RISC-V Hardware Synthesizable Processor Design Test and Verification Using User-Friendly Desktop Application

Hyogeun An
Hanbat National University, Daejeon, South Korea. E-mail: ahnhyogean@gmail.com

Sudong Kang
Hanbat National University, Daejeon, South Korea. E-mail: dongdonge9555@gmail.com

Guard Kanda
Hanbat National University, Daejeon, South Korea. E-mail: guardkanda@gmail.com

Kwangki Ryoo*
Hanbat National University, Daejeon, South Korea. E-mail: kkryoo@gmail.com

Received September 26, 2021; Accepted December 19, 2021
ISSN: 1735-188X
DOI: 10.14704/WEB/V19I1/WEB19305

Abstract

Although the RISC-V ISA has not been around for long, it is a processor architecture that has been highlighted by many businesses and individuals for its low-cost and rapid pace of development. They are open-source-synthesizable hardware processors with minimal functionality that is ideal for current IoT applications involving simple sensors and actuator controls. Due to some qualities of hardware, they can operate in areas where software programs and applications cannot be used whereas, these software programs that run on such hardware equally help in understanding how hardware operates. This paper, therefore, proposes and discusses the design, implementation, and internal verification and test platform for a Reduced Instruction Set Code-V's (RISC-V) Instruction Set Architecture (ISA), using an interactive desktop program for a 32-bit single-cycle processor. This paper developed a system that functions as interactive assistance to RISC-V's ISA design and debugger using a more user-friendly desktop UI application. The uniqueness of this design is the flexibility of testing and debugging that is possible through either the software interface or through hardware peripherals such as Universal Asynchronous Receiver/Transmitter (UART) protocols in FPGA or even both. These peripherals allow users to view the contents of the register files and RAM being utilized by the implemented processor on the FPGA. The proposed desktop User Interface program monitors and controls the sequential processing and states of a 32-bit single-cycle RISC-V processor's operation on an FPGA. Contents of the proposed processor's registers and memory are displayed alongside other temporal or internal data. Internal components such as Program Counters (PC), Random Access Memory (RAM), are displayed all through the proposed User Interface (UI) program and also through various peripherals on the FPGA board. The software program is implemented using C# programing language through Microsoft Visual Studio 2019 Integrated Development Environment (IDE). The proposed hardware synthesizable processor core is implemented using Verilog Hardware Description Language (HDL) and synthesized with Xilinx Integrated Synthesis Environment (ISE) version 14.7. The proposed processor and its corresponding hardware test modules occupy 6476 Look-Up-Tables (LUT) and operate at a maximum frequency of 49MHz and its operation is verified on a Field Programmable Gate Array (FPGA). The proposed processor and its test platform can serve as a good educational tool as well as a help for processor design engineers both experienced and beginners.

Keywords

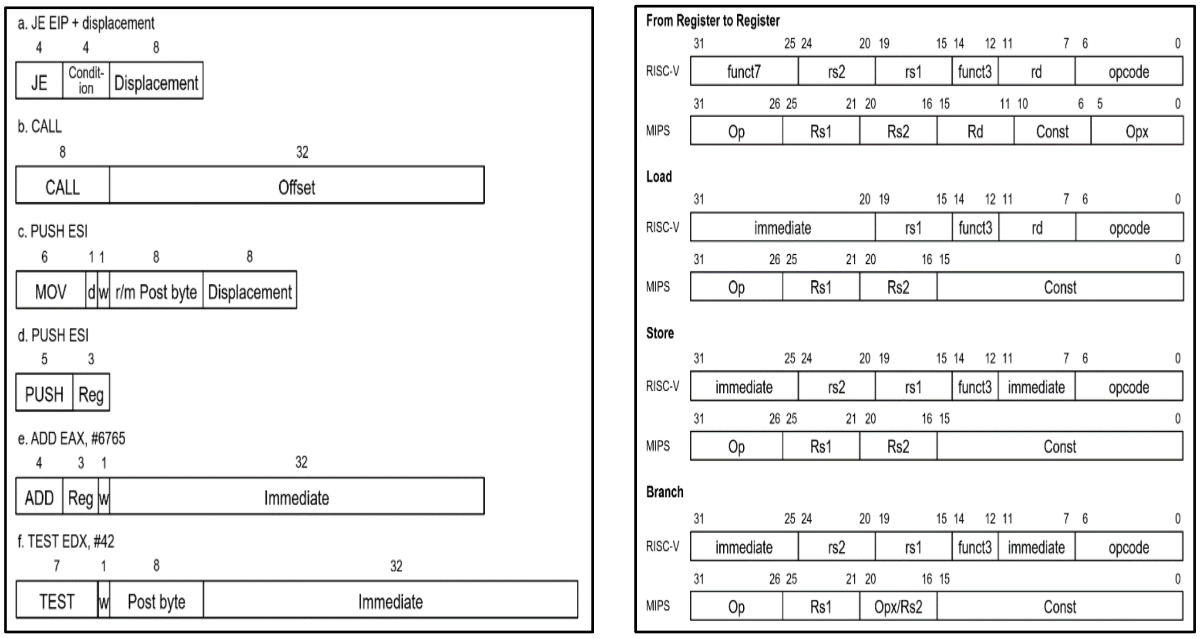
RISC-V, Desktop Application, FPGA, Cross Verification, ISA.

Introduction

The Central Processing Unit (CPU) handles various forms of arithmetic and logical instruction execution and memory access. The unique set of instructions that each processor group has are called ISA. ISA is a specific set of instructions that can only be compiled by a dedicated compiler and then converted into machine code. The current Information Technology (IT) market is demanding lower power and higher performance embedded systems (Ryu, Lee, 2021). Examples are research on using Internet of Things (IoT) low-cost Mixed OXide (MOX) gas sensors technology (Bruno et al., 2021) and Ultra-low Power Embedded Unsupervised Learning Smart Sensor technology (Gies et al., 2021). Among embedded systems, processors are essential components. However, commercial ISAs (CPU, RISC-V, 2020)(Degirmen, 2019) of companies with unique processor architectures such as Advanced RISC Machines (ARM), x86 with ISA products are either patent-pending, expensive to purchase a license, which makes them not available to researchers and hobbyists. Although such licenses prevent designers from implementing proprietary ISA, it also limits the use of only the processor cores provided by these companies. These restrictions help prevent other companies from spying on a company. Not all, it also helps prevent competition, plagiarism, and increases reliability (Swarup, 2014). However, these advantages equally hinder the development of architectural performance and security. To address these shortcomings, the University of California, Berkeley developed a new ISA design known as RISC-V ISA (Andrew, 2011), which makes it easier for researchers and hobbyists to use or implement a processor without having to purchase commercial licenses. RISC-V is a Reduces Instruction Set Code (RISC)-based processor architecture. Examples of RISC-based processor architectures include ARM and Microprocessor without

Interlocked Pipeline Stages (MIPS). The architecture contrasted with the RISC-based processor is a Complex Instruction Set Computer (CISC)-based processor. Examples are x86 and x64, which can process many and complex tasks. CISC also has a lot of instruction, so it is highly productive for micro developers. In addition, if the instruction supported at the hardware level is supported by the subsequent processor because there is no concept of optimal instruction, it is not necessary to pay much attention to the compatibility of the instruction, so it is possible to continue to develop rapid design changes and innovative architectures. The structure of the RISC architecture by International Business Machine's (IBM) Institute of Yorktown, New York, USA proved that about 20% of the instructions in the computer handle more than 80% of the total work. By doing so, the concept of RISC was first raised in 1974, and since then, the XT model of IBM PC, which was announced in 1980, has increased to the market share level of RISC. The term RISC itself is named by David Peterson, a professor at the University of California Berkeley. RISC is a simple processor that minimizes instructions and requires compatibility problems, software complexity, and size to optimize compilers. However, even if all companies developed and used RISC-based processors to create products and user licenses, the companies had a lot of development costs and time along with a market that is already dominated by the CISC-based processors. Thus, despite these advantages, RISC has not been noticed. At the time, the RISC-based processor was opened to the general public by opensource licensing in 2010 at the University of UC Berkeley, to improve the functionality and performance of the RISC-V. RISC-V has a very systematic and efficient instruction set. As shown in Figure 1(a), the CISC instruction format of the x86 processor has different instruction sizes, making the structure not constant and having many instructions. Also, pipeline bubbles occur when pipelined, which reduces processing efficiency. Figure 1(b) on the contrary, shows a comparison of some instruction set between the RISC-based MIPS architecture most mounted on embedded boards and the RISC-based RISC-V.

As shown in Figure 1(b), all RISC-based instruction sets are fixed length, however, the position of the OPCODE, which introduces the biggest difference, is different. The OPCODE of MIPS is located in the upper bits, while that of RISC-V is in the lower bits. This position of the OPCODE in RISC-V implies that is more variable because the other sizes of the fields can easily be adjusted to accommodate an increased bus width. For this reason, the RISC-V is more compatible as it can easily fit in any generationʹs design. As mentioned above, the RISC-V instruction set is emerging to be a RISC-based architecture that is more efficiently designed than other ISA architectures including the RISC.



