DUBLIN CITY UNIVERSITY
SCHOOL OF ELECTRONIC ENGINEERING

Implementation and Performance Analysis of the RSIC-V RV32I Architecture

Zhuang Miao

August 2024

BACHELOR OF ENGINEERING

IN

ELECTRONIC AND COMPUTER ENGINEERING

MAJORING IN

Systems & Devices

Supervised by Dr X. Wang

Acknowledgements

I would like to express my deepest gratitude to Dr Xiaojun Wang for his invaluable guidance and expertise throughout this project. We have set a time to meet every week to answer my questions about the project. Set and lead tasks for me at each stage. His insightful feedback was pivotal in the completion of this work.

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the source cited in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the DCU Academic Integrity and Plagiarism at https://www.dcu.ie/system/files/2020-09/1_-_integrity_and_plagiarism_policy_ovpaa-v4.pdf and IEEE referencing guidelines found at https://loop.dcu.ie/mod/url/view.php?id=1401800 .

Name: Zhuang Miao Date: 01/04/2024

Abstract

In this project, we delve into the implementation of the RISC-V RV32I architecture and its performance evaluation with the PRESENT-80 encryption algorithm. This study builds on RISC-V's inherent principles of simplicity, efficiency, and scalability by carefully building a custom single-cycle processor designed to operate the RV32I instruction set. Using SystemVerilog for processor design and Vivado for simulation, we carefully analysed the processor's ability to execute PRESENT-80 and juxtaposed it with a direct hardware implementation of the cryptographic algorithm. Through comparative analysis, the project sheds light on processor execution times, instruction lengths, and overall performance, with a focus on optimizing code for greater efficiency. The findings highlight the strong potential of the RISC-V RV32I architecture in secure data processing applications and advocate further exploration of specialized processor designs to optimize performance. This exploration not only broadens the understanding of RISC-V's suitability for cryptography, but also lays the foundation for future advances in processor design tailored to specific computing tasks.

Table of Contents

Table of Figures

Chapter 1 - Introduction

Contemporary computer architecture adheres to the principles delineated by the Instruction Set Architecture (ISA). Within this framework, the Reduced Instruction Set Computer (RISC-V) emerges as a novel open-source ISA, developed by the University of California, Berkeley, and first unveiled in 2014. Among its various implementations, the RISC-V 32-bit integer (RV32I) architecture stands out as the most comprehensive, offering a 32-bit subset of the RISC-V that is distinguished by its simplicity, efficiency, and scalability when juxtaposed with alternative architectures [2, 3]. The primary challenge and objective of this paper is the implementation of PRESENT-80 on the RISC-V platform. Although the GNU Compiler Collection (GCC[1]) facilitates the translation of C++ code into RV32I [4], the essence of this project is centred on the implementation of PRENSENT-80 using RV32I that has been optimized independently. The analytical section of this paper provides a comparative study on the code length of RV32I generated by GCC versus that produced through independent optimization, in addition to examining the runtime performance on a processor operating at maximum frequency and the actual runtime in a C++ program.

To establish the operating environment for the RISC-V RV32I, a bespoke Central Processing Unit (CPU) was devised, alongside its subsequent simulation and testing, utilizing SystemVerilog. This CPU is designed to process the machine code corresponding to a given instruction in a single cycle. The construction of the CPU commenced with the development of its fundamental modules, which include, but are not limited to, an Arithmetic Logic Unit (ALU), instruction memory, register memory, control unit, adder, and multiplexers. For the theoretical underpinning of these core concepts integral to the CPU's design, '*Digital Design and computer architecture: RISC-V edition'* by Harris, S.L. and Harris, D.M [5] served as a pivotal resource.

The evolution of encryption technology has spanned many years, during which a plethora of technologies have emerged. These technologies include the Data Encryption Standard (DES), Advanced Encryption Standard (AES), Rivest-Shamir-Adleman (RSA), Twofish, and numerous other algorithms. Each of these technologies presents distinct differences and possesses its unique set of advantages. The significance of encryption technology cannot be overstated, especially in its role in safeguarding private data transmitted across networks and in protecting national security through military-grade encryption systems. Within the scope of this project, the PRESENT-80 encryption algorithm is highlighted as an exemplar of an ultra-lightweight block cipher [1]. This paper will primarily focus on the encryption of 64-bit state plaintext utilizing the RISC-V instruction set, although the underlying theory is equally applicable to 128-bit states. The implementation of PRESENT-80 is envisaged in a hardware design context using SystemVerilog and the RISC-V instruction set, alongside a software design in C++.

[1] GCC is GNU Compiler Collection, an optimizing compiler by GUN Project

Chapter 2 - The RISC-V RV32I Computer Architecture

In this chapter, several key technics relate RISC-V instructions set and computer architecture have been used to develop and implement in this project.

2.1 RISC-V RV32I

RISC-V architecture provides a comprehensive set of instructions for execution on the CPU, which are transformed into machine code adhering to its specific formatting conventions. These instructions adhere to a fixed or similar format and are categorized into four primary types: register instruction (R-type), immediate instruction (I-type), store/branch instruction (S/B-type), and upper immediate/jump instruction (U/J-type). All the instructions of RISC-V RV32I can be found in Table 5: Full Set of the RV32I Instructions.

R-type instructions, such as *add* and *sub*, involve operations on three registers, underscoring their function in executing arithmetic or logical operations directly between registers. Immediate instruction (I-type), represented by instructions like *addi, lw, slli* among others, incorporates the use of an immediate value alongside a source and destination address, facilitating operations where constants are integral to the execution logic.

Store/Branch (S/B-type) instructions, including *sw, sh,* and *sb*, are designed for operations that involve two addresses combined with an immediate value, typically used in data storage and conditional branching scenarios.

Finally, U/J-type instructions are characterized by their operation on a destination register and an immediate value, notable for the size of this immediate value being 20-bit, exemplified by the *lui* instruction. This category is particularly focused on tasks that involve setting upper bits of a register or facilitating jumps in the program flow.

For a completely summary table of all the instructions 32-bit format can be found in figure below. Whereas in this project certain types and instructions in 32-bit format mentioned above will be majority being used and implemented in CPU. There is also a website tool to transfer RISC-V instruction to machine code [12].

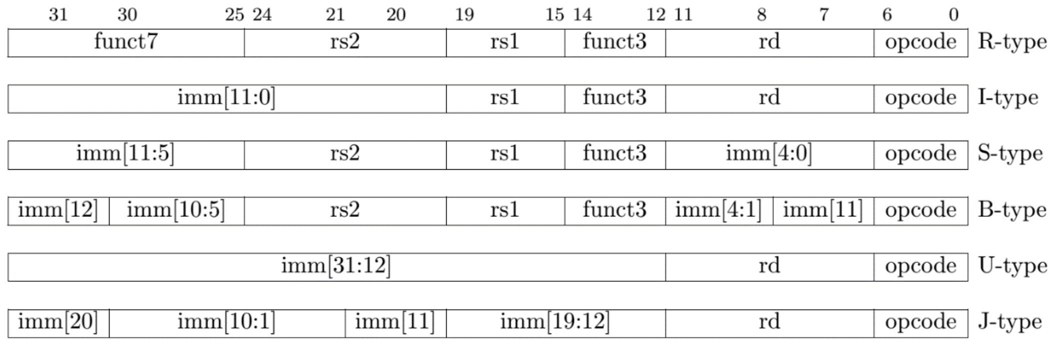


Figure 1: Summary of Instructions to Machine Code Bits Locations [6]

The format of instructions not only contain return address(*rd*) or resource(*rs*), immediate value(*imm*) with variety size of bits, there are other groups of bits inside the format are equally important such as *funct7*, *funct3*, and *opcode*, which in control of ALU, control unit, data memory block, extend block, and several multiplex blocks select bit. There is more detail of how those group effect the behaviour of the CPU in following sections. In this project the customized CPU will only interesting the fifth bit of *funct7*, *funct3*, and *opcode*. This is due to customized CPU only supports with certain instructions.

The instructions slated for use and implementation within the customized CPU for this project are detailed in Table 2: Implemented Instructions Set. Despite the CPU's architectural simplification, the instructions retain their conventional format, ensuring consistency with the RISC-V standard. The subsequent section will elucidate the application of these instructions for the implementation of the PRESENT-80 encryption algorithm.

The resources and return addresses involved in the instructions include not only reading conventional RAM, but also very commonly used register table. This register can store 32 pieces of 32-bit data. More strictly speaking, the first storage location (0x0) of the register can only be zero, and the data in other locations will change according to the instruction. Each piece of data has their name which use in instructions. The register table shown in Figure 10.

2.2 CPU – Single Cycle Processor

Implementing the CPU provides an operating environment for running RISC-V on the one hand, and on the other hand, by running the instruction set corresponding to PRESENT-80, the running time can be predicted for comparison. There are many types of CPUs mentioned in the *Digital Design and computer architecture: RISC-V edition*, such as Single Cycle Processor, multicycle processor, and pipeline processor, as they are all implemented the subset of RISC-V instructions and share core components, but they are difference in how they operate instruction [5]. In this project, a single cycle processor was implemented. Because single cycle processor provides a straightforward architecture and clock cycles per instruction (CPI) of 1 which is excellent scenario to obtain more accurate predict runtime.

This CPU is composed of multiple modules and able to process 32-bit instructions. Those components are Instruction Memory, Register File, Control Unit, ALU, Data Memory, Extend, adders, and three multiplexers. The overall top level of this architecture shown in figure below. Equivalent schematic of this project implementation by Vivado also shown as follows. They are same concept but figure 2 is easier to understand and clear.

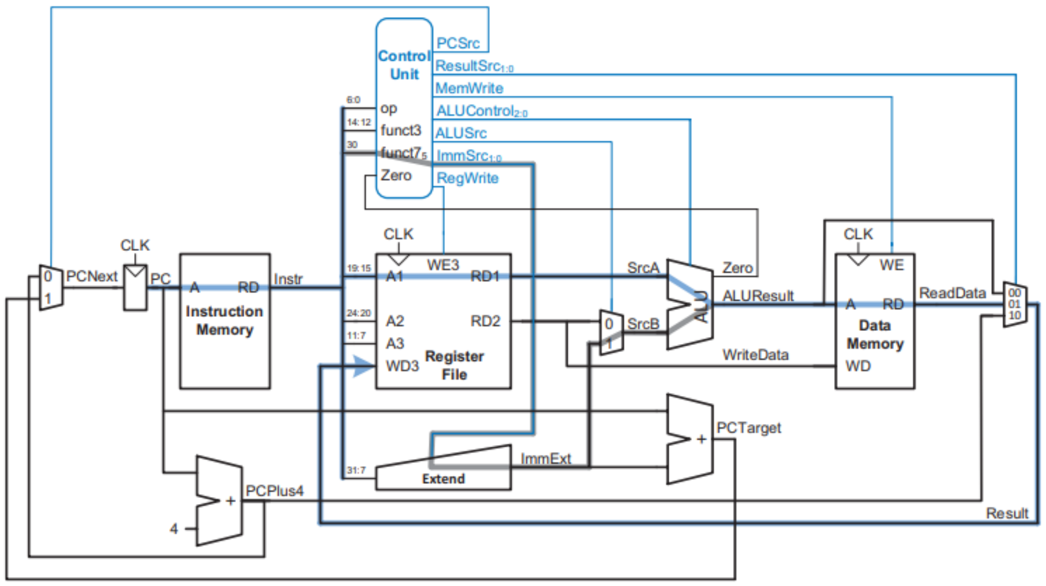


Figure 2: Top Level of CPU Implementation [5]

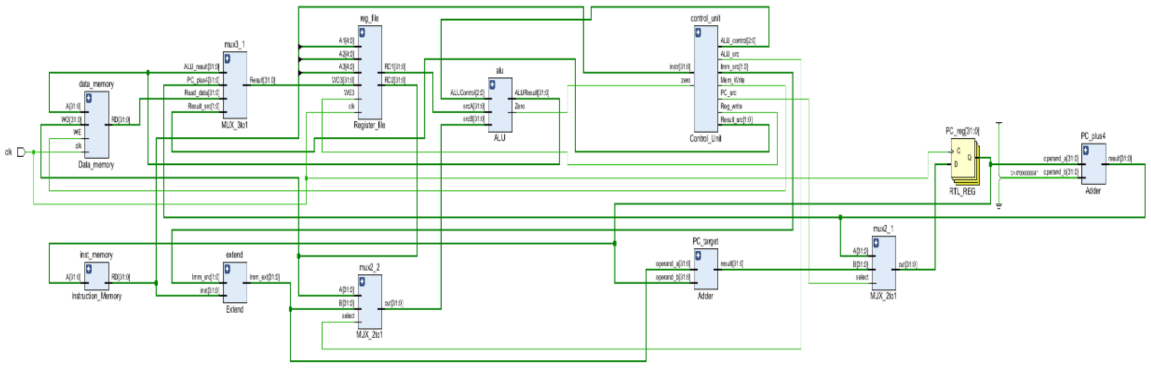


Figure 3: Schematic Top Level of CPU Implementation by Vivado

All the detail of blocks has been summary in Table 3: Modules Inside CPU with Details and overall architecture shown in figure above, the overall structure of the CPU and the functions of each module are noticeably clear. But this CPU needs more details to be specified. In terms of data memory, adder, multiplexers, and register files are quite easy to understand and very conventional. The special thing about this CPU is that it can analyse instructions to complete a variety of different operations, thanks to the Control Unit module.

The Control Unit consists of two parts, the ALU Decoder and Main Decoder. The overall structure of those modules shown as below. The figure 4 is drawn based on the *Digital Design and computer architecture: RISC-V edition*. The structure provided by the book is only suitable for a certain number of instructions. As the development of this project, the CPU has been updated to work with more instructions.

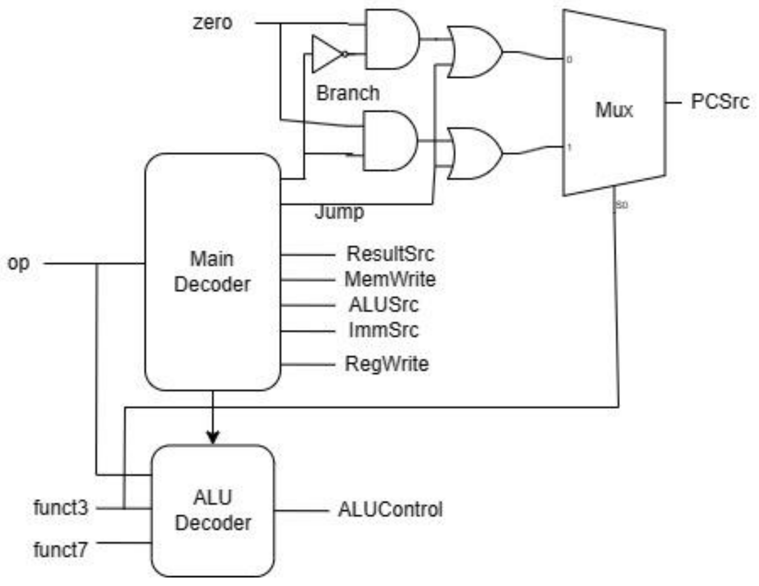


Figure 4: Updated Control Unit Structure [5]

As part of the control unit, the main decoder takes 6-bit opcode as input, output with control ports which listed in table 3. Inside the main decoder block, one truth table was relying on shown in Figure 13.

Similarly, the ALU decoder block has a truth table to help with, shown in Table 4. From Figure 1, depending on the instruction type, the index range of the extend section on the instruction will also be different. Therefore, the Extend module obtains the corresponding immediate value according to the instruction type. The identification method for Extend module also relies on the truth table as shown in Figure 14.

2.3 C++ programme & SystemVerilog HDL[2]

C++ is a very common programming language as an extension of the C language. C and C++ are both low-level languages, very flexible and closer to embedded systems. The reason for choosing C++ in this project is that the machine code it generates through the GCC compiler corresponds to RISC-V [4] (the RISC-V generated will have certain differences depending on the version of the compiler). In section 4.5 Performance evaluation and comparisonanalysed and compared in this project includes the comparison of the number of instructions generated by using C++ to implement PRESENT and the number of self-improvement RISC-V instructions.

SystemVerilog is an HDL that use for design, simulation, was standardise in IEEE 1800 at 2009. SystemVerilog as an extension of Verilog with more features for simulation and

[2] HDL - hardware description language

verification, such as Universal Verification Methodology (UVM). In this project, SystemVerilog will be used to implement and simulating of the CPU.

