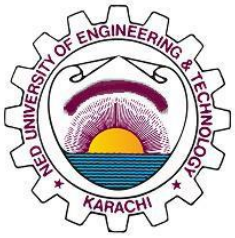
UNDERGRADUATE FINAL YEAR PROJECT REPORT

Department of Electronic Engineering

NED University of Engineering and Technology



Implementing AES Encryption and Decryption on RISC-V Processor

Group Number: 27 **Batch: 2021 – 2025**

Group Member Names:

Muhammad Tariq Waseem	EL-21056
Mirza Musab Baig	EL-21057
Rameez Nawaz	EL-21061
Anas Uddin	EL-21067

Approved by

……...

Miss Mariyum

Lecturer

Project Advisor

Author's Declaration

We declare that we are the sole authors of this project. It is the actual copy of the project that was accepted by our advisor(s) including any necessary revisions. We also grant NED University of Engineering and Technology permission to reproduce and distribute electronic or paper copies of this project.

Signature and Date	Signature and Date	Signature and Date	Signature and Date
.....	.....	.....	.....
.	.	.	.
M.Tariq Waseem	Mirza Musab baig	Rameez Nawaz	AnasUddin
EL-21056	EL-21057	EL-21061	EL-21067
tariqwaseem3@gmail.com	bmusab123@gmail.com	rameez.nawaz02@gmail.com	uddinanas66@gmail.com

Statement of Contributions

Muhammad Tariq Waseem (EL-21056) acted as group leader and led the development and integration of the RISC-V processor. He designed and implemented a 6-stage pipelined processor with a 3-stage multiplier and ensured proper interfacing with the AES Hardware Accelerator. He also contributed to the RTL development of the system and guided the team's overall progress and technical direction.

Mirza Musab Baig (EL-21057)was responsible for the initial literature review, surveying both symmetric and asymmetric cryptographic methods. He worked with Rameez on implementing the AES-128 Encryption and Decryption algorithm in Matlab and Verilog. He also developed the RSA key generation logic, contributed to the documentation, and worked on outlining the future start up road map based on the project.

Rameez Nawaz (EL-21061) collaborated with Musab in designing and implementing the AES algorithm in hardware. He played the main role in developing the RTL for the AES Hardware Accelerator and verifying its functional integration with the processor. He also assisted in simulation and performance testing.

Anas Uddin (EL-21067) supported Tariq in the RISC-V processor development and contributed significantly to documentation and system validation. He was also responsible for converting high-level MATLAB code into RISC-V assembly instructions to ensure end-to-end functionality from simulation to hardware implementation.

All members actively participated in team meetings, testing, simulations, documentation, and final presentation of the project.

Executive Summary

Problem Statement

The project addresses the need for secure and efficient data encryption in embedded systems, where existing AES implementations on RISC-V often fail to fully utilize the processor's capabilities, resulting in slower performance and unreliable.

Background Information

In today's digital world, protecting data is critical across sectors like finance, healthcare, and IoT. AES is a widely accepted encryption standard for its strong security, while RISC-V offers a flexible, open-source processor architecture ideal for customized hardware solutions. The combination of both offers an opportunity to create a high-performance, secure embedded system.

Methodology Used to Solve the Problem

The project started by developing both single-cycle and pipelined RISC-V processors using Quartus Prime software and Verilog. Pipelined processors with 5-stage and 6-stage designs were implemented for better performance. For multiplication tasks, both Carry Ripple Adder (4-stage multiplier) and Carry Save Adder (3-stage multiplier) designs were tested.
In parallel, the AES algorithm was implemented, consisting of SubBytes, ShiftRows, MixColumns, and AddRoundKey modules. The functionality of these modules was first validated using MATLAB simulations. Then, Register Transfer Level (RTL) designs were developed and integrated into the RISC-V processor. The processor communicates with the AES module using a memory-mapped interface, enabling encryption and decryption directly through processor commands.

Major Findings

The 6-stage pipelined processor combined with the 3-stage pipelined multiplier achieved the highest performance, reaching 166.53 million instructions per second (MIPS), outperforming the 5-stage and single-cycle designs. The hardware implementation of AES worked correctly and efficiently, verified through both simulation and actual FPGA tests. The system also included VGA integration for output display, showing smooth coordination between the processor and encryption hardware.

Conclusions

The project successfully demonstrates the integration of AES encryption and decryption modules into a RISC-V processor, achieving high speed, low power consumption, and strong data security. The design is scalable and adaptable for modern embedded systems where secure, real-time data processing is essential.

Acknowledgments

We would like to express our sincere gratitude to Sir Faheem for his continuous guidance, support, and valuable feedback throughout the course of our project. His expertise, timely suggestions, and encouragement have been instrumental in helping us overcome challenges and successfully complete our work.

We are also deeply thankful to Miss Mariyum for her constant assistance and motivation at every stage of the project. Her insightful advice, technical support, and dedication have greatly contributed to the smooth progress and successful completion of our research and implementation.

Table of Contents

Catalog

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Full Form
AES	Advanced Encryption Standard
RTL	Register Transfer Level
FPGA	Field Programmable Gate Array
VGA	Video Graphics Array
PLL	Phase-Locked Loop
I/O	Input/Output

United Nations Sustainable Development Goals

The Sustainable Development Goals (SDGs) are the blueprint to achieve a better and more sustainable future for all. They address the global challenges we face, including poverty, inequality, climate change, environmental degradation, peace and justice. There is a total of 17 SDGs as mentioned below.

- ☐ No Poverty
- ☐ Zero Hunger
- ☐ Good Health and Well being
- ☐ Quality Education
- ☐ Gender Equality
- ☐ Clean Water and Sanitation
- ☐ Affordable and Clean Energy
- ☑ Decent Work and Economic Growth
- ☑ Industry, Innovation and Infrastructure
- ☐ Reduced Inequalities
- ☐ Sustainable Cities and Communities
- ☐ Responsible Consumption and Production
- ☐ Climate Action
- ☐ Life Below Water
- ☐ Life on Land
- ☑ Peace and Justice and Strong Institutions
- ☐ Partnerships to Achieve the Goal

Similarity Index Report

Following students have compiled the final year report on the topic given below for partial fulfillment of the requirement for Bachelor's degree in Electronic Engineering.

Project Title **Implementing AES Encryption and Decryption on a RISC-V Processor.**

S.No.	Student Name	Seat Number
1.	Muhammad Tariq Waseem	EL-21056
2.	Mirza Musab Baig	EL-21057
3.	Rameez Nawaz	EL-21061
4.	Anas Uddin	EL-21067

This is to certify that Plagiarism test was conducted on complete report, and overall similarity index was found to be less than 20%, with maximum 5% from single source, as required.

Signature and Date

.....

Miss Mariyum

Chapter 01
INTRODUTION

1.1 Introduction

1.1.1 Overview of Cryptography

Cryptography is the science of protecting information through secure communication techniques. The Advanced Encryption Standard (AES) is a widely adopted encryption method known for its efficiency and robust security. Unlike its predecessor, the Data Encryption Standard (DES), AES supports larger key sizes (128, 192, and 256 bits), making it resistant to brute-force attacks. AES is commonly employed in sectors requiring high levels of data security, such as banking, health care, and industrial systems.

1.1.2 Introduction to RISC-V Processors

RISC-V is an open-source Reduced Instruction Set Computing (RISC) architecture that provides a flexible and scalable platform for hardware and software development. It stands out due to its modular design, which allows extensions to meet specific needs without the constraints of proprietary licensing. RISC-V's adaptability makes it an ideal candidate for implementing secure encryption algorithms like AES.

1.2 Significance and Motivation

1.2.1 Importance of Data Security

In today’s interconnected world, ensuring the confidentiality, integrity, and availability of data is paramount. Encryption algorithms like AES play a critical role in protecting sensitive information from unauthorized access, especially in industries such as finance, health-care, and IoT.

1.2.2 Relevance of RISC-V in Modern Systems

As a versatile and cost-effective architecture, RISC-V is increasingly used in embedded systems, IoT devices, and high-performance computing. Its open-source nature enables rapid innovation and customization, which is essential for developing cutting-edge solutions.

1.2.3 Bridging Performance Gaps

Existing AES implementations often fail to fully utilize the capabilities of RISC-V processors, leading to inefficiencies in power consumption and processing speed. This project seeks to bridge these gaps by creating an optimized AES module for RISC-V platforms.

1.3 Aims and Objective

1.3.1 Key Objectives of the Project

- Develop an AES encryption and decryption module tailored to the RISC-V architecture.
- Minimize the clock cycles required for efficient encryption and decryption.
- Optimize the implementation for reduced memory usage and power consumption.
- Ensure robust security against common cryptographic attacks.

1.3.2 Expected Outcomes

- A functional AES module integrated into a RISC-V processor.
- Improved performance metrics, including higher throughput and lower power usage.
- Detailed documentation and analysis to support future enhancements.

1.4 Methodology

1.4.1 Development of RISC-V Core

- Design and implement a RISC-V core using FPGA tools such as Quartus Prime.
- Integrate essential components, including memory modules and peripherals.

1.4.2 Implementation of AES Algorithm

- Develop encryption and decryption modules aligned with the RISC-V instruction set.
- Incorporate cryptographic operations such as key expansion, substitution, permutation, and mixing.

1.4.2 Integration and Testing

• Combine the AES modules with the RISC-V core.

• Test functionality with diverse data sets to validate encryption accuracy and efficiency.

1.4.3 Performance Validation

• Measure the system's throughput and power consumption.

• Conduct scenario-based testing to assess real-world applicability, such as secure data transmission.

1.5 Report Outline

1.5.1 Structure of the Report

The report is structured into eight chapters, each authored by a team member, to provide a comprehensive overview of the project.

1.5.2 Summary of Chapters

Chapter 1: Introduction

Lays the foundation by discussing background, motivation, and methodology.

Chapter 2: RISC-V Overview

Explores the architecture and its implementation.

Chapter 3: AES Implementation

Details the development process of the AES modules.

Chapter 4: Comparative Analysis and Design Improvement

Describes the integration process and performance evaluation.

Chapter 5: RSA Key Generation Algorithm

Explanation and implementation of RSA key generation, highlighting its role in secure asymmetric encryption.

Chapter 6:Register Transfer Level (RTL) Design for AES Encryption and Decryption on RISC-V

Hardware implementation of AES-128 algorithm integrated with a custom RISC-V processor at RTL level

Chapter 7: Secure Embedded Cryptographic System

Full system integration of RISC-V and AES hardware accelerator validated through simulation and FPGA deployment.

Chapter 8: Conclusion and future goals

Summary of project outcomes with proposed directions for technical expansion and entrepreneurial development.

Chapter 02
Implementation of RISC-V Processor

2.1 Introduction

This Chapter will explain the differences between a single-cycle processor, where each instruction completes in one clock cycle, and a pipelined processor, where multiple instructions are executed simultaneously at different stages.

The detail of how both processor types are designed and implemented, including their data paths, control units, Hazards, steps to reduce the Hazards and instruction execution processes their timing results and simulation.

2.1.1 What is RISC-V?

RISC-V is an open-source Reduced Instruction Set Computing (RISC) architecture designed to be simple, extensible, and modular. Unlike proprietary ISA (e.g., ARM, x86), RISC-V is freely available for academic, industrial, and commercial use, encouraging innovation and cost-effectiveness. It provides a base instruction set (e.g., RV32I for 32-bit processors) with optional extensions like M (Multiplication and Division), A (Atomic Operations), F (Single precision Floating-point) and D(Double precision Floating point). It provides a size flexibility as well ranging from the compressed version of 16 bits extending it to a 64 bits and 128 bits as well.

Extension	Description
M	Integer multiplication and division
F	Single-precision floating-point
Q	Quad-precision floating-point
C	Compressed instructions
A	Atomic instructions
B	Bit manipulations
L	Decimal floating-point
J	Dynamically translated languages
T	Transactional memory
P	Packed-SIMD instructions
V	Vector operations

Table 2.1 RISC-V Extensions

2.1.2 Significance of Implementing RISC-V Processors

The simplicity of RISC-V makes it ideal for learning and experimentation in computer architecture. It is highly modular and scalable, enabling implementations ranging from small embedded devices to powerful processors. The open-source nature allows researchers and developers to create custom processors without licensing restrictions, making it a preferred choice for educational and industrial projects. Implementing RISC-V offers hands-on experience in designing processors while staying aligned with modern architecture trends.

2.1.3 Single Cycle RISC-V Processor

A single-cycle processor is a CPU design in which every instruction completes execution in one clock cycle, regardless of its complexity. This means the cycle time must be long enough to accommodate the slowest instruction (e.g., load/store or branch). The Clock frequency is calculated from the time consumed by the critical path (slowest path).

- Each instruction fetches, decodes, executes, accesses memory (if needed), and writes back the result within one cycle.
- Simplified control unit due to the absence of overlapping instruction execution.
- Inefficient in terms of clock speed, as the critical path determines the cycle time.
- Performance Issue: Instructions with shorter execution times (e.g., ALU operations) must wait for the longest instruction to complete, leading to wasted cycles.

2.1.4 Pipelined RISC-V Processor

A pipelined processor divides instruction execution into distinct stages (e.g., Fetch, Decode, Execute, Memory Access, Write Back). Multiple instructions are executed concurrently by overlapping their stages, similar to an assembly line.

- Instructions are broken down into stages, with each stage executed in parallel.
- Requires pipeline registers to store intermediate data between stages.
- Increases CPU throughput by allowing one instruction to complete at each clock cycle.
- More complex control due to challenges like pipeline hazards (data, control, and structural)

Pipelining is implemented to improve the performance and efficiency of processors. It significantly enhances instruction throughput with disturbing the latency just a bit.

Improved Throughput:

In a single-cycle processor, only one instruction completes per clock cycle. In a pipelined processor, after the pipeline is filled, an instruction completes every clock cycle, effectively increasing the Instruction Per Cycle (IPC)

.

Efficient Utilization of Hardware:

Different parts of the processor (e.g., ALU, memory) are used simultaneously for different instructions, ensuring hardware is not idle.

Reduced Cycle Time:

The clock cycle in a pipelined processor is determined by the slowest pipeline stage, which is shorter than the critical path in a single-cycle processor.

Single-cycle processors are simpler to design but inefficient in terms of performance due to their long cycle time. Pipelined processors, though more complex, offer a significant performance boost by overlapping instruction execution and increasing throughput. This is why pipelining is a critical feature in modern CPU designs, ensuring processors meet the demands of high-performance applications.

2.2 RISC-V ISA

An Instruction Set Architecture (ISA) is the interface between hardware and software in a computer system, defining the set of instructions that a processor can execute. It specifies the operations, data types, registers, memory addressing modes, and the format of instructions. The ISA is crucial because it determines the functionality and performance of a processor while enabling compatibility between hardware and software. A well-designed ISA, like RISC-V, balances simplicity and flexibility, ensuring efficient hardware implementation and ease of software development, making it foundation for building scalable and high-performance computing systems.

2.2.1 Architectural Overview

The single-cycle processor executes one instruction per clock cycle, balancing simplicity and functionality. Its main components include:

Program Counter (PC)

Responsible for maintaining the address of the current instruction. Updated sequentially or via branch/jump instructions.

Instruction Memory

Fetches the instruction based on the PC. Memory is implemented using the IP catalog for efficient resource utilization.

Register File

Provides two read ports and one write port for accessing the 32 general-purpose registers. Inputs and outputs are directly tied to instruction decoding.

ALU

Performs arithmetic and logical operations based on the control signals and input data.

Data Memory

Handles read/write operations during load/store instructions.

Control Unit

Decodes the instruction and generates control signals that orchestrate the data flow across the processor.

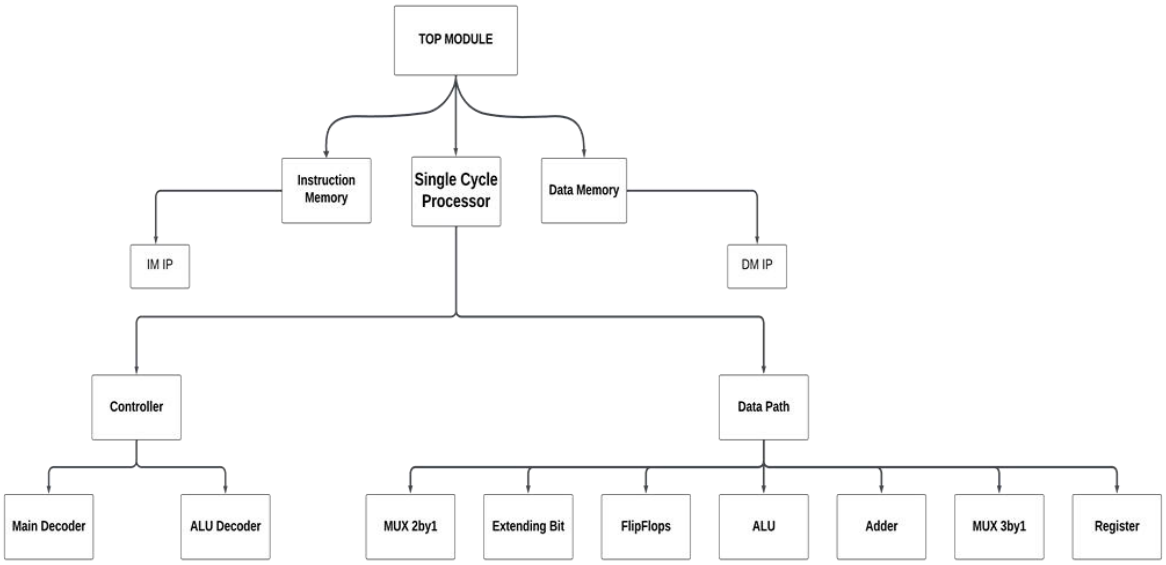


Figure 2.1 RISC-V Design Hierarchy

The implementation of this processor is carried out in Quartus Prime software, following a modular bottom-to-top design approach. Each component, such as the Program Counter (PC), ALU, Register File, Control Unit, Instruction Memory, and Data Memory, is developed and tested individually to ensure correctness and functionality. These modules are then systematically integrated to form the complete processor. This hierarchical methodology ensures scalability, simplifies debugging, and allows for efficient resource utilization, with memory blocks generated using the IP catalog for optimized space usage.