(a) (b)

Figure 1. (a) Typical x86 Instruction Format (David, 2017), (b). Instruction Formats of RISC-V and MIPS (David, 2017)

As a sign of endorsement and efficiency, Samsung electronics recently announced its adoption of RISC-V as a processor for their 5G chip and Complementary Metal Oxide Semiconductor (CMOS) image sensors (Samsung, RISC-V, 2019). Other open-source processors implementing RISC-V ISA include Freedom 300 (SiFive, platform, 2016) and PICORV32 (PicoRV32, 2019). These processors are relatively complex, with pipelined structures that result in high throughput compared to low hardware area (Dennis , Ryoo 2019). To contribute to reducing the complexity and increasing the understanding of RISC-V processors, this paper proposes and implements the functionality of the RISC-V ISA processor alongside using hardware peripherals of the FPGA board to monitor the internal operation of the 32-bit single-cycle RISC-V ISA core. Additionally, through the UART protocol, a proposed desktop UI program is also designed to control and validate the RISC-V ISA core on the desktop. The contributions of this paper are as follows:

1. A description of controls and implementation of a 32-bit, single-cycle RISC-V ISA core using Verilog HDL.
2. Proposal and design of desktop UI program to control and validate the implementation of processor cores on desktops. This is a necessary feature to provide an easy way to input instruction and observe the processor's data from components such as register files, RAM, PCs during operation through the personal

computer

3. The design of the processor to execute instruction data sent from the Desktop UI program in real-time.
4. The Implementation of data transmission and reception between UI and FPGA using the UART protocol.

The rest of this paper is organized as follows: Section 2 describes some related papers. Section 3 presents the functionality of the implemented RISC-V processor core. Section 4 describes serial communication between the proposed processor core and the proposed UI. While section 5 describes the desktop UI program. Section 6 describes hardware resources utilization and experiment using the desktop UI program. Finally, conclusions and future works of the paper are discussed in Section 7.

Related Work

There have been several academic-based research regarding the design of open-source RISC-V processors since its introduction. Through the continual analysis and review of some proposed architectures, the RISC-V community has seen a drastic improvement in both design and use. Concerning processor verification, which is one of the focuses of this research, (Oleksiak et al., 2019) in their paper presented a test and verification methods for RISC-V ISA which had similar ideas on the verification as compared to the research of this paper. The authors of (Oleksiak et al., 2019) designed an emulator that accepted a Design Under Test (DUT) processor module and compared its operations by way of instruction execution to a golden model that exists in the emulator. Although this design provides information on the type of error and the expected result of each instruction execution, it did not permit a step-by-step trace of the executed instruction to localize the exact source of the error. Not all, the proposed verification module had also not been verified on FPGA. It is therefore difficult for designers and developers to understand and use such a system. In addition to this design, a similar design by (Schiavone et al., 2018) utilized an open-source evolutionary optimizer to create functional test programs that constantly improve verification test sets. However, like the design in (Oleksiak et al., 2019), the design lacked a graphical user interface which limited the flexibility of testing and increased the time of debugging.

Don et. al. in their paper proposed a test and verification system that used a Text-LCD hardware peripheral component to display the PC value and other computation results. The design of (Don , 2017) is the most similar to the design of this paper because, unlike the previous two designs mentioned, it utilized an instruction memory to hold instruction data. However, it only included a single peripheral component and did not have a user-friendly graphical interface. Although these existing designs, in one way or the other, enabled the test and verification of RISC-V processors, they lacked in flexibility such as the ability to step-by-step trace an error, the flexibility of building and inputting addition instructions into memory, and the co-usage of a user-friendly graphical interface to interact with the internal components of the processor under test. Based on these limitations of the aforementioned designs, this paper proposes a test and verification system that incorporates the aforementioned flexibility by including both hardware peripheral test components and a software user interface for an enhanced and easy-to-use system that makes the understanding and application of RISC-V architecture simpler.

Proposed RISC-V Processor Core Implementation

The proposed implementation reads and processes instruction data from memory. The implemented processor is a single cycle architecture which implies that the designed processor executes an instruction in one cycle, spans one rising edge and two falling edges of the clock. There are two operation modes proposed in this research. First is the continuous real-time execution of instruction data and the second is a step-by-step execution of instruction data through a single pulse clock. The two operation modes make the verification and testing process faster. The reason for using two modes is to compensate for the shortcomings of either mode. By compensating for the shortcomings, faster verification or testing can be performed. The main components of the proposed processor are the core processor block and the instruction data memory or RAM.

RAM

RAM is the main memory device that stores processors working data or machine code. It allows data to be read and written to any memory location. The input ports of the implemented RAM module are ***clk***, ***clka***, ***addra***, ***web***, ***dinb***, ***load_data***. The output ports of the RAM module are ***doutbw***, ***douta***, ***doutb***, ***load_add***. RAM modules have true dual-port block memory of 32x2048 and two additional registers. The two registers are controlled by the main clock of the FPGA board and are connected to the output port of the block memory to output them based on the rising edge of the clock. Port ***addra*** is the address to which data is read or written, ***web*** is write-enable signal and ***dinb*** is data port. ***load_data*** is a port for load instruction and has a built-in circuit that immediately uses the output data to perform a load instruction as the address of the block RAM to output that value. Figure 2 shows the architecture of RAM.

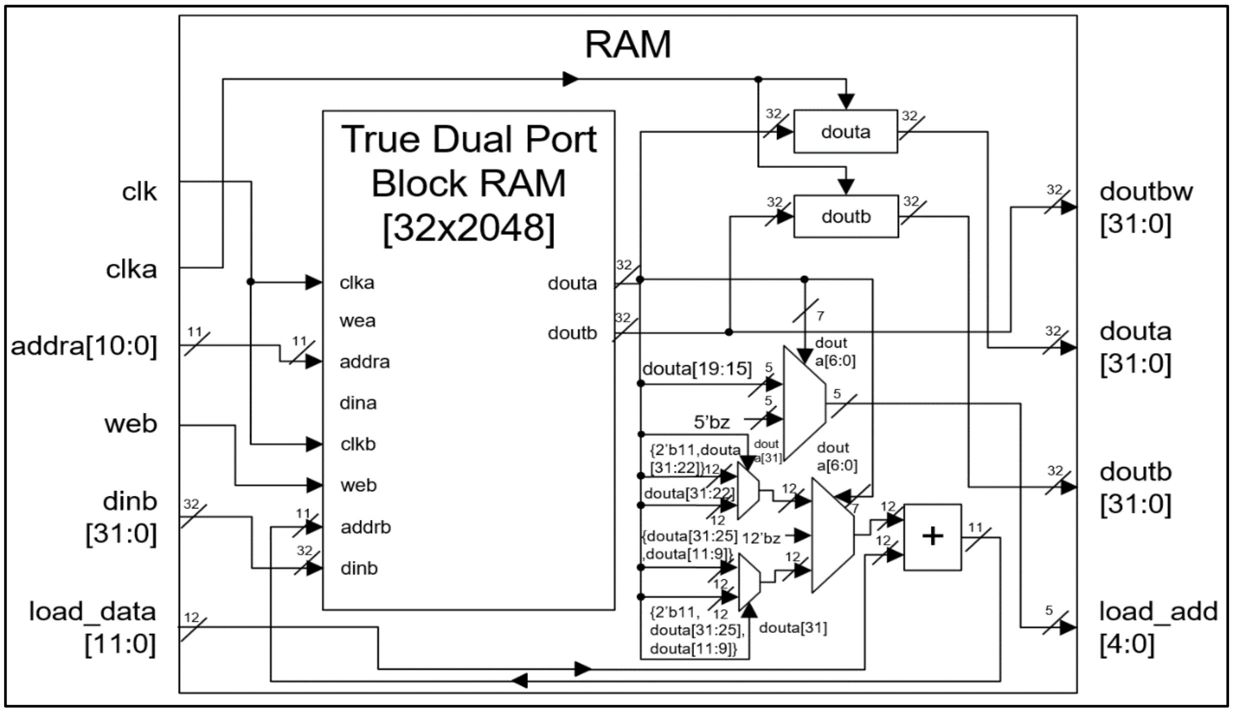


Figure 2. The architecture of RAM

Implemented Processor

The processor block is a hardware component that responds to and processes the basic instructions that drive a computer or any hardware circuit. The implemented processor is composed of a PC, Register File, Data Modification module, Arithmetic and Logic Unit (ALU), and Control and Status Registers. The processor module contains a controller that separates the instruction data received from RAM and executes the appropriate actions that match the code while it calculates the value of the PC. The input ports of the processor module are ***clk***, ***rst***, ***inst***, ***dataout***, ***doutbw***, ***load_add***. The output ports of the processor module are the ***memwro***, ***datain***, ***in***, ***load_data***. The ***inst*** port is the port used to receive instruction data from RAM and the ***dataout*** port to obtain the required values from the load instruction. The ***load_add*** port is the address of the register for executing a store instruction. The ***memwro*** port and ***datain*** port are write enabled signals and data respectively for performing store instruction. The ***in*** port serves as a load of the next instruction with PC values. Figure 3 shows the architecture of the processor.