2.4 Vivado – Synthesis and Analysis of HDL

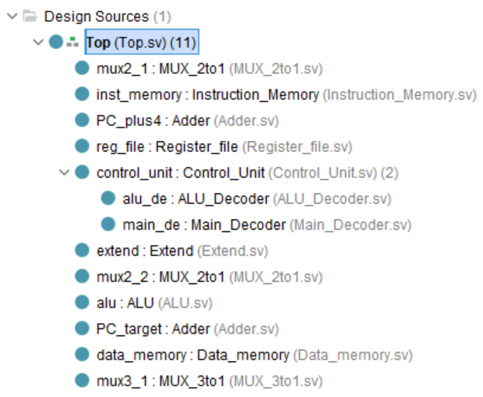
Vivado is a software that use for synthesis and analysis of HDL. It was first introduced in 2012, provide integrated design environment (IDE) that modelling the design. In this project, this software has been using for simulation and synthesis. From the simulation, it will show the behaviour of the chip and provided features that helps to debug. In synthesis, we can see how the design is performed, and it also provides predicted worst negative slack (WNS). From there the best clock cycle can be calculated [8]. With best clock period, predicted running time is also can be calculated. Figure 3: Schematic Top Level of CPU Implementation by Vivado as an example of Vivado RTL ANALYSIS feature.

Chapter 3 – RV32I Instruction Set Architecture Implementation and Testing

3.1 Implementation

The main components and overall structure diagram of the CPU are fully introduced in the above chapters. Following that, the CPU can be implemented. First, a default empty project was created name "RISC-V" in Vivado. And then created all the modules that mentioned in Table 3: Modules Inside CPU with Details. All the source code of each module are shown in Design Source. There also has a GitHub website with same content created while the project is developing [9].

A top-level design consists of block level design. In this project, a file call Top.sv connect all the sub-blocks by initial wires and regs. All the connections can refer to figure 6. It is worth noting that in this file, `(* DONT_TOUCH = "true|yes" *) (* keep = "true" *)` is not part of the design. It's because in the later analysis, Vivado's SYNTHESIS feature was used. SYNTHESIS will automatically optimize the design, like CPU design which will cause part of the design to be streamlined. In order to avoid automatic optimization of the design, the above code needs to be added [10]. The design hierarchy shown in figure below. Now, the Top.sv is set as top of the design with sign at front.

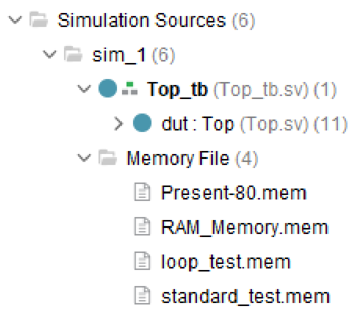


```
Design Sources (1)
  Top (Top.sv) (11)
    mux2_1 : MUX_2to1 (MUX_2to1.sv)
    inst_memory : Instruction_Memory (Instruction_Memory.sv)
    PC_plus4 : Adder (Adder.sv)
    reg_file : Register_file (Register_file.sv)
    control_unit : Control_Unit (Control_Unit.sv) (2)
      alu_de : ALU_Decoder (ALU_Decoder.sv)
      main_de : Main_Decoder (Main_Decoder.sv)
    extend : Extend (Extend.sv)
    mux2_2 : MUX_2to1 (MUX_2to1.sv)
    alu : ALU (ALU.sv)
    PC_target : Adder (Adder.sv)
    data_memory : Data_memory (Data_memory.sv)
    mux3_1 : MUX_3to1 (MUX_3to1.sv)
```

Figure 5: RISC-V Design Hierarchy in Vivado

3.2 Testing

Before running the test, the project needs to specify a testbench in the simulation file, which is called Top_tb.sv. The main purpose of this file is to provide a suitable test environment for top-level design. According to the code, it can be found that the file has clock (1-bit) input connected to the CPU and the clock period is set to 10ns. After that, the instruction memory block in the CPU needs to be loaded with a specific instruction set. The testbench has two instruction sets, which store in Standard_test.mem [5] and Loop_test.mem. Those memory files need to be created as memory file type that only use for simulation. The simulation files hierarchy shown in figure below.



```
Simulation Sources (6)
  sim_1 (6)
    Top_tb (Top_tb.sv) (1)
      dut : Top (Top.sv) (11)
    Memory File (4)
      Present-80.mem
      RAM_Memory.mem
      loop_test.mem
      standard_test.mem
```

Figure 6: RISC-V Simulation Hierarchy in Vivado

To loading these memory files, there is a function call `$readmemh()` that can rewrite specify block data in the design with external documents. There are two key parameters for this function, first parameter is the name of the file with double quote, second parameter is the location of the memory block. This function should be called in the initial block at the testbench, this will guarantee at every beginning of the test the block has load the data from the memory file. Additionally, there are stop conditions when the simulation has started. The always block will constantly check and finish the simulation if the CPU achieve the stop conditions. Once, conditions are meet, from the console will display simulation succeeded message.

3.2.1 Test from RISC-V test book [5]

This test is referred from *Digital Design and computer architecture: RISC-V edition* [5]. The purpose of this test is to combine some of the instructions covered in Table 2 in one test. Since the book illustrates the result after executing instructions, the registers should contain the same results after executing these instructions to ensure the accuracy of the CPU.

After running the test, select some important signals of blocks to the waveform to see, there are programme counter (PC), RD (reading data in instruction memory block) which show which instruction is executed, and whole register table. According to Figure 10, rename the register table to make it easier to read the waveform. In order to distinguish instructions more clearly, blue markers are added between each instruction. The overall waveform of this test shown in Figure 20. To clearly shows the data, the figure only displays beginning section of waveform as example.

As expected, the terminal displayed the message that the simulation was successful. The same result obtained from the register table, and it can be verified that each command is running correctly. Taking the first instruction as an example, the machine code of the instruction is 0x00500113 (addi x2, x0, 5), which simply adds x0 to the number 5 and stores the result in x2. As shown in the waveform, the yellow marker locates at the end this instruction. The value on x2(sp) has changed to 5. It was verified that each instruction ran successfully according to the above checking method.

To better understand how dose the instructions achieve step by step, an equivalent C++ code was created name Standard_test.cpp in Appendix B. From this programme, each register with different value while the programme is running can be easier capture.

3.2.2 Loop Test

Like the previous test, but this test focus on testing the blt (branch less than), jal (jump) instruction. These two instructions can be combined to form a loop structure. To ensure those instructions work properly will help later implementation of PRESENT-80. Similar to the previous step, the specific memory file needs to be loaded, name Loop_test.mem. The waveform for this test can be found in the Figure 21. The markers in the waveform are located at the start of each loop. The values change after each loop is recorded. Compared to the changes with the corresponding C++ code (Loop_test.cpp), the changes are matched, and behavior of the blocks are as expected.

Chapter 4 - PRESENT-80 implementation on RV32I ISA

4.1 PRESENT-80

Compared with other encryption modules, PRESENT provides fewer gate equivalents 1300 – 2600 (GE) [7]. This encryption algorithm can encrypt 64-bit data with key. Key length can be 80-bit or 128-bit. The key length in this project is 80-bit. PRENSENT's encryption process requires 3 layers, the first is addRoundKey, substitution-box (S-box), and permutation layer (p-layer).

The addRoundKey is providing 64-bit output key[3] in overall 80-bit key, to join xor gate with 64-bit state. The result will go to S-box layer. The S-box layer is a 4-bit converter. For example, 4-bit state with value “2” input to S-box, the output will be “6”, more details shown in Figure 11: S-box Details. As the size of the state is sixty-four bits, it needed sixteen s-box converters to line up together.

Permutation layer is a 64-bits conversion like S-box layer, but instead of converting values, P-layer relocated bit position. The detail of P-layer can be found in Figure 12. To fully understand of this figure, e.g., if a bit in state at index “5” will relocated to index “17”.

The plaint text needs to go through these three layers 31 times. The 80-bit key will also be updated in each round after addRoundKey. To update the key, firstly it rotates circular shift to left 61 bits. Then the leftmost of 4 bits will be updated by S-box. Last, the 5 bits at key index 15-19 will be updated by its value xor gate with round counter which contains value from 1 to 31. Overall process of PRESENT-80 shown in figure 7 below.

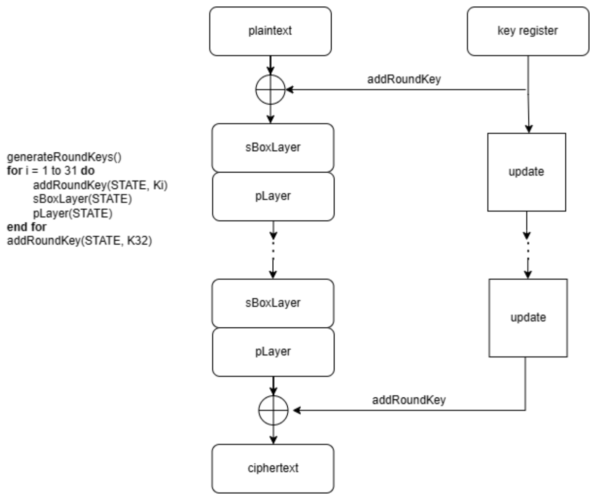


Figure 7: PRESENT-80 Algorithmic Description [7]

[3] Output key is assigned to key [79:16]

4.2 RV32I Instruction Set used for PRESENT-80 implementation

Implementing PRESENT-80 uses the existing RISC-V instruction set. According to the structure of the encryption algorithm, encryption can be divided into three general steps (S-box, p-layer, update key). This foundation structure will be loop through 31 times and finally generate the ciphertext.

4.2.1 S-box

The function of S-box is converting input 4-bit data into corresponding 4-bit data base on the Figure 11: S-box Details. The S-box table is pre-stored in the RAM in the data memory block, and the starting address is 0x0. To load the transformation details to RAM requires `$readmemh()`, using RAM_Memory.mem[4] as memory file. For PRESEN-80, 16 S-boxes are used to convert the 64-bit state to the new 64-bit or 16 conversions at 4-bit intervals. Since this CPU uses the RV32I structure, the registers and RAM store data as 32-bit a unit. Therefore, the 64-bit state will be stored in two 32-bit stores. Registers a0 and a1 will be used as the storage addresses of state high and state low, and t0 and t1 are the result addresses of new state high section and new state low section.

Firstly, register t0 and t1 need to be cleared to ensure that the new state stores correct information. Set the constant 31 to the t3 register. This constant stands for the largest index of the register will be use as part of the loop stop condition. Initialize the bit counter to 0 use register t4. The main purpose of this register is to record the progress of the conversion, in range from 0 to 31.

Upon finishing the preliminary setup, the program progresses to the conversion phase, which is segmented into two distinct loops: one for converting the high state and another for the low state. These loops share a structural framework but use on different register addresses. The termination of each loop is governed by the `blt` instruction, which serves as a loop-ending condition. The loop concludes when the highest index (t3) falls below the bit counter (t4).

In the next steps for the high state conversion, the a0 register, which has the high state, undergoes a rightward shift based on the bit counter (t4), with the resultant value stored in the temporary register t2. This operation aims to isolate the rightmost 4 bits of t2 for conversion. Following this, the *andi* instruction is applied to mask the t2 register, keeping only its rightmost 4 bits, with t2 holding the resultant value. Given that RISC-V is byte-addressable, with the last two bits of a word address always being zero, the t2 data, combined with the lw instruction, is used to retrieve the corresponding value from the s-box table, with the outcome stored back in t2.

[4] This memory file has S-box, p-layer, key, and state data.

The value in t2 is then shifted back in alignment with the bit counter (t4) and integrated into the high state (t0) through the *or* instruction. Subsequently, the bit counter is incremented by 4, and the program employs the *jal* instruction to loop back to the start, thereby iterating through the conversion process for both the high and low states in a systematic manner. This approach exemplifies a structured and efficient method for state conversion within the specified encryption algorithm.

When the bit counter(t4) value reaches 31, the loop will terminate and enter the state low transition. Before the loop starts, the program will reset the bit counter (t4) to 0. The new loop is same as the earlier loop, but the conversion target becomes state low. The overall conversion idea and register usage can be understood based on the following picture. The overall instructions can be found in S-box section of Present-80.mem.

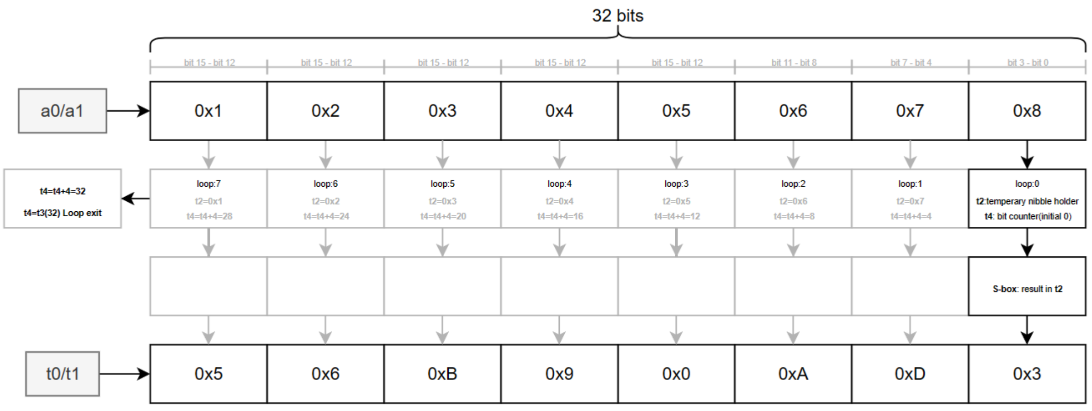


Figure 8: S-box Conversion Detail and Registers Usage

4.2.2 P-layer

The permutation layer moves bits inside the 64-bit state. The exchange detail table can be found in Figure 12. After s-box, the new state is stored in t0 and t1, so a0 and a1 need to be used as storage addresses of new data in the p-layer. First, clean these two registers, initialize the bit counter (t4) to 0, and set t3 as the maximum index 63. Clear a3 to prepare to store bit new location.

The implementation of p-layer mainly relies on a cycle of 63 reincarnations. The loop will traverse the entire state, based on the table to find the new location of the current bit, use instruction blt to decide whether the new location is in high state or low state, and then store the bit, and then continue looping until the last location. Detailed steps are as follows.

After entering the loop, need to decide whether to convert to High state or Low state. The state being converted will be stored in t0. Shift t0 to the right. The number of shifts is based on the data stored in the bit counter. This step is to keep the rightmost bit of t0 as the target of conversion. Then use instruction andi to mask t0 so that t0 only stores the information of the conversion target bit. Then use the bit counter (t4) and the information in RAM to find the new location of the bit. Use the blt command to decide whether the new location is in the high or low state. If it is stored in the low state, directly shift the bit to the left to the new location, and then integrate it with the low state. If the location is in state high, you need to subtract 32 from the location data, and then use this location to store it in state high. The following instructions will jump to the beginning of the loop to continue a new loop. The detailed flow chart is shown below.

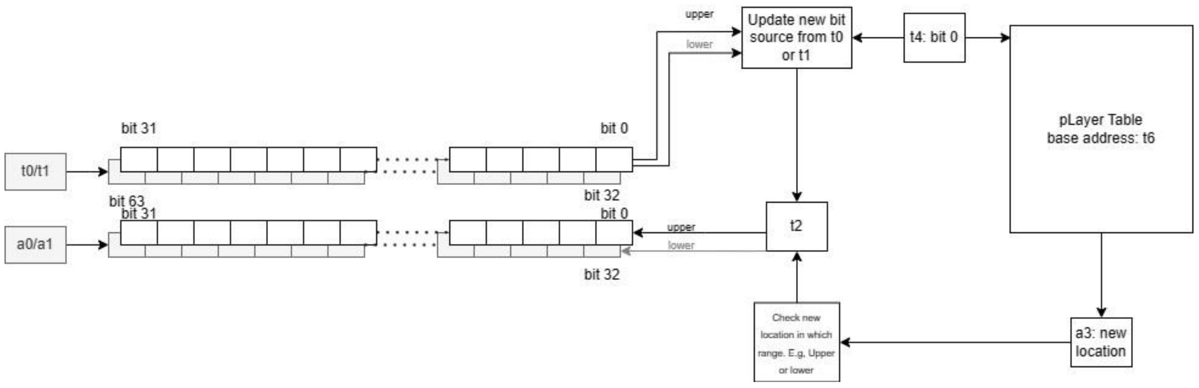

upper
lower
Update new bit source from t0 or t1
t4: bit 0
pLayer Table base address: t6
bit 31
bit 0
t0/t1
bit 63
bit 32
bit 31
bit 0
a0/a1
upper
lower
t2
bit 32
Check new location in which range. E.g. Upper or lower
a3: new location

Figure 9: P-layer Implementation Details

4.2.3 update key

Since the PRESENT of this project uses an 80-bit key, and the maximum storage unit is 32 bits, 3 registers are needed to store the complete key information. s2 stores key [79:48], s3 stores key [47:32], and s4 stores key [31:0]. The conversion process requires the storage of three temporary keys, which correspond to the earlier sequence and require the use of registers t2, t3, and t4.