2.2.2 RV32I Instruction Set Format

The base RV32I instruction set in RISC-V consists of 47 instructions, designed to cover all essential operations required for a general-purpose processor. These instructions are categorized into six types: R-type, I-type, S-type, B-type, U-type, and J-type, each serving specific purposes such as arithmetic, logical operations, memory access, control flow, and immediate value handling.
In total, there are three major types of data transfer operations:

1. Register-to-Memory: Instructions that store data from registers into memory.
2. Memory-to-Register: Instructions that load data from memory into registers.
3. Register-to-Register: Instructions that perform operations directly between registers.

The RISC-V ISA simplifies these operations through a load-store architecture, where only memory-to-register (Load) and register-to-memory (Store) instructions interact with memory. These include:

- Load Instructions (LB, LH, LW, LBU, LHU) for accessing memory.
- Store Instructions (SB, SH, SW) for writing data to memory.

For register-to-register transfers, RISC-V supports a wide range of arithmetic and logical operations like ADD, SUB, AND, OR, and more. This streamlined approach reduces complexity, enabling efficient hardware design and execution.

Out of the total instructions in RISC-V, 12 are dedicated to register-to-memory and memory-to-register transfers, while the remaining instructions support register-to-register operations, branches, immediate values, and control flow. This balanced and minimalistic design ensures simplicity while maintaining functionality and performance.

In RISC-V, each instruction is encoded in a 32-bit fixed-length format, divided into specific fields that define its operation and operands. All types of instruction follow the same distribution of field eventually each type use what it requires the remaining

will be used as immediate field or function three or function seven to identify same type of instruction with only one opcode. These fields include:

1. Opcode (7 bits): Specifies the operation type (e.g., arithmetic, memory, branch).
2. rd (5 bits): Indicates the destination register for storing results.
3. funct3 (3 bits): Defines sub-operations or variations of the main instruction (e.g., different arithmetic operations).
4. rs1 and rs2 (5 bits each): Identify the source registers for the operation.
5. funct7 (7 bits): Used for further sub-operation encoding or instruction extension.
6. Immediate (varies): Encodes constant values for immediate-type instructions or addresses for branching and jumping

The Format of every type of Instruction are following.

R-type (Register-Register)

Used for arithmetic and logical operations between registers. The R-Type instruction format is

7 Bits	5 Bits	5 Bits	3 Bits	5 Bits	7 Bits
Funct7	**rs1**	**rs2**	**Funct3**	**rd**	**Opcode**

Table 2.2 R-Type Instruction

I-type (Immediate)

Handles operations with immediate values (constants) and loads from memory. The I-Type instruction format is:

12 Bits	5 Bits	3 Bits	5 Bits	7 Bits
Immediate	**rs2**	**Funct3**	**rd**	**Opcode**

Table 2.3 I-Type Instruction

S-type (Store) & B-type(Branch)

S-type is used to store data from a register to memory while B-type is used for conditional branching based on comparison results. Both have the same format of storing the Instruction but have different Opcode the instruction format is:

7 Bits	5 Bits	5 Bits	3 Bits	5 Bits	7 Bits
Immediate	**rs1**	**rs2**	**Funct3**	**Immediate**	**Opcode**

Table 2.4 S-Type Instruction

U-type(Upper Immediate) & J-type(Jump)

U-type is used to loads a 20-bit immediate into the upper bits of a register and jump is used for unconditional jumps to an specific address.

20 Bits	5 Bits	7 Bits
Immediate	**rd**	**Opcode**

Table 2.5 U-Type Instruction

Here's a table summarizing all instruction types in RV32I, categorized by format, with their uses:

Type	Instructions	Purpose / Use
R-Type	ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU	Perform arithmetic, logical, and shift operations between two registers.
I-Type	ADDI, ANDI, ORI, XORI, SLLI, SRLI, SRAI, SLTI, SLTIU, LB, LH, LW, LBU, LHU, JALR	Immediate operations, register-to-memory loads, and indirect jumps.

S-Type	SB, SH, SW	Store data from register to memory.
B-Type	BEQ, BNE, BLT, BGE, BLTU, BGEU	Conditional branch instructions for control flow (loops, if-else).
U-Type	LUI, AUIPC	Load upper immediate values or compute PC-relative addresses.

Table 2.6 RV-32I base Instructions and their Types

2.3 Single Cycle Processor Implementation

This section explains the design and implementation of a single-cycle processor, highlighting the architectural hierarchy and functionality of each component. Using a top-to-bottom approach, the design includes memory generated from the IP catalog for optimized space usage. The detailed implementation of every block (Verilog Code), along with the top level module, is provided in the appendix.

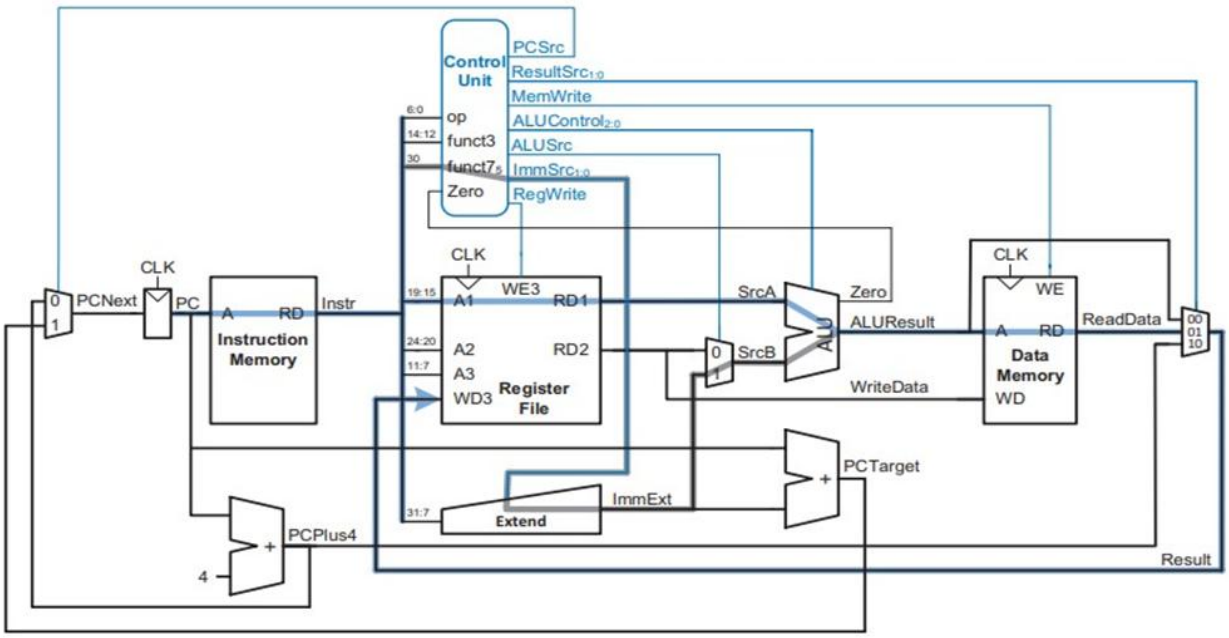


Figure 2.2 Single Cycle Processor Architecture with Critical Path [1]

2.3.1 Data Path

The processor's data path in a single-cycle design manages the flow of data required to execute an instruction entirely within one clock cycle. The **Program Counter (PC)** directs the instruction memory to fetch the current instruction, while the control unit immediately decodes it to generate the necessary control signals. The **ALU** performs the specified operation, such as arithmetic or logical computation, directly using data fetched from the register file or immediate values. For memory-related instructions, the data memory is accessed to read or write values as required. The result, whether from the ALU or data memory, is then written back to the register file. Multiplexers handle data selection at key points, and the control signals orchestrate the entire process, ensuring that all operations complete within the same clock cycle.

2.3.2 Control Unit and Memory Design

Control Unit Logic:

Responsible for decoding the instruction's opcode and function fields to generate signals such as:

- ALUSrc for selecting the second ALU operand.
- RegWrite for enabling writes to the register file.
- MemWrite for interacting with the data memory.

Memory Design:

Memory blocks (instruction and data) are implemented using the IP catalog, ensuring minimal space usage compared to flip-flop-based designs. This design choice balances resource usage and performance, especially for larger instruction sets.

2.3.3 Design Hierarchy and Methodology

The processor design follows a modular, bottom-to-top approach:

1. The top-level module integrates all components, ensuring seamless interaction.
2. Each block (PC, Register File, ALU, Control Unit) is designed and verified individually.

3. Simulation and synthesis confirm the correct operation and timing constraints.

The verilog code of all the module including top modules is present at the appendix.

2.4 Pipelined Processor Implementation

Pipelining is a technique used in processor design to improve performance by dividing the execution of instructions into several stages, allowing multiple instructions to be processed simultaneously. This is achieved by overlapping the execution of instructions, much like an assembly line in a factory. In a pipelined processor, different parts of the instruction execution process are handled by specialized hardware units, each dedicated to a specific stage.

A typical pipeline consists of five stages:

1. **Instruction Fetch (IF)**: The processor retrieves the next instruction to be executed from instruction memory using the program counter (PC).
2. **Instruction Decode (ID)**: The instruction is decoded to identify its type, extract necessary operands, and generate control signals.
3. **Execute (EX)**: The operation specified in the instruction is performed. This may involve arithmetic or logical computations performed by the ALU.
4. **Memory Access (MA)**: If the instruction requires interaction with memory (e.g., load or store), the appropriate data is accessed or updated in this stage.
5. **Write Back (WB)**: The result of the operation is written back to the register file, making it available for subsequent instructions.

This division into stages allows multiple instructions to flow through the processor at once, increasing throughput and making the design more efficient than a single-cycle processor.

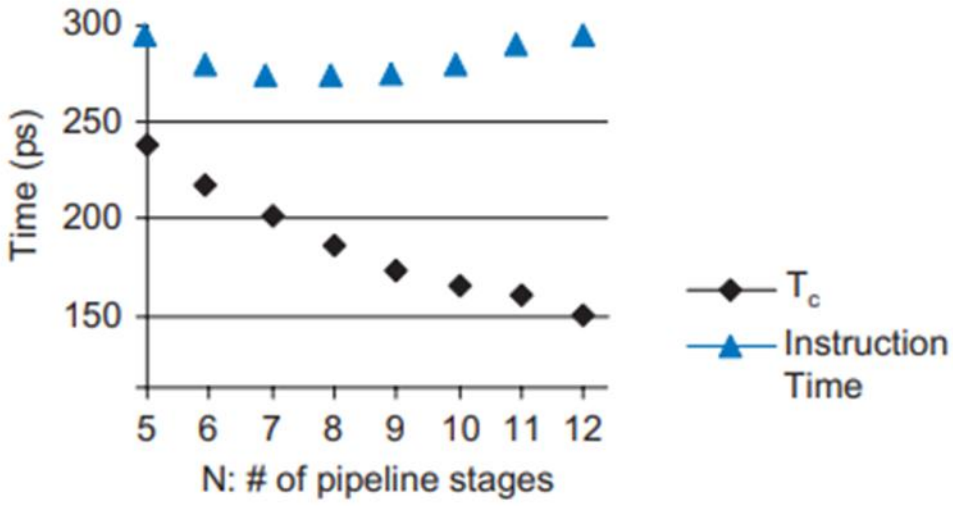


Figure 2.3 Instruction Time vs Pipelined stages [1]

The graph illustrates the relationship between the number of pipeline stages (N) and two metrics: **clock cycle time (Tc)** and **instruction execution time**. As the number of pipeline stages increases, the clock cycle time (Tc) decreases due to finer granularity in task division. However, the overall instruction execution time remains constant, as the latency for completing an instruction does not reduce with additional stages, highlighting the balance between pipelining efficiency and diminishing returns.

2.4.1 Pipeline Data Path

The data path in a pipelined processor is an enhanced version of the single-cycle data path, with additional registers inserted between stages to store intermediate results. These pipeline registers ensure that each stage operates independently and passes its output to the next stage in the pipeline.

Compared to a single-cycle processor, the pipelined data path has the following differences:

- **Pipeline Registers**: Registers between stages (e.g., IF/ID, ID/EX, EX/MA, MA/WB) are introduced to hold data and control signals.
- **Stage-Specific Logic**: Each stage in the pipeline has dedicated hardware for its operations. For example, the ALU is primarily used in the execute stage, while the memory unit is active during the MA stage.

- **Concurrency**: Multiple instructions are in different stages of execution at the same time, requiring coordination of control signals across the pipeline.

This modular structure not only improves instruction throughput but also allows for better scalability in design.

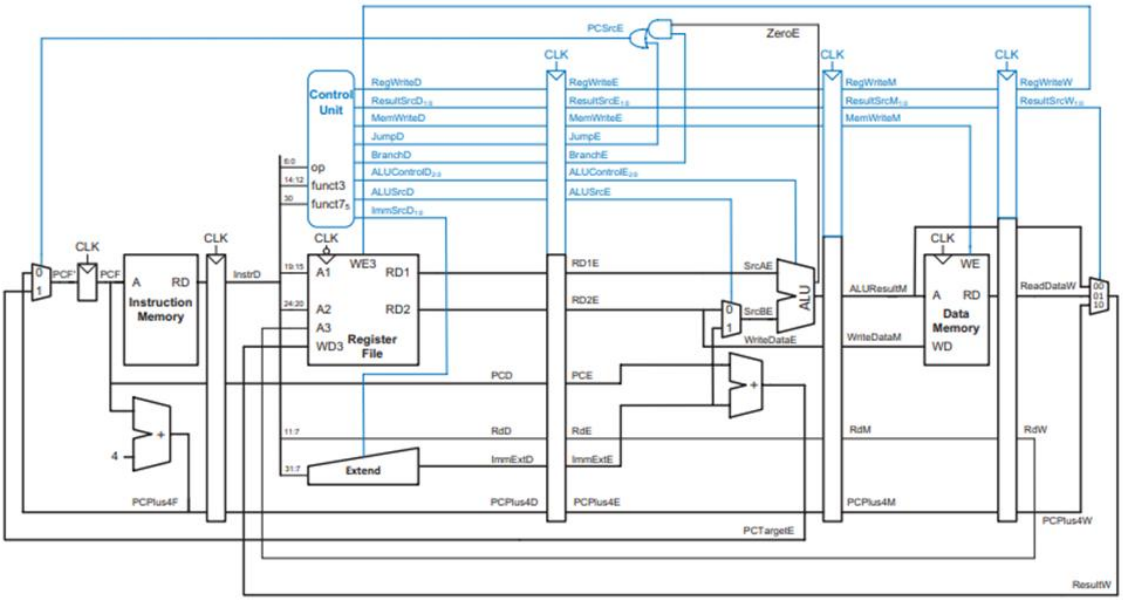


Figure 2.4 5 stage Pipelined Processor Architecture [1]

2.4.2 Pipeline Control

The control signals in a pipelined processor are generated during the decode stage and carried through the pipeline via the pipeline registers. Each stage uses only the control signals relevant to its operation.

Managing control signals in a pipeline involves careful planning to ensure correct operation. For example:

- Signals such as **RegWrite** and **MemRead** are generated during the ID stage and propagated to subsequent stages to perform the intended action.
- Additional logic is included to handle pipeline hazards, such as stalls or flushing instructions.

By distributing the control signals across pipeline stages, the processor maintains a smooth execution flow while ensuring correct results.

2.4.3 Pipeline Hazards

Pipelining introduces new challenges, known as hazards, which can disrupt instruction execution. These include:

Structural Hazards

Occur when two instructions compete for the same hardware resource.

1. **Memory access** for fetching an instruction and data simultaneously).
2. **Register File** to be used at the same time for read and Write Operation in decode and Write back stage.

Solutions include resource duplication or dividing resources into separate units (e.g., separate instruction and data memory). In our Processor a two port memory has been used for Structural hazard 1 and a Register file which will be Writing at the first half cycle of the clock and Read at second half cycle of the clock.

Data Hazards

Happen when an instruction depends on the result of a previous instruction that has not yet completed. Common solutions are:

1. **Forwarding (Bypassing)**: Forwarding units pass data from later pipeline stages to earlier ones to resolve dependencies.
2. **Pipeline Stalls**: Insert "bubbles" (idle cycles) to delay execution until data is ready.
3.

Control Hazards

Arise from branch or jump instructions that alter the program counter, potentially causing the wrong instructions to be fetched. Solutions include:

1. **Branch Prediction**: Guess the outcome of branches to keep the pipeline filled.
2. **Flushing**: Clear instructions from the pipeline when a branch is mispredicted.