Register File

A Register File is a memory device that temporarily stores data calculated by the processor. The input ports of the register file are the ***clk***, ***rst***, ***datad***, ***ind***, ***ina***, ***inb***, ***regw***, ***load_add***. The output ports of the register file are ***outa***, ***outb***, ***load_data***. This module embeds a 32x32 array register and each register is clock-controlled. In addition, the ***datad***, ***ind***, ***regw*** signals are used to control the register circuit respectively for data to be stored, address to store data, and register write enable signals. The ***ina***, ***inb*** inputs can be used to obtain the values of the ***outa***, ***outb*** of that address, and ***load_add*** and ***load_data*** to perform the load instruction functions. Figure 4 shows the proposed architecture of the Register File.

Program Counter (PC)

The PC is a register within the processor that has the address or location of the next instruction to be executed. The input ports of the PC module are the ***clk***, ***rst***, and ***in***. The output port of the PC module is ***out***. The PC module calculates the PC value on the falling edge. Therefore, it is necessary to output values on the rising edge. The main clk of the FPGA board was inserted to resolve this. Figure 5 shows the proposed architecture of the PC implementation.

Arithmetic Logic Unit (ALU)

An ALU module is a module that is responsible for all the arithmetic and logical operations to be performed on the data received. Such operations include **AND**, **OR**, **ADD**, **SHIFT**, **SUB**, **NOT**, **BLT**, **BGE**, **NOR**. The input ports of the ALU module are the ***Aina***, ***Ainb***, and ***alucon***. The output port of the ALU module is ***Aout***. It has the function of performing the desired computation based on the ***alucon*** signal. The arithmetic or logical operations are performed using values of ***Aina*** and ***Ainb*** from the data modification as operands and, ***Aout*** as the result of the operation. Figure 6 shows the implemented architecture of the ALU.

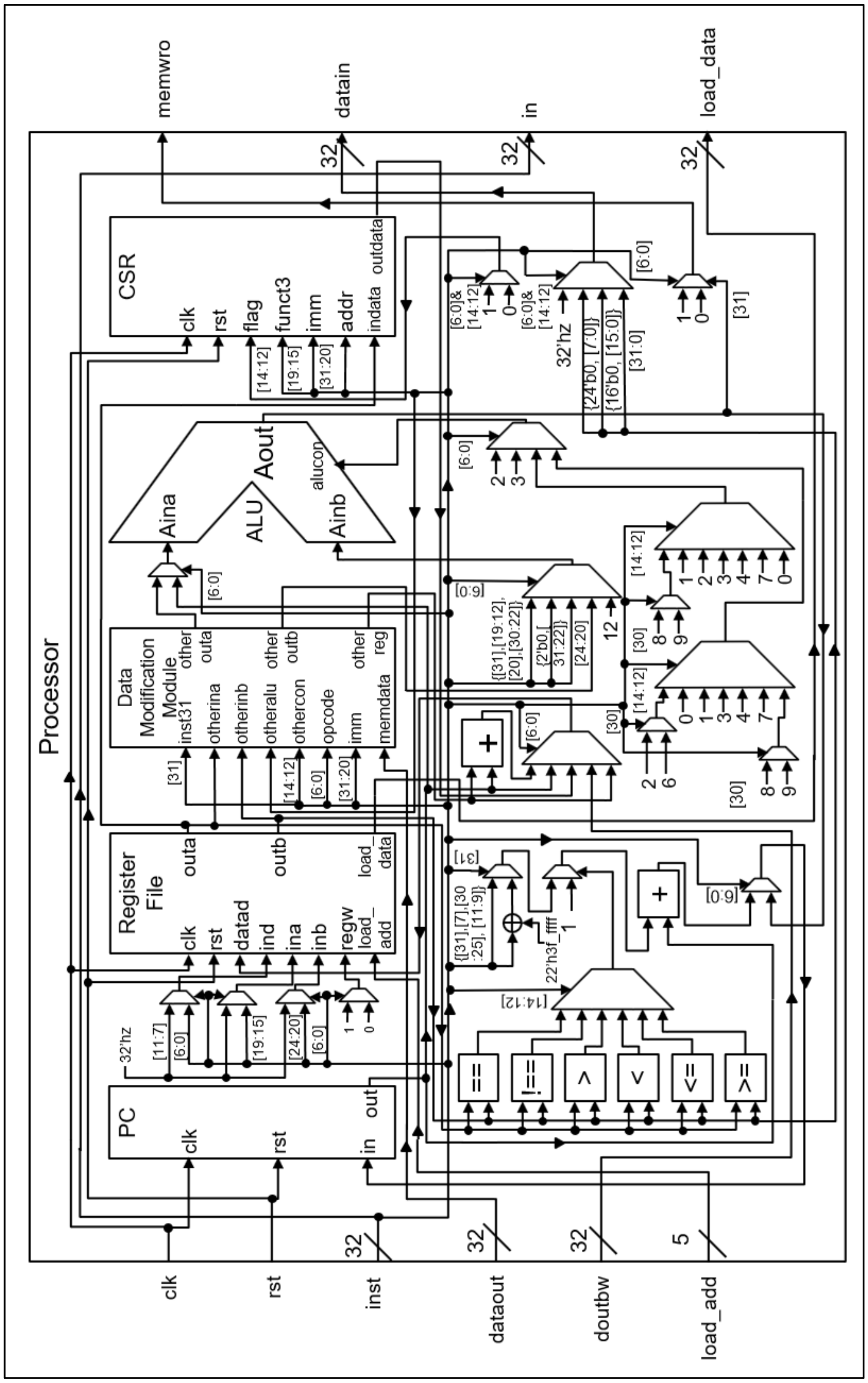


Figure 3. The architecture of the Processor

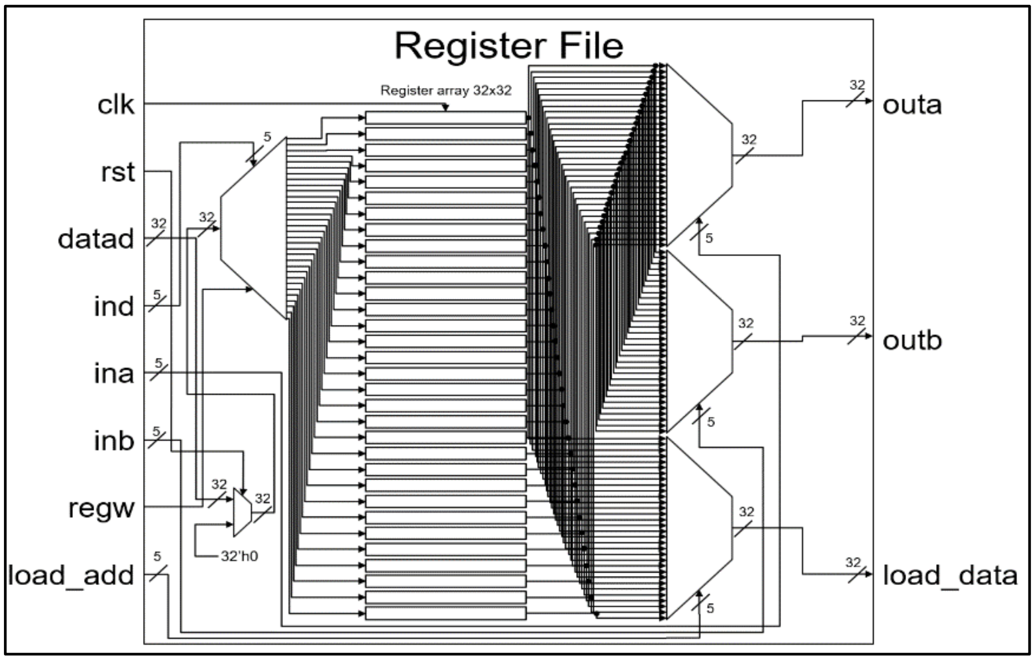


Figure 4. The architecture of Registers and Register Convention

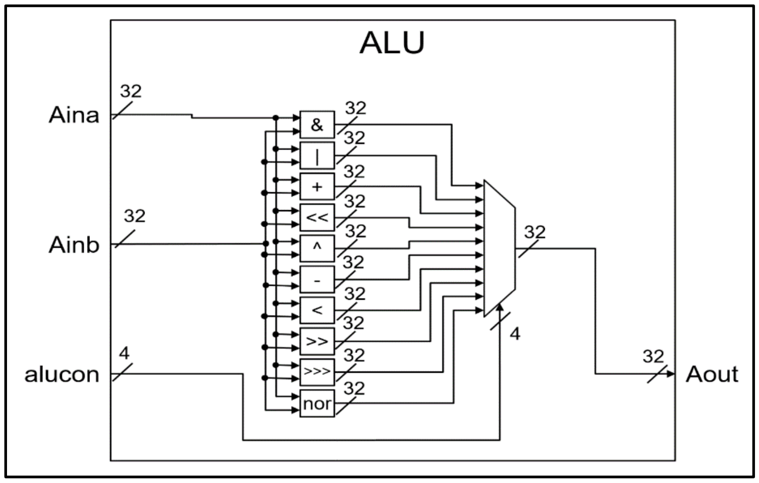


Figure 6. The architecture of ALU

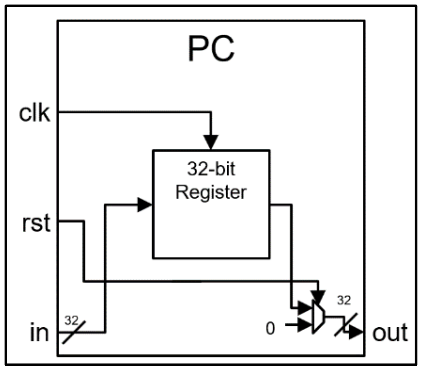


Figure 5. The architecture of PC

Control and Status Register (CSR)Conclusion

CSR is a register that stores various status information within a processor. The input ports of the CSR module are the ***clk***, ***rst***, ***flag***, ***funct3***, ***imm***, ***addr***, ***indata***. The output port of the CSR module is ***outdata***. RISC-V defines a separate address space of 4096 for CSRs so can have at most 4096 CSRs. RISC-V only allocates a part of the address space so can add custom CSRs in unused addresses. Also, not all CSRs are required for all implementations. The module is a register with an array of 12x12-bits and all registers are controlled by the clock and can be activated using a flag signal. The CSR calculates the values of ***imm*** port, ***ina*** port through ***funct3*** port, and stores them in the register. The value of addr is used to write or read the data in the register array. The ***outdata*** port is used to convert a 12-bit output to 32-bit output. Figure 7 shows the proposed architecture of CSR implementation.