According to Figure 15, in relation to remapping the new key. t2 consist of s4 [18:0] and s2 [31:20]; t3 holds s2 [21:3]; t4 holds s2 [2:0], s3 [15:0], and s4 [31:19]. Before storing the shift results, need apply xor for s3 [6:2] and loop counter, and store the result in the same position. To update the t2 register, shift s4 and s2 to correct position and mask with non-interest bits, then merge them together with instruction or. Also, apply s-box to the left most 4-bit same as s-box layer. The t3 is easy to deal with, as only shift s2 to certain position and apply mask. Updating t4 is same as t2, shift s2, s3, s4 with correct index and mask with useless bits, merge them all with instruction or. The key update process can be summarized as Figure 15.

With the above three major steps, still need to be linked together and connected through a large loop. At the beginning of each cycle, the key needs to be selected as 64-bit as the output, and then the state and key are OR gated. The registers involved in the entire process and their purpose at each stage are summarized in the Figure 16.

4.2.4 Testing

Load memory file (Present-80.mem) and RAM information (RAM_Memory.mem) in the specified block. Use Vivado to simulate the CPU the waveform as shown in the Figure 22: PRESENT-80 Waveform. The highlighted signal (s5) in the figure stands for the counter of each cycle. As expected, the entire program completed 31 cycles, and the test successfully encrypted the original text. By seeing the inside of each cycle inside blue markers, as shown in the Figure 23: PRESENT-80 Waveform Selection Range. The distinct stages between each blue mark are s-box, p-layer, and key update. They all work as expected. After that, I changed the state and key in the RAM file and conducted more tests, and the test results were still same as expectation.

4.3 C++ implementation of PRESENT-80

The initial sections of the PRESENT_80.cpp import necessary libraries for input/output operations, vector usage, and fixed-width integer types, setting the stage for the cipher's operations. Two crucial components as arrays, the details of substitution box (S-box) and the permutation box (P-box), are defined globally.

The `keySchedule` function is tasked with generating the round keys from the initial 80-bit key, split into a 64-bit key Upper for output and a 16-bit key Lower. It proceeds to create 32 round keys, incorporating operations like bitwise rotations and substitutions using the S-box. This key schedule is a pivotal part of the overall code. Finally, the `presentEncrypt` function embodies the encryption process, accepting a 64-bit plaintext and the round keys as inputs. It performs 31 rounds of substitution (via the S-box), permutation (according to the P-box), and round key addition, followed by a final procedure to xor current state with the round key.

Inside the main function, an example test with state and key are all 0. After executing function `presentEncrypt`, manually checked the cipher text is same as expected. The programme also successfully passes other customized inputs.

4.4 SystemVerilog HDL implementation of PRESENT-80

As part of the comparison, the project needs to use SystemVerliog to implement PRESEN-80. Still using the same encryption algorithm, the design part consists of three parts: S-box (64-bit), p-layer, and KeySchedule. The composition of S-box (64-bit) is quite simple, consisting of 16 small S-boxes (4-bit). The p-layer uses hardcode to assign each bit. In KeySchedule, there is an s-box (4-bit) and assign statements for shifting. When the next clock cycle starts, KeySchedule will output a new key. In the top-level design, connect the above components together.

All source files are stored in Appendix B: Design Source. Compared with the CPU simulation file, the simulation file of this design will be simpler. The top design only needs to be input the original text, key, and clock, and the ciphertext is used as the output. In the initialization block, the customized original text and key data are included. The simulation files can be found in the Top.sv. The waveform of this design shown in Figure 24: HDL PRESENT-80 Waveform.

4.5 Performance evaluation and comparison

With the help of Vivado feature SYNTHESIS, the Design Timing Summary can be found in below for PRESENT-80 and CPU implementation. From this timing report, the largest clock frequency can be found by the instruction [8].

Before running synthesis feature, inside both Vivado projects, we need specify clock constraints shown in Figure 19: Constraints for Designs. As there might occur errors in Timing Report in Vivado if without the constrain. So, the overall performance of these two design summaries in below table.

Design	Max Freq/Min Period	Total cycles	Total time
CPU	5.413 ns	33,000	178,629 ns
PRESENT-80	1.942 ns	31	60.202 ns

Table 1: Comparison Runtime of CPU and PRESENT-80

From above table, it can be found that the time it takes for the CPU to process one encrypted data at the maximum frequency is about three thousand times that of PRESENT-80. In addition, the runtime test is performed on the equivalent C++ program. Using `std::chrono` provided by the C++11 version, the time required to run the program can be obtained [11]. After multiple rounds of testing, the average runtime was 11 microseconds, which is 11,000 nanoseconds. Compared with the time consumed by the above CPU, the CPU's running time is 1.6 times longer.

In addition, the number of instructions can also be used as an area for comparison. There are 550 instructions by converting C++ source code into corresponding RISC-V RV32I instructions [6]. Custom instructions require 97 instructions to implement PRESENT-80 base on Present-80.mem.

Chapter 5 - Results and Discussion

This chapter presents the results obtained from the implementation of the RISC-V RV32I architecture and the performance analysis of the PRESENT-80 cipher on this platform. The outcomes are discussed in the context of the objectives outlined in the introduction and subsequent design and implementation chapters.

5.1 Implementation Results

The implementation of the RISC-V RV32I architecture, as detailed in previous chapters, was successfully achieved. The custom CPU was designed to support a subset of the RV32I instruction set, crucial for the operation of the PRESENT-80 encryption algorithm. The architecture was synthesized and tested using the Vivado toolset, demonstrating functional correctness through various simulation tests.

The implementation of the PRESENT-80 ciphers on the RISC-V architecture involved both software (C++) and hardware (SystemVerilog) approaches. These implementations were aimed at encrypting 64-bit plaintext using an 80-bit key, utilizing a subset of RV32I instructions specifically selected and optimized for this purpose.

The performance analysis compared the execution time of the PRESENT-80 encryption in three environments: a custom CPU running the RISC-V instruction set, a direct SystemVerilog hardware implementation, and a C++ software implementation.

- Custom CPU Implementation: The execution on the custom CPU demonstrated the functional capability of the RISC-V architecture to run cryptographic algorithms efficiently. However, due to the single-cycle nature of the CPU and scaler of overall design, the execution time was significantly longer compared to the hardware implementation.
- SystemVerilog Hardware Implementation: The direct hardware implementation of PRESENT-80 with the fastest execution time, benefiting from parallel processing capabilities and hardware optimization. This implementation highlighted the advantages of executing cryptographic algorithms directly on hardware for maximum efficiency.
- C++ Software Implementation: The software implementation provided a baseline for performance comparison and demonstrated the versatility and ease of implementing cryptographic algorithms in a high-level programming language. While not as fast as hardware implementations, it offered a more flexible and easily modifiable platform for algorithm development and testing.

5.2 Discussion

The results underline the efficiency and flexibility of the RISC-V RV32I architecture in implementing cryptographic algorithms. The architecture's simplicity and modularity facilitated the adaptation of the PRESENT-80 cipher in both software and hardware contexts.

However, the performance analysis clearly indicates the superiority of hardware implementations for cryptographic applications, with direct SystemVerilog implementations outperforming software counterparts in terms of speed. This outcome supports the argument for specialized cryptographic hardware, especially in environments where encryption speed is critical.

Additionally, the comparison between the custom CPU and software implementations provides valuable insights into the trade-offs between flexibility and performance. While the software approach offers ease of development and modification, dedicated hardware can significantly enhance performance, a crucial factor for real-time applications.

Chapter 6 – Ethics

This chapter delves into the ethical considerations associated with the implementation and analysis of the RISC-V RV32I Architecture. Given the expanding role of RISC-V in the technological landscape, it's imperative to address its ethical implications. Our exploration spans the open-source paradigm, potential societal impacts, and security considerations, guided by the principles of beneficence, nonmaleficence, and justice

6.1 Open-Source Contribution and Collaboration

RISC-V's open-source nature embodies a commitment to collective progress in the field of computing architectures. This openness fosters a collaborative environment where knowledge is shared freely, contributing to the global community's growth. However, it also raises ethical questions about the fair distribution of advancements and the responsibility of contributors to ensure the integrity and reliability of shared knowledge.

Recommendation: Contributors should adhere to lofty standards of transparency and accountability, ensuring that modifications or enhancements are meticulously documented and peer reviewed.

6.2 Security and Privacy

The ubiquity of computing technologies in personal and professional spheres heightens the importance of security and privacy. The ethical design and implementation of RISC-V architectures must prioritize these aspects to protect against malicious exploitation, especially given the architecture's role in potentially sensitive applications.

Recommendation: Adopt a security-by-design approach, integrating robust encryption and privacy-preserving features from the outset. Continuous vulnerability assessments and updates are essential to safeguard users' security and privacy.

Chapter 7 - Conclusions and Further Research

7.1 Conclusions

This project embarked on the ambitious journey of implementing and analysing the performance of the RISC-V RV32I architecture, focusing on a custom CPU to accommodate the PRESENT-80 encryption algorithm. The initiative aimed at exploring the architectural simplicity, efficiency, and scalability inherent to the RISC-V framework, with a particular emphasis on the encryption application. Key findings from this exploration include:

- **Implementation Success**: The project successfully designed and implemented a simplified RISC-V RV32I based CPU, showing the architecture's adaptability and potential for educational and research applications. The CPU could execute a subset of the RV32I instruction set, tailored to support the PRESENT-80 encryption algorithm.
- **Performance Insights**: Through detailed synthesis and analysis using the Vivado platform, the study provided empirical data on the CPU's performance metrics. The insights included clock cycle efficiency, instruction execution times, and the comparative analysis of the hardware-implemented algorithm versus its software counterpart in C++.
- **Encryption Efficiency**: Implementing the PRESENT-80 encryption algorithm within the RISC-V RV32I architecture illustrated the feasibility of using RISC-V for encryption tasks. The project highlighted the algorithm's low resource requirements and showed the practicality of hardware-based encryption for lightweight, secure communication systems.
- **Educational Value**: By documenting the design and implementation process, this project contributes valuable educational resources for understanding RISC-V architecture and encryption technologies. The detailed breakdown of the CPU components and encryption algorithm implementation serves as a practical guide for students and researchers alike.

7.2 Future Research

The exploration of the RISC-V RV32I architecture in the context of encryption presents several avenues for future research:

1. **Architectural Extensions**: Future work could explore the integration of additional RISC-V extensions, such as the multiplication and atomic instruction sets, to enhance the CPU's capabilities and performance for a broader range of applications.
2. **Security Enhancements**: Investigating the incorporation of security-focused features, such as secure boot, hardware-based random number generation, and instruction traceability, could fortify the architecture's resilience against emerging cyber threats.
3. **Power Efficiency**: Power consumption analysis and optimization remain critical for embedded and IoT devices. Future studies could delve into power-saving techniques and the design of low-power RISC-V implementations.

4. **Comparative Analysis**: Expanding the comparative analysis to include other encryption algorithms and their performance on the RISC-V architecture would offer deeper insights into the platform's versatility and efficiency in cryptographic applications.

References

[1] Bogdanov, A. *et al.* (no date) ‘Present: An ultra-lightweight block cipher’, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pp. 450–466. doi:10.1007/978-3-540-74735-2_31.

[2] Waterman, A. *et al.* (2014) *The RISC-V instruction set manual. volume 1: User-level ISA, version 2.0* [Preprint]. doi:10.21236/ada605735.

[3] Kanter, D., 2016. RISC-V offers simple, modular ISA. *Microprocessor Report*, *1*, pp.1-5.

[4] Five Embeddev (2019) *RISC-V compile targets, GCC, Five EmbedDev*. Available at: https://five-embeddev.com/toolchain/2019/06/26/gcc-targets/ (Accessed: 19 March 2024).

[5] Harris, S.L. and Harris, D.M. (2022) *Digital Design and computer architecture: RISC-V edition*. Cambridge, MA: Morgan Kaufmann.

[6] Godbolt, M. (no date) *Compiler explorer, Compiler Explorer*. Available at: https://godbolt.org/ (Accessed: 30 March 2024).

[7] Bogdanov, A. *et al.* (no date a) ‘Present: An ultra-lightweight block cipher’, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pp. 450–466. doi:10.1007/978-3-540-74735-2_31.

[8] *Vivado: Finding the ‘Maximal Frequency’ after synthesis* (no date) *01signal*. Available at: https://www.01signal.com/vendor-specific/xilinx/vivado-minimal-period/ (Accessed: 26 February 2024).

[9] Miao, Z. (2024) *Zhuang-Alfie/RISC-V-32I, GitHub*. Available at: https://github.com/Zhuang-Alfie/RISC-V-32I (Accessed: 27 January 2024).

[10] (No date) *AMD Customer Community*. Available at: https://support.xilinx.com/s/article/54778?language=en_US (Accessed: 27 February 2024).

[11] GfG (2023) *Measure execution time of a function in C++, GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/measure-execution-time-function-cpp/ (Accessed: 30 March 2024).

[12] Davis, L.@ U. (no date) *Rvcodec.js · RISC-V instruction encoder/decoder*. Available at: https://luplab.gitlab.io/rvcodecjs/ (Accessed: 31 January 2024).

Appendix A: Table and Figures

Table 2: Implemented Instructions Set

Instruction	Type	Example Usage	Description
addi	I-type	addi s0, s1, 0x5	s0 = s1 + 0x5
lw	I-type	lw s0, 0x5(s1)	Load word to s0 at address s1 with offset 0x5
srli	I-type	srli s0, s1, 0x5	s0 = s1 << 5
slli	I-type	slli s0, s1, 0x5	s0 = s1 >> 5
or	R-type	or s0, s1, s2	s0 = s1 | s2
xor	R-type	xor s0, s1, s2	s0 = s1 ^ s2
blt	B-type	blt s0, s1, 0x5	Branch if s0 < s1, with pc[5] increment 0x5
sll	B-type	sll s0, s1, s2	s0 = s1 >> s2
srl	B-type	srl s0, s1, s2	s0 = s1 << s2
jal	J-type	jal s0, 0x5	Jump to instruction: pc += (s0 + 0x5)
sub	R-type	sub s0, s1, s2	s0 = s1 - s2

Table 3: Modules Inside CPU with Details

Module Name	Description	Input Port(s)	Output Port(s)
Instruction Memory	Instruction Memory handles holding the binary instructions that make up a program. When the processor is running a program, it sequentially reads instructions from this memory	**A** – 32-bit address	**RD** – 32-bit instruction
Register File	Register File is the central hub for fast data storage (Register Table shown in figure 2) and access during instruction execution	**A1** – 5-bit resource 1 **A2** – 5-bit resource 2 **A3** – 5-bit write data address **WD3** – 32-bit write data **clk** – 1-bit clock **WE3** – 1-bit write enable bit	**RD1** – 32-bit read data 1 **RD2** – 32-bit read data 2
Control Unit	This module holds and connects two sub-modules which are ALU decoder and main decoder. It distributed the Instruction data to those two modules, generate result to further control other modules. More details of those modules in their description	**instr** – 32-bit instruction **zero** – 1-bit zero flag	**PC_src** – 1-bit PC source control **Reg_write** – 1-bit register write enable **Imm_src** – 2-bit immediate source control **ALU_src** – 1-bit ALU source control **ALU_control** – 3-bit ALU control **Mem_Write** – 1-bit memory write enable **Result_src** – 2-bit result source control

[5] Pc: programme counter, one memory block in CPU store currently instruction index.