By carefully addressing these hazards, pipelined processors can maintain high instruction throughput without compromising correctness.

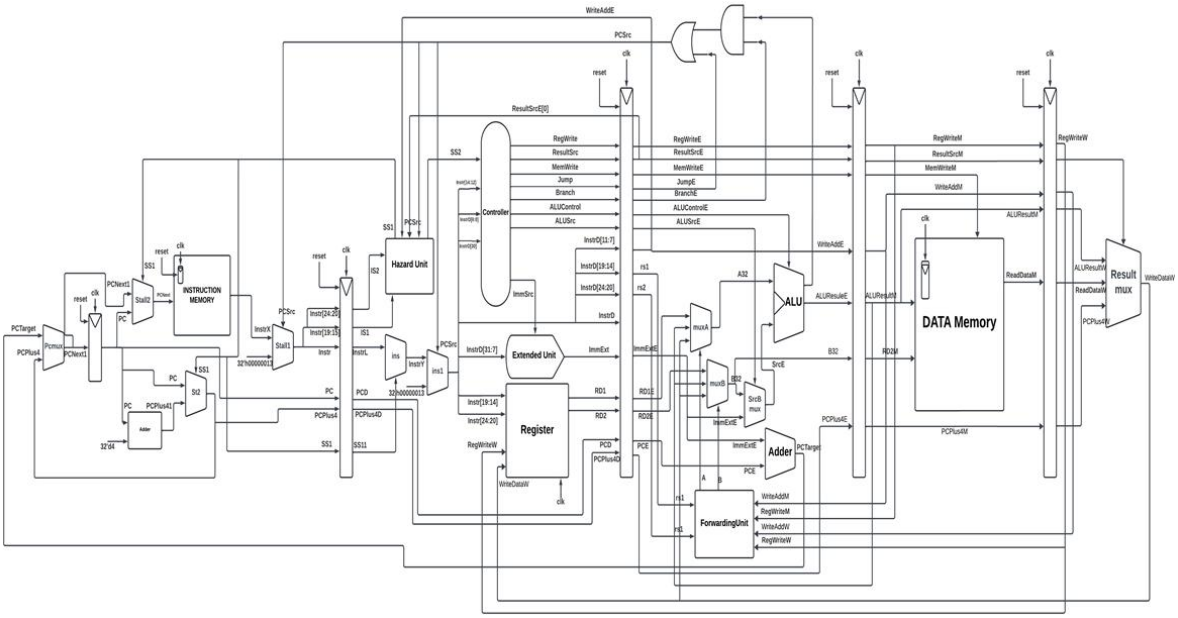


Figure 2.5 Pipelined Processor with Hazard Mitigation

The diagram represents the data path of a pipelined processor after incorporating mechanisms to resolve hazards. The design divides instruction execution into five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). To address **data hazards**, forwarding (bypassing) units are included, allowing intermediate results to be passed directly to dependent stages without waiting for write-back. Pipeline registers isolate each stage, ensuring data and control signal integrity across cycles. **Control hazards** are mitigated using branch prediction and flushing mechanisms, visible in the control logic and pipeline registers. Structural hazards are avoided by designing independent resources, such as separate instruction and data memory units. The overall design demonstrates efficient pipelining with optimized throughput and minimal stalls.

CHAPTER 03
IMPLEMENTATION OF AES ALGORITHM

3.1 Introduction

In 2001, the National Institute of Standards and Technology (NIST) introduced the Advanced Encryption Standard (AES). This symmetric block cipher was developed to serve as the successor to the Data Encryption Standard (DES) and is now widely adopted across various applications.

AES carries out all its operations on 8-bit bytes. Specifically, arithmetic processes such as addition, multiplication, and division are executed within the finite field $GF(2^8)$.

Essentially, a field is a mathematical set that allows addition, subtraction, multiplication, and division operations, with the results always remaining within the same set.

3.2 AES Structure

3.2.1 General Structure:

The AES algorithm processes data in blocks of 128 bits, which is equal to 16 bytes. The key size—either 16, 24, or 32 bytes (corresponding to 128, 192, or 256 bits)—determines whether the version is AES-128, AES-192, or AES-256. Both the encryption and decryption procedures operate on one 128-bit block at a time. This block is first placed into a structure known as the State array, where it is transformed through several rounds. Once all rounds are complete, the State array is transformed into the final output matrix, as shown in Figure 3.1.

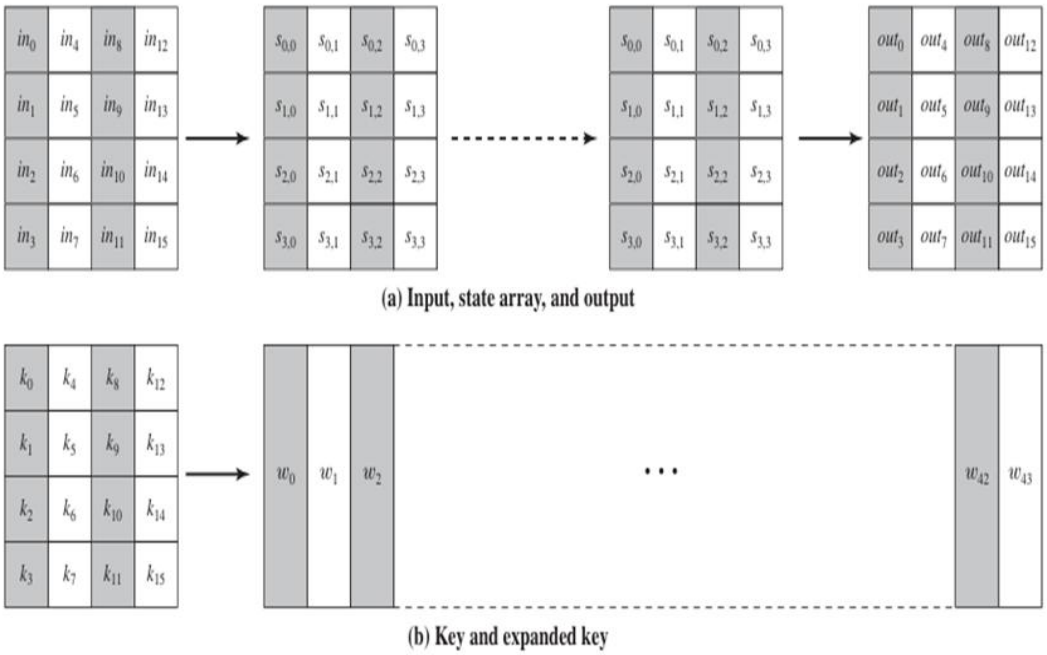


Figure 3.1 Input, State Array, Output , Key and Expanded key

The encryption key is represented as a square matrix of bytes, which undergoes a process known as key expansion. This expands the key into a series of words, forming what is called the key schedule. For a 128-bit key, the key schedule consists of 44 words, each comprising four bytes, as shown in Figure 3.1.

It is important to note that the matrix is arranged in column-major order. For example, the initial four bytes of a 128-bit plaintext are placed into the first column of the input matrix, followed by the next four bytes in the second column, and so forth. In the same way, the first word—comprising four bytes—of the expanded key fills the first column of the key schedule matrix. This structured layout helps maintain a consistent and orderly flow of both data and key material during encryption.

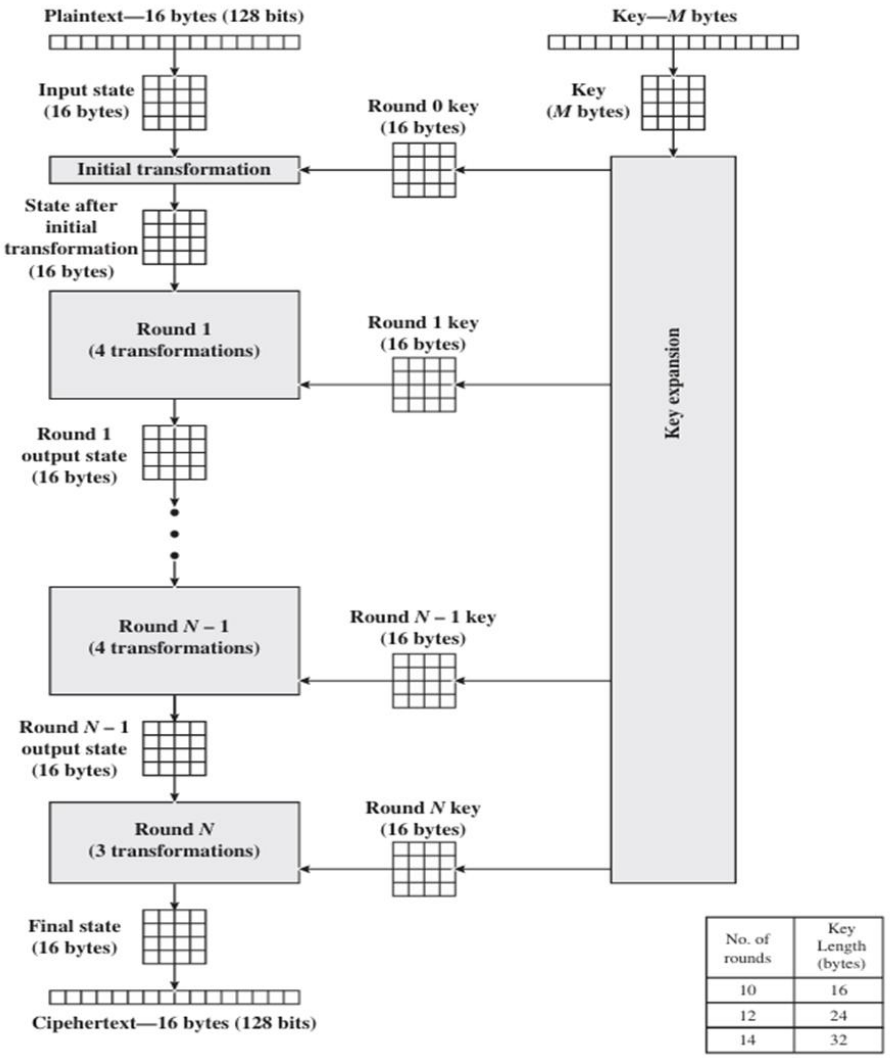


Figure 3.1 AES Algorithm Structure

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext Block Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14
Round Key Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded Key Size (words/bytes)	44/176	52/208	60/240

Table 3 Indication of no. of bits used for corresponding no. of rounds

3.3 Implementation of AES

Before exploring the inner workings of AES, it's helpful to understand some key aspects of its overall structure. The input key is expanded into an array consisting of forty-four 32-bit words. In each round, four of these words (128 bits in total) are used as the round key, as shown in Figure 3.2. The AES algorithm comprises four main stages—one involving permutation and the remaining three involving substitution:

- **Substitute bytes:** Applies an S-box to individually replace each byte in the block
- **Shift-rows:** Performs a straightforward permutation of the rows.
- **Mix-columns:** Uses finite field arithmetic over GF(2^8) to apply a substitution across columns.
- **Add Round-key:** Executes a bitwise XOR between the current block and a portion of the expanded key.

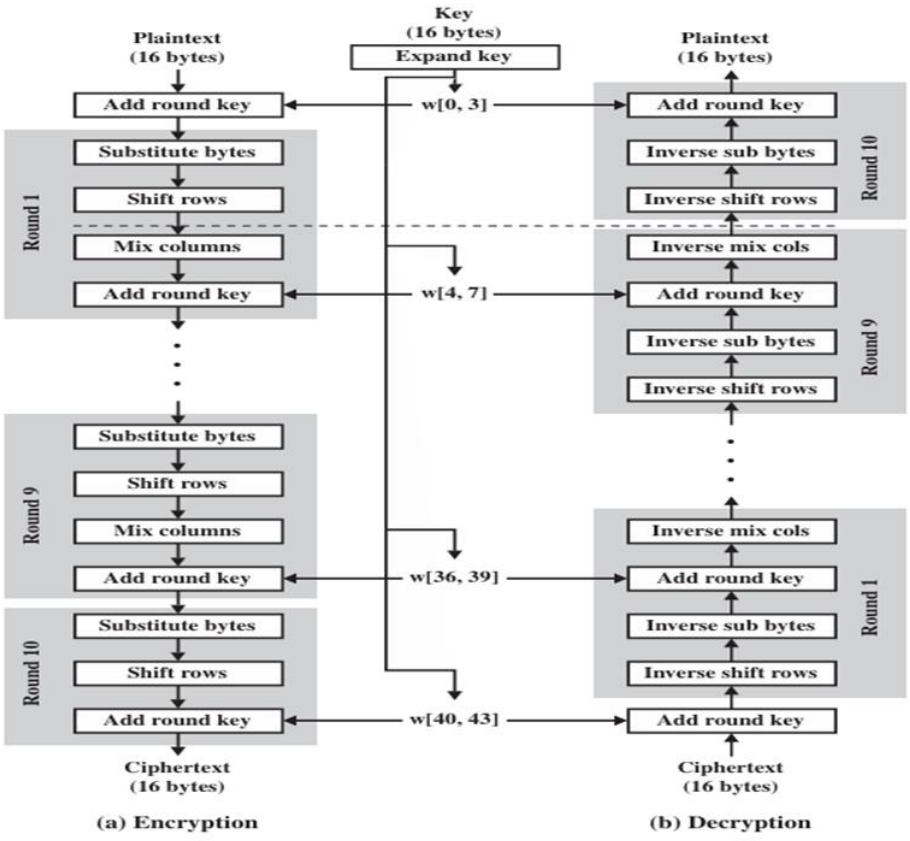


Figure 3.2 AES Algorithm Structure of 10 rounds

The encryption process starts with the Add Round Key stage, followed by nine rounds that incorporate all four transformation stages. The final, tenth round includes only three of these stages. The complete structure of an encryption round is illustrated in Figure 3.3. Notably, only the Add Round Key step utilizes the encryption key. Therefore, the algorithm begins and concludes with this stage. Including any of the other transformations at the start or end would not enhance security, as they are reversible even without access to the key.

The Add Round Key step functions similarly to a Vernam cipher and, on its own, does not offer strong security. The remaining three stages introduce confusion, diffusion, and non-linearity; however, since they do not involve the key, they cannot ensure security by themselves. The AES process can be seen as a cycle of XOR-based encryption (through the Add Round Key stage) followed by a series of transformations that scramble the data (the other three stages), and then another XOR operation, continuing in this pattern. This alternating structure is both secure and computationally efficient.

Every stage in the AES algorithm is designed to be reversible. During decryption, the inverse operations of Substitute Byte, ShiftRows, and MixColumns are applied. As for the AddRoundKey stage, its reversal is straightforward reapplying the same round key using XOR restores the original block, due to the property that XORing a value twice with the same key returns the original value (i.e., if $A = B \oplus K$, then $B = A \oplus K$).

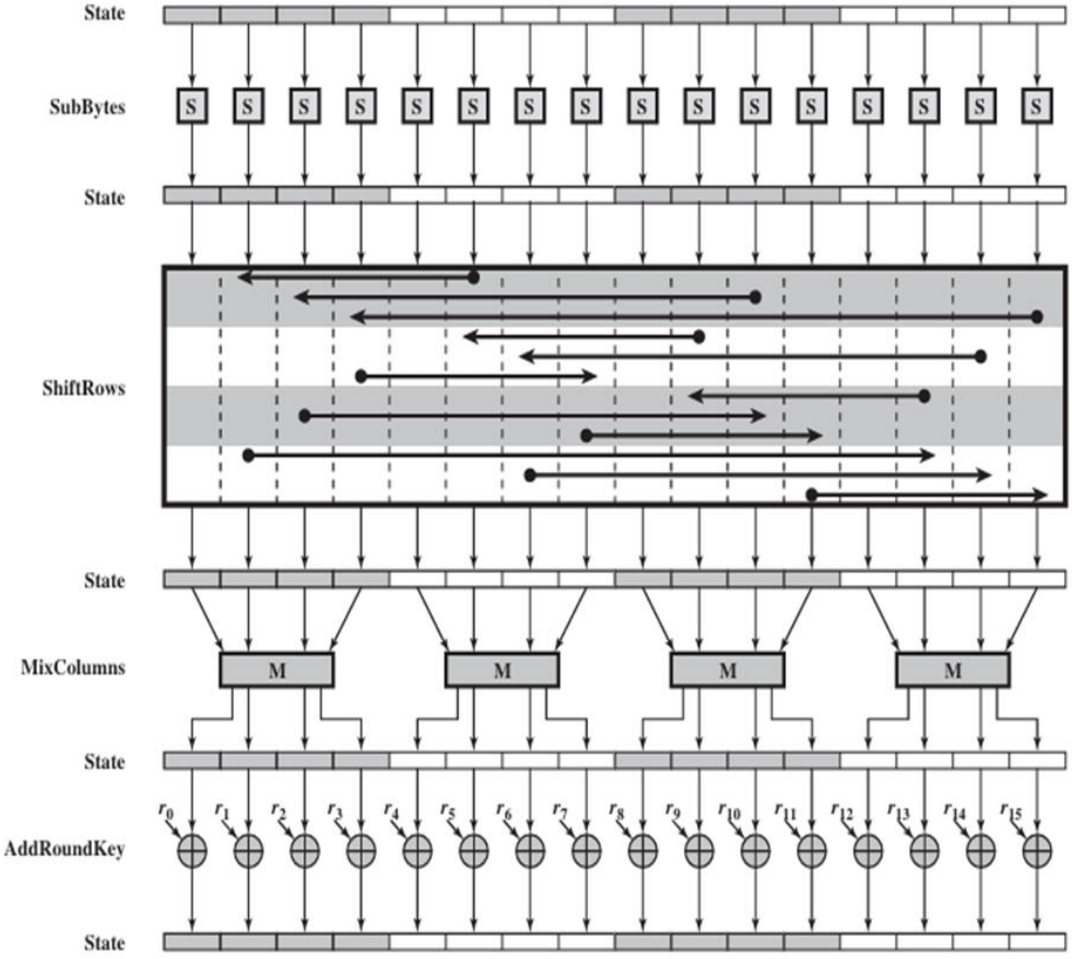


Figure 3.3 Step - by - step stages of a single round of AES

Like many block ciphers, AES decryption utilizes the expanded key in the reverse sequence. However, the decryption process differs from the encryption process due to the specific architectural design of AES.