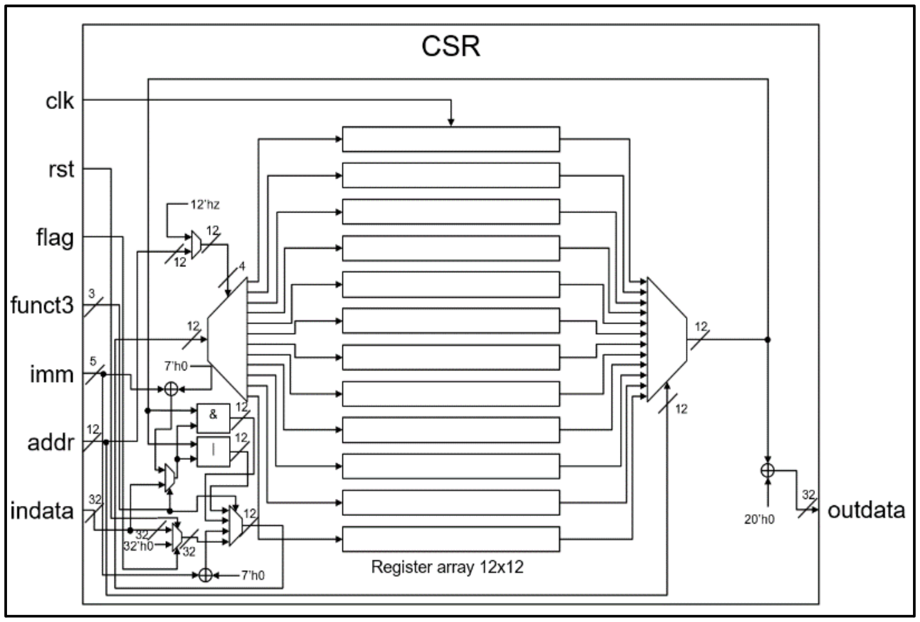


Figure 7. The architecture of CSR

Data Modification Module

The Data Modification Module splits the data into bytes, handles signed and unsigned calculations, and is responsible for some other value calculations. The input ports of the Data Modification Module are the ***inst31***, ***otherina***, ***otherinb***, ***otheralu***, ***othercon***, ***opcode***, ***imm***, ***memdata***. The output ports of the data modification module are the ***otherouta***, ***otheroutb***, and ***otherreg***. The control signals for the Data Modification Module are the ***othercon***, ***opcode***, and ***imm***. The ***otherina*** and ***otherinb*** signals are data from the Register File and ***memdata*** signal is the write/read back data from memory. The internal circuitry of the Data Modification Module consists of logic that converts 32-bit data to signed format and logic that divides 32-bit data into 8-bit, 16-bit, and 24-bit. The ***otherouta***, ***otheroutb*** output signals are connected as inputs to the ALU block and the ***otherreg*** output signal is connected as an input to the Register File block. Figure 8 shows the proposed architecture of the data modification module.

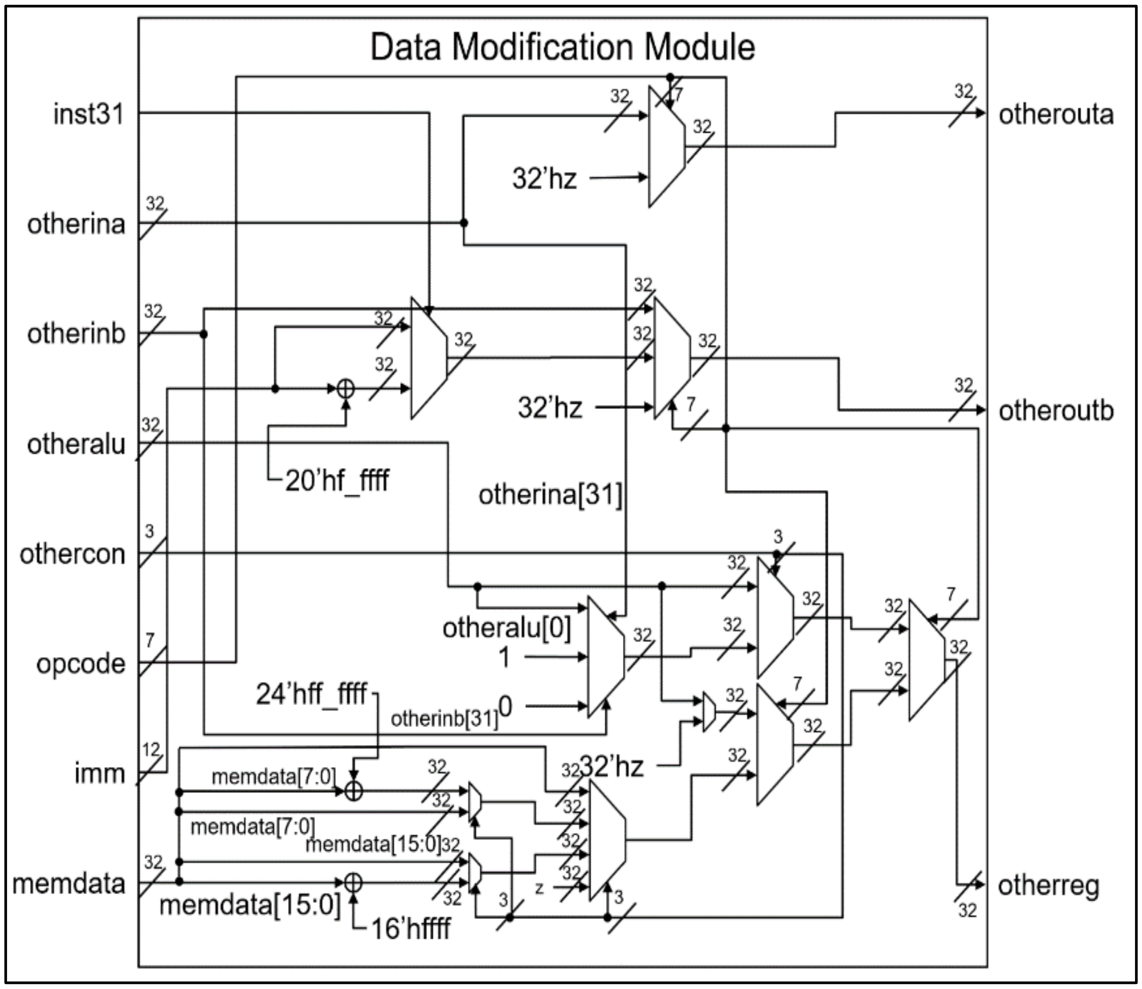


Figure 8. The architecture of the Data Modification Module

Top

The top module is the wrapper module that interconnects and controls the RAM and processor. The input ports of the top module include ***clk*** and ***rst***. The processor manipulates data on both the rising and falling edges. Therefore, at least a twice faster clock is required for RAM and registers to operate properly. The top module has a clock divider and a wire to connect the RAM and processor. Figure 9 shows the architecture of the top module.