ALU Decoder	The ALU Decoder is a critical part in a RISC-V RV32I based single-cycle processor, enabling the flexible and dynamic operation of the ALU by translating instruction codes into specific ALU control signals.	**funct3** – instruction [14:12] **funct7** – instruction [30] **op5** – instruction [5] **ALU_op** – 2-bit ALU operation control	**ALU_control** – 3-bit ALU control
Main Decoder	This part is essential for the functioning of the processor, as it translates the binary instruction codes into a set of actionable commands that control the data path and the operation of the processor's various elements, such as the ALU, multiplexers, register file, and memory access units.	**op** – 7-bit opcode from instruction	**Reg_write** – register write enable **Imm_src** - 2-bit immediate source control **ALU_src** - 1-bit ALU source control **Mem_Write** – 1-bit memory write enable **Result_src** – 2-bit result source control **Branch** – branch flag **ALU_op** - 2-bit ALU operation control **Jump** – Jump flag
ALU	Arithmetic Logic Unit (ALU)is responsible for carrying out operations such as addition, subtraction, logical operations (AND, OR, XOR, NOT), and shift operations, which are fundamental to the execution of various instruction types within the RISC-V RV32I architecture, including arithmetic, logical, control, and data transfer instructions	**scrA** – 32-bit source A **scrB** – 32-bit source B **ALUControl** – 3-bit ALU control	**zero** – zero flag **ALUResult** – 32-bit ALU result
Data Memory	The Data Memory part serves as the primary storage area for data used and produced by the program being executed.	**A** – 32-bit reading address **WD** – writing address **clk** – clock **WE** – write enable flag	**RD** – reading data
Extend	The immediate value extender is a key part in a RISC-V RV32I based single-cycle processor, ensuring that immediate values embedded within instructions are correctly sign-extended to 32 bits.	**inst** – 32-bit instruction **Imm_src** – 2-bit immediate source	**Imm_ext** – 32-bit immediate extended
MUX 2to1	multiplexer (mux) 2-to-1 is a digital switch that allows one of two input signals to be selected and sent to the output, based on the value of a control signal.	**A -** 32-bit data A **B** – 32-bit data B **select** – 1-bit control signal	**out** – 32-bit output
MUX 3to1	multiplexer (mux) 3-to-1 is a digital switch that allows one of three input signals to be selected and sent to the output, based on the value of a control signal.	**ALU_result** – 32-bit ALU result **Read_data** – 32-bit reading data **PC_plus4** – 32-bit result of PC add 0x4 **Result_src** – 2-bit result control signal	**Result** – 32-bit result
Adder	adder is a circuit to perform the arithmetic addition of binary numbers	**operand_a** – 32-bit input A **operand_a** – 32-bit input B	**result** – 32-bit result

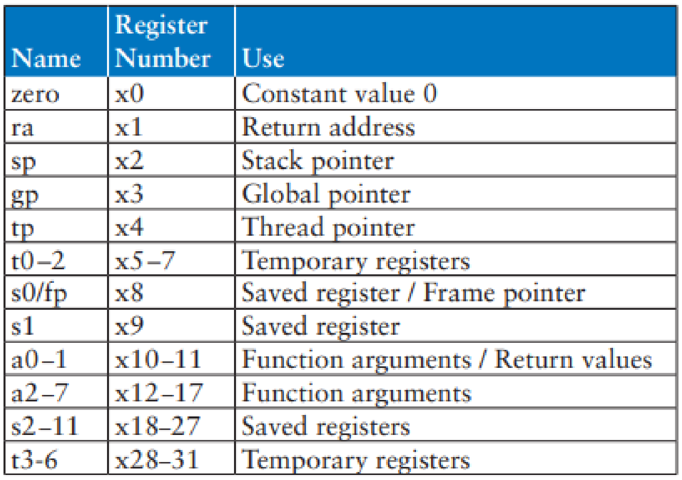
Table 4: ALU Decoder Truth Table

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000(add)	lw, sw
01	000	x	001(sub)	beq
	100	x	101(slt)	blt
10	000	00,01,10	000(add)	add
	000	11	001(sub)	sub
	001	x	100(sll)	sll
	010	x	101(bit operation)	slt
	100	x	111(xor)	xor
	101	x	110(srl)	srl
	110	x	011(srl)	or
	111	x	010(and)	and

Table 5: Full Set of the RV32I Instructions

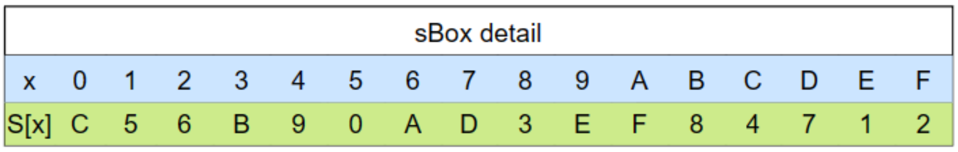
Instruction	**Type**	**Example Usage**	**Description**
addi	I-type	addi rd, rs1, imm	rd = rs1 + imm (Add immediate)
lw	I-type	lw rd, offset(rs1)	Load word from address **rs1 + offset** to **rd**
srli	I-type	srli rd, rs1, imm	rd = rs1 >> imm (Logical right shift immediate)
slli	I-type	slli rd, rs1, imm	rd = rs1 << imm (Logical left shift immediate)
andi	I-type	andi rd, rs1, imm	rd = rs1 & imm (AND immediate)
ori	I-type	ori rd, rs1, imm	rd = rs1
xori	I-type	xori rd, rs1, imm	rd = rs1 ^ imm (XOR immediate)
slti	I-type	slti rd, rs1, imm	Set **rd** to 1 if **rs1 < imm** (signed), else 0
sltiu	I-type	sltiu rd, rs1, imm	Set **rd** to 1 if **rs1 < imm** (unsigned), else 0
add	R-type	add rd, rs1, rs2	rd = rs1 + rs2 (Add)
sub	R-type	sub rd, rs1, rs2	rd = rs1 - rs2 (Subtract)
sll	R-type	sll rd, rs1, rs2	rd = rs1 << rs2 (Shift left logical)
slt	R-type	slt rd, rs1, rs2	Set **rd** to 1 if **rs1 < rs2** (signed), else 0
sltu	R-type	sltu rd, rs1, rs2	Set **rd** to 1 if **rs1 < rs2** (unsigned), else 0
xor	R-type	xor rd, rs1, rs2	rd = rs1 ^ rs2 (XOR)
srl	R-type	srl rd, rs1, rs2	rd = rs1 >> rs2 (Shift right logical)
sra	R-type	sra rd, rs1, rs2	rd = rs1 >> rs2 (Shift right arithmetic)
or	R-type	or rd, rs1, rs2	rd = rs1
and	R-type	and rd, rs1, rs2	rd = rs1 & rs2 (AND)
beq	B-type	beq rs1, rs2, offset	Branch to PC + offset if **rs1 == rs2**
bne	B-type	bne rs1, rs2, offset	Branch to PC + offset if **rs1 != rs2**

blt	B-type	blt rs1, rs2, offset	Branch to PC + offset if **rs1** < **rs2** (signed)
bge	B-type	bge rs1, rs2, offset	Branch to PC + offset if **rs1** >= **rs2** (signed)
bltu	B-type	bltu rs1, rs2, offset	Branch to PC + offset if **rs1** < **rs2** (unsigned)
bgeu	B-type	bgeu rs1, rs2, offset	Branch to PC + offset if **rs1** >= **rs2** (unsigned)
jal	J-type	jal rd, offset	Jump to PC + offset and store return address in **rd**
jalr	I-type	jalr rd, rs1, offset	Jump to **rs1 + offset** and store return address in **rd**
sw	S-type	sw rs2, offset(rs1)	Store word from **rs2** to address **rs1 + offset**
lui	U-type	lui rd, imm	Load upper immediate to **rd**
auipc	U-type	auipc rd, offset	Add upper immediate to PC and store result in **rd**



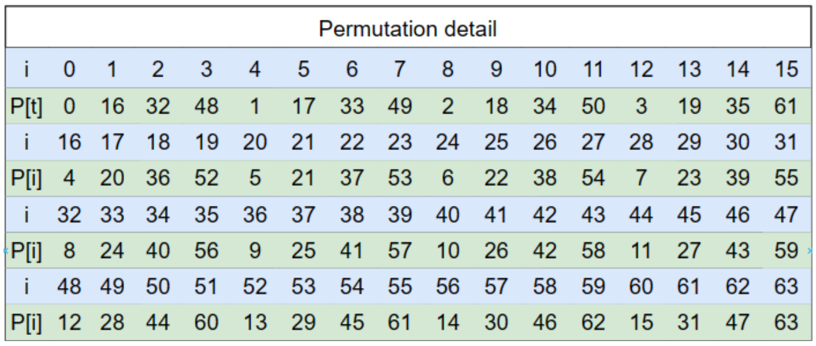
Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

Figure 10: Register Table [5]



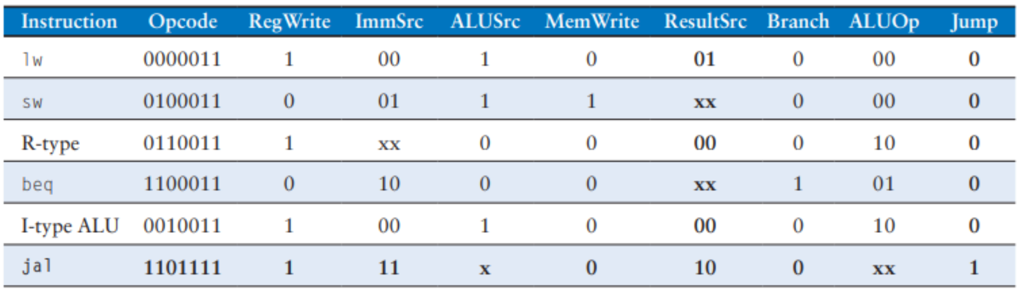
sBox detail																
x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Figure 11: S-box Details



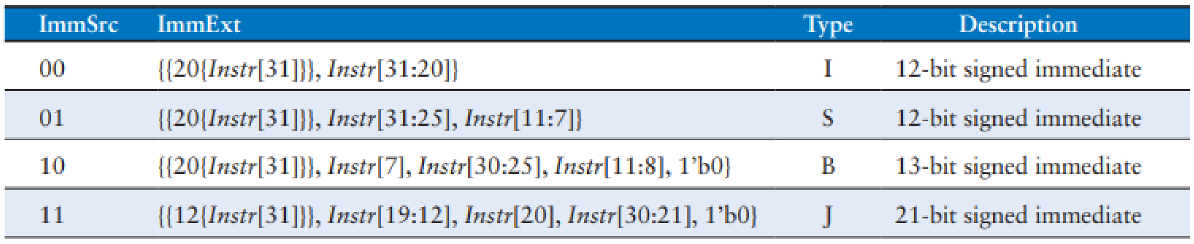
Permutation detail																
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P[t]	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	61
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
P[i]	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
P[i]	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
P[i]	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Figure 12: Permutation Layer Detail



Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
lw	0000011	1	00	1	0	01	0	00	0
sw	0100011	0	01	1	1	xx	0	00	0
R-type	0110011	1	xx	0	0	00	0	10	0
beq	1100011	0	10	0	0	xx	1	01	0
I-type ALU	0010011	1	00	1	0	00	0	10	0
jal	1101111	1	11	x	0	10	0	xx	1

Figure 13: Main Decoder Truth Table [5]



ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1’b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1’b0}	J	21-bit signed immediate

Figure 14: Extension Truth Table [5]

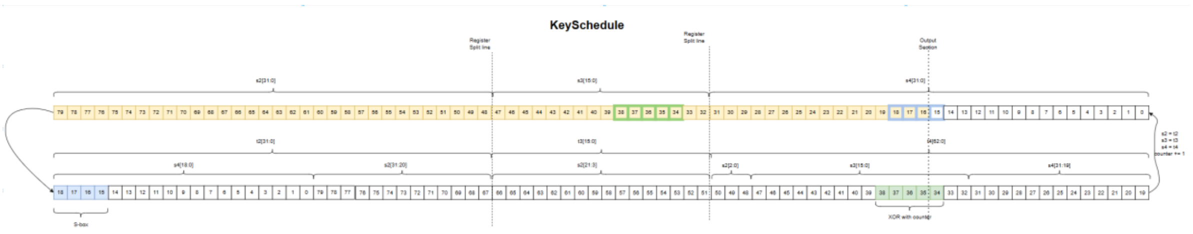
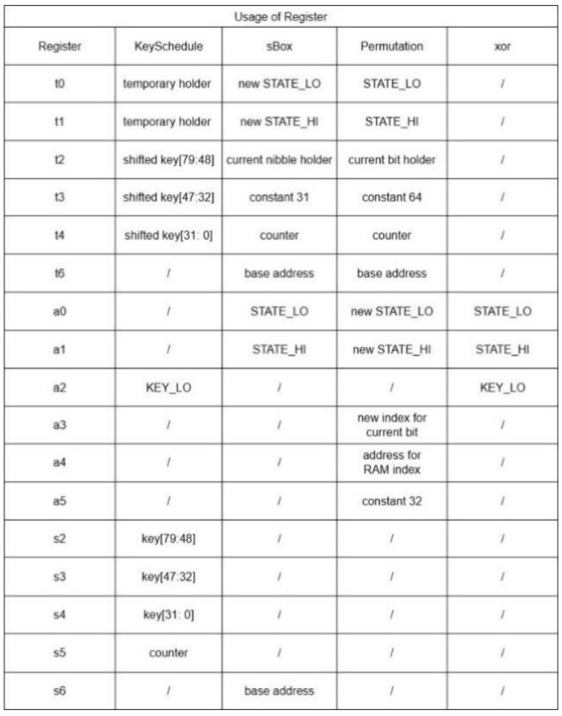
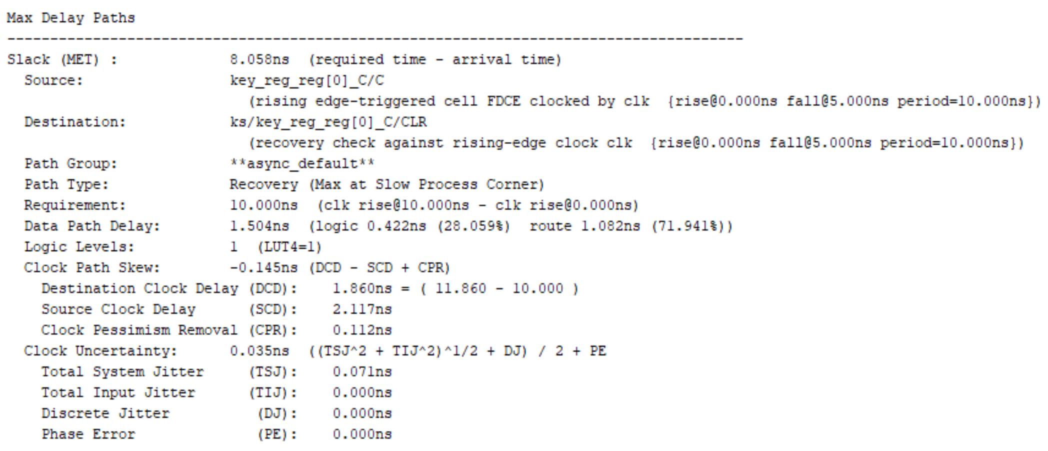


Figure 15: Key Update Remap Details



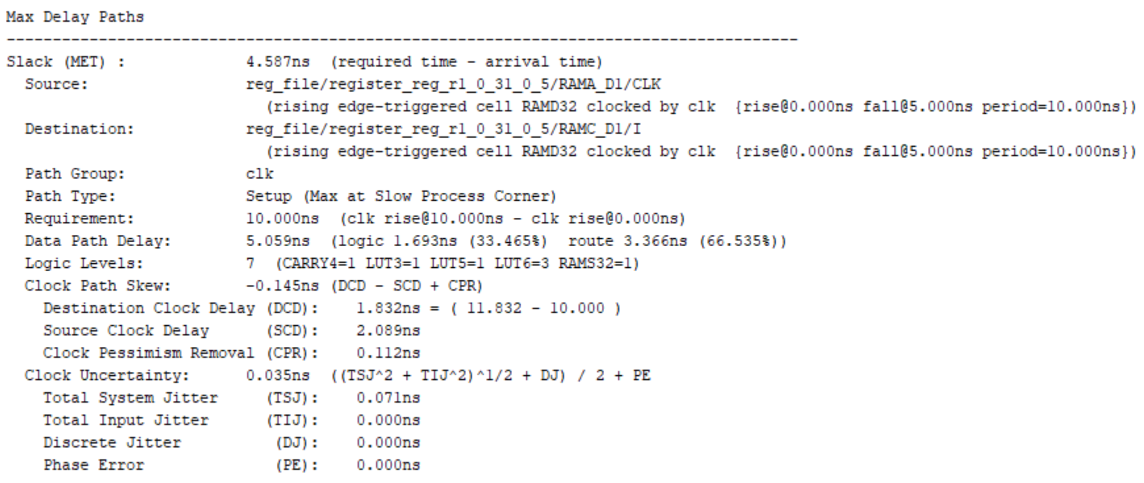
Usage of Register				
Register	KeySchedule	sBox	Permutation	xor
t0	temporary holder	new STATE_LO	STATE_LO	/
t1	temporary holder	new STATE_HI	STATE_HI	/
t2	shifted key[79:48]	current nibble holder	current bit holder	/
t3	shifted key[47:32]	constant 31	constant 64	/
t4	shifted key[31: 0]	counter	counter	/
t6	/	base address	base address	/
a0	/	STATE_LO	new STATE_LO	STATE_LO
a1	/	STATE_HI	new STATE_HI	STATE_HI
a2	KEY_LO	/	/	KEY_LO
a3	/	/	new index for current bit	/
a4	/	/	address for RAM index	/
a5	/	/	constant 32	/
s2	key[79:48]	/	/	/
s3	key[47:32]	/	/	/
s4	key[31: 0]	/	/	/
s5	counter	/	/	/
s6	/	base address	/	/