After confirming that each of the four stages can be reversed, it becomes clear that the decryption process successfully restores the original plaintext. Figure 3 illustrates encryption and decryption progressing in opposite vertical directions. At corresponding points across the horizontal axis (such as the dashed line in the figure), the State remains identical for both encryption and decryption.

In both the encryption and decryption processes, the final round includes only three stages. This design choice is a direct result of AES's unique structure and is essential to ensure that the cipher can be reversed correctly.

3.3.1 Substitute Bytes Transformation

Forward and Inverse Transformation:

The Sub-bytes transformation, used in the encryption process, is essentially a table look-up operation (see Figure 5). AES utilizes a 16×16 matrix known as the S-box, which contains a rearranged set of all 256 possible 8-bit byte values. During this step, each byte in the State is replaced by a corresponding value from the S-box. This is done by interpreting the byte's leftmost 4 bits as the row index and the rightmost 4 bits as the column index. These indices point to a unique entry in the S-box. For instance, the byte {95} corresponds to row 9 and column 5 in the S-box, which holds the value {2A}. Thus, {95} is substituted with {2A}.

```
%subbyte
function out = sub_byte(in, sbox)
    out = zeros(4, 4);  % Initialize output matrix
    for j = 1:4
        for i = 1:4
            byte = in(i,j);  % Get byte value
            row = bitshift(byte, -4);  % First nibble (high 4 bits)
            col = bitand(byte, 15);  % Second nibble (low 4 bits)
            out(i, j) = sbox(row + 1, col + 1);  % Substitute using S-box
        end
    end
end
```

Figure 3.4 Matlab code for Sub-byte stage

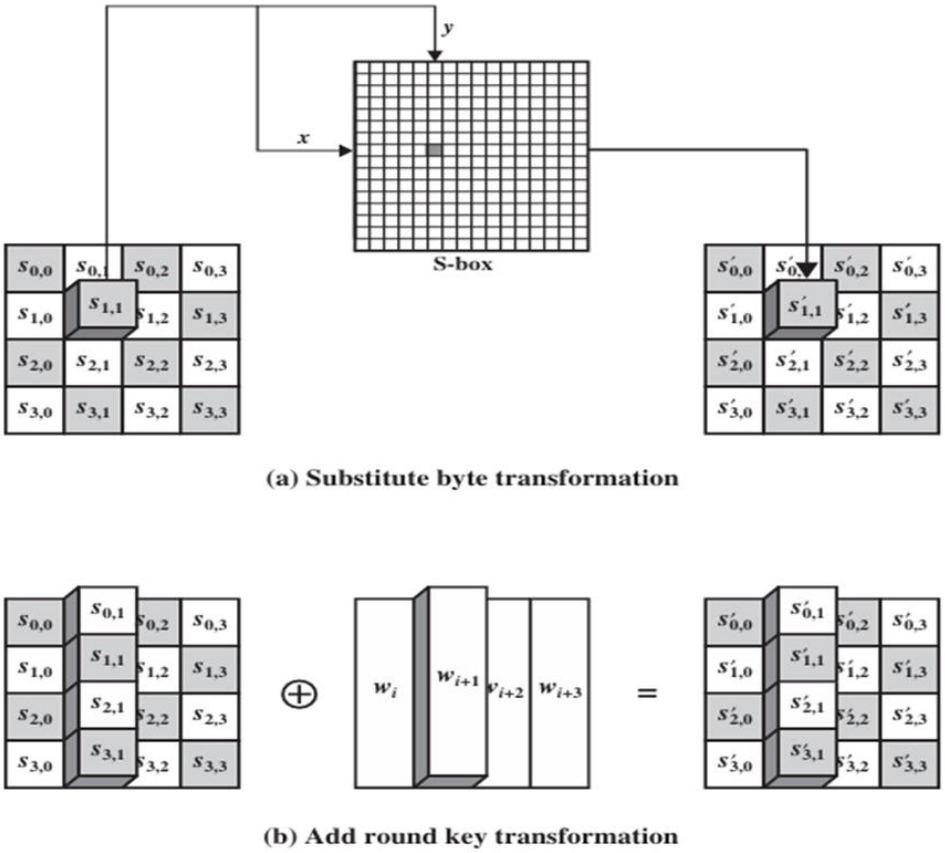


Figure 3.5 Visual representation of Sub-byte and Add round key stage

3.3.2 Shift Rows Transformation

Forward and Inverse Transformation:

The ShiftRows transformation, illustrated in Figure 7, involves shifting the rows of the State matrix in a specific pattern. The first row remains unchanged. The second row undergoes a circular left shift by one byte, the third row by two bytes, and the fourth row by three bytes. This step rearranges the data within the State and contributes to diffusion. An example of this transformation is shown below.

87	F2	4D	97
EC	6E	4C	90
4A	C3	46	E7
8C	D8	95	A6

→

87	F2	4D	97
6E	4C	90	EC
46	E7	4A	C3
A6	8C	D8	95

Figure 3.6 Visual Representation of Shift Rows stage

The inverse shift row transformation, called Inverse Shift Rows, performs the circular shifts in the opposite direction for each of the last three rows, with a 1-byte circular right shift for the second row, and so on.

```
%shift_rows
function out=shift_rows(in)
out=zeros(4,4);
        out(1,:)=[in(1,1) in(1,2) in(1,3) in(1,4)];
        out(2,:)=[in(2,2) in(2,3) in(2,4) in(2,1)];
        out(3,:)=[in(3,3) in(3,4) in(3,1) in(3,2)];
        out(4,:)=[in(4,4) in(4,1) in(4,2) in(4,3)];
end
```

Figure 3.7 Matlab code for Shift Rows stage

3.3.3 Mix Columns Transformation

The MixColumns transformation, applied during the encryption process, works on each column of the State matrix independently. In this step, every byte in a column is transformed into a new value derived from a combination of all four bytes in that column. This operation is performed using matrix multiplication over the finite field GF(2^8). Each resulting byte is calculated as the sum of products between the corresponding row and column elements. The MixColumns operation for a single column of the State can be represented as follows.

```
%mix_columns
function out = mix_columns(const_mat, in)
    out = zeros(4, 4);
     for i = 1:4
        % Precompute some mul operations to reduce redundancy
        mul_1_1 = mul(const_mat(1,1), in(1,i));
        mul_1_2 = mul(const_mat(1,2), in(2,i));
        mul_2_2 = mul(const_mat(2,2), in(2,i));
        mul_2_3 = mul(const_mat(2,3), in(3,i));
        mul_3_3 = mul(const_mat(3,3), in(3,i));
        mul_3_4 = mul(const_mat(3,4), in(4,i));
        mul_4_1 = mul(const_mat(4,1), in(1,i));
        mul_4_4 = mul(const_mat(4,4), in(4,i));

        % Compute the output columns with reduced redundant operations
        out(i,1) = bitxor(bitxor(mul_1_1, mul_1_2), bitxor(in(3,i), in(4,i)));
        out(i,2) = bitxor(bitxor(in(1,i), mul_2_2), bitxor(mul_2_3, in(4,i)));
        out(i,3) = bitxor(bitxor(in(1,i), in(2,i)), bitxor(mul_3_3, mul_3_4));
        out(i,4) = bitxor(bitxor(mul_4_1, in(2,i)), bitxor(in(3,i), mul_4_4));
    end

end
```

Figure 3.8 Matlab code for Mix Columns stage

```
function result=mul(a,b)
result=0x00;
while (b~=0)
    if (b&&1==1)
        result=bitxor(result,a);
    end

    if (a&&0x80==1)
       a=bitshift(a,-1);
       a=bitxor(a,0x11b);
    end
    if (a&&0x80~=1)
       a=bitshift(a,-1);
    end
b=bitshift(b,1);
end
```

Figure 3.9 Matlab code for multiply function in Mix Columns stage

3.3.5 Add Round Key Transformation

The AddRoundKey transformation, used during encryption, involves performing a bitwise XOR between the 128-bit State and the 128-bit round key. As illustrated in Figure 5.5b, this process can be interpreted as a column-wise operation, where each column of four bytes in the State is XORed with one word (32 bits) from the round key. Alternatively, it can be seen as applying XOR at the byte level. Below is an example demonstrating the AddRoundKey operation.

47	40	A3	4C
37	D4	70	9F
94	E4	3A	42
ED	A5	A6	BC

$\oplus$

AC	19	28	57
77	FA	D1	5C
66	DC	29	00
F3	21	41	6A

$=$

EB	59	8B	1B
40	2E	A1	C3
F2	38	13	42
1E	84	E7	D6

Figure 3.10 Visual representation of Add round key stage

In this example, the first matrix represents the State, while the second represents the corresponding round key. The inverse AddRoundKey operation is the same as the forward one, since the XOR function is self-inverse—applying it twice with the same key restores the original value.

```
%add_round_key
function out =add_round_key(in,key,round)
out=zeros(4,4);
    for j=1:4
      for i=1:4
    out(j,i)=bitxor(in(j,i),key(j,round*4+i));
      end
    end


end
```

Figure 3.11 Matlab code for Add round key stage

CHAPTER 04
COMPARITIVE ANALYSIS AND DESIGN IMPROVENMENT

4.1 Introduction

In modern computing, the performance of a processor plays a pivotal role in determining the efficiency of a system. Analyzing and enhancing processor performance is critical to meet the growing demands for faster and more reliable computations. By evaluating various architectures and optimizing key components, such as multipliers, designers can significantly improve processing speed and efficiency while addressing bottlenecks.

This chapter focuses on comparing different processor architectures, including single-cycle and pipelined designs, to analyze their performance under varying configurations. Special attention is given to the integration of multipliers, where two designs—the Carry Ripple Adder and the Carry Save Adder—are examined. The goal is to demonstrate how these multiplier implementations impact the performance of single-cycle and pipelined processors, and to identify improvements that optimize overall efficiency.

4.2 Performance of a Processor

The performance of a processor is a crucial factor in determining the efficiency and speed of a computing system. It reflects how well the processor executes instructions within a given period. Analyzing processor performance allows architects to identify bottlenecks and improve design efficiency. Performance is typically evaluated using metrics such as clock frequency, instructions per cycle, and execution time.

4.2.1 Key Metrics for Measuring Performance

Clock Frequency (MHz or GHz):

The operating speed of the processor, representing the number of cycles it can execute per second. Higher frequencies generally indicate faster processors, but performance also depends on architectural efficiency.

CPI (Clock per Instruction)

The average number of clock cycles required to execute a single instruction. CPI accounts for factors like instruction complexity, pipeline stalls, and memory latency. Lower CPI values indicate better performance.

$$\text{CPI} = \frac{\text{Total Clock Cycles}}{\text{Total Instructions Executed}}$$

Instruction per Second (IPS)

The number of instructions the processor executes in one second. It is derived from the clock frequency and CPI:

$$\text{IPS} = \frac{\text{Clock Frequency}}{\text{CPI}}$$

Execution Time

The time taken to execute a specific number of instructions, often measured for benchmarks or real workloads. For example, the time to execute one billion instructions can be calculated as:

$$\text{Execution Time} = \frac{\text{Number of Instructions}}{\text{IPS}}$$

4.2.2 Performance of Single Cycle vs Pipelined Processor

Single Cycle Processor

For single-cycle processors, CPI is typically 1, as each instruction completes in one cycle. The Maximum Frequency on which the design can run properly is **98.1Mhz** calculated through timing analyzer of Quartus Prime Software. So IPS for Single cycle processor will be

IPS = 98.1M (instruction per second)

5 Stage Pipelined Processor

The Maximum clock frequency for the 5 stage Pipelined processor is **139Mhz** calculated through timing analyzer of Quartus Prime Software. For the CPI lets

calculate it so lets assume a case.

There are approximately 22% loads, 13% stores, 12% branches, 3% jumps, and 50% R- or I-type ALU instructions out of which 30% load instruction follows with a register used in load as destination, 20% of the branches are taken, 20% of R-Type and I-Type follows with the same register used as destination by the R and I type and 30% of R-Type and I-Type instruction following after one instruction contain the same register used as destination by the R and I type. Now the CPI would be.

Introduce **1 stall cycle** for 30% of the loads.
Effective stalls = 22% × 30% = 6.6%

When branches are taken, **2 stall cycles** are introduced.
Effective stalls = 12% × 20% × 2 = 2.4%

No stall Cycle for R-type and I-type as there is a forwarding unit present.

CPI = 1 + 0.066 + 0.024 = 1.07

IPS = 139/1.07 = 129.9M (instruction/second)

6 Stage Pipelined Processor

The Maximum clock frequency for the 5 stage Pipelined processor is **206.9Mhz** calculated through timing analyzer of Quartus Prime Software. For the CPI lets calculate it so lets assume a case.
Taking the same example for this as well so
Introduce 2 **stall cycle** for 30% of the loads.
Effective stalls = 22% × 30% x 1 = 13.2%
When branches are taken, 3 **stall cycles** are introduced.
Effective stalls = 12% × 20% × 2 = 3.6%

Introduce 1 **stall cycle** for the R and I-type follow with a same register instruction
Effective stalls = 50% × 20% × 1 = 10%

No stall cycle for the R and I-type followed by the second instruction with same register.

CPI = 1 + 0.132 + 0.036 + 0.1= 1.268

IPS = 206.9/1.268 = 163.17M (instruction/second)

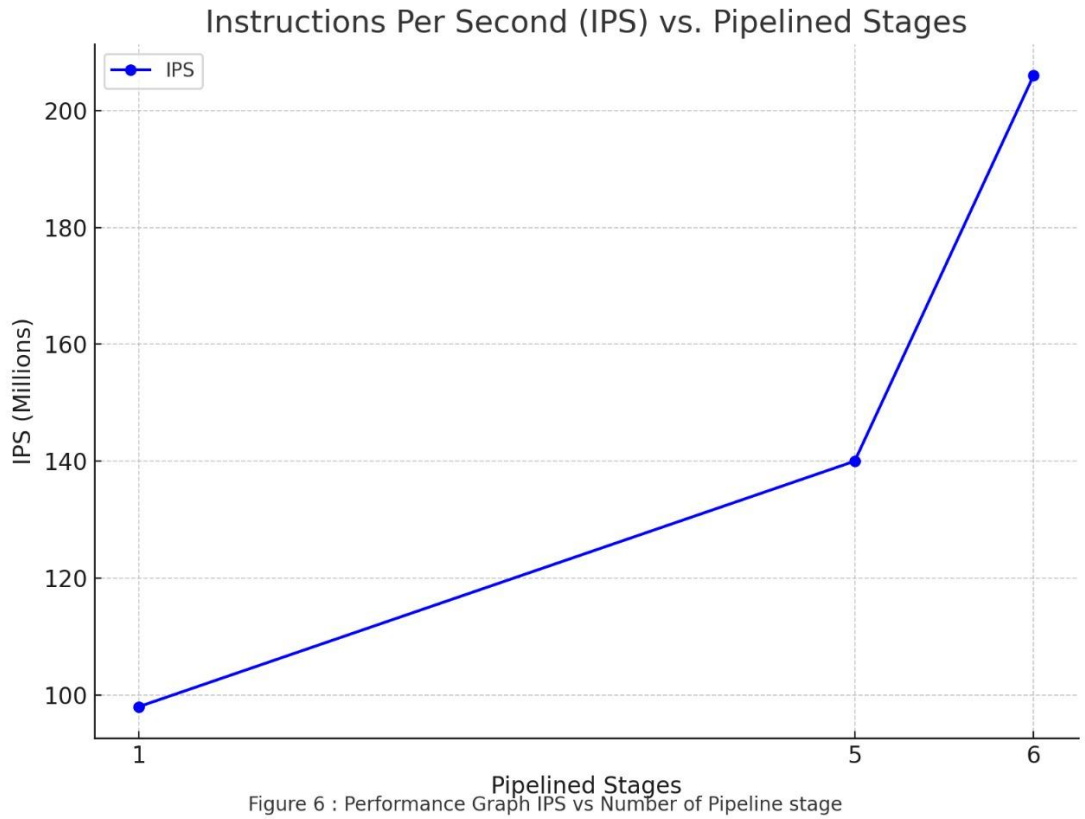


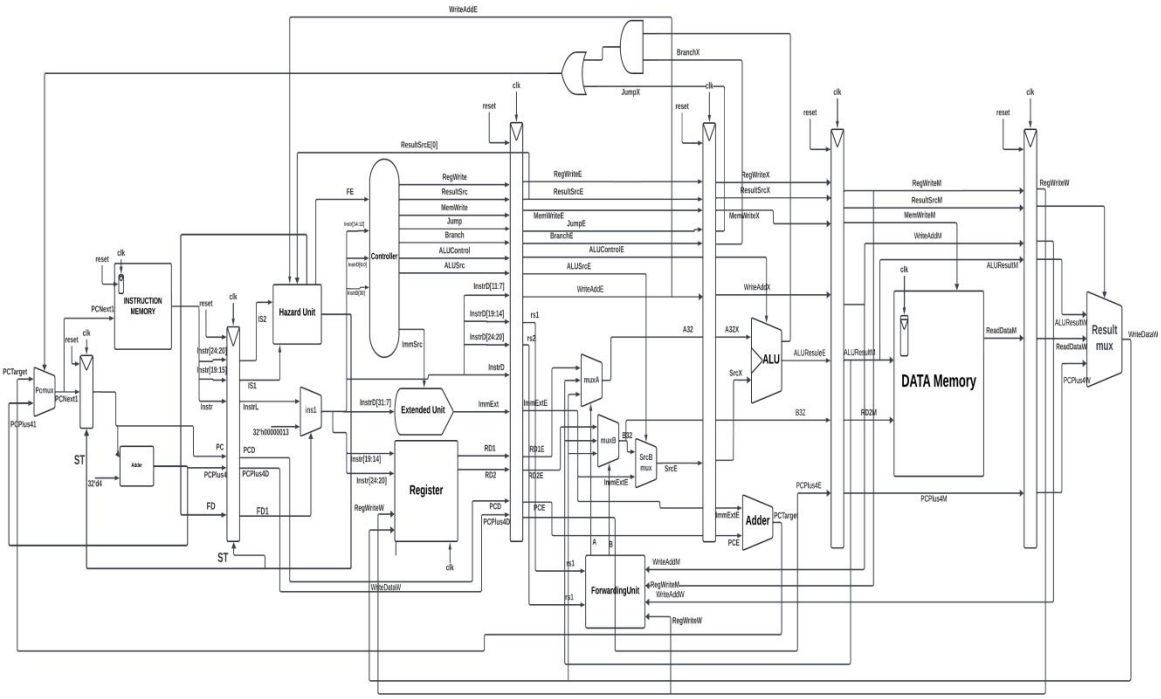
Figure 6 : Performance Graph IPS vs Number of Pipeline stage