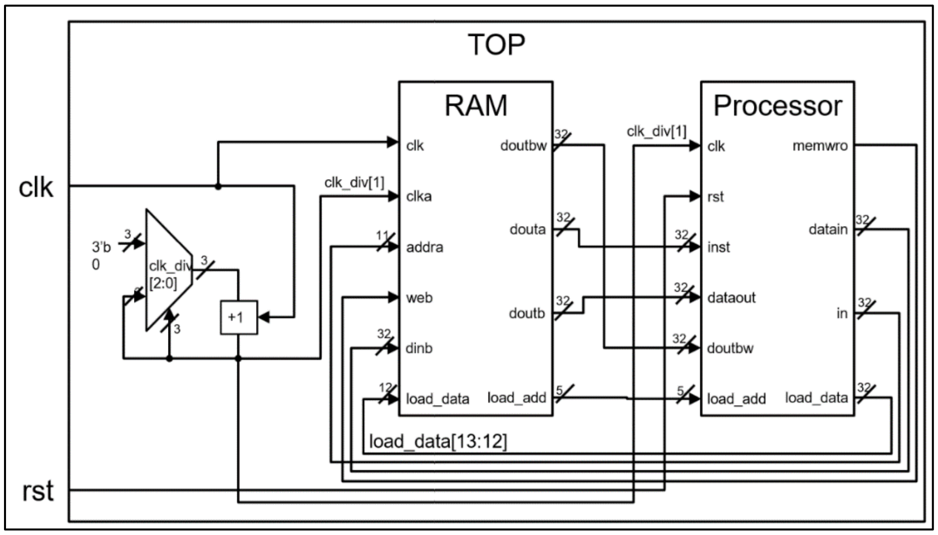


Figure 9. The architecture of Top Module

Serial Communication

A series of schedules are required to connect the proposed user-application program to communicate with the processor. The communication protocols shown in Figure 10 are used to exchange communication between the hardware and software over UART. Firstly, for the UI program to send data, it breaks it into 4-bit blocks and converts it into ASCII code from 0 to F before sending it to FPGA. During communication using UART, all data transmissions start with initiation from the UI program except for PC value transmission. The data sent over UART is always kept at a size of 48 bits for consistency. All unused bit locations are treated as zero and ignored. The program also uses bits 7 to 13 from the protocol as the address to which instruction data is to be stored and the remainder as instruction data. When receiving data from the FPGA board, the program converts it into ASCII code, splitting it into 4 bits each. The data sent from the FPGA to the program has varying data sizes which include 32-bits x 5 of instruction data, 32-bits x 32 of RAM data, and 32-bits x 1 of PC.

Instruction Data Transfer Protocol from UI FPGA

The first 2 most significant protocol bits are the bits that determine if a protocol is an instruction data transmission, RAM data transmission, or Register data transmission. As shown in Figure 10, The instruction data transmission selection bit is 2'b01. The next 11-bits are RAM address data. The following 32-bits are instruction data. The rest of the 3 remaining bits are unused bits and are filled with 3'b000. When the FPGA receives an instruction data transfer of this form, it recognizes that it is an instruction data transfer protocol through the first 2 bits of data received. Subsequently, the hardware generates a write enable signal that enables and stores the data to the RAM.

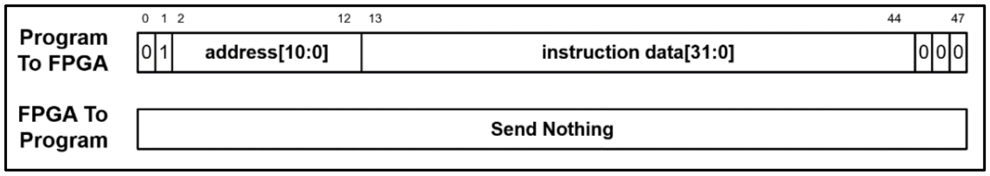


Figure 10. Instruction Send Protocol

Register Refresh Protocol

As shown in Figure 11, the upper 2 bits for Register refresh are 2'b10 with the remaining bits treated as 0 and consisting of 48-bits in total. When the UI program sends a transmission request through the register refresh protocol, the hardware module receives the data and decodes it as a register refresh through the top 2 bits. Subsequently, register file data are obtained by sequentially accessing all addresses in the register file from 0 to 31 and their data contents transferred to the UI program via UART. The total amount of data transmitted is 128 bytes.

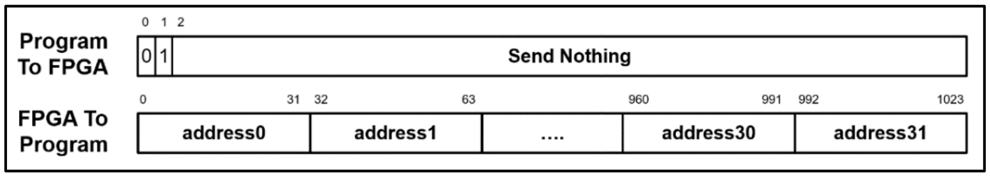


Figure 11. Register Refresh Protocol

RAM Refresh Protocol

The 2 most significant bit of this protocol is 2'b11 which represents RAM selection for a refresh. This is followed by an 11-bit RAM address, and the remaining bits are treated as 0s. When the UI program sends a transmission protocol request for RAM refresh through the protocol, the hardware module receives the data and recognizes that it is a RAM refresh request through the upper 2 bits. It then accesses the RAM data of the top two and bottom two RAM addresses from the RAM address sent to the FPGA. The FPGA transfers the RAM data to the computer via UART in order of the larger addresses. The total amount of data sent is 20 bytes during RAM refresh. Figure 12 shows the register refresh protocol.Conclusion

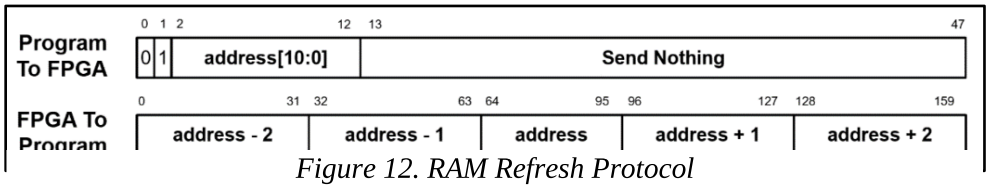

Program To FPGA
0 1 2 12 13 47
0 1 address[10:0] Send Nothing
FPGA To Program
0 31 32 63 64 95 96 127 128 159
address - 2 address - 1 address address + 1 address + 2

Figure 12. RAM Refresh Protocol

PC Transmission Protocol

There is no data transferred from the UI program to the FPGA, and if the button clock is pressed on the FPGA, the FPGA transmits the current PC value. The total amount of data sent through UART is 4 bytes during the PC update. Figure 13 shows the PC transmission protocol.

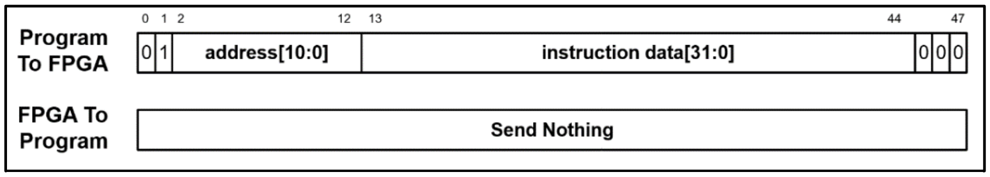
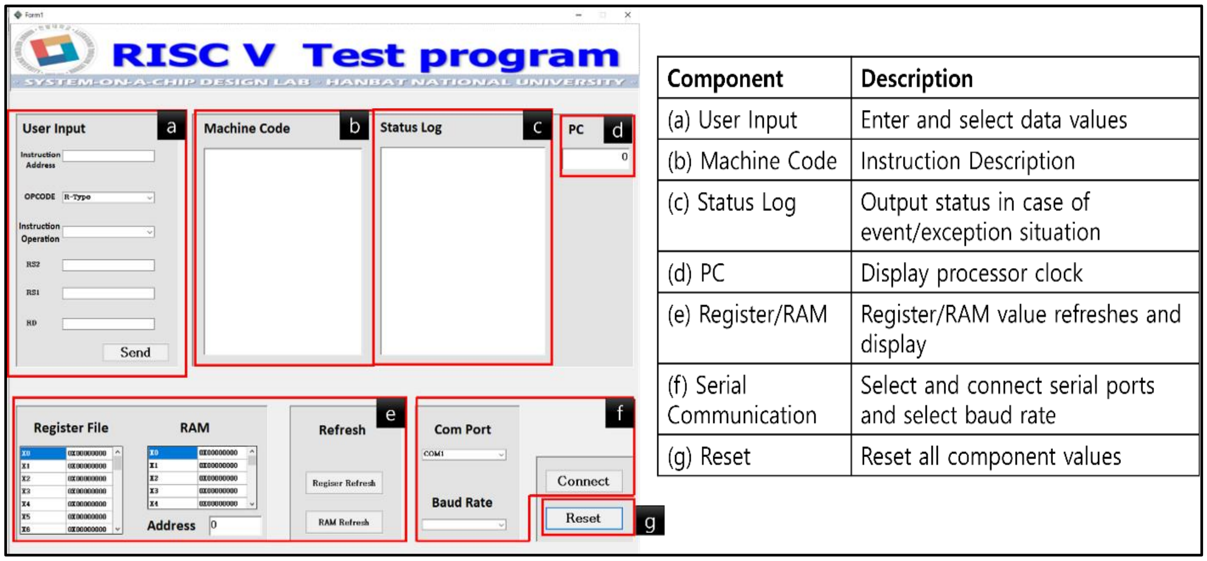


Figure 13. PC Transfer Protocol

Desktop UI Program

The desktop UI program was designed with the C# program language. It allows a user, through a desktop, to control and verify the functionality of a processor core. Figure 14 shows the desktop UI program.