Figure 16: Register Usage at Each Layer



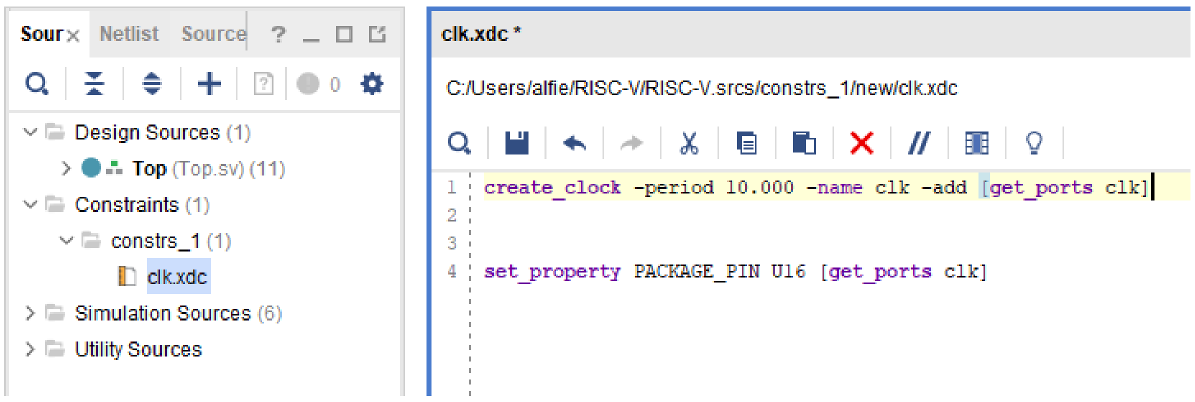
```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             8.058ns  (required time - arrival time)
  Source:                 key_reg_reg[0]_C/C
                            (rising edge-triggered cell FDCE clocked by clk  {rise@0.000ns fall@5.000ns period=10.000ns})
  Destination:            ks/key_reg_reg[0]_C/CLR
                            (recovery check against rising-edge clock clk  {rise@0.000ns fall@5.000ns period=10.000ns})
  Path Group:             **async_default**
  Path Type:              Recovery (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk rise@10.000ns - clk rise@0.000ns)
  Data Path Delay:        1.504ns  (logic 0.422ns (28.059%)  route 1.082ns (71.941%))
  Logic Levels:           1  (LUT4=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    1.860ns = ( 11.860 - 10.000 )
    Source Clock Delay      (SCD):    2.117ns
    Clock Pessimism Removal (CPR):    0.112ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Figure 17: PRESENT-80 HDL Timing Summary



```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             4.587ns  (required time - arrival time)
  Source:                 reg_file/register_reg_r1_0_31_0_5/RAMA_D1/CLK
                            (rising edge-triggered cell RAMD32 clocked by clk  {rise@0.000ns fall@5.000ns period=10.000ns})
  Destination:            reg_file/register_reg_r1_0_31_0_5/RAMC_D1/I
                            (rising edge-triggered cell RAMD32 clocked by clk  {rise@0.000ns fall@5.000ns period=10.000ns})
  Path Group:             clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            10.000ns  (clk rise@10.000ns - clk rise@0.000ns)
  Data Path Delay:        5.059ns  (logic 1.693ns (33.465%)  route 3.366ns (66.535%))
  Logic Levels:           7  (CARRY4=1 LUT3=1 LUT5=1 LUT6=3 RAMS32=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    1.832ns = ( 11.832 - 10.000 )
    Source Clock Delay      (SCD):    2.089ns
    Clock Pessimism Removal (CPR):    0.112ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Figure 18: CPU Timing Summary



C:/Users/alfie/RISC-V/RISC-V.srcs/constrs_1/new/clk.xdc

```
create_clock -period 10.000 -name clk -add [get_ports clk]


set_property PACKAGE_PIN U16 [get_ports clk]
```

Figure 19: Constraints for Designs

Appendix B: Source Codes

Vivado: RISC-V

Design Source

Top.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Top
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This module is the top level of all the modules.
//////////////////////////////////////////////////////////////////////////////////

(* DONT_TOUCH = "true|yes" *)   //Avoid synthesis optimizing circuit
module Top(
    (* keep = "true" *)input clk
);

    wire [31:0] instr;

    wire PC_src, Mem_Write, ALU_src, Reg_write, zero;
    wire [1:0] Imm_src, Result_src;
    wire [2:0] ALU_control;

    wire [31:0] Imm_ext, RD1, RD2, out, ALUResult, RD_DM, Result_mux3_1, result_PC_plus4,
result_PC_target, out_mux2_1;

    reg [31:0] PC_next, PC;
    reg [31:0] constant_4;

    assign PC_next = out_mux2_1;

    // Flip-flop generate next PC
    always @(posedge clk)
            PC <= PC_next;

    initial begin
        constant_4 = 'b100;     // assign to constant = 4
        PC = 0;
    end

    (* DONT_TOUCH = "true|yes" *)MUX_2to1 mux2_1 (//Avoid synthesis optimizing circuit
        .A(result_PC_plus4),
        .B(result_PC_target),
        .select(PC_src),
        .out(out_mux2_1)
    );

    (* DONT_TOUCH = "true|yes" *)Instruction_Memory inst_memory(//Avoid synthesis optimizing
circuit
        .A(PC),
        .RD(instr)
    );

    (* DONT_TOUCH = "true|yes" *)Adder PC_plus4 (//Avoid synthesis optimizing circuit
        .operand_a(PC),
        .operand_b(constant_4),
        .result(result_PC_plus4)
    );

    (* DONT_TOUCH = "true|yes" *)Register_file reg_file(//Avoid synthesis optimizing circuit
        .A1(instr[19:15]),
        .A2(instr[24:20]),
        .A3(instr[11:7]),
        .WD3(Result_mux3_1),
        .clk(clk),
        .WE3(Reg_write),
        .RD1(RD1),
        .RD2(RD2)
    );

    (* DONT_TOUCH = "true|yes" *)Control_Unit control_unit(//Avoid synthesis optimizing
circuit
        .instr(instr),
        .zero(zero),
        .PC_src(PC_src),
        .Reg_write(Reg_write),
        .Imm_src(Imm_src),
        .ALU_src(ALU_src),
        .ALU_control(ALU_control),
        .Mem_Write(Mem_Write),
        .Result_src(Result_src)
    );

    (* DONT_TOUCH = "true|yes" *)Extend extend (//Avoid synthesis optimizing circuit
        .inst(instr),
        .Imm_src(Imm_src),
        .Imm_ext(Imm_ext)
    );

    (* DONT_TOUCH = "true|yes" *)MUX_2to1 mux2_2 (//Avoid synthesis optimizing circuit
        .A(RD2),
        .B(Imm_ext),
        .select(ALU_src),
        .out(out)
    );

    (* DONT_TOUCH = "true|yes" *)ALU alu (//Avoid synthesis optimizing circuit
        .srcA(RD1),
        .srcB(out),
        .ALUControl(ALU_control),
        .Zero(zero),
        .ALUResult(ALUResult)
    );

    (* DONT_TOUCH = "true|yes" *)Adder PC_target (//Avoid synthesis optimizing circuit
        .operand_a(Imm_ext),
        .operand_b(PC),
        .result(result_PC_target)
    );

    (* DONT_TOUCH = "true|yes" *)Data_memory data_memory (//Avoid synthesis optimizing
circuit
        .A(ALUResult),
        .WD(RD2),
        .clk(clk),
        .WE(Mem_Write),
        .RD(RD_DM)
    );

    (* DONT_TOUCH = "true|yes" *)MUX_3to1 mux3_1 (//Avoid synthesis optimizing circuit
        .ALU_result(ALUResult),
        .Read_data(RD_DM),
        .PC_plus4(result_PC_plus4),
        .Result_src(Result_src),
        .Result(Result_mux3_1)
    );
endmodule
```

ALU.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: ALU
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: Arithmetic Logic Unit (ALU)is responsible for carrying out operations such
as addition, subtraction,
//              logical operations (AND, OR, XOR, NOT), and shift operations, which are
fundamental to the execution
```

```
//              of various instruction types within the RISC-V RV32I architecture, including
arithmetic, logical,
//              control, and data transfer instructions
//////////////////////////////////////////////////////////////////////////////////


module ALU(
    input signed [31:0] srcA,
    input signed [31:0] srcB,
    input [2:0] ALUControl,

    output Zero,
    output logic [31:0] ALUResult
    );

    assign Zero = (ALUResult == 0);

     always_comb begin
        ALUResult = 32'b0;
        case (ALUControl)
                                                       // operations    - (RV32I
Instructions)
            3'b000: ALUResult = srcA + srcB;           // add           - (lw, sw, add)
            3'b001: ALUResult = srcA - srcB;           // sub           - (beq, sub)
            3'b010: ALUResult = srcA & srcB;           // and           - (and)
            3'b011: ALUResult = srcA | srcB;           // or            - (or)
            3'b100: ALUResult = srcA << srcB[4:0];     // sll           - (sll)
            3'b101: ALUResult = {31'b0, srcA < srcB};  // bit operation - (slt)
            3'b110: ALUResult = srcA >> srcB[4:0];     // srl           - (srl)
            3'b111: ALUResult = srcA ^ srcB;           // xor           - (xor)
            default: ALUResult = 32'b0;
        endcase
    end

endmodule
```

MUX_2to1.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: MUX_2to1
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: multiplexer (mux) 2-to-1 is a digital switch that allows one of two input
signals to
//               be selected and forwarded to the output, based on the value of a control
signal.
//////////////////////////////////////////////////////////////////////////////////

module MUX_2to1(
    input [31:0]  A,    // input data A
    input [31:0]  B,    // input data B
    input select,       // control signal

    output [31:0] out   // output data
    );

    assign out = select? B : A;

endmodule
```

MUX_3to1.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: MUX_3to1
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: multiplexer (mux) 3-to-1 is a digital switch that allows one of three input
signals to
//              be selected and forwarded to the output, based on the value of a control
signal.
//////////////////////////////////////////////////////////////////////////////////

module MUX_3to1(
    input [31:0] ALU_result,    // Input 1
    input [31:0] Read_data,     // Input 2
    input [31:0] PC_plus4,      // Input 3
    input [1:0] Result_src,     // Control signal

    output logic [31:0] Result  // Output
    );

    always_comb begin
        case(Result_src)
            2'b00 : Result = ALU_result;
            2'b01 : Result = Read_data;
            2'b10 : Result = PC_plus4;
            default : Result = 32'bx;
        endcase
    end
endmodule
```

Instruction_Memory.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Instruction_Memory
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: Instruction Memory is responsible for holding the binary instructions that
// make up a program. When the processor is running a program, it sequentially reads
instructions from this memory
//////////////////////////////////////////////////////////////////////////////////

module Instruction_Memory(
    input [31:0] A,             // 32-bit address
    output logic [31:0] RD      // 32-bit binary instruction
    );

    logic [31:0] memory [100:0];

    // Resize A to satisfiy when the address is increasing by "1"
    assign RD = memory[A[31:2]];

    initial $readmemh("SimpleTest.mem", memory);

endmodule
```

Adder.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Adder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: adder is a circuit to perform the arithmetic addition of binary numbers
//////////////////////////////////////////////////////////////////////////////////

module Adder(
    input [31:0] operand_a,
    input [31:0] operand_b,

    output [31:0] result
    );
    assign result = operand_a + operand_b;

endmodule
```

Data_memory.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Data_memory
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: The Data Memory component serves as the primary storage area for data
//              used and produced by the program being executed.
//////////////////////////////////////////////////////////////////////////////////

module Data_memory(
    input [31:0] A,         // Reading address
    input [31:0] WD,        // Writing address
    input clk,
    input WE,               // Writing enable
    output logic [31:0] RD  // Reading data
    );

    logic [31:0] RAM [100:0];

    assign RD = RAM[A[31:2]];

    //Synchronous write, synchronous read
    always_ff @(posedge clk) begin
        // Last two bits of word address are always zero. Because RISC-V byte addressable.
        if(WE) RAM[A[31:2]] <= WD;
    end

    // Initial RAM memory with S-box table, permutation layer table, key, state
    initial $readmemh("RAM_Memory.mem", RAM);

endmodule
```

Control_Unit.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Control_Unit
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This module contains and connects two sub-modules which are ALU decoder and
main decoder.
//              It allocated the Instruction data to those two modules, generate result to
further control
//              other modules such as ALU, PC_src etc base on the instruction.
//              More details of those modules can be seen in their description
//////////////////////////////////////////////////////////////////////////////////

module Control_Unit(
    input [31:0] instr,
    input zero,

    output logic PC_src,
    output logic Reg_write,
    output logic [1:0] Imm_src,
    output logic ALU_src,
    output logic [2:0] ALU_control,
    output logic Mem_Write,
    output logic [1:0] Result_src
    );

    wire [1:0] ALU_op;
    wire branch;
    wire jump;
    always_comb
        case(instr[14:12])
            3'b100 : PC_src = (!zero & branch) | jump;
            default : PC_src = (zero & branch) | jump;
        endcase

    ALU_Decoder alu_de (.funct3(instr[14:12]), .funct7(instr[30]), .op5(instr[5]),
.ALU_op(ALU_op), .ALU_control(ALU_control));

    Main_Decoder main_de(.op(instr[6:0]), .Reg_write(Reg_write), .Imm_src(Imm_src),
.ALU_src(ALU_src), .Mem_Write(Mem_Write), .Result_src(Result_src), .Branch(branch),
.ALU_op(ALU_op), .Jump(jump));

endmodule
```

ALU_Decoder.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: ALU_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: The ALU Decoder is a critical component in a RISC-V RV32I based single-cycle
//              processor, enabling the flexible and dynamic operation of the ALU by
translating
//              instruction codes into specific ALU control signals.
//////////////////////////////////////////////////////////////////////////////////

module ALU_Decoder(
    input [2:0] funct3,         // from instr[14:12]
    input funct7,               // from instr[30]
    input op5,                  // from instr[5]
    input [1:0] ALU_op,

    output logic [2:0] ALU_control
    );

    always_comb begin
        case(ALU_op)
            2'b00 : ALU_control = 3'b000;       // lw, sw
            2'b01 : case(funct3)
                        3'b000 :  ALU_control = 3'b001;       // beq
                        3'b100 :  ALU_control = 3'b101;       // blt
                        default : ALU_control = 3'bx;
                    endcase
            2'b10 : case(funct3)
                        3'b000 : case({op5,funct7})
                                    2'b00, 2'b01, 2'b10 : ALU_control = 3'b000;     // add
                                    2'b11 : ALU_control = 3'b001;                   // sub
                                    default : ALU_control = 3'bx;
                                 endcase
                        3'b001 : ALU_control = 3'b100;          // sll
                        3'b010 : ALU_control = 3'b101;          // slt
                        3'b100 : ALU_control = 3'b111;          // xor
                        3'b101 : ALU_control = 3'b110;          // srl
                        3'b110 : ALU_control = 3'b011;          // or
                        3'b111 : ALU_control = 3'b010;          // and
                        default : ALU_control = 3'bx;
                    endcase
            default : ALU_control = 3'bx;
        endcase
    end

endmodule
```

Main_Decoder.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Main_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This component is essential for the functioning of the processor, as it
//              translates the binary instruction codes into a set of actionable commands
//              that control the data path and the operation of the processor's various
//              elements, such as the ALU, multiplexers, register file, and memory access
units.
//////////////////////////////////////////////////////////////////////////////////


module Main_Decoder(
    input [6:0] op,

    output logic Reg_write,
    output logic [1:0] Imm_src,
    output logic ALU_src,
    output logic Mem_Write,
    output logic [1:0] Result_src,
    output logic Branch,
    output logic [1:0] ALU_op,
    output logic Jump
    );

    always_comb begin
        case(op)
            7'b0000011 : {Reg_write,Imm_src,ALU_src,Mem_Write,Result_src,Branch,ALU_op,Jump}
= 11'b1_00_1_0_01_0_00_0;      // lw
            7'b0100011 : {Reg_write,Imm_src,ALU_src,Mem_Write,Result_src,Branch,ALU_op,Jump}
= 11'b0_01_1_1_00_0_00_0;      // sw
            7'b0110011 : {Reg_write,Imm_src,ALU_src,Mem_Write,Result_src,Branch,ALU_op,Jump}
= 11'b1_xx_0_0_00_0_10_0;      // R-type
            7'b1100011 : {Reg_write,Imm_src,ALU_src,Mem_Write,Result_src,Branch,ALU_op,Jump}
= 11'b0_10_0_0_00_1_01_0;      // beq
            7'b0010011 : {Reg_write,Imm_src,ALU_src,Mem_Write,Result_src,Branch,ALU_op,Jump}
= 11'b1_00_1_0_00_0_10_0;      // I-type ALU
            7'b1101111 : {Reg_write,Imm_src,ALU_src,Mem_Write,Result_src,Branch,ALU_op,Jump}
= 11'b1_11_0_0_10_0_00_1;      // jal
            default : {Reg_write,Imm_src,ALU_src,Mem_Write,Result_src,Branch,ALU_op,Jump} =
11'bx;
        endcase
    end

endmodule
```

Extend.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Extend
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: The immediate value extender is a key component in a RISC-V RV32I based
//              single-cycle processor, ensuring that immediate values embedded within
//              instructions are correctly sign-extended to 32 bits.
//////////////////////////////////////////////////////////////////////////////////

module Extend(
    input [31:0] inst,

    input [1:0] Imm_src,

    output logic [31:0] Imm_ext
    );

    always_comb begin
        Imm_ext = 32'b0;
        case(Imm_src)
            2'b00: Imm_ext = { {21{inst[31]}}, inst[31:20] };
// I
            2'b01: Imm_ext = { {21{inst[31]}}, inst[30:25], inst[11:7] };
// S
```