Figure 4.1 Performance Graph IPS vs Number of Pipeline stage

The 6-stage pipelined processor demonstrates superior performance compared to its counterparts, achieving an IPS of 163.17 million, which significantly surpasses the 5-stage (129.9M IPS) and single-cycle (98.1M IPS) architectures. This improvement is attributed to enhanced instruction throughput facilitated by deeper pipelining, which reduces the execution time per instruction. Despite the added complexity of an additional stage, the 6-stage design effectively balances instruction latency and overall system performance, making it the most efficient architecture among the three.

The architectural diagram of the Six stage Pipelined Processor is attached below where ALU stage is pipeline in 2 namely Execute and X stage.

Figure 4.2 6 stage Pipelined Processor Design



While the 6-stage pipelined processor strikes an optimal balance between performance and complexity, extending the pipeline to 7 or more stages introduces diminishing returns. To achieve a 7-stage pipeline, stages such as instruction fetch or memory access would need to be split further. This division increases the number of intermediate stages, which, in turn, leads to higher stall cycles due to additional data hazards and control dependencies. As a result, the overall throughput may not improve significantly and could even degrade due to the increased penalty of pipeline stalls. Therefore, going beyond 6 stages is not advisable, as the performance gains plateau while the complexity and latency penalties escalate.

4.3 3-Stage Pipelined Multiplier with Carry Save Adder

This design include carry save adders (CSAs) to optimize the partial product accumulation process, significantly reducing carry propagation delays. The multiplier

achieves a maximum frequency of 234 MHz, making it faster than the previous 4-stage design. It operates by performing AND operations on the inputs, generating partial products, and efficiently summing them through CSA stages, followed by a final addition to produce the 64-bit output. While this approach slightly increases area consumption due to the added CSA logic, the performance gain justifies the trade-off, especially in high-speed application.

4.3.1 Design Overview

The 3-stage pipelined multiplier incorporates carry save adders (CSAs) to optimize the performance of the multiplication operation. The use of CSAs reduces the carry propagation delay by breaking the addition process into smaller, faster stages, thereby enabling higher operating frequencies.

Input Handling

The multiplier begins by performing AND operations between the bits of inputs A[31:0] and B[31:0], generating partial products.

Carry Save Adder Stages

The partial products are processed through multiple levels of carry save adders to accumulate intermediate sums and carries efficiently.

Final Summation

At the final stage, the accumulated sum and carry values are passed through a pipeline register and combined using an adder to produce the 64-bit output C[63:0].

4.3.2 Performance Metrics

Maximum Frequency

The multiplier achieves a peak frequency of **234 MHz**, significantly higher than the previously discussed 4-stage design.

Area Consumption

The use of carry save adders introduces a slight increase in area consumption due to the additional logic elements required for carry-save operations. However, this trade-off is justified by the substantial performance gain.

4.3.3 Integration into Pipelines

When integrated into a **6-stage** or **5-stage pipelined processor**, the 3-stage pipelined multiplier with carry save adders does not impact the maximum frequency of the processors. The frequencies remain at **207 MHz** for the 6-stage design and **139 MHz** for the 5-stage design. This indicates that the multiplier's frequency capabilities are well above the processor's operational limits, ensuring seamless integration without introducing bottlenecks. This compatibility highlights the multiplier's efficiency and adaptability across different pipelined architectures.

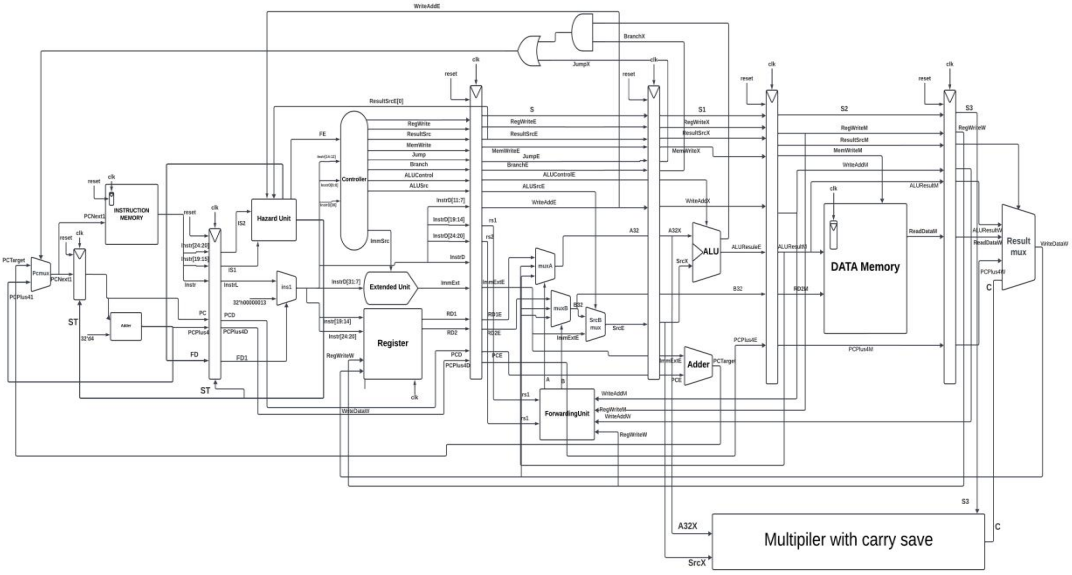


Figure 4.3 6 stage Pipelined Processor with 3 stage Multiplier

The 3-stage pipelined multiplier with carry save adders is a superior choice compared to the 4-stage multiplier with carry ripple adders. While it has slightly higher area consumption, its significantly higher speed makes it more suitable for our design. Since we are implementing the processor on an FPGA, where achieving higher performance is critical, the 3-stage carry save multiplier is the preferred option to ensure optimal speed and efficiency in the overall system.

4.4 4-Stage Pipelined Multiplier

The 4-stage pipelined multiplier is a highly efficient architecture for performing multiplication operations, optimized for a maximum operating frequency of 137 MHz. This design leverages pipeline registers to divide the multiplication process into distinct stages, ensuring improved throughput and reduced latency by allowing multiple multiplication operations to be executed simultaneously in a pipelined manner.

4.4.1 Architecture Overview

Stage 1: Partial Product Generation

The inputs, A[31:0] and B[31:0], are used to generate 32 partial products using bitwise AND operations. Each bit of operand B is ANDed with the entire operand A, resulting in 32 rows of partial products.

Stage 2: First-Level Addition

The generated partial products are passed to a set of carry ripple adders in parallel, which reduce the 32 rows into intermediate sums. This stage organizes the addition in a hierarchical structure to minimize delay and balance the workload across the pipeline.

Stage 3: Intermediate Reduction

The intermediate sums are further processed through additional adders, reducing the number of operands while maintaining accuracy. Pipeline registers are strategically placed to store intermediate results and ensure proper timing alignment.

Stage 4: Final Addition

In the final stage, the reduced partial products are combined to generate the final 64-bit product, C[63:0]. This stage completes the computation, with the result ready at the output after all pipeline stages have completed their tasks.

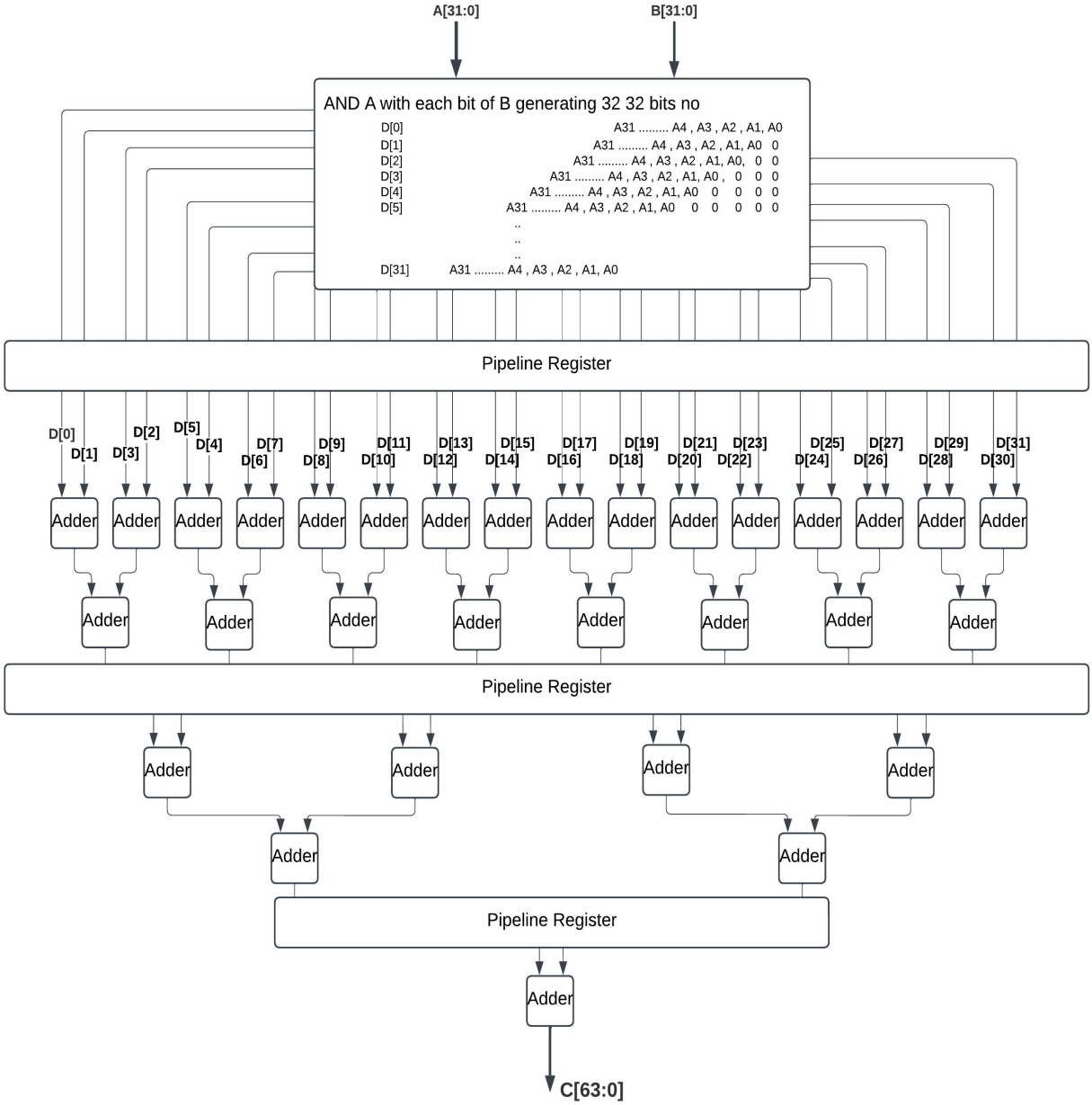


Figure 4.4 : 4-Stage Multiplier Design

4.4.2 Performance Analysis

Max Frequency

The 4-stage pipelined multiplier achieves a maximum frequency of 137 MHz, indicating the time required for a single stage is well-optimized.

Throughput

The pipelined nature of the design ensures high throughput, as each stage operates independently, allowing for a new multiplication operation to be initiated every clock cycle.

Latency

While the design introduces a latency of 4 cycles, the increased throughput compensates for the delay, making it suitable for high-performance computing applications.

This architecture balances complexity and performance, providing a robust solution for multiplication operations in pipelined processor designs. Its effectiveness can be attributed to the use of pipeline registers and an organized hierarchical addition structure.

4.4.3 Integration into a 6-Stage Pipelined Processor

When integrated into the 6-stage pipelined processor, the 4-stage pipelined multiplier operates at a maximum frequency of 137 MHz. This demonstrates the seamless compatibility of the multiplier with the existing 6-stage architecture, ensuring high performance without compromising the processor's efficiency.

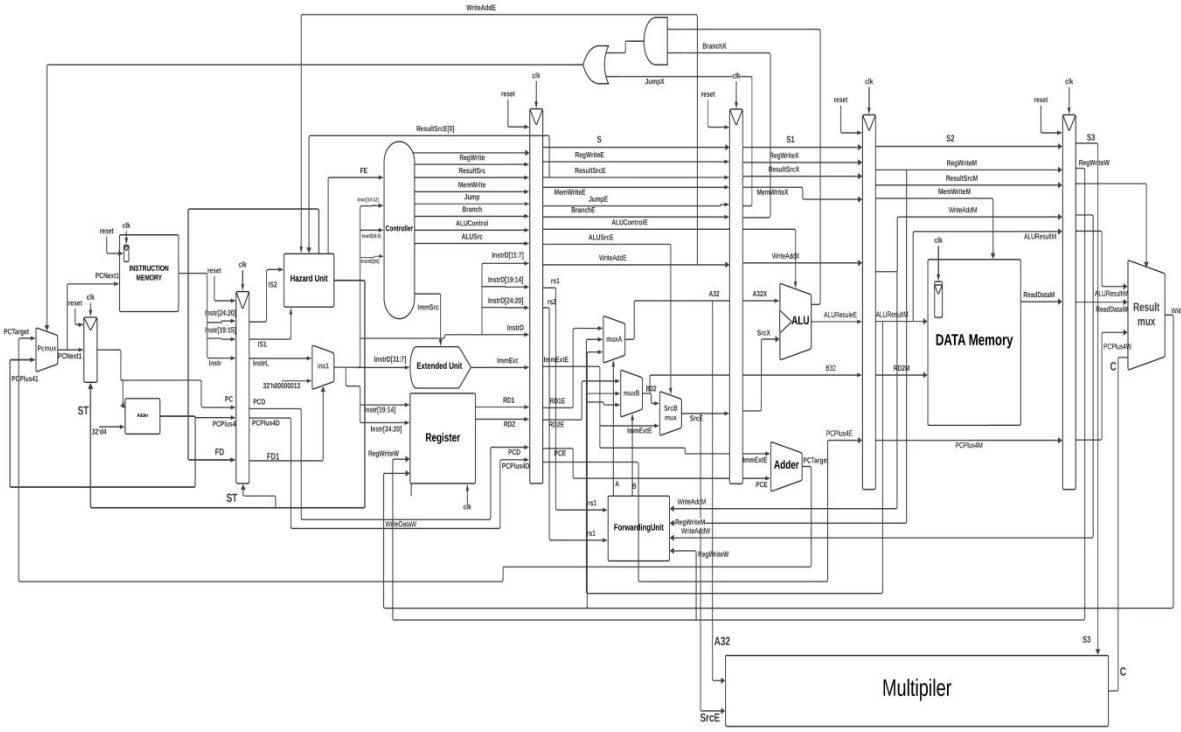


Figure 4.5 6 stage Pipelined Processor with 4-stage multiplier

However, this multiplier cannot be directly implemented in a 5-stage pipelined processor due to architectural constraints. To adapt it for a 5-stage pipeline, the multiplier design would need to be reduced to 3 stages. While this modification allows integration, it results in a significant reduction in performance, with the maximum frequency dropping to 89 MHz. This highlights the trade-offs involved in adapting multiplier designs for different pipeline architectures and underscores the superiority of the 6-stage processor in maintaining high frequencies with advanced components.