Component	Description
(a) User Input	Enter and select data values
(b) Machine Code	Instruction Description
(c) Status Log	Output status in case of event/exception situation
(d) PC	Display processor clock
(e) Register/RAM	Register/RAM value refreshes and display
(f) Serial Communication	Select and connect serial ports and select baud rate
(g) Reset	Reset all component values

Figure 14. Desktop UI Program and Describes the Components

User Input

To send instruction data from the desktop UI program to the processor core, input values are set and then transferred. The instruction address section sends instruction data along with the instruction so that the instruction is stored at a specified address in RAM. The OPCODE section uses the combo-box to list the types from R-type to J-type OPCODEs.

Table 2 shows the OPCODE values and their named labels based on the combo-box selected values. The instruction operation is listed using the combo-box for each type of instruction operation. The values of Funct3 and Funct7 are stored according to the selected index value. Table 1 shows the index and the corresponding Funct3 and Funct7 values of each OPCODE type listed in the OPCODE combo-box. Also, based on the selected OPCODE type, two or three source or destination text-boxes labeled RS2/IMM, RS1, or RD are used, to capture the instruction values to be sent. When the send button is clicked, the OPCODE, Funct3, Funct7, and source-destination register text-box inputs are combined to construct an instruction according to chosen OPCODE type and instruction operation. The built instruction data is converted to hexadecimal format and sent to the FPGA via the UART.

Machine Code

When the send button is pressed, the instruction address, instruction operation, RS2/IMM, RS1, and RD are converted in Assembly and Machine codes (hexadecimal format) and updated in the machine code text-box. Figure 15 is a depiction that shows an example of machine code constructed according to user input values.

Status Log

When an event occurs in a desktop UI program or when an exception is encountered, the Status Log text area is updated with the appropriate log information based on the event/exception that occurred. Table 3 list the log notification or information that correspond to events or possible exceptions that may occur. Figure 15 shows the text of the status log output from the desktop UI program.

Table 1. Table 1.Funct3 and Funct7 Values for Each Type of Operation

Type	Operation	Funct7	Funct3	Type	Operation	Funct7	Funct3
R	ADD	0000000	000	L	LB	Not used	000
	SUB	0100000	000		LH	Not used	001
	SLL	0000000	001		LW	Not used	010
	SLT	0000000	010		LBU	Not used	100
	SLTU	0000000	011		LHU	Not used	101
	XOR	0000000	100	S	SB	Not used	000
	SRL	0000000	101		SH	Not used	001
	SRA	0100000	101		SW	Not used	010
	OR	0000000	110	SB	BEQ	Not used	000
	AND	0000000	111		BNE	Not used	001
I	ADDI	Not used	000		BLT	Not used	100
	SLTI	Not used	010		BGE	Not used	101
	SLTIU	Not used	011		BLTU	Not used	110

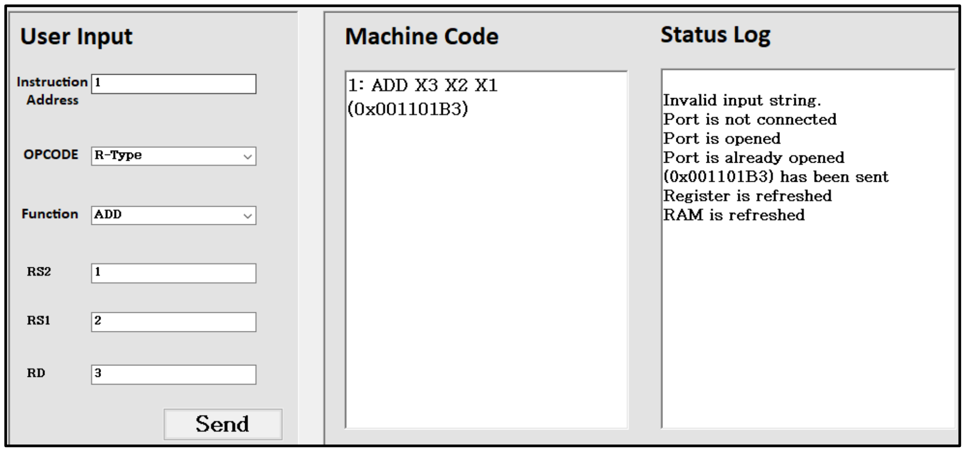
	XORI	Not used	100		BGEU	Not used	111
	ORI	Not used	110	U1	LUI	Not used	Not used
	ANDI	Not used	111	U2	AUIPC	Not used	Not used
	SLLI	Not used	001	J1	JAL	Not used	Not used
	SRLI	Not used	101	J2	JALR	Not used	000
	SRAI	Not used	101	-	-	-	-

Table 2. OPCODE Values and Named Labels

Type	OPCODE	Label 1	Label 2	Label 3
R	0110011	RS2	RS1	RD
I	0010011	IMM	RS1	RD
L	0000011	IMM	RS1	RD
S	0100011	IMM	RS2	RS1
SB	1100011	IMM	RS2	RS1
U1	0110111	IMM	RD	Not used
U2	0010111	IMM	RD	Not used
J1	1101111	IMM	RD	Not used
J2	1100111	IMM	RS1	RD

Table 3. The output of Text According to Event or Exception Processing

Event/Exception	Log Information Text
Clicking send button	Hex instruction data has been sent
Clicking register refresh button	Register is refreshed
Clicking RAM refresh button	RAM is refreshed
Clicking connect button	Port is opened/Port is already opened
When the communication port is not connected	Port is not connected
Clicking send button without incorrect value or value in user input	Invalid input string



Instruction Address: 1
OPCODE: R-Type
Function: ADD
RS2: 1
RS1: 2
RD: 3

```
1: ADD X3 X2 X1
(0x001101B3)
```

```
Invalid input string.
Port is not connected
Port is opened
Port is already opened
(0x001101B3) has been sent
Register is refreshed
RAM is refreshed
```

Figure 15. Output Value of Machine Code and Status Log's Rich Text-Box

Register/RAM Refresh in UI

When the register refresh button is clicked, register values for addresses 0 to 31 in the processor core are read and updated in the data grid view at one time. When the RAM refresh button is clicked, the address value entered in the Address text-box and the address value-2, address value-1, address value, address value+1, address value+2 locations are read from the processor and updated in the data grid view. The reason for refreshing data values for only five addresses is that RAM has 2048 addresses, which is cumbersome in retrieving all data at once. Table 4 shows the data retrieved from the FPGA according to the address for RAM.

Table 4. Scope of Data Refresh Based on Address for RAM

Address	Arrange
0, 1	0, 1, 2, 3, 4
2046, 2047	2043, 2044, 2045, 2046, 2047
Others	Address -2, Address -1, Address, Address +1, Address +2

Serial Communication

The communication port (comport) combo box lists the serial ports that are available when the program runs. The port to be used, and its corresponding baud rate is selected from the combo box to set up communication. The connect button is pressed to establish the communication link between the FPGA and the UI program.Conclusion.

Verification Module of RISC-V Processor

The verification module is designed to perform processor verification through the hardware on which the RISC-V RV32I is programmed or implemented. Verification module provides visual display and control using multiple hardware peripheral components including LEDs, 7-segment, text-LCD, dot-matrix, dip-switches, RS232, push-buttons, and keypad. By using these components, any verification can be performed to visualize the actual internal operations of the processor under test and directly control them from the FPGA. Figure 16 shows the components of the verification module, and Table 5 describes the functions of each component.