```
            2'b10: Imm_ext = { {20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0 };
// B
            2'b11: Imm_ext = { {12{inst[31]}}, inst[19:12], inst[20], inst[30:25],
inst[24:21], 1'b0 };    // J
            default: Imm_ext = 32'b0;
        endcase
    end

endmodule
```

Simulation Source

Top_tb.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Top_tb
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This is top level testbench.
//////////////////////////////////////////////////////////////////////////////////

module Top_tb;
    reg clk;

    Top dut(.clk(clk));

    // Generate clock
    always #5 clk = !clk;

    initial begin
        // Initial clock to zero
        clk = 0;

        // Initial RAM memory with S-box, permutation layer table
        $readmemh("RAM_Memory.mem", dut.data_memory.RAM);

        // Loding Instructions
        // Here are three tests, uncomment one of then to use, same as sucess condiction in
always block
        // $readmemh("loop_test.mem", dut.inst_memory.memory);
        // $readmemh("standard_test.mem", dut.inst_memory.memory);
        $readmemh("Present-80.mem", dut.inst_memory.memory);
    end

    always @(negedge clk) begin
//        if(dut.reg_file.A3 === 10 & dut.reg_file.WD3 === 7) begin
//loop_test success condiction
//        if(dut.data_memory.A[31:2] === 25 & dut.data_memory.WD === 25) begin
//standard_test success condiction
        if(dut.reg_file.register[11] === 32'h5579c138 & dut.reg_file.register[10] ===
32'h7b228445) begin                     //loop_test success condiction
            $display("Simulation succeeded");
            #10 $finish;
        end
    end
endmodule
```

RAM_Memory.mem

```
c           //0 -> c    S-box table
5           //1 -> 5
6           //2 -> 6
b           //3 -> b
9           //4 -> 9
0           //5 -> 0
a           //6 -> a
d           //7 -> d
3           //8 -> 3
e           //9 -> e
```

```
f           //a -> f
8           //b -> 8
4           //c -> 4
7           //d -> 7
1           //e -> 1
2           //f -> 2    end table
0000_0000           //lower plaintext
0           //0 -> 0(0x0)   P-layer table
10          //1 -> 16(0x10)
20          //2 -> 32(0x20)
30          //3 -> 48(0x30)
1           //4 -> 1(0x1)
11          //5 -> 17(0x17)
21          //6 -> 33(0x33)
31          //7 -> 49(0x31)
2           //8 -> 2(0x2)
12          //9 -> 18(0x12)
22          //10 -> 34(0x22)
32          //11 -> 50(0x32)
3           //12 -> 3(0x3)
13          //13 -> 19(0x13)
23          //14 -> 35(0x23)
33          //15 -> 61(0x33)
4           //16 -> 4(0x4)
14          //17 -> 20(0x14)
24          //18 -> 36(0x24)
34          //19 -> 52(0x34)
5           //20 -> 5(0x5)
15          //21 -> 21(0x15)
25          //22 -> 37(0x25)
35          //23 -> 53(0x35)
6           //24 -> 6(0x6)
16          //25 -> 22(0x16)
26          //26 -> 38(0x26)
36          //27 -> 54(0x36)
7           //28 -> 7(0x7)
17          //29 -> 23(0x17)
27          //30 -> 39(0x27)
37          //31 -> 55(0x37)
8           //32 -> 8(0x8)
18          //33 -> 24(0x18)
28          //34 -> 40(0x28)
38          //35 -> 56(0x38)
9           //36 -> 9(0x9)
19          //37 -> 25(0x19)
29          //38 -> 41(0x29)
39          //39 -> 57(0x39)
A           //40 -> 10(0xA)
1A          //41 -> 26(0x1A)
2A          //42 -> 42(0x2A)
3A          //43 -> 58(0x3A)
B           //44 -> 11(0xB)
1B          //45 -> 27(0x1B)
2B          //46 -> 43(0x2B)
3B          //47 -> 59(0x3B)
C           //48 -> 12(0xC)
1C          //49 -> 28(0x1C)
2C          //50 -> 44(0x2C)
3C          //51 -> 60(0x3C)
D           //52 -> 13(0xD)
1D          //53 -> 29(0x1D)
2D          //53 -> 45(0x2D)
3D          //54 -> 61(0x3D)
E           //55 -> 14(0xE)
1E          //56 -> 30(0x1E)
2E          //57 -> 46(0x2E)
3E          //58 -> 62(0x3E)
F           //59 -> 15(0xF)
1F          //60 -> 31(0x1F)
2F          //61 -> 47(0x2F)
3F          //62 -> 63(0x3F)    end tabel
0000_0000        //upper plaintext
0000_0000       //key_HI
0000_0000       //key_MID ["xxxx_0000": "x" part will not be use]
0000_0000       //key_LOW
```

Present-80.mem

```
0x00100a93          //addi s5, 1(zero)                                # s5: Initial Constant (1)
0x02000793          //addi a5, zero, 32                     # a5: constant value for substration with bit counter(32)
0x01f00813          //addi a6, zero, 31                     # a6: constant value for keySchedule total iteration(31)
0x04400b93          //addi s7, zero, 68                                         # s7: permutation base address(68)
0x14802903          //lw   s2, 328(zero)          `Loading Key                  # s2: key[79-48]
0x14c02983          //lw   s3, 332(zero)                                        # s3: key[47-32]
0x15002a03          //lw   s4, 336(zero)                                        # s4: key[31-0]
0x04002503          //lw   a0, 64(zero) `Loading PlantText  # a0: STATE_LO
0x14402583          //lw   a1, 324(zero)                                        # a1: STATE_HI
0x010a5613          //srli a2, s4, 16                                 # a2: s4[31:16]     -> s4[15:0]
0x01099293          //slli t0, s3, 16                                 # t0: s3[15:0] -> s3[31:16]
0x00566633          //or   a2, a2, t0                                 # a2: merge s4 and s3
0x00c54533          //xor  a0, a0, a2   `xor                          # a0: the result of STATE_LO XOR KEY_LO
0x0125c5b3          //xor  a1, a1, s2                                 # a1: the result of STATE_HI XOR KEY_HI
0x15584c63          //blt  a6, s5, 344  `[Loop-3]           # Beginning of the whole cipher process
0x00000293          //addi t0, zero, 0                      # t0: empty t0 prepare for ciphertext-------------------S-box
0x00000313          //addi t1, zero, 0                                # t1: empty t1 prepare for ciphertext
0x01f00e13          //addi t3, zero, 31                               # t3: Maximum index of a register(31)
0x00000e93          //addi t4, zero, 0                                # t4: bit counter initialized to 0
0x03de4263          //blt  t3, t4, 32   `[Loop-0]           # Jump to the end if t3 < t4
0x01d553b3          //srl  t2, a0, t4                                 # t2: shift a0 to right to hold the new nibble
0x00f3f393          //andi t2, t2, 0xF                                # t2: update t2 to isolate other nibbles
0x00239393          //slli t2, t2, 2    `Apply S-box                  # t6: multiply 4 for lw instruction
0x0003a383          //lw   t2, 0(t2)                                  # t2: S-box substitution for the current nibble
0x01d393b3          //sll  t2, t2, t4                                 # t2: shifted nibble to correspond position
0x0053e2b3          //or   t0, t2, t0                                 # t0: merge new nibble to current encryped text
0x004e8e93          //addi t4, t4, 4                                  # t4: Update the bit counter
0xfe1ff06f          //jal  zero,-32               `Jump to [loop-0]   # Jump back to continute new iteration
0x00000e93          //addi t4, zero, 0                                # t4: bit counter re-initialized to 0
0x03de4263          //blt  t3, t4, 36   `[Loop-1]           # Jump to the end if t3 < t4
0x01d5d3b3          //srl  t2, a1, t4                                 # t2: shift a1 to right to hold the new nibble
0x00f3f393          //andi t2, t2, 0xF                                # t2: update t2 to isolate other nibbles
0x00239393          //slli t2, t2, 2                                  # t6: multiply 4 for lw instruction t6
0x0003a383          //lw   t2, 0(t2)                                  # t2: S-box substitution for the current nibble
0x01d393b3          //sll  t2, t2, t4   `Store new nibble   # t2: shifted nibble to correspond position
0x0063e333          //or   t1, t2, t1                                 # t0: merge new nibble to current encryped text
0x004e8e93          //addi t4, t4, 4                                  # t4: Update the bit counter
0xfe1ff06f          //jal  zero,-32 `Jump to [loop-1]# Jump back to continute new iteration------------------S-box END
0x03f00e13          //addi t3, zero, 63           # t3: Last bit index(63)--------------------------------Permutation
0x00000e93          //addi t4, zero, 0                                # t4: bit counter initialized to 0
0x00000693          //addi a3, zero, 0                                # a3: current bit new location
0x00000513          //addi a0, zero, 0                                # a0: empty a0 prepare for ciphertext
0x00000593          //addi a1, zero, 0                                # a1: empty a1 prepare for ciphertext
0x05de4263          //blt  t3, t4, 68   `[Loop-2]           # Jump to the end if t3(64) < t4(0-64)
0x00fec463          //blt  t4, a5, 8                                  # Skip next line if (t4 < a5(32))
0x00030293          //addi t0, t1, 0                                  # Update upper 32-bit plain text to t0
0x01d2d3b3          //srl  t2, t0, t4                                 # t2: shift t0 to right to holds new bit at LSB
0x0013f393          //andi t2, t2, 0x1                                # t2: update t2 isolate the other bits
0x002e9713          //slli a4, t4, 2                                  # a4: multiply 4 with t4 for lw instruction
0x01770733          //add  a4, a4, s7                                 # a4: add a4 with base address s7
0x00072683          //lw   a3, 0(a4)                        # a3: p-layer substitution for the location of current bit
0x00f6ca63          //blt  a3, a5, 20                                 # check new location in range(32 - 63)
0x40f686b3          //sub  a3, a3, a5   `IF Range(32 - 63)  # a3: substrat 32 bit in a3
0x00d393b3          //sll  t2, t2, a3                                 # t2: shifted nibble to correspond position in a1
0x00b3e5b3          //or   a1, t2, a1                                 # t0: merge new nibble to current encryped text
0x00c0006f          //jal  zero,12                                              # Skip duplicate storage
0x00d393b3          //sll  t2, t2, a3   `IF Range(0 - 31)   # t2: shifted nibble to correspond position in a0
0x00a3e533          //or   a0, t2, a0                                 # t0: merge new nibble to current encryped text
0x001e8e93          //addi t4, t4, 1                                  # t4: Update the bit counter
0xfc1ff06f          //jal  zero,-64`Jump to [loop-2]# Jump back to continute new iteration------------------Permutation END
0x00da1393          //slli t2, s4, 13   `Shifted t2# t2: shift s4 to left 13 bits--------------------------KeySchedule
0x01395293          //srli t0, s2, 19                                 # t0: shift s2 to right 19 bits
0x0072e3b3          //or   t2, t0, t2                                 # t2: merge t0 and t2 to t2
0x0029d293          //srli t0, s3, 2    `Apply XOR                    # t0: shigt s3 to right 2 bits
0x01f2f293          //andi t0, t0, 0x1F                               # t0: isolate first 5 bits
0x0152c2b3          //xor  t0, t0, s5                                 # t0: xor t0 with counter
0x00229293          //slli t0, t0, 2                                  # t0: re-arrange t0 to left 2 bits
0xf839f993          //andi s3, s3, 0xFF83                                       # S3: clear bit[6:0] prepare for updating
0x0059e9b3          //or   s3, s3, t0                                 # S3: update S3 with result of XOR
0x00395293          //srli t0, s2, 3    `Shifted t3                   # t0: shift s2 to right 2 bits
0x0ff2fe13          //andi t3, t0, 0xff                               # t3: save bit 7 - 0 from t0
0x0082d293          //srli t0, t0, 8                                  # t0: shift t0 to right 8 bits
0x0ff2f293          //andi t0, t0, 0xff                               # t0: save bit 7 - 0 from t0
0x00829293          //slli t0, t0, 8                                  # t0: shift t0 to right 8 bits
0x01c28e33          //add  t3, t0, t3                                 # t3: merge t0 and t3 to t2
0x00797e93          //andi t4, s2, 7    `Shifted t4                   # t4: save bit 2 - 0 ftom s2
0x01de9e93          //slli t4, t4, 29                                 # t4: shift t4 to left 29 bits
0x0ff9f293          //andi t0, s3, 0xff                               # t0: save bit 7 - 0 from s3
0x0089d313          //srli t1, s3, 8                                  # t1: shift s3 left 8 bits
0x0ff37313          //andi t1, t1, 0xff                               # t1: save bit 7 - 0 from t0
0x00831313          //slli t1, t1, 8                                  # t1: shift t1 back left 8 bits
0x005362b3          //or   t0, t1, t0                                 # t0: merge t0 and t1 to t0
0x00d29293          //slli t0, t0, 13                                 # t0: re-arrange t0 shift 13 bits to left
0x01d2eeb3          //or   t4, t0, t4                                 # t4: merge t0 and t4
0x013a5293          //srli t0, s4, 19                                 # t0: shift s4 to right 19 bits
0x01d2eeb3          //or   t4, t0, t4                                 # t4: merge t0 and t4
0x01a3d293          //srli t0, t2, 26   `Apply S-box                  # t0: move t2[31:28] to t2[5:2]
0x0002a283          //lw   t0, 0(t0)                                  # t0: S-box with t0 and save in t0
0x00439393          //slli t2, t2, 4                                  # t2: shift t2 to left 4 bits
0x0043d393          //srli t2, t2, 4                                  # t2: shift t2 to right 5 bits
0x01c29293          //slli t0, t0, 28                                 # t0: re-arrange t0 to most left
0x0053e3b3          //or   t2, t2, t0                                 # t2: merge t2 and to t2
0x00038933          //add  s2, t2, zero                               # s2: update s2 from t2
0x000e09b3          //add  s3, t3, zero                               # s3: update s3 from t3
0x000e8a33          //add  s4, t4, zero                               # s4: update s4 from t4
0x001a8a93          //addi s5, s5, 1                                  # s5: increase s5 with 1
0xea5ff06f          //jal  zero, -348`Jump[loop-3]# Jump back to continute new iteration------------------KeySchedule END
```

Loop_test.mem

```
01800513       //addi a0, zero, 24
00300593       //addi a1, zero, 3
00000393       //addi t2, zero, 0
00058E33       //add t3 a1 zero
01C54863       //blt a0, t3, 16
00138393       //addi t2, t2, 1
00BE0E33       //add , t3, t3, a1
FF5FF06F       //jal zero, -12
00038533       //add a0, t2, zero
```

Standard_test.mem [5]

```
00500113 //addi x2, x0, 5
00C00193 //addi x3, x0, 12
FF718393 //addi x7, x3, -9
0023E233 //or x4, x7, x2
0041F2B3 //and x5, x3, x4
004282B3 //add x5, x5, x4
02728863 //beq x5, x7, 48
0041A233 //slt x4, x3, x4
00020463 //beq x4, x0, 8
00000293 //addi x5, x0, 0
0023A233 //slt x4, x7, x2
005203B3 //add x7, x4, x5
402383B3 //sub x7, x7, x2
0471AA23 //sw x7, 84(x3)
06002103 //lw x2, 96(x0)
005104B3 //add x9, x2, x5
008001EF //jal x3, 8
00100113 //addi x2, x0, 1
00910133 //add x2, x2, x9
0221A023 //sw x2, 32(x3)
00210063 //beq x2, x2, 0
```

Vivado: PRESENT-80

Design Source

PRESENT80Cipher.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Main_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This is top level design of PRESENT-80
//////////////////////////////////////////////////////////////////////////////////


module PRESENT80Cipher(
    input logic clk,                // clock
    input logic rstn,               // reset (negative)
    input logic [79:0] key,         // key
    input logic [63:0] plaintext,   // plaintext/state
    output logic [63:0] ciphertext  // ciphertext
    );

    logic [79:0] key_reg;
    logic [63:0] sbox_result, round_key, xor_result, state, state_new;

    assign xor_result = round_key ^ state;      // xor layer
    assign ciphertext = xor_result;

    Sbox_64 sbox_64(                            // S-box layer
        .in(xor_result),
        .out(sbox_result)
    );

    pLayer pl(                                // Permutation layer
        .in(sbox_result),
        .out(state_new)
    );

    KeySchedule ks(                           // Addroundkey
        .clk(clk),
        .rstn(rstn),
        .key(key_reg),
        .round_key(round_key)
    );