4.5 Performance of 6 Stage Pipelined Processor with Different Multiplier

To evaluate the performance of a 6-stage pipelined processor, we calculate the CPI (Cycles Per Instruction) and IPS (Instructions Per Second) for two multiplier designs: a 3-stage pipelined multiplier and a 4-stage pipelined multiplier. The performance is compared using instruction mix statistics, stalls due to hazards, and clock frequencies.

4.5.1 With 4 stage Multiplier

Consider the Same example that was discussed in performance of a processor that was There are approximately 20% loads, 13% stores, 12% branches, 3% jumps, and 50% R- or I-type ALU instructions out of which 30% load instruction follows with a register used in load as destination, 20% of the branches are taken, 15% of R-Type and I-Type follows with the same register used as destination by the R and I type, 27% of R-Type and I-Type instruction following after one instruction contain the same register us destination by the R and I type and 10% multiply instruction with 4% followed by the same register used as destination by the Mult. Now the CPI would be.

Introduce 2 **stall cycle** for 30% of the loads.
Effective stalls = 20% × 30% x 2 = 12%

When branches are taken, 3 **stall cycles** are introduced.
Effective stalls = 12% × 20% × 3 = 3.6%

Introduce 1 **stall cycle** for the R and I-type follow with a same register instruction
Effective stalls = 50% × 15% × 1 = 7.5%

Introduce 3 **stall cycles** are when a Mult instruction is follow by same destination register
Effective stalls = 10% × 4% × 3 = 1.2%

No stall cycle for the R and I-type followed by the second instruction with same register.

CPI = 1 + 0.12 + 0.036 + 0.075 + 0.012 = 1.243

IPS = 137/1.243 = 110.22 Million (instruction/second)

4.5.2 With 3 stage Multiplier

Considering the same example CPI would remain same but with the Max frequency IPS would be different so

IPS = 207/1.243 = 166.53 Million (instruction/second)

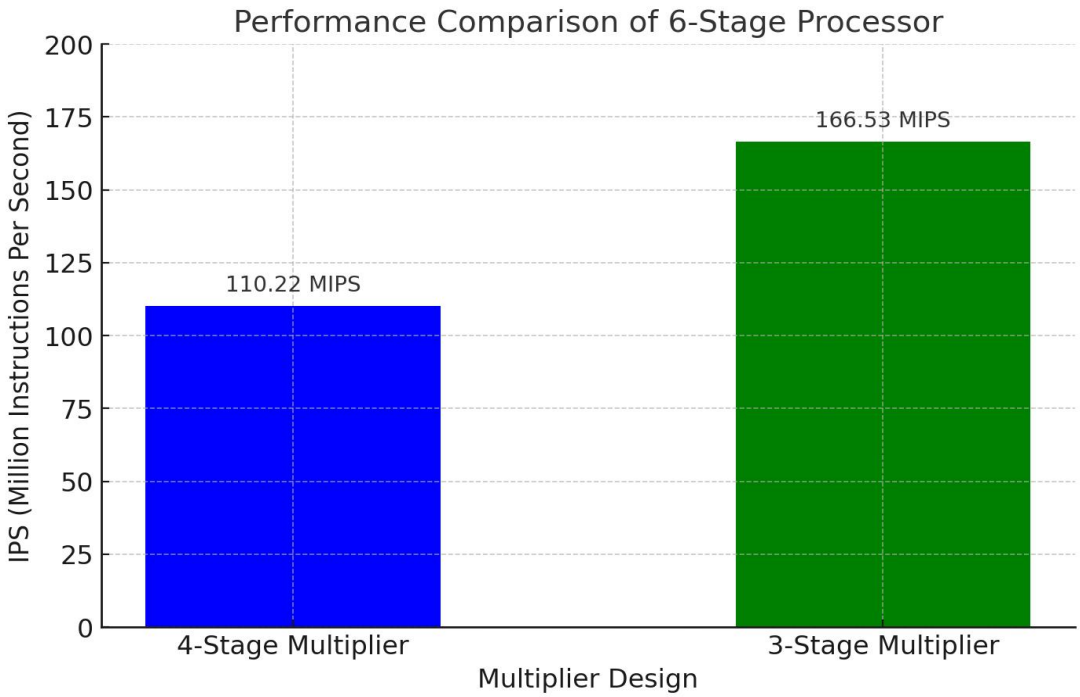


Figure 4.6 Performance Comparison of 6-Stage processor with different Multipliers

Here is the performance comparison plot of a 6-stage pipelined processor using 4-stage and 3-stage multipliers. The 3-stage multiplier achieves a significantly higher IPS (166.53 MIPS) compared to the 4-stage multiplier (110.22 MIPS), reflecting the superior speed of the 3-stage design for FPGA implementations where performance is the primary concern.

CHAPTER 05

RSA Key Generation Algorithm

5.1. Introduction

RSA (Rivest–Shamir–Adleman) is one of the most widely adopted public-key cryptographic algorithms, first introduced in 1977. It falls under asymmetric encryption, where a pair of keys — one public and one private — is used for encryption and decryption respectively. Unlike symmetric encryption algorithms like AES that use the same key for both operations, RSA provides a more secure key exchange and is often used for authentication, digital signatures, and secure key distribution.

5.2. General Overview

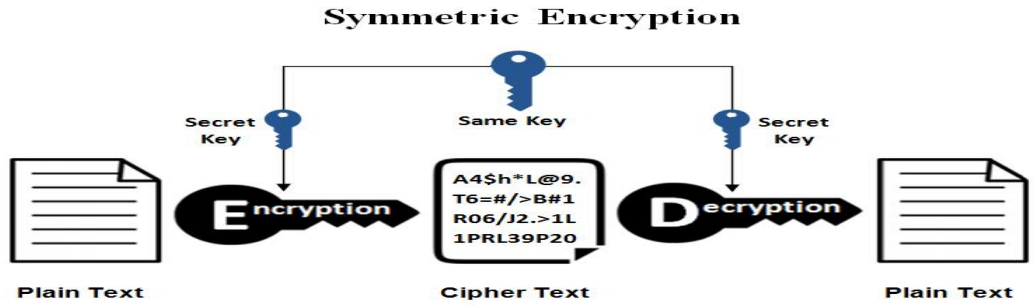


Figure 5.1 Symmetric Key Encryption process

Cryptography is divided into **symmetric** and **asymmetric** systems. Symmetric encryption uses **one key** for both encryption and decryption — it's fast but risky due to **key sharing issues**.

Asymmetric encryption solves this by using **two keys**: a **public key** for encryption and a **private key** for decryption. This makes communication more secure, especially over open networks

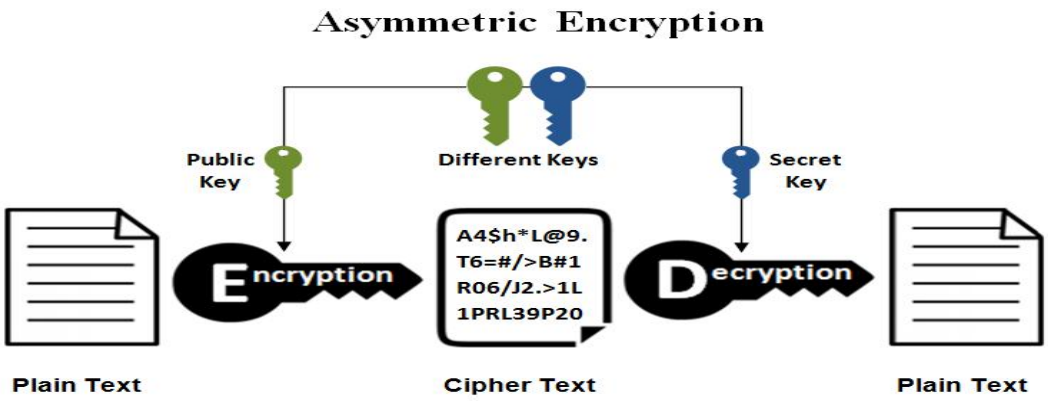


Figure 5.2 Asymmetric Key Encryption process

RSA (Rivest–Shamir–Adleman) is a popular Asymmetric algorithm. It's based on the mathematical challenge of factoring large prime numbers, making it secure and reliable. RSA is commonly used for secure key exchange, digital signatures, and authentication in systems like SSL/TLS, email encryption, and smart cards.

5.3. Implementation of RSA

RSA works in three main steps:

5.3.1 Key Generation

- ✓ Choose two large prime numbers p and q.
- ✓ Compute $n = p \times q$ and $\varphi(n) = (p-1)(q-1)$.
- ✓ Select a public exponent e (commonly 65537) such that $\gcd(e, \varphi(n)) = 1$.
- ✓ Calculate the private exponent d using the extended Euclidean algorithm so that $d \times e \equiv 1 \pmod{\varphi(n)}$.
- ✓ Public key = (e, n), Private key = (d, n).

5.3.2 Encryption

- ✓ Convert the message to an integer M such that M < n.
- ✓ Compute ciphertext: C = M^e mod n.

5.3.3 Decryption

- ✓ Recover original message: M = C^d mod n.

RSA ensures security through the difficulty of factoring large numbers, making it ideal for **secure key exchange and authentication**.

5.4. MATLAB Simulation for Validation

Before implementing RSA in Verilog, a MATLAB simulation was developed to validate the algorithm's functional correctness. The simulation included the complete RSA flow: key generation, encryption, and decryption. Standard mathematical functions were used to generate large prime numbers, compute public-private key pairs, and perform modular exponentiation.

```
p = 61;
q = 53;

n = p * q;

phi_n = (p - 1) * (q - 1);

e = 17;

d = modInverse(e, phi_n);

disp('Public Key:');
disp(['e = ', num2str(e), ', n = ', num2str(n)]);

disp('Private Key:');
disp(['d = ', num2str(d), ', n = ', num2str(n)]);

function inv = modInverse(a, m)
    [g, x, ~] = gcdExtended(a, m);
    if g ~= 1
        error('Modular inverse does not exist!');
    else
        inv = mod(x, m);
    end
end

function [g, x, y] = gcdExtended(a, b)
    if a == 0
        g = b;
        x = 0;
        y = 1;
        return;
    end
    [g, x1, y1] = gcdExtended(mod(b, a), a);
    x = y1 - floor(b / a) * x1;
    y = x1;
end
```

Figure 5.3 Matlab code for RSA Key generation Algorithm

Name	Value
d	2753
e	17
n	3233
p	61
phi_n	3120
q	53

Figure 5.4 Matlab Result, Private and Public RSA Keys

This simulation served as a **baseline reference model** for verifying the correctness of future hardware (Verilog) implementation. Sample messages were encrypted using the public key and then decrypted with the private key, ensuring that the original plaintext was accurately recovered. The consistency between encrypted-decrypted pairs in MATLAB confirmed the algorithm's reliability and helped identify potential edge cases before moving to RTL design.

5.5. Conclusion

RSA remains a cornerstone of modern cryptography, offering a reliable method for secure communication through public and private key pairs. The MATLAB-based simulation confirmed the algorithm's correctness and provided a strong foundation for hardware implementation. Overall, RSA plays a critical role in secure systems, and understanding its operation is essential for designing robust embedded cryptographic architectures.

Chapter 06
Register Transfer Level (RTL) Design for AES Encryption and Decryption on RISC-V

6.1 Introduction

Register Transfer Level (RTL) design serves as a foundational abstraction in digital circuit design, particularly when implementing cryptographic algorithms like the Advanced Encryption Standard (AES). In the context of integrating AES with a RISC-V processor, RTL design facilitates the precise modeling of data flow and control logic, ensuring efficient and secure encryption and decryption processes.

6.2 Overview of AES Algorithm

The AES algorithm is a symmetric block cipher standardized by NIST, operating on fixed-size blocks of 128 bits and supporting key sizes of 128, 192, or 256 bits. The algorithm comprises multiple rounds of processing, each involving the following transformations:

6.2.1 SubBytes:

In the AES algorithm, SubBytes is a nonlinear byte substitution step where each byte in the 4x4 state matrix is replaced using an S-box (Substitution Box). This operation increases the confusion in the cipher, which is essential for cryptographic strength. The substitution is done independently for each of the 16 bytes using a look-up table.

6.2.2 ShiftRows:

The ShiftRows step is one of the core transformations in the AES encryption process. It is designed to rearrange the bytes within the state matrix, helping to enhance diffusion across columns. Specifically, it cyclically shifts the rows by varying offsets—while the first row stays unchanged, the second shifts by one byte, the third by two, and the fourth by three. This operation spreads byte influence across multiple columns, increasing resistance to crypt analytic attacks by mixing data further before the next transformation step.

6.2.3 MixColumns:

MixColumns is a transformation in AES that operates on each column of the state matrix to ensure diffusion—meaning, a small change in the input causes widespread changes in the output. This step applies finite field (Galois Field GF(2^8)) arithmetic to combine the four bytes of each column. The result is that each output byte depends on all four input bytes of the same column, effectively spreading information and making the cipher more secure against crypt analysis.

6.2.4 AddRoundKey:

The AddRoundKey step is designed to combine the AES state with a key that is specific to the current encryption round. This is achieved using a bitwise XOR operation, which introduces key-dependent confusion into the encryption process. By applying a unique portion of the expanded key in each round, this step ensures that the output becomes increasingly difficult to predict or reverse without the correct key. Implementing these steps in hardware requires careful RTL design to ensure correct functionality and performance.

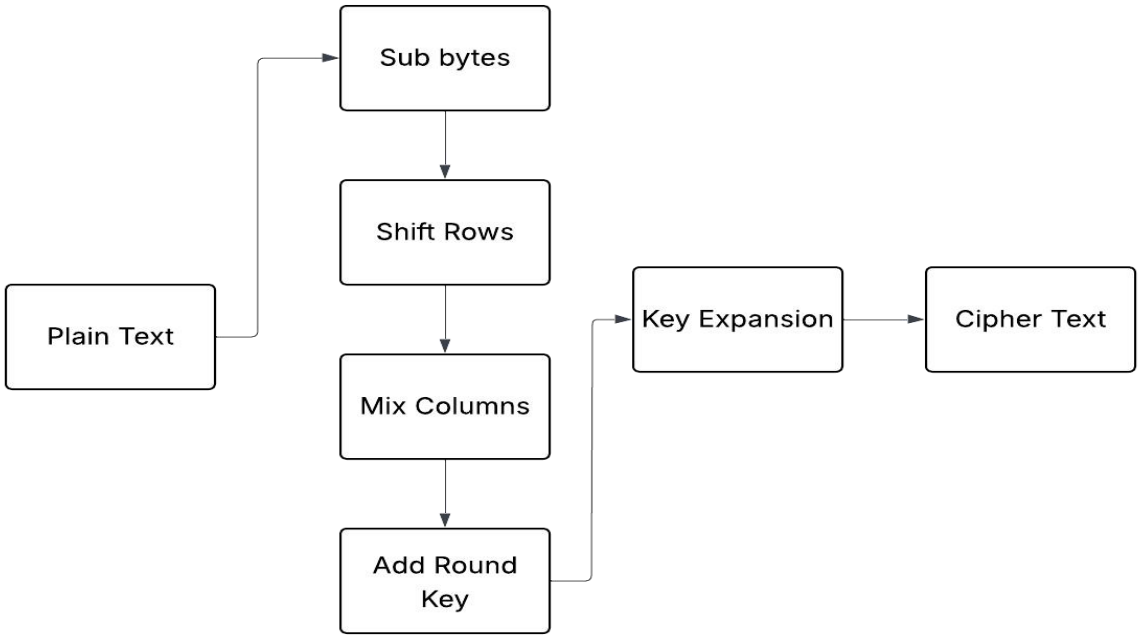


Figure 6.1 AES Encryption/Decryption RTL Block Diagram

6.3 MATLAB Simulation for Functional Validation

To ensure the correctness of each AES transformation stage implemented in RTL, a MATLAB-based simulation was developed. This served as a reference model to verify the outputs of both encryption and decryption processes. The simulation mirrors the internal steps of AES, such as SubBytes, ShiftRows, MixColumns, and AddRoundKey across multiple rounds.

The results from the MATLAB model were compared against the RTL outputs for consistency. As shown in the images, intermediate states and final outputs (both cipher and decrypted text) align with expected behavior, confirming the logical accuracy of the implemented RTL design.