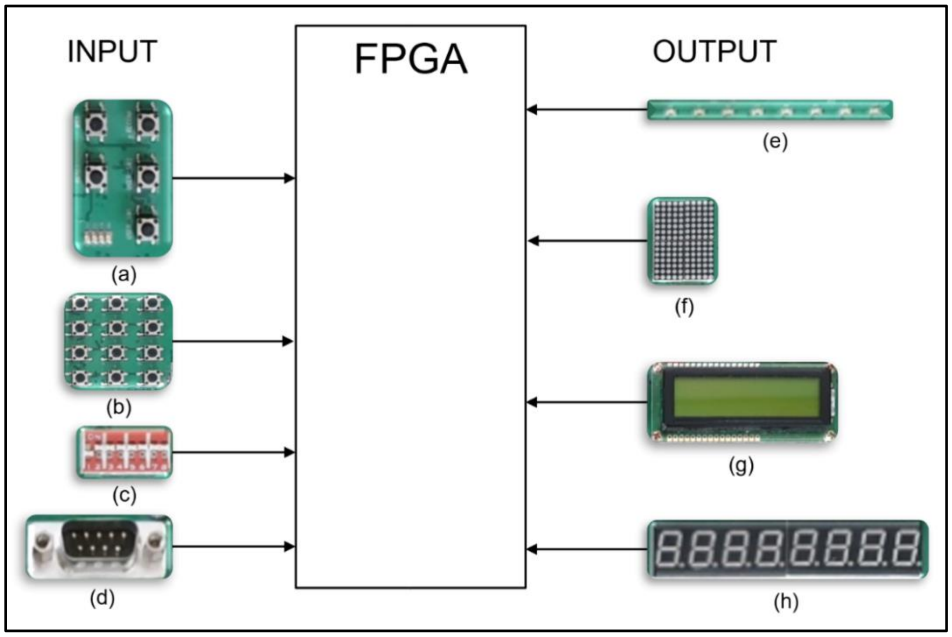


Figure 16. Overall Connectivity of the Designed Peripherals

Table 5. Functional Description of Each Component

Component	Description
(a) Push-Button	The button on the left produces a signal that corresponds to only one clock pulse when the button is pressed to the clock generator. Press the button on the right with the instruction address selector to increase the address by one in two 7-segments from the right, which will initialize to zero as the value goes up to 31.
(b) Keypad	Enter the input value with the keypad corresponding to 0 through 9. If entered incorrectly, the button in the lower-left corner (*) clears the entered value. When the correct value is entered, press the button in the lower right corner (#) to send the input and prepare to receive the input from the next code.
(c) Dip-Switches	Five switches from the left are components that set the address of the register file. The switch on the far right is a component that controls the real-time and non-real-time operation modes of the processor under verification.
(d) RS232 Port	This component is required to communicate with a program that performs verification using UART. Use the components above to carry out UART communications.
(e) LED	The function that displays the current value of PC. Up to 255 can be expressed.
(f) Dot-Matrix	Use only a total of 32 dot matrices from the top left to the bottom. The information displayed is observed by adjusting the value using dip-switches from 1 to 5, where the dip-switches represent the address of the register and the value of the dot matrix corresponds to the data of that register address.
(g) Text LCD	Text LCD is 16x2 in size. On the left side of the first row of the text LCD, the 2nd to 4th LCD is the code that is entered. The other six text LCDs display the input values for that code in decimal. On the left side of the second row, 2 through 5, are the commands of that type and 7 through 14 are the values of the entered 32-bit hexadecimal values.
(h) 7-Segment	Six 7-segments from the left are MMIO components with addresses of 0x80000001. The other two 7-segments display the instruction address to store.

Verification of Proposed Processor Implementation on an FPGA Board

The implemented RISC-V processor and the corresponding verification hardware modules were designed and tested on the HBE-SoC-IPD FPGA board from Hanback Electronics. The test board is equipped with a Virtex-4 XC4VLX80 FPGA device. The test board uses LEDs, 7-segments, text-LCD, dot-matrix, dip-switches, RS232, push-buttons, and keypad components to monitor the internal operation of the processor under test. Figure 16 describes the overall connectivity of the designed peripherals. The test module controls all components except the RS232 ports which are controlled by the UART module.

Execution Flow of UI Program

The desktop UI program was developed with MS's Visual Studio 2019 IDE and was designed as a window form using the C# programming language. Figure 17 shows a flowchart of the execution flow of the UI program. When the desktop UI program runs, the GUI components are initialized, and available serial ports are automatically listed in the combo box of the com port. The user selects the right port and baud rate, clicks the connect button to connect with FPGA through UART communication. The instruction address is entered, and the desired data is entered in the source-destination register text-boxes according to the OPCODE and instruction operation selected. When the send button is pressed, the set user inputs are transformed into a 12-byte hexadecimal string that combines the instruction address, instruction data, and control bit. This hexadecimal data is then sent to the FPGA. Each string sent is bit-converted according to the value of the ASCII code, and the instruction data is stored in RAM at the address of the instruction address. As the PC increases, instructions stored in RAM are executed by the processor, and values of the PC are displayed in the desktop UI program.

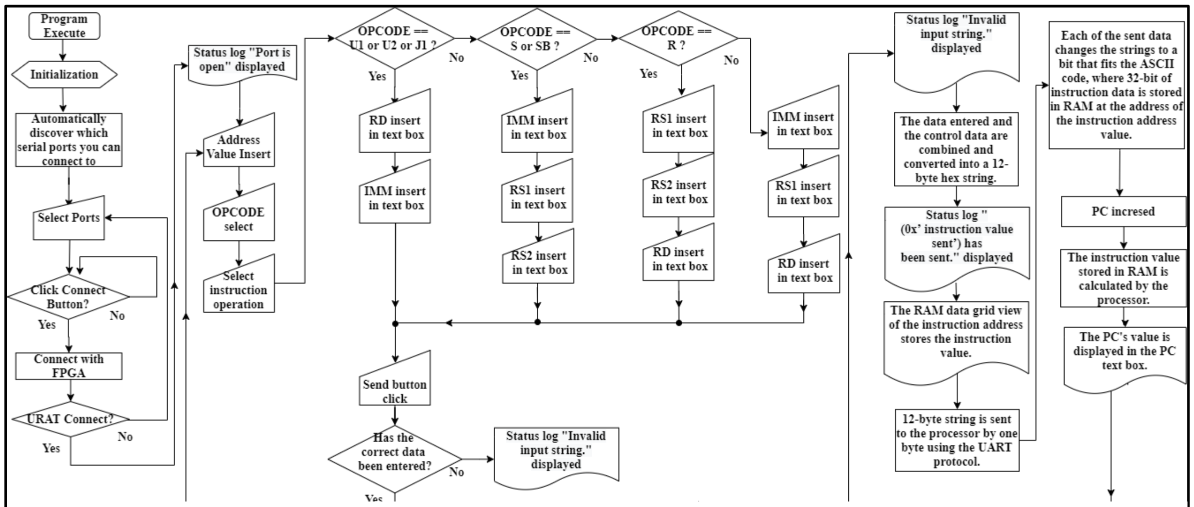

Program Execute
Initialization
Automatically discover which serial ports you can connect to
Select Ports
Click Connect Button?
Yes
No
Connect with FPGA
URAT Connect?
Yes
No
Status log "Port is open" displayed
Address Value Insert
OPCODE select
Select instruction operation
OPCODE == U1 or U2 or J1 ?
Yes
No
RD insert in text box
IMM insert in text box
OPCODE == S or SB ?
Yes
No
IMM insert in text box
RS1 insert in text box
RS2 insert in text box
OPCODE == R ?
Yes
No
RS1 insert in text box
RS2 insert in text box
RD insert in text box
IMM insert in text box
RS1 insert in text box
RD insert in text box
Send button click
Has the correct data been entered?
No
Yes
Status log "Invalid input string." displayed
Status log "Invalid input string." displayed
The data entered and the control data are combined and converted into a 12-byte hex string.
Status log " (0x' instruction value sent') has been sent." displayed
The RAM data grid view of the instruction address stores the instruction value.
12-byte string is sent to the processor by one byte using the UART protocol.
Each of the sent data changes the strings to a bit that fits the ASCII code, where 32-bit of instruction data is stored in RAM at the address of the instruction address value.
PC incresed
The instruction value stored in RAM is calculated by the processor.
The PC's value is displayed in the PC text box.

Figure 17. Flowchart of the Execution flow of UI Program

Figure 18 shows a flowchart for receiving Register/RAM data values from the FPGA to the desktop UI program. When a user clicks the register refresh button in the desktop UI program, Register values for addresses 0 to 31 are read from FPGA and displayed in the data grid view. The RAM locations that are updated based on the input address are explained and illustrated in Table 4.

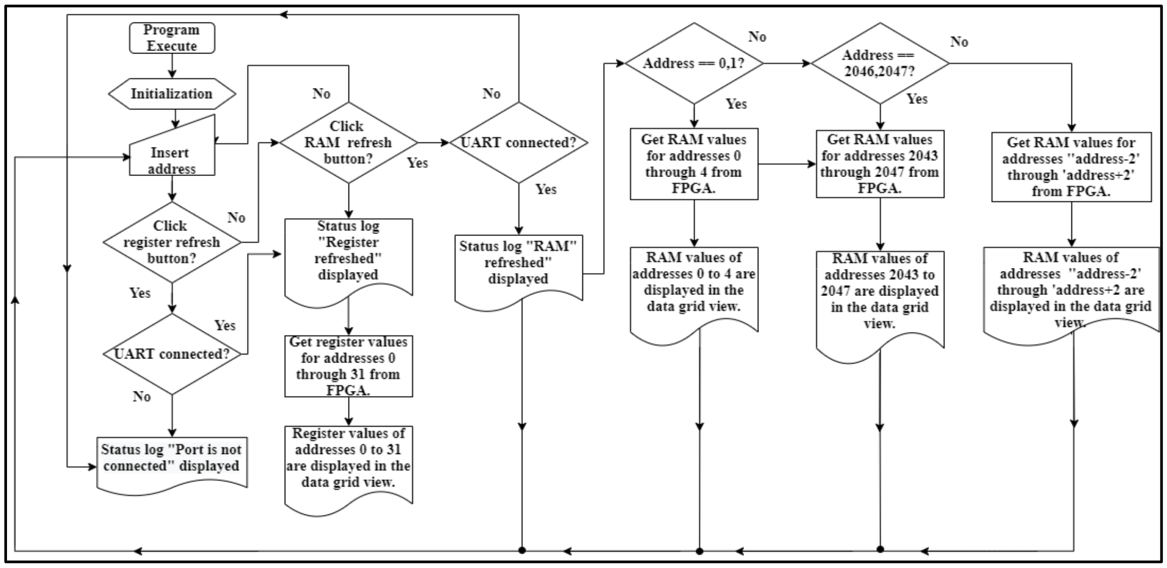
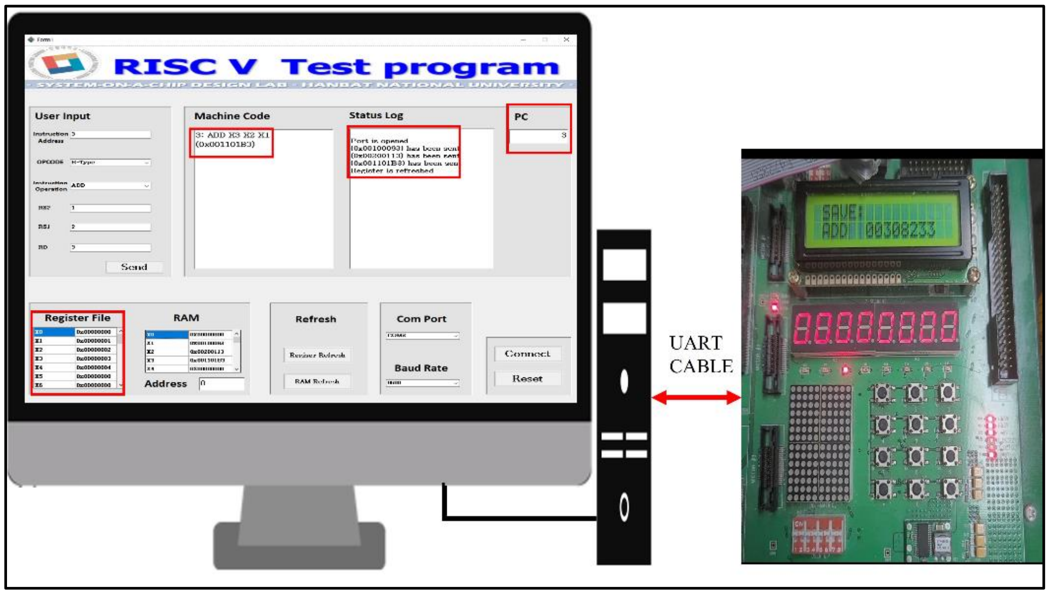