    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            key_reg <= key;
            state <= plaintext;
        end else begin
            state <= state_new;
        end
    end

endmodule
```

Sbox_64.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Main_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This is a Sbox design with 64-bit input
//////////////////////////////////////////////////////////////////////////////////


module Sbox_64(
    input logic [63:0] in,
    output logic [63:0] out
    );
    Sbox sbox_1(.in(in[3:0]), .out(out[3:0]));
    Sbox sbox_2(.in(in[7:4]), .out(out[7:4]));
    Sbox sbox_3(.in(in[11:8]), .out(out[11:8]));
    Sbox sbox_4(.in(in[15:12]), .out(out[15:12]));
    Sbox sbox_5(.in(in[19:16]), .out(out[19:16]));
    Sbox sbox_6(.in(in[23:20]), .out(out[23:20]));
    Sbox sbox_7(.in(in[27:24]), .out(out[27:24]));
    Sbox sbox_8(.in(in[31:28]), .out(out[31:28]));
    Sbox sbox_9(.in(in[35:32]), .out(out[35:32]));
    Sbox sbox_10(.in(in[39:36]), .out(out[39:36]));
    Sbox sbox_11(.in(in[43:40]), .out(out[43:40]));
    Sbox sbox_12(.in(in[47:44]), .out(out[47:44]));
    Sbox sbox_13(.in(in[51:48]), .out(out[51:48]));
    Sbox sbox_14(.in(in[55:52]), .out(out[55:52]));
    Sbox sbox_15(.in(in[59:56]), .out(out[59:56]));
    Sbox sbox_16(.in(in[63:60]), .out(out[63:60]));
endmodule
```

Sbox.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Main_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This is a Sbox design with 4-bit input
//////////////////////////////////////////////////////////////////////////////////


module Sbox(
    input logic [3:0] in,
    output logic [3:0] out
    );

    always_comb begin
        case(in)
            4'h0: out = 4'hC;
            4'h1: out = 4'h5;
            4'h2: out = 4'h6;
            4'h3: out = 4'hB;
            4'h4: out = 4'h9;
            4'h5: out = 4'h0;
            4'h6: out = 4'hA;
            4'h7: out = 4'hD;
            4'h8: out = 4'h3;
            4'h9: out = 4'hE;
            4'hA: out = 4'hF;
            4'hB: out = 4'h8;
            4'hC: out = 4'h4;
            4'hD: out = 4'h7;
            4'hE: out = 4'h1;
            4'hF: out = 4'h2;
            default: out = 4'hx;
        endcase
    end
endmodule
```

pLayer.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Main_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This is a Permutation layer design
//////////////////////////////////////////////////////////////////////////////////


module pLayer(
    input logic [63:0] in,
    output logic [63:0] out
    );

    assign out[0] = in[0];
    assign out[1] = in[4];
    assign out[2] = in[8];
    assign out[3] = in[12];
    assign out[4] = in[16];
    assign out[5] = in[20];
    assign out[6] = in[24];
    assign out[7] = in[28];
    assign out[8] = in[32];
    assign out[9] = in[36];
    assign out[10] = in[40];
    assign out[11] = in[44];
    assign out[12] = in[48];
    assign out[13] = in[52];
    assign out[14] = in[56];
    assign out[15] = in[60];
    assign out[16] = in[1];
    assign out[17] = in[5];
    assign out[18] = in[9];
    assign out[19] = in[13];
    assign out[20] = in[17];
    assign out[21] = in[21];
    assign out[22] = in[25];
    assign out[23] = in[29];
    assign out[24] = in[33];
    assign out[25] = in[37];
    assign out[26] = in[41];
    assign out[27] = in[45];
    assign out[28] = in[49];
    assign out[29] = in[53];
    assign out[30] = in[57];
    assign out[31] = in[61];
    assign out[32] = in[2];
    assign out[33] = in[6];
    assign out[34] = in[10];
    assign out[35] = in[14];
    assign out[36] = in[18];
    assign out[37] = in[22];
    assign out[38] = in[26];
    assign out[39] = in[30];
    assign out[40] = in[34];
    assign out[41] = in[38];
    assign out[42] = in[42];
    assign out[43] = in[46];
    assign out[44] = in[50];
    assign out[45] = in[54];
    assign out[46] = in[58];
    assign out[47] = in[62];
    assign out[48] = in[3];
    assign out[49] = in[7];
    assign out[50] = in[11];
    assign out[51] = in[15];
    assign out[52] = in[19];
    assign out[53] = in[23];
    assign out[54] = in[27];
    assign out[55] = in[31];
    assign out[56] = in[35];
    assign out[57] = in[39];
    assign out[58] = in[43];
    assign out[59] = in[47];
    assign out[60] = in[51];
    assign out[61] = in[55];
    assign out[62] = in[59];
    assign out[63] = in[63];
endmodule
```

KeySchedule.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Main_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This is a KeySchedule design, generate key at every clock cycle
//////////////////////////////////////////////////////////////////////////////////


module KeySchedule(
    input logic clk,                // Clock signal
    input logic rstn,               // Asynchronous reset
    input logic [79:0] key,         // 80-bit master key input
    output logic [63:0] round_key  // 64-bit round key output
);

    reg [79:0] key_reg, key_shift, key_new; // Registers to hold the different key states
    reg [4:0] round_counter, xor_result;    // Round counter, xor result
    wire [3:0] sbox_out;                    // Output from the S-box module

    // Instantiate the S-box module
    Sbox sbox(
        .in(key_shift[79:76]),      // Connect the highest 4 bits of the key to the S-box
input
        .out(sbox_out)              // Connect the S-box output
    );

    assign key_shift = {key_reg[18:0], key_reg[79:19]};                    // shift 60 bit
to right
    assign xor_result = key_shift[19:15] ^ round_counter;                  // xor with
counter
    assign key_new = {sbox_out,key_shift[75:20],xor_result,key_shift[14:0]};// merge
different parts to new state
    assign round_key = key_reg[79:16];                                     // assign
key_reg to output
```

```
    // Key schedule operation
    always @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            key_reg <= key; // Load the master key on reset
            round_counter <= 1;    // Reset the round counter
        end else begin
            // Assign key_reg to new key
            key_reg <= key_new;

            // Increment the round counter
            round_counter <= round_counter + 1'b1;
        end
    end

endmodule
```

Simulation Source

Top.sv

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// University: Dublin City University
// Supervisor: Xiaojun Wang
// Student Name: Zhuang Miao
// Module Name: Main_Decoder
// Project Name: Implementation and Performance Analysis of the RSIC-V RV32I Architecture
// Description: This is a Testbench for top-level design
//////////////////////////////////////////////////////////////////////////////////


module Top;

    logic clk;
    logic rstn;
    logic [79:0] key;
    logic [63:0] plaintext, ciphertext;

    PRESENT80Cipher cipher(
        .clk(clk),
        .rstn(rstn),
        .key(key),
        .plaintext(plaintext),
        .ciphertext(ciphertext)
    );

    always #5 clk = ~clk;

    initial begin
        // Test 1(expect ciphertext: 64'h3333DCD3_213210D2)
        plaintext <= 64'hFFFF_FFFF_FFFF_FFFF;
        key <= 80'hFFFF_FFFF_FFFF_FFFF_FFFF;
        // Test 2(expect ciphertext: 64'h5579C138_7B228445)
//        plaintext <= 64'h0000_0000_0000_0000;
//        key <= 80'h0000_0000_0000_0000_0000;
        clk <= 0;
        rstn <= 0;

        #10 rstn <= 1;
    end

    always @(cipher.ks.round_counter)
        if (cipher.ks.round_counter == 1'b0)    // Finish simulation when counter is 0
            #10 $finish;


endmodule
```

C++ Source Code

Standard_test.cpp

```
// 00500113 addi x2, x0, 5
// 00C00193 addi x3, x0, 12
// FF718393 addi x7, x3, -9
// 0023E233 or x4, x7, x2
// 0041F2B3 and x5, x3, x4
// 004282B3 add x5, x5, x4
// 02728863 beq x5, x7, 48
// 0041A233 slt x4, x3, x4
// 00020463 beq x4, x0, 8
// 00000293 addi x5, x0, 0
// 0023A233 slt x4, x7, x2
// 005203B3 add x7, x4, x5
// 402383B3 sub x7, x7, x2
// 0471AA23 sw x7, 84(x3)
// 06002103 lw x2, 96(x0)
// 005104B3 add x9, x2, x5
// 008001EF jal x3, 8
// 00100113 addi x2, x0, 1
// 00910133 add x2, x2, x9
// 0221A023 sw x2, 32(x3)
// 00210063 beq x2, x2, 0

#include <iostream>
#include <vector>

int main() {
    // Simulating registers as variables
    int x2 = 5, x3 = 12, x4, x5, x7, x9;

    // Simulating memory with a vector (for sw and lw instructions)
    std::vector<int> memory(128); // Size based on max offset used + register size

    x7 = x3 - 9;
    x4 = x7 | x2;
    x5 = x3 & x4;
    x5 = x5 + x4;

    // Conditional branch translated to if statement
    if (x5 == x7) {
        // Branch target instruction logic here (skipping for simplicity)
    }

    x4 = x3 < x4 ? 1 : 0;

    if (x4 == 0) {
        // Next instruction logic here (skipping for simplicity)
    }

    x5 = 0;
    x4 = x7 < x2 ? 1 : 0;
    x7 = x4 + x5;
    x7 = x7 - x2;

    // Store and Load translated to vector operations
    memory[x3 + 84 / sizeof(int)] = x7; // Assuming 4 bytes per int for address calculation
    x2 = memory[96 / sizeof(int)]; // Assuming x0 is the start of the memory vector

    x9 = x2 + x5;

    // Jump and link (jal) translated to function call if needed
    x2 = 1;
    x2 = x2 + x9;
    memory[x3 + 32 / sizeof(int)] = x2;

    // Printing final values for verification
    std::cout << "x2: " << x2 << ", x3: " << x3 << ", x4: " << x4 << ", x5: " << x5 << ",
x7: " << x7 << ", x9: " << x9 << std::endl;
    return 0;
}
```

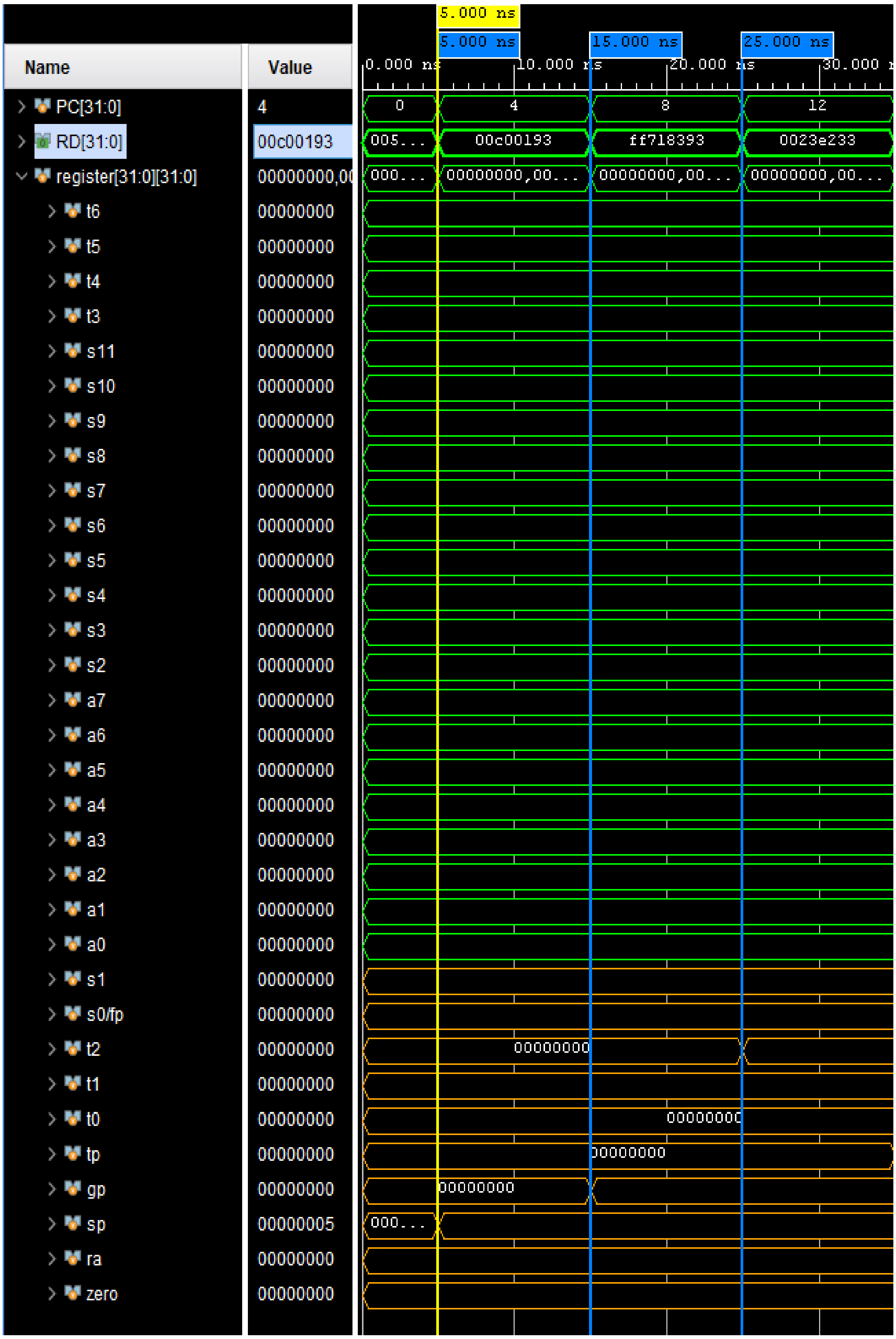
Loop_test.cpp

```
// 01800513    //addi a0, zero, 24
// 00300593    //addi a1, zero, 3
// 00000393    //addi t2, zero, 0
// 00058E33    //add t3 a1 zero
// 01C54863    //blt a0, t3, 16
// 00138393    //addi t2, t2, 1
// 00BE0E33    //add , t3, t3, a1
// FF5FF06F    //jal zero, -12
// 00038533    //add a0, t2, zero

#include <iostream>

int main() {
    int a0 = 24;
    int a1 = 3;
    int t2 = 0;
    int t3 = a1; // Initialize t3 to the same value as a1, which is 3

    while (a0 > t3) { // Loop until a0 is less than or equal to t3
        t2++;        // Increment t2
        t3 += a1;    // Increment t3 by the value of a1
    }

    a0 = t2; // At the end, copy the value of t2 to a0

    std::cout << "Final value of a0: " << a0 << std::endl;

    return 0;
}
```

PRESENT_80.cpp

```
#include <iostream>
#include <vector>
#include <cstdint>
#include <chrono>
using namespace std::chrono;

// The S-box
int SBox[16] = {0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD, 0x3, 0xE, 0xF, 0x8, 0x4, 0x7, 0x1,
0x2};

// The P-layer permutation
const uint8_t PBox[64] = {
    0, 16, 32, 48, 1, 17, 33, 49, 2, 18, 34, 50, 3, 19, 35, 51,
    4, 20, 36, 52, 5, 21, 37, 53, 6, 22, 38, 54, 7, 23, 39, 55,
    8, 24, 40, 56, 9, 25, 41, 57, 10, 26, 42, 58, 11, 27, 43, 59,
    12, 28, 44, 60, 13, 29, 45, 61, 14, 30, 46, 62, 15, 31, 47, 63
};

// Key scheduling for PRESENT-80
std::vector<uint64_t> keySchedule(uint64_t keyUpper, uint16_t keyLower) {
    std::vector<uint64_t> roundKeys(32);

    for (int i = 0; i < 32; ++i) {
        // Take 64 bits for the round key
        roundKeys[i] = keyUpper;

        // XOR round counter before shifted
        uint64_t xorResult = static_cast<uint64_t>(((keyUpper >> 18) & 0x1f) ^ (i+1)) << 18;
        keyUpper = (keyUpper & 0xFFFFFFFFFF83FFFF) | xorResult;

        // Key update
        // Rotate 61 bits to the left
        uint64_t tempUpper = keyUpper;
        uint16_t tempLower = keyLower;
        keyUpper = (tempUpper << 61) | (tempUpper >> 19) | (static_cast<uint64_t>(tempLower)
<< 45);
        keyLower = static_cast<uint16_t>((tempUpper >> 3) & 0xFFFF);

        // Apply S-box to the leftmost 4 bits of the key
        int topNibble = static_cast<int>((keyUpper >> 60) & 0xF);
        keyUpper = (keyUpper & 0x0FFFFFFFFFFFFFFF) | (static_cast<uint64_t>(SBox[topNibble])
<< 60);
    }

    return roundKeys;
}

// The PRESENT encryption algorithm
uint64_t presentEncrypt(uint64_t plaintext, const std::vector<uint64_t>& roundKeys) {
    uint64_t state = plaintext;

    for (int round = 0; round < 31; ++round) {
        // Add round key
        state ^= roundKeys[round];

        // S-box layer
        for (int i = 0; i < 16; ++i) {
            uint8_t nibble = (state >> (4 * i)) & 0xF;
            state = (state & (~(0xFULL << (4 * i)))) | (static_cast<uint64_t>(SBox[nibble])
<< (4 * i));
        }

        // P-layer
        uint64_t newState = 0;
        for (int i = 0; i < 64; ++i) {
            newState |= ((state >> i) & 0x1) << PBox[i];
        }
        state = newState;
    }

    // Last round (add round key only)
    state ^= roundKeys[31];

    return state;
}

int main() {
    uint64_t keyUpper = 0x0; // Lower 64 bits of the 80-bit key
    uint16_t keyLower = 0x0;             // Upper 16 bits of the 80-bit key
    uint64_t plaintext = 0x0; // 64-bit plaintext
    // time_t start, end;

    auto start = high_resolution_clock::now();
    // Generate round keys
    std::vector<uint64_t> roundKeys = keySchedule(keyUpper, keyLower);

    // Encrypt the plaintext
    uint64_t ciphertext = presentEncrypt(plaintext, roundKeys);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(stop - start);
    std::cout << duration.count()  << " microseconds"<< std::endl;
    std::cout << "Ciphertext:: " << std::hex << ciphertext<< std::endl;

    return 0;
}
```