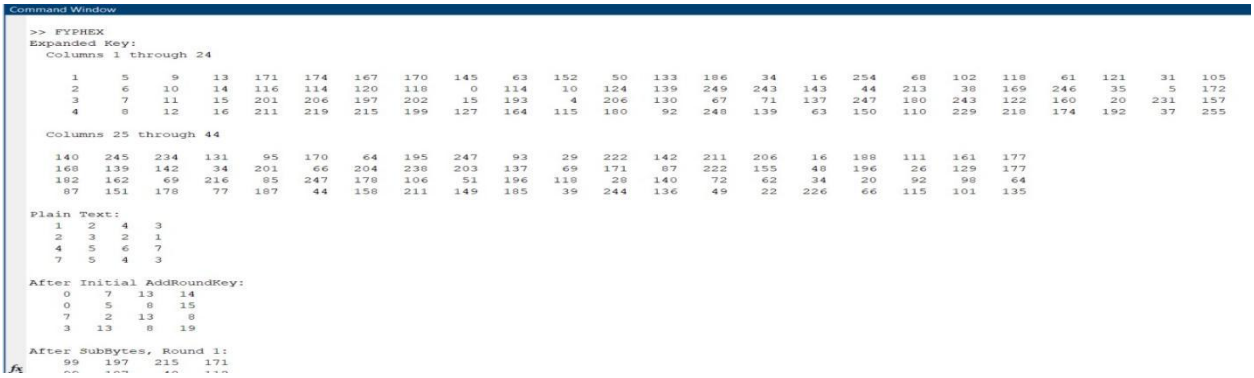


Figure 6.2 AES Encryption Algorithm RTL Matlab Code

In this figure the it shows us the the simple expanded key and the whole data is about 128 bytes. Having a matrix of 4x4 each. There is a 4x4 Plain text that works as a input. At first we do the AddRoundKey for round 1

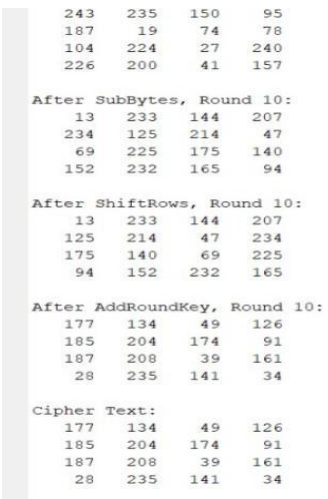


Figure 6.3 AES Encryption Algorithm RTL Matlab Code , Sub-bytes, Shift-rows, Add-round-key, Cipher Text

After doing the whole cycle of for 10 rounds we get the Cipher text.

```
After inv_MixColumns, Round 2:
    99   197   215   171
   107    48   118    99
   215    48   197   119
   125   123   215    48

After inv_ShiftRows, Round 1:
    99   197   215   171
    99   107    48   118
   197   119   215    48
   123   215    48   125

After inv_SubBytes, Round 1:
     0     7    13    14
     0     5     8    15
     7     2    13     8
     3    13     8    19

After inv_AddRoundKey, Round 1:
     1     2     4     3
     2     3     2     1
     4     5     6     7
     7     5     4     3

decrypted Text:
     1     2     4     3
     2     3     2     1
     4     5     6     7
     7     5     4     3
```

Figure 6.4 AES Decryption Algorithm RTL Matlab Code

After getting encrypted text we do the whole process backwards from round 10 to 1, we finally get the decrypted text.

6.4 RTL Design of AES Components (MATLAB code to RISC-V Assembly language)

6.4.1. SubByte Module

```
sub_byte:
    la t0, sbox              # Load address of S-box
    la t1, state             # Load address of the state matrix

sub_byte_loop:
    lb t2, 0(t1)             # Load a byte from the state
    add t2, t0, t2           # Calculate address in S-box
    lb t2, 0(t2)             # Substitute byte using S-box
    sb t2, 0(t1)             # Store substituted byte back in state
    addi t1, t1, 1           # Move to next byte
    blt t1, state+16, sub_byte_loop  # Repeat for all 16 bytes

    ret

sub byte codes
```

```
%subbyte
function out = sub_byte(in, sbox)
    out = zeros(4, 4);  % Initialize output matrix
    for j = 1:4
        for i = 1:4
            byte = in(i,j);  % Get byte value
            row = bitshift(byte, -4);  % First nibble (high 4 bits)
            col = bitand(byte, 15);  % Second nibble (low 4 bits)
            out(i, j) = sbox(row + 1, col + 1);  % Substitute using S-box
        end
    end
end
```

Figure 6.5 Sub-Bytes MATLAB code to RISC-V Assembly Conversion

The RISC-V assembly implementation of SubBytes is structured to mimic the logic of the MATLAB function while optimizing it for hardware-level execution.

Initialization: The program begins by loading the memory addresses of both the S-box and the AES state matrix into temporary registers (t0 and t1).

Processing Loop: A loop runs through each of the 16 bytes in the state. Within the loop:

- A byte is fetched from the state matrix.
- Its value is used as an offset to find the corresponding substituted byte in the S-box.
- The new substituted value is then written back to the same position in the state matrix.

Pointer Increment: After each substitution, the state matrix pointer moves to the next byte.

Loop Control: The loop continues until all 16 bytes (128 bits) of the state have been processed.

This approach ensures efficient substitution using direct memory operations, making it suitable for fast and secure AES hardware acceleration.

6.4.2. Shift Rows Module

```
shift_rows:
    la t0, state

    # Row 1: Rotate left by 1
    lb t1, 4(t0)
    lb t2, 5(t0)
    lb t3, 6(t0)
    lb t4, 7(t0)
    sb t2, 4(t0)
    sb t3, 5(t0)
    sb t4, 6(t0)
    sb t1, 7(t0)
    ret
```

shift rows code

```
%shift_rows
function out=shift_rows(in)
out=zeros(4,4);
        out(1,:)=[in(1,1) in(1,2) in(1,3) in(1,4)];
        out(2,:)=[in(2,2) in(2,3) in(2,4) in(2,1)];
        out(3,:)=[in(3,3) in(3,4) in(3,1) in(3,2)];
        out(4,:)=[in(4,4) in(4,1) in(4,2) in(4,3)];
end
```

Figure 6.6 Shift Rows MATLAB code to RISC-V Assembly Conversion

The RISC-V code implements the rotation of the second row (index 1) of the state matrix to the left by one byte. Here's how it works:

Address Setup: The base address of the AES state matrix is loaded into register t0.

Byte Loading: The instruction set sequentially loads the four bytes of the second row (located at offsets 4, 5, 6, and 7 from the base) into registers t1 through t4.

Row Rotation: The rotation effect is achieved by storing the loaded bytes back into the memory at shifted positions. For example, the byte originally at position 5 is written to 4, the byte at 6 goes to 5, and so on, completing a leftward circular shift.

Termination: After the updated bytes are stored, the function ends with a ret instruction.

This logic closely reflects the behavior of the MATLAB version, where rows of the state matrix are shifted left by an increasing number of positions per row. In RTL/hardware terms, such row-wise rotations help spread the byte values more uniformly across the encryption rounds.

6.4.3. Mixed Column Module

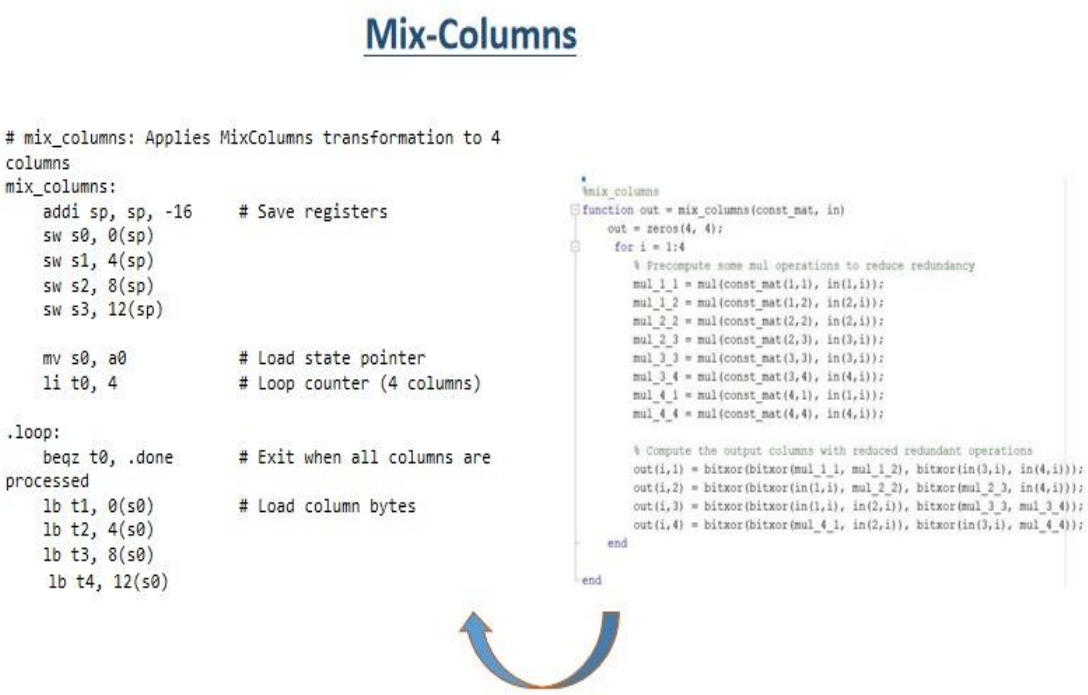


Figure 6.7 Mix Columns MATLAB code to RISC-V Assembly Conversion

```
# Compute new column values using MixColumns matrix
   mv a0, t1            # 2*t1 + 3*t2 + t3 + t4
   call gmul2
   mv s1, a1
   mv a0, t2
   call gmul3
   xor s1, s1, a1
   xor s1, s1, t3
   xor s1, s1, t4

   mv a0, t2            # 2*t2 + 3*t3 + t1 + t4
   call gmul2
   mv s2, a1
   mv a0, t3
   call gmul3
   xor s2, s2, a1
   xor s2, s2, t1
   xor s2, s2, t4
```

```
mv a0, t3            # 2*t3 + 3*t4 + t1 + t2
    call gmul2
    mv s3, a1
    mv a0, t4
    call gmul3
    xor s3, s3, a1
    xor s3, s3, t1
    xor s3, s3, t2

    mv a0, t4        # 2*t4 + 3*t1 + t2 + t3
    call gmul2
    mv s4, a1
    mv a0, t1
    call gmul3
    xor s4, s4, a1
    xor s4, s4, t2
    xor s4, s4, t3
```

Figure 6.8 Intermediary Mix columns MATLAB code to RISC-V Assembly Conversion

```
  # Store back transformed column
    sb s1, 0(s0)
    sb s2, 4(s0)
    sb s3, 8(s0)
    sb s4, 12(s0)

    addi s0, s0, 1      # Next column
    addi t0, t0, -1
    j .loop

.done:
    lw s0, 0(sp)        # Restore registers
    lw s1, 4(sp)
    lw s2, 8(sp)
    lw s3, 12(sp)
    addi sp, sp, 16
    ret
```

```
# gmul2: Multiplies a byte by 2 in GF(2^8)
gmul2:
    slli a1, a0, 1        # Multiply by 2
    andi t1, a0, 0x80     # Check if MSB is set
    beqz t1, .done        # If not, return
    xor a1, a1, 0x1B      # If MSB=1, XOR with
AES polynomial
.done:
    ret

# gmul3: Multiplies a byte by 3 in GF(2^8)
gmul3:
    call gmul2
    xor a1, a1, a0
    ret
```

Figure 6.9 G-mul function Mix columns MATLAB code to RISC-V Assembly Conversion

The RISC-V implementation of MixColumns starts by setting up the environment and then performs column-wise transformations:

Saving Registers: The first few lines push the contents of general-purpose registers onto the stack to preserve them during execution.

State Pointer and Counter: The base address of the AES state matrix is moved into a register, and a loop counter is initialized to handle all 4 columns.

Column Byte Loading: Within the loop, four bytes from a single column of the state matrix are loaded into temporary registers (t1 to t4). These represent one column of the 4x4 AES state matrix.

Matrix Multiplication (implied in extended code): Although not shown in full here, the next step would typically involve multiplying the loaded bytes with fixed values from the AES MixColumns matrix (such as 2, 3, 1, 1 in GF(2^8)).

Combining Results: After the multiplications, the results are XORed (bitwise addition in GF arithmetic) to compute the final column output.

Loop Execution: The process repeats for each of the four columns, with the pointer incrementing after each set of four bytes.

This hardware-level execution mirrors the MATLAB function, which performs multiplications and XORs to mix each column. The assembly version is carefully crafted to handle each transformation efficiently using memory operations and bitwise logic, ideal for hardware or low-level firmware implementations

6.4.4. Add Round Key Module

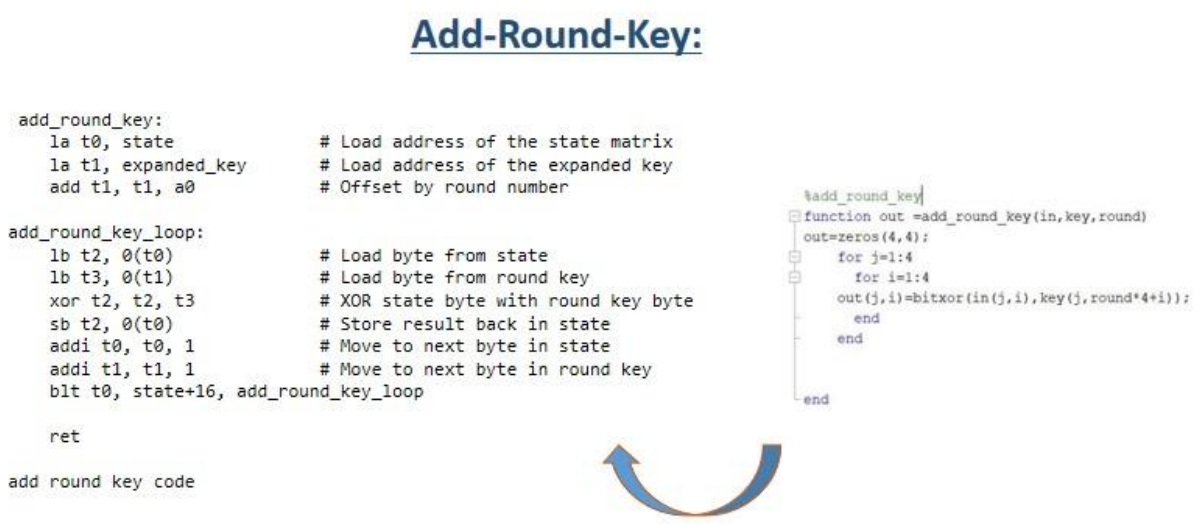


Figure 6.9 Add-round-key MATLAB code to RISC-V Assembly Conversion

Address Initialization: The starting address of the AES state matrix is placed into register t0, while the beginning of the expanded key is loaded into t1. The round number, stored in a0, is used to calculate the correct position within the expanded key for the current round.

State-Key Integration: The loop goes through all 16 bytes of the state. During each iteration, one byte from the state and its corresponding round key byte are fetched.

Bitwise Combination: Each pair of bytes (from the state and round key) is combined using an XOR operation. This new value replaces the original byte in the state matrix, effectively embedding the round-specific key into the state.

Function Exit: After processing all bytes, the loop exits, and the function concludes with a ret instruction.

This process closely aligns with the MATLAB version, where the state is updated by XORing it with the round key. From a hardware perspective, this transformation is implemented using parallel XOR gates, allowing for efficient and high-speed operation during encryption cycles.

6.5 RTL Modules

6.5.1 Key Expansion

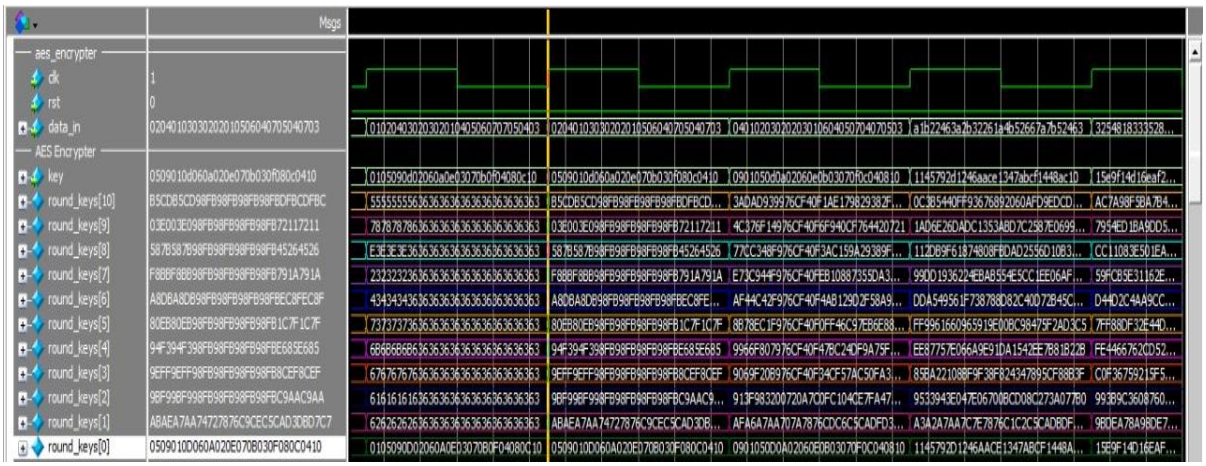


Figure 6.10 AES Encryption 10 Rounds Key Expansion RTL Module

Key Functionalities:

Input/Output Interface

- Takes a 128-bit cipher key and produces 11 round keys (each 128 bits) for AES-128.
- Operates on a clock (clk) and reset (rst) for synchronization.

Core Operations

- Initial Key Loading: The first 4 words (128 bits) are directly derived from the input cipher key.
- SubWord & RotWord: For subsequent keys, the last word of each 4-word group undergoes:
- Byte substitution (using S-box via sub_word module).