Program Execute
Initialization
Insert address
Click register refresh button?
No
Yes
UART connected?
Yes
No
Status log "Port is not connected" displayed
Click RAM refresh button?
No
Yes
Status log "Register refreshed" displayed
Get register values for addresses 0 through 31 from FPGA.
Register values of addresses 0 to 31 are displayed in the data grid view.
UART connected?
No
Yes
Status log "RAM" refreshed" displayed
Address == 0,1?
No
Yes
Get RAM values for addresses 0 through 4 from FPGA.
RAM values of addresses 0 to 4 are displayed in the data grid view.
Address == 2046,2047?
No
Yes
Get RAM values for addresses 2043 through 2047 from FPGA.
RAM values of addresses 2043 to 2047 are displayed in the data grid view.
Get RAM values for addresses "address-2' through 'address+2' from FPGA.
RAM values of addresses "address-2' through 'address+2 are displayed in the data grid view.

Figure 18. Flowchart for Receipt of Register/RAM Data Values from FPGA by Desktop UI Program

Figure 19 shows the interconnection between the implemented RISC-V ISA and the custom-designed UI program. To test the addition of the digits 1 and 2 to obtain a result of 3, First, using the desktop UI program, input the Instruction Address 1, choose I-type from the OPCODE combo-box and select the ADDI from the Instruction Operation combo-box. Next input the IMM value of 1 which is the first digit(operand) for the addition operation to the destination register(RS1 value of 0, and RD value of 1). When the send button is clicked, an instruction for storing 12’h1 in register address 1 is saved to RAM address 1. For the second instruction to store 12’h2 at register address 2 the same steps are repeated as the previous but Instruction is saved at RAM address 2(IMM=2, RS1=0, RD=2). Thirdly, the user input the Instruction Address value of 3, chooses R-type at OPCODE, chooses ADD from the Instruction Operation combo-box. Input RS1 value for source register 1, RS2 value for source register 2, and RD value for destination register 3. When the send button is clicked, an instruction for computing the sum of the value of register addresses 1 and 2 into register address 3 is stored in RAM address 3 is built (RS1 = 2, RS2 = 2, RD = 3). As the PC increases, the instruction stored in the RAM is executed sequentially. When the register refresh button is clicked, a data value for each address of the register is displayed. The result values were all correct. Conversely, it was confirmed that the operation was performed by storing instruction data in RAM on the FPGA board and that the correct value was displayed even when the refresh button was pressed in the UI program. Therefore, it was confirmed that there was an interconnection between the RISC-V ISA implementation UI programs.



3: ADD X3 X2 X1
(0x001101B3)

Port is opened
(0x00100093) has been sent
(0x00200113) has been sent
(0x001101B3) has been sent
Register is refreshed

Figure 19. Implementation of RISC-V ISA Using Desktop UI Program

Hardware Synthesis Results of RISC-V Verification Module

The proposed processor and hardware verification architecture was designed and verified by Xilinx 14.7. The design occupied 6476 LUTs and operated at a maximum frequency of 49.114 Mhz. In comparison to the architecture in (Don, 2017), the proposed design utilized 16 percent more LUT's due to the use of multiple verification modules. However, the maximum operating frequency increased by 53 percent. Table 6 shows a summary of the comparison of the area and speed of this design with the design of (Don, 2017).

Table 6. Compares Size and Speed

Design	Processor Type and Peripherals	Area (LUTs)	Frequency (MHz)
(Don,2017)	RV32I Processor, Text-LCD	5578	32
This work	RV32I Processor, Text-LCD, RS232 port, Push-Button, 7-Segment, LED, Dot-Matrix, Keypad, Dip-Switches	6476	49

Conclusion

This paper proposed a hardware implementation of RISC-V synthesizable processor and a user-friendly desktop UI program. The proposed processor with its features makes it easier to understand and control the internal operation of a RISC-V processor under design, using either hardware peripheral components of the FPGA or a user-friendly GUI application. The proposed architecture is capable of quickly processing a large number of instructions through the proposed desktop UI software program which communicates via UART. Not all, the proposed system can also serve as a guide to researchers and designers who want to design, verify or perform a simple test of a single cycle RISC-V processor. Regarding limitation, the proposed system cannot build instructions through higher-level languages like C and C++ that enable the proposed system to function additionally as a RISC-V compiler or toolchain. The next direction of this research will seek to provide a system that can provide users with a much-increased convenience by incorporating a compiler within the software program to accommodate higher-level languages such as assembly, machine code which will result in an integrated RISC-V software and hardware development and test platform.

References

Ryu, J.W., & Lee, J.H. (2021). An Efficient Vehicle License Plate Recognition System Based on Embedded Systems. *Journal of Next-generation Convergence Technology Association*, *5*(1), 22-27.

Bruno, C., Licciardello, A., Nastasi, G.A.M., Passaniti, F., Brigante, C., Sudano, F., Faulisi, A., & Alessi, E. (2021). Embedded Artificial Intelligence Approach for Gas Recognition in Smart Agriculture Applications Using Low-Cost MOX Gas Sensors. *2021 Smart Systems Integration*, 1-5.

Gies, V., Marzetti, S., Barchasz, V., Barthelemy, H., & Glotin, H. (2021) Ultra-low Power Embedded Unsupervised Learning Smart Sensor for Industrial Fault Classification. *2020 IEEE International Conference on Internet of Things and Intelligence System* , 181-187.

The Linux 'RISC-V' singularity in the CPU world has begun to break through . (2020). https://zdnet.co.kr/view/?no=20201201123035.

Degirmen, D. (2019). Open Hardware: Initial Experiences with Synthesizing Open Cores.

Swarup, B., Michael, S.H., Mainak, B., & Seetharam, N. (2014). Hardware trojans attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, *102*(8), 1229-1247.

Andrew, W., Yunsup, L., David, P., & Krste, A.(2011) The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley.

Samsung to Use SiFive RISC-V Cores for SoCs, Automotive, 5G Applications . (2019). https://www.anandtech.com/show/15228/samsung-to-use-riscv-cores.

SiFive introduces E300 & U500 opensource chip platforms . (2016). https://www.electronicproducts.com/sifive-introduces-e300-u500-open-source-chip-platforms/#.

PicoRV32 - A Size-Optimized RISC-V CPU . (2019). Retrieved from https://github.com/YosysHQ/picorv32.

Dennis, A.N.G., & Ryoo, K.K. (2019). Selecting a synthesizable RISC-V processor core for low-cost hardware devices. *Journal of Information Processing Systems*. *15*(6), 1406-1421.

Oleksiak, A., Cieslak, S., Marcinek, K., & Pleskacz, W.A. (2019). Design and Verification Environment for RISC-V Processor Cores. *Mixed Design of Integrated Circuits and Systems*, 206-209.

Schiavone, P.D., Sanchez, E., Ruospo, A., Minervini, F., Zaruba, F., Haugou, G., & Benini, L. (2018). An Open-Source Verification Framework for Open-Source Cores: A RISC-V Case Study. *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 43-48.

Don, K.D., Ayushi, P., Virk, S.S., Sajal, A., Tanuj, S., Arit, M., & Kailash, C.R. (2017). Single-cycle RISC-V microarchitecture processor and its FPGA prototype. *Proceedings of 7th International Symposium on Embedded Computing and System Design*, 1-5.

David, A.P., & John, L.H. (2017). Computer organization and design: The hardware/software interface. RISC-V 5th edn, Elsevier.