Appendix C: Vivado Simulation Waveforms



Name	Value
PC[31:0]	4
RD[31:0]	00c00193
register[31:0][31:0]	00000000,00
t6	00000000
t5	00000000
t4	00000000
t3	00000000
s11	00000000
s10	00000000
s9	00000000
s8	00000000
s7	00000000
s6	00000000
s5	00000000
s4	00000000
s3	00000000
s2	00000000
a7	00000000
a6	00000000
a5	00000000
a4	00000000
a3	00000000
a2	00000000
a1	00000000
a0	00000000
s1	00000000
s0/fp	00000000
t2	00000000
t1	00000000
t0	00000000
tp	00000000
gp	00000000
sp	00000005
ra	00000000
zero	00000000

Figure 20: Partial Waveform of Test

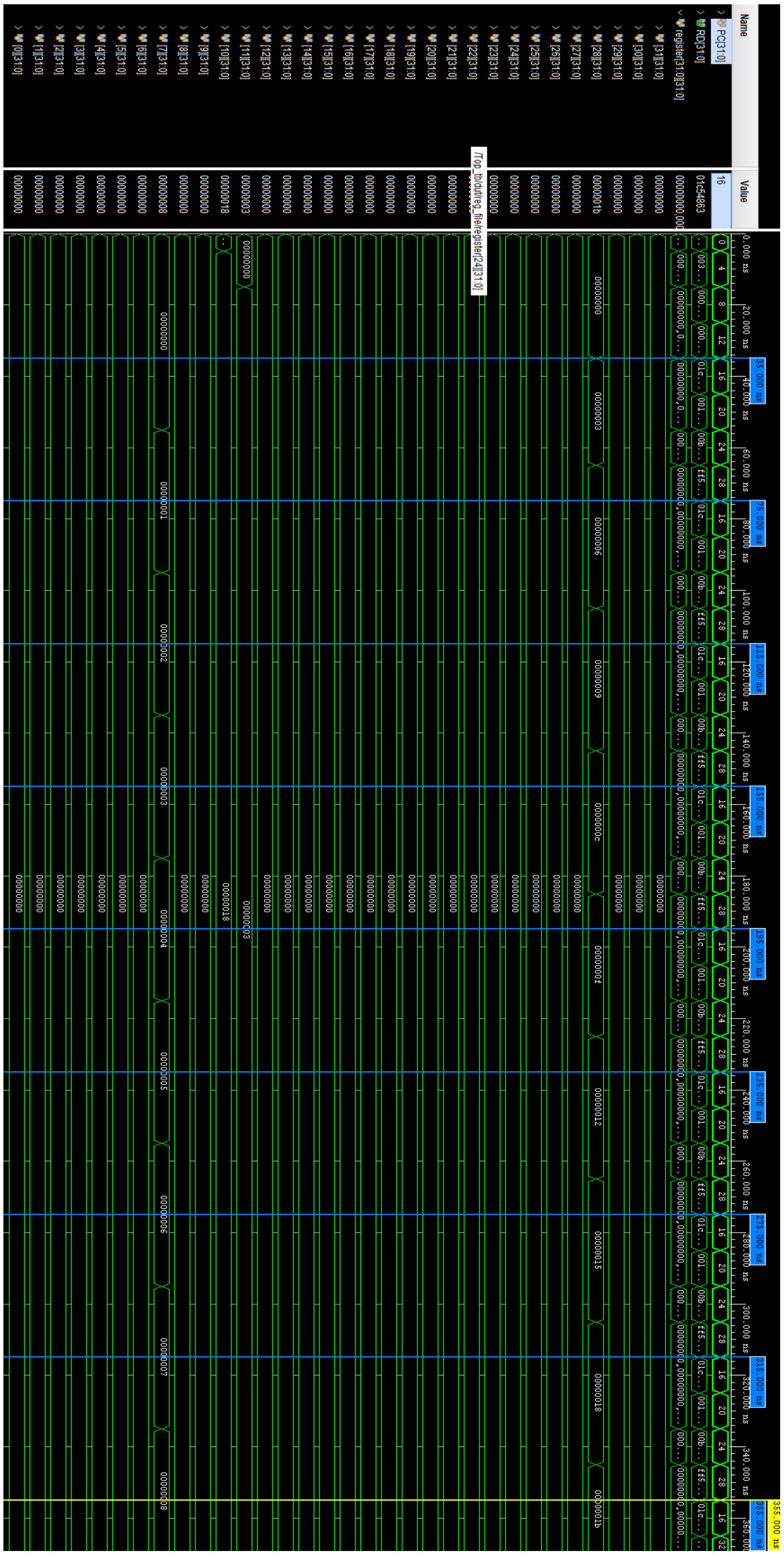


Figure 21: Loop_test Waveform

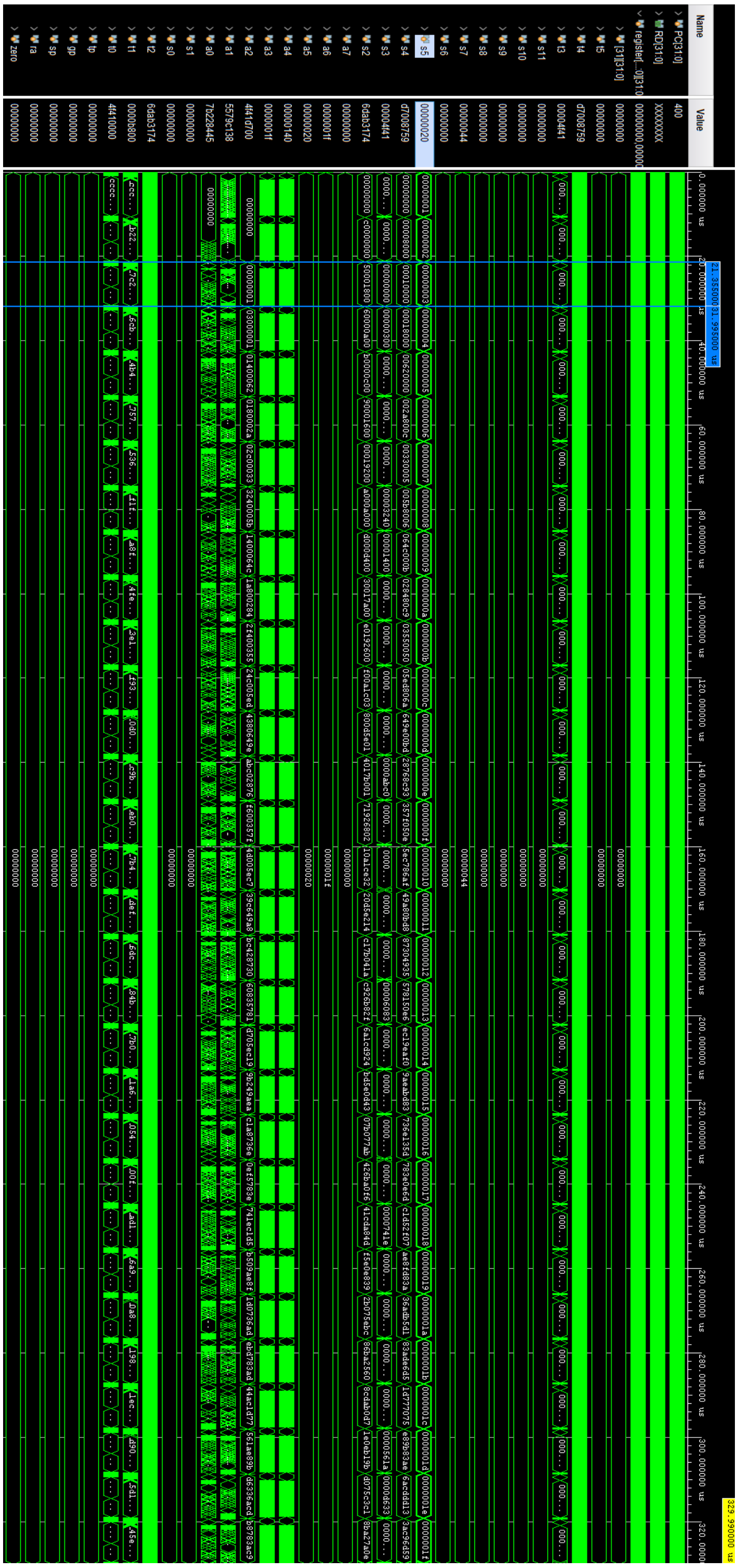


Figure 22: PRESENT-80 Waveform

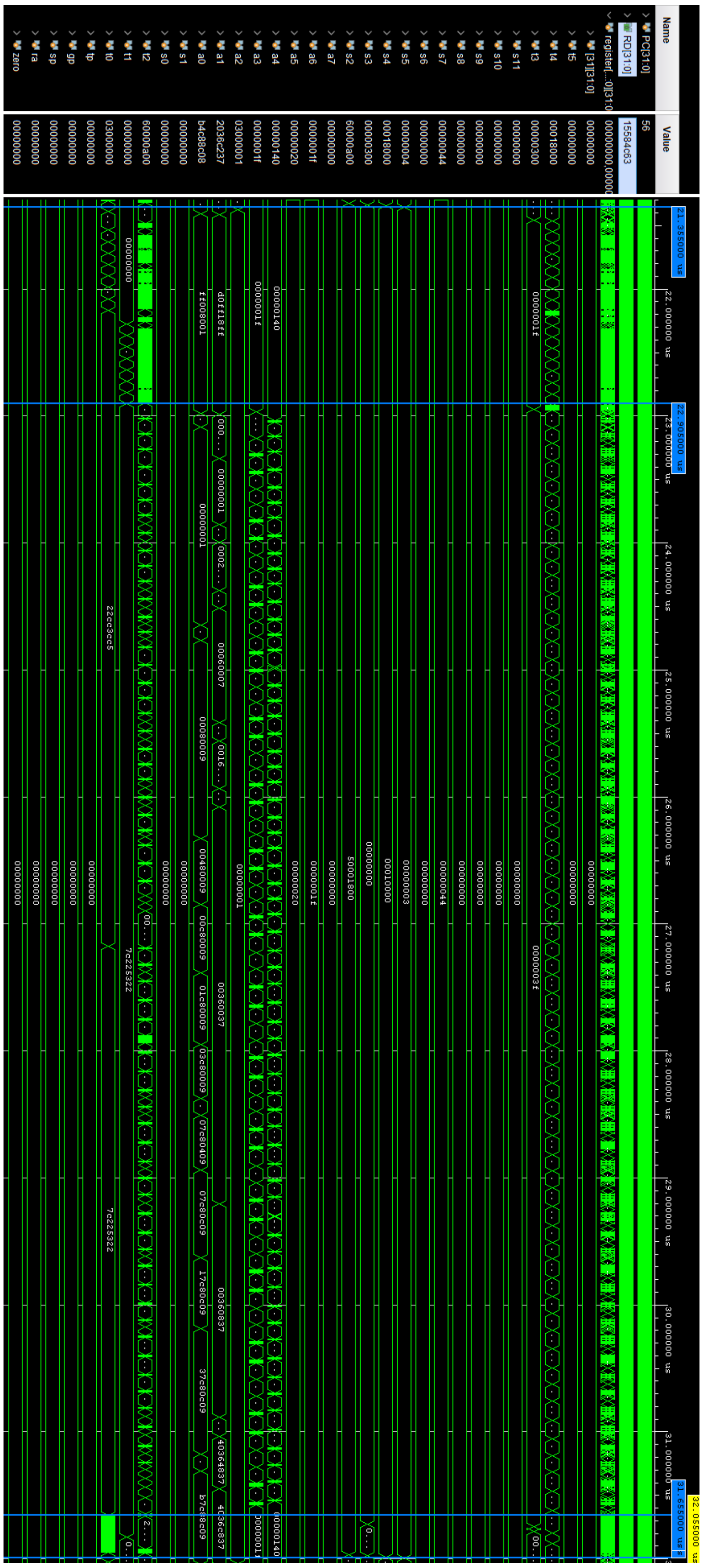
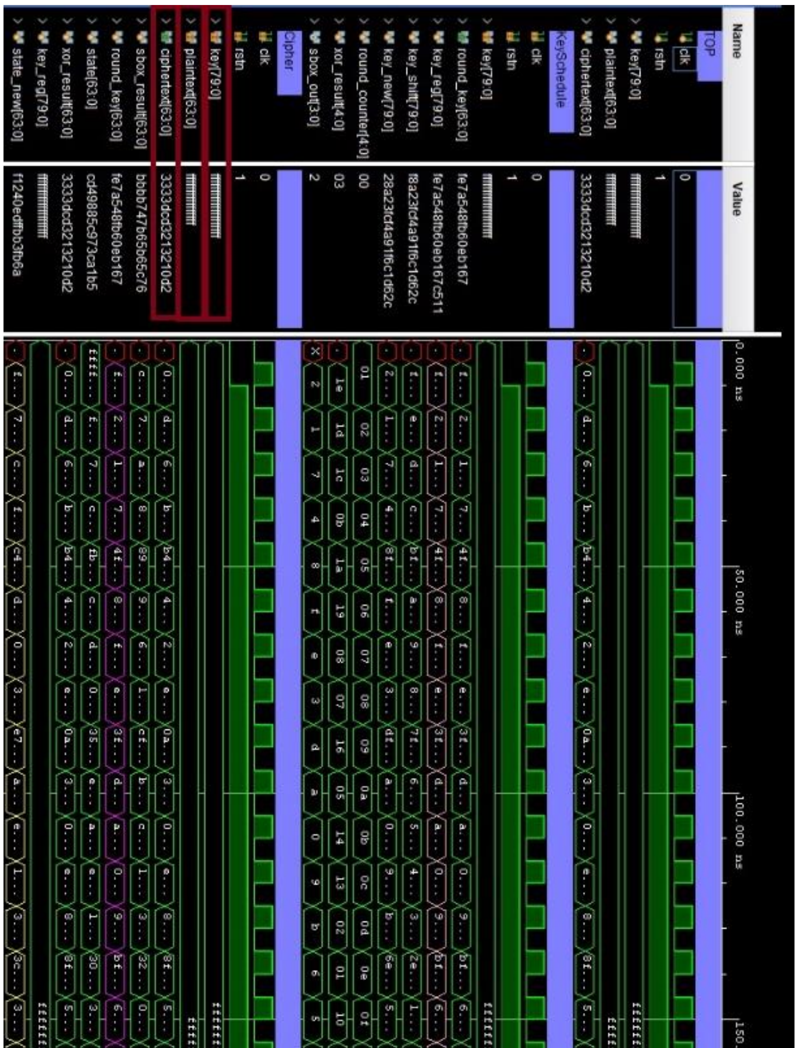


Figure 23: PRESENT-80 Waveform Selection Range



```
Name                 Value
TOP
  clk                0
  rstn               1
  key[79:0]          ffffffffffffffffffff
  plaintext[63:0]    ffffffffffffffff
  ciphertext[63:0]   3333dcd3213210d2
KeySchedule
  clk                0
  rstn               1
  key[79:0]          ffffffffffffffffffff
  round_key[63:0]    fe7a548fb60eb167
  key_reg[79:0]      fe7a548fb60eb167c511
  key_shift[79:0]    f8a23fcf4a91f6c1d62c
  key_new[79:0]      28a23fcf4a91f6c1d62c
  round_counter[4:0] 00
  xor_result[4:0]    03
  sbox_out[3:0]      2
Cipher
  clk                0
  rstn               1
  key[79:0]          ffffffffffffffffffff
  plaintext[63:0]    ffffffffffffffff
  ciphertext[63:0]   3333dcd3213210d2
  sbox_result[63:0]  bbbb747b65b65c76
  round_key[63:0]    fe7a548fb60eb167
  state[63:0]        cd49885c973ca1b5
  xor_result[63:0]   3333dcd3213210d2
  key_reg[79:0]      ffffffffffffffffffff
  state_new[63:0]    f1240edffbb3fb6a
```

Figure 24: HDL PRESENT-80 Waveform