32018				0000001	8				FFFFF000		0000F	6000		FFC	32018	

Figure 6..2- Load Store -Type Result

Simulation of JAL and JALR Instructions

Assembly Code Used for Testing and Simulations of JAL Instruction:

addi x4,x3,2 addi x4,x3,2 addi x4,x3,2 addi x4,x3,2 loop: addi x5,x4,1 addi x7,x5,1000



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

	addi addi addi	x7,x5,100 x7,x5,100 x7,x5,100 x7,x5,100	0 0 0						
Signals	addi Waves	x7,x5,100	0						
Time		100 ns 200	ns 300	ns 400	0 ns 50	0 ns 600	ns 700 ns	800 ns	900 ns 1 i
clk2=									
clk1=									
IF_ID_IR[31:0] =>	*****	00+ xxxxxxx 00+ xxxxx	xxx 00+ xxxxxxxx 00+ x	xxxxxxx 00+ xxxxxxx	3E+ xxxxxxxx 3E+ xxxx	xxxx 3E+ xxxxxxxx 3E+	xxxxxxxx 3E+ xxxxxxxx 3E+	xxxxxxxx FE+ xxxxxxxx 00+ xxx	xxxxx 3E+ xxxxxxxx 3E+ xxxxxxxx
IF_ID_NPC[31:0] =)	XXX	4 XXX 8 XXX	12 XXX 16 X	XX 20 XXX .	24 XXX 28 XXX	32 XXX 36	XXX 40 XXX 44	XXX 48 XXX 20 XXX	24 XXX 28 XXX
ID_EX_A[31:0] =	*****	00000000		0000002	0000003			xxxxxxxx 00000	002 0000003
ID_EX_NPC[31:0] =)	XXX	4 XXX 8 XXX	12 XXX 16	XXX 20 XXX	24 XXX 28 XX	X <u>32</u> XXX 36	XXX 40 XXX 44	XXX 48 XXX 20 X	XX 24 XXX 28 XXX
a[31:0]=4	XXX	0		2	3			44 2	(3
b[31:0] =-	XXX	2		1	1000			-28 1	1000
EX_MEM_ALUOUT[31:0] =	XXX	2		3	1003			16 3	1003
MEM_WB_ALUOUT[31:0]=	XXX	2		3	1003			(16)3	1003
pc[31:0] =	XXX 0	XXX 4 XXX 8	XXX 12 XXX	16 XXX 20	XXX 24 XXX	28 XXX 32 XXX	36 XXX 40 XXX	44 XXX 16 XXX	20 XXX 24 XXX 28 X

jal x1, loop

Figure 7.1 JAL & JALR Result

Assembly Code Used for Testing and Simulations of JALR Instruction:

addi x4,x3,2 addi x4,x3,2 addi x4,x3,2 addi x4,x3,2 addi x5,x4,1 addi x7,x5,1000 addi x7,x5,1000 addi x7,x5,1000 addi x7,x5,1000 addi x7,x5,1000 addi x7,x5,1000 jalr x20,x15,8



||Volume 11, Issue 7, July 2022||

| DOI:10.15680/IJIRSET.2022.1107174 |

Signals	Waves																															
Time		100 ns			200	ns		30	0 ns			400 n	S		5	00 n	S		600	ns		70	10 ns			800	ns		90	0 ns		1
clk1=																																
c1k2 =(
IF_ID_IR[31:0] =:	*****	00+	XXXXXX)	x x 00+	*****	XX 00+	XXXXX	XXX 00+	*****	xx 00+	*****	x 3E+	XXXX	xxxx 3E	+ XXX	xxx)	x 3E+	*****	xx 3E+	*****	x 3E+	XXXXXXXX	3E+	XXXXXX)	(X 00+	xxxx	xxx 00+	××××××	xx 3E+	xxxxxx	x 3E+	XXXXXXX
IF_ID_NPC[31:0] =	XXX	4	XXX	8	XXX	12	XXX	16	XXX	20	XXX	24	XXX	28	3 XXX		32	XXX	36	XXX	40	XXX	44	XXX	48	XXX	20	XXX	24	XXX	28	(XX
ID_EX_A[31:0] =	XXX	0								2			3												8		2		3			
ID_EX_B[31:0] =	XXX																															
ID_EX_NPC[31:0] =)	XXX	4	XXX	8	XXX	1	2 XX)	(16	XXX	2	XXX		24 X)	(X	28 X	XX	32	2 XXX	36	XXX	40	XXX	44	XXX	48	XX	2	20 XXX	24	XXX	28	XXX
a[31:0]=	XXX		0								2		3													8		2		3		
b[31:0]=	XXX		2								1		1000													8		1		1000		
EX_MEM_ALUOUT[31:0] =:	XXX		2								3		100	3												16		3		1003		
MEM_WB_ALUOUT[31:0] =:	XXX		2								3		10	003												16		3		1003		
pc[31:0] =	XXX Ø	XXX	4	XXX	8	ХХХ		12 XXX	16	XXX	20	XX)	(24 X)	XX	28	XXX	32	XXX	36	XXX	40	XXX	44	XXX		6 XXX	20	XXX	24	XXX	28

Figure 7.2 JAL & JALR Result

V. CONCLUSION

In this research paper we successfully implemented the working and structure of different types of instructions used in our RISC-V processor. We were able to successfully complete the micro coding of 31 different instructions using Verilog HDL to form our RISC-V processor. We have tested the functionality and verified the working of our RISC-V through proper simulations and test cases. We have also successfully deployed our processor design on a FPGA.

RISC-V processors being open source & containing very few instructions set to be micro coded in order to form a functional processor thus any application specification need can be met with shorter development time and reduced power consumption due to simpler hardware implementation.

Due to the changing trends in the semiconductor development which is shifting from a complex approach to a simpler architecture and more innovative architecture which will utilize the increasing transistor budget.

Thus this project can be further developed and extended for any specific application such as AI, signal processing, etc.

REFERENCES

- "SHAKTI Processors: An Open-Source Hardware Initiative," N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, V. Kamakoti, 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), Kolkata, India, January 2016.
- [2]. "RISC-V Offers Simple, Modular ISA," D. Kanter, The Linley Group MICROPROCESSOR report, March 2016.
- [3]. "Design of the RISC-V Instruction Set Architecture," A. Waterman, Technical Report No. UCB/EECS-2016-1, EECS Department, University of California, Berkeley, January 2016.
- [4]. "The RISC-V Instruction Set," Andrew Waterman, Yunsup Lee, Rimas Avizienis, Henry Cook, David Patterson, Krste Asanovic, Poster at the Symposium on High Performance Chips (HotChips-25), Stanford, CA, August 2013.
- [5]. "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA," Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.
- [6]. The RISC-V Instruction Set Manual, Andrew Waterman, Krste Asanovi, SiFive Inc. CS Division, EECS Department, University of California, Berkeley.
- [7]. RISC-V ASSEMBLY LANGUAGE Programmer Manual Part I, SHAKTI Development Team at IIT Madras '20.





Impact Factor: 8.118





INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

IN SCIENCE | ENGINEERING | TECHNOLOGY





www.ijirset.com