- Rotation (cyclic shift).
- Rcon XOR: A round constant (Rcon) is applied to introduce non-linearity.
- Recursive XOR Chaining: Each new word is derived by XORing previous words with transformed intermediates.

Pipelined Structure

- Uses sequential logic (always_ff @(posedge clk) for step-by-step key derivation.
- Generates 10 round keys iteratively (44 words total, grouped into 11 keys).

Reset Handling

- Clears all registers (exp_key, out) when reset (rst) is active.

Design Highlights

- Efficiency: Parallel sub-word transformations and XOR operations reduce latency.
- Modularity: The sub_word module (S-box) is instantiated for reuse.
- Scalability: Easily extensible for AES-192/256 with adjusted loop bounds.

6.5.2 Sub Modules (subByte, ShiftRows, MixColumns, AddRoundKey)

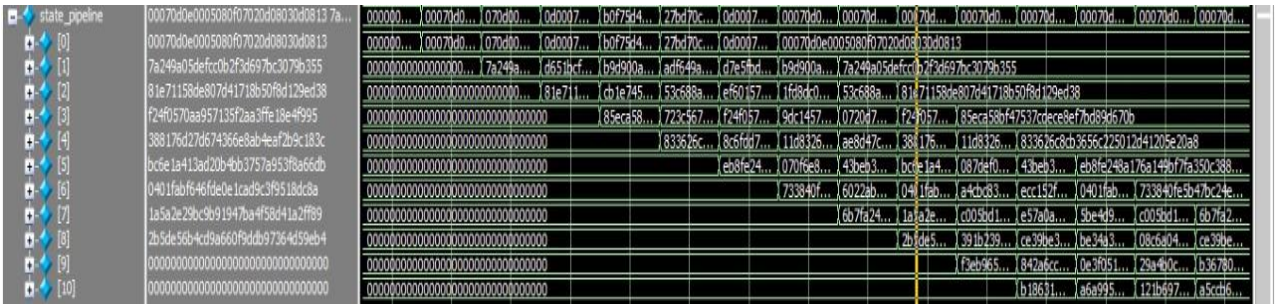


Figure 6.11 AES Encryption sub-byte, shift-rows, mix-columns, add-round-key RTL Module

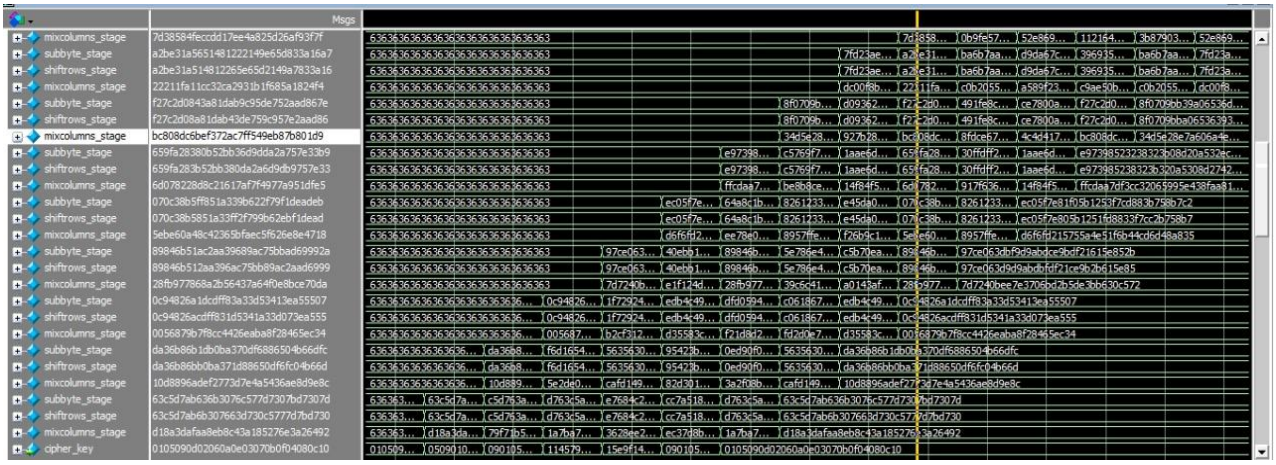


Figure 6.12 AES Encryptor 10 Rounds State Pipeline RTL Module

6.5.3 AES Encypter

The AES Encrypter module is responsible for performing the full AES-128 encryption in hardware using RTL design principles. This module accepts a 128-bit plaintext input along with a 128-bit cipher key. The encryption process begins with the key expansion block, which generates eleven round keys from the original cipher key. These round keys are essential because AES performs multiple rounds of transformation, and each round requires its own key. The key expansion is handled by a separate module that continuously derives the necessary keys before the main encryption starts.

Once the round keys are ready, the encryption begins with an initial AddRoundKey stage. In this step, the input data is simply XORed with the original cipher key to mix

the plaintext and key. The result is stored in the first stage of a pipeline register, preparing it for the subsequent rounds. The design uses sequential flip-flops that are triggered by the clock signal to store the intermediate data at each stage. This pipelined structure allows the encryption process to handle multiple data blocks efficiently, where each block proceeds stage by stage without interfering with others.

The main body of the AES algorithm consists of nine rounds. In each round, three core operations are performed: SubBytes, ShiftRows, and MixColumns. The SubBytes operation substitutes each byte using a predefined substitution box (S-box), introducing non-linearity into the system. ShiftRows then shifts the rows of the matrix to shuffle the data and spread byte positions. Following this, MixColumns mixes the bytes across columns to further enhance data diffusion. After completing these transformations, the state is XORed with the appropriate round key for that round. Each round's output is again stored in the pipeline using flip-flops to maintain proper data synchronization.

The final round differs slightly from the previous ones. It follows the same SubBytes and ShiftRows operations but skips MixColumns, as required by the AES specification for the last round. The processed data is finally XORed with the last round key, producing the encrypted output. A separate valid signal is pipelined alongside the data to ensure that the module accurately indicates when the output data is valid and ready. The reset signal is used to clear all the registers and bring the system back to its initial state whenever necessary, ensuring clean operation on every new encryption cycle.

Overall, this design combines pipelining, sequential logic, and modular blocks to achieve efficient and reliable AES encryption in hardware. The careful synchronization of data using flip-flops allows the module to handle high-speed operations without data collision or timing issues.

6.5.4 AES Decrypter

The AES Decrypter module is designed to perform the full AES-128 decryption process in hardware using RTL design techniques. Similar to the encryption module, it takes a 128-bit cipher text and the same 128-bit cipher key as inputs. The decryption process begins with a key expansion module, which generates all 11 round keys. However, since AES decryption requires applying these keys in reverse order compared to encryption, the module uses internal storage (key_storage) to store and correctly align the keys for each round during decryption.

Before starting decryption, the input cipher text is first delayed and stored in pipeline registers (data_in_store and valid_in_store) to match the timing with key generation and internal processing. This ensures proper synchronization between data and key flow. The decryption starts with an initial AddRoundKey operation where the input cipher text is XORed with the last round key (round_keys[10]), essentially reversing the initial encryption step.

The core decryption process consists of nine rounds, where each round applies the

Figure 6.13 AES Decryptor 10 Rounds State Pipeline RTL Module

Inverse operations of encryption. Each round first performs the Inverse ShiftRows operation, which reverts the byte shifts done during encryption, followed by the Inverse SubBytes that reverses the byte substitution using the inverse S-box. After these two operations, the round key is applied using XOR. The round keys are carefully selected from key_storage to ensure that the correct key is applied in each round in the reverse sequence. Finally, Inverse MixColumns is performed to undo the column mixing from encryption. Each round's output is stored in pipeline registers to maintain synchronization across the stages.

The final round is slightly different, as it skips the Inverse MixColumns operation. It applies only the Inverse ShiftRows and Inverse SubBytes, followed by the last AddRoundKey using the first round key. The pipelined valid signals ensure that the module indicates when valid decrypted data is available at the output. Throughout the design, flip-flops are used to store intermediate states and valid signals, ensuring proper timing and preventing data collision. The reset signal is included to clear all internal registers, preparing the system for fresh decryption operations.

In summary, the AES Decrypter module mirrors the encryption process but carefully reverses the order of operations. It combines modular design, pipelining, and careful key scheduling to perform efficient and reliable decryption in hardware.

6.5.5 Key Storage

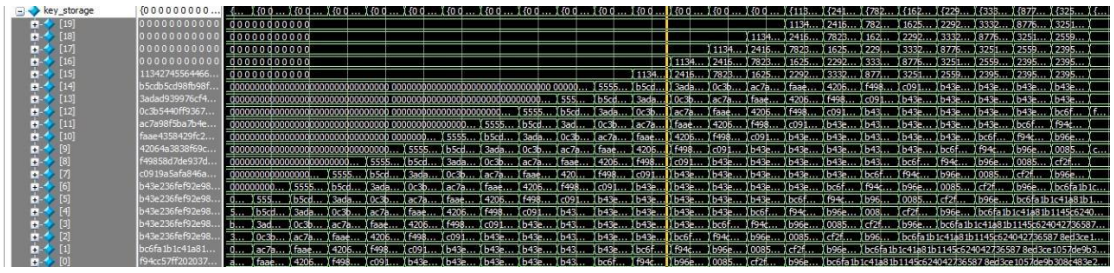


Figure 6.14 AES Key Storage RTL Module

6.6. Conclusion

RTL design plays a pivotal role in the hardware implementation of cryptographic algorithms like AES on RISC-V processors. By meticulously designing each component at the RTL level, we can achieve efficient, secure, and high-performance encryption and decryption Functionalities. The integration of these modules with RISC-V processors further enhances the system's capability to handle cryptographic operations, making it suitable for applications requiring robust security.

Chapter 7

Secure Embedded Cryptographic System

This chapter presents the design and validation of our secure embedded cryptographic system built around a RISC-V processor interfaced with a custom AES hardware accelerator. The system is implemented on the DE1-SoC FPGA and is capable of handling encryption and decryption instructions issued directly from the processor. The design uses a modular architecture allowing efficient memory-mapped interaction between the processor and the AES unit.

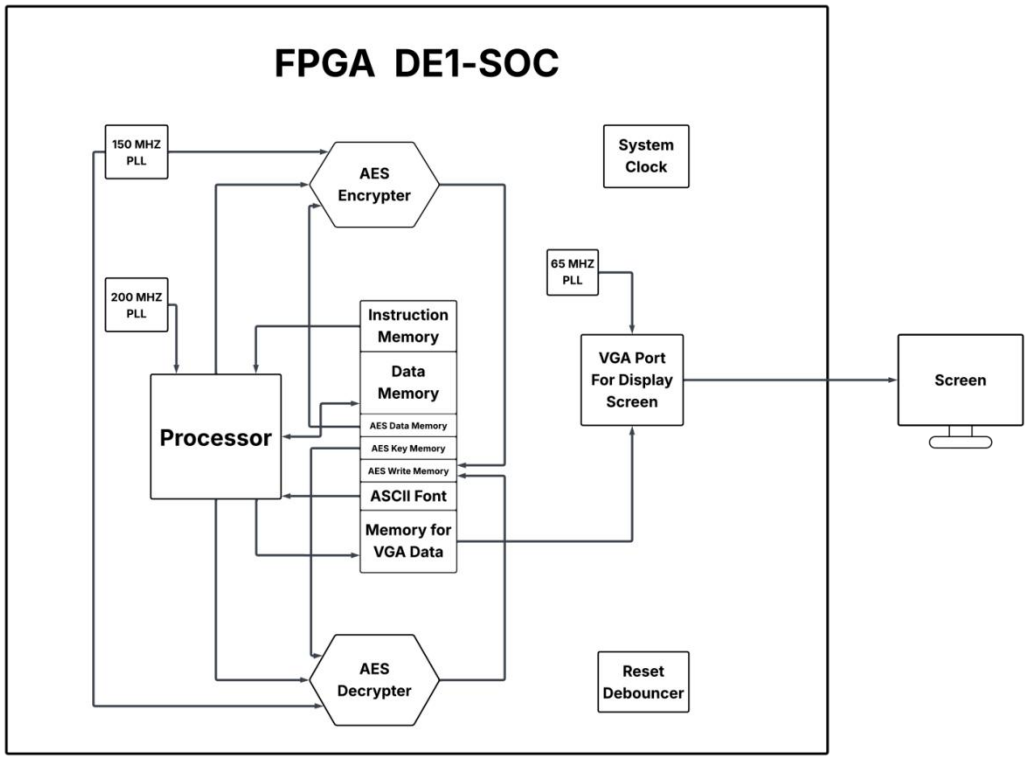


Figure 7.1 Secure Embedded Cryptographic System Block Diagram

7.1 Integration:

7.1.1 System Overview

The main architecture of our design is illustrated in the block diagram (Fig. X.X). It includes the following components:

- **RISC-V Processor:** Acts as the controller and decision-making unit.
- **AES Encrypter and Decrypter Modules**: Dedicated hardware accelerators for symmetric encryption and decryption.
- **Instruction and Data Memory:** Includes sections for AES input/output, keys, and display data.
- **Memory-Mapped I/O Interface:** Facilitates communication between processor and AES.
- **VGA Interface and PLLs:** For displaying decrypted output on screen and managing clock domains.

The processor is responsible for issuing commands to the AES hardware accelerator. This is done through a well-defined protocol involving memory-mapped registers and dedicated control signals.

The Main System Clock is **50MHz**, while the Clock for Processor is **200MHz**, AES Encrypter/Decrypter **150 MHZ** and VGA **65MHz** converted through a PLL

7.2 RISC-V to AES Accelerator Communication

The processor sends encryption/decryption instructions to the AES unit by providing

- AES Encrypt/Decrypt command
- Write Enable
- Data Address (source)
- Key Address
- Write Address (destination)
- Write Data

Before an encryption or decryption command is issued, the processor sets:

- Register 29 → Data Address
- Register 30 → Key Address
- Register 31 → Write Address

These registers are directly wired to the AES hardware. Thus, once the command is triggered, the AES accelerator fetches the appropriate data and key automatically.

7.3 Command Protocol and Control Signals

The memory-mapped interface uses a specific address (address 77) to trigger encryption or decryption:

Bit 10 HIGH → Execute Encryption
Bit 11 HIGH → Execute Decryption
Bits [0–9] → Represent the number of 32-bit words to process

For data retrieval, the processor accesses address 56 using a Read Data command, which fetches the output from the AES write buffer.

This protocol ensures streamlined control where a single write to a mapped register can trigger a full encryption or decryption sequence without software-side data handling overhead.7.4 MMIO PORT READ WRITE ADDRESS AND MAPPING

7.4.1 AES ENCRPT AND DECRYPT METHOD

To give a encrypt command from RISC-V to AES 5 types of data should be placed and assigned at correct location

1. Read Data Address (from where AES will fetch its data to encrypt)
2. Read KEY Address (from where AES will fetch its key use to encrypt)
3. Write Data Address (where AES will write the Encrypted Data)
4. No of Words (how many consecutive words should be encrypted “1 word = 128

bits")

5. Encrypt and Decrypt Bit set to 1

Registers	Data
X29	Read Data Address
X30	Read Key Address
X31	Write Data Address

Table 7.1 Registers Read Write Table

Store word command will be used to send Encrypt and Decrypt Signal line with writing address of 77

sw x14 , 77(x0) #the no words and encrypt and decrypt bits are in x14

Remaining bits	Decrypt bit	Encrypt bit	no of words
31:12	11	10	9:0

Table 7.2 Memory mapped interface Triggers Encryption/Decryption

RISC-V commands AES hardware accelerator by writing data on x77 address, If 11^{th} bit is high then Decryption is triggered in the AES Hardware accelerator, if the 10^{th} bit is high then Encryption is triggered. Remaining 0-9 bits determine the total no.of words

7.4.2 Write VGA Port

The write for VGA port is 7756 in decimal when we write any data it will select the 8 bit ASCII code from it it can be 7:0 or 15:8 or 23:16 or 31:24 depending on what is store in x28's last 2 bit. We will use store word command for this like

```
addi    x12, x0 , 2000
addi    x13, x0 , 2000
addi    x14, x0 , 2000
addi    x15, x0 , 1756
add     x16, x12, x13
add     x17, x14, x15
addi    x5 , x0 , 0
add     x5 , x16, x17     #address 7756
addi   x28, x0,0
addi   x9,  x0, 54
sw      x9 , 0(x5)      #writing at VGA byte by byte
```

This code will write the ASCII character 54 in VGA memory the screen address of VGA is sequential so what ever we write will be display sequentially.

7.4.3 READ and WRITE AES memory

There are three memories in AES namely Data , key and Output storing memory from which Data and Key can be written by the processor while output can only be written by AES encrypted and decrypted data while all three memories can be read by the processor.

READING

The base address for reading is 56 in decimal which is 0x38 in hex so last two bits is used to define which memory data is asked by the processor if its 00 then output memory data 01 then key memory data if 10 the data memory data.

Base Address			Reading Different Memory
31:8	7:4	3:2	1:0
0000 0000 0000 0000 0000 0000	0011	10	00
0000 0000 0000 0000 0000 0000	0011	10	01
0000 0000 0000 0000 0000 0000	0011	10	10

Table 7.3 Reading Data from AES Memory

WRITING

If 16 bit of x29 is high the data will be written at Key memory if its low the data will be written at Data memory the base address to write at AES memories is 56 so the expected command will look like

```
addi   x29, x0  , 2000
addi   x24, x0  , 0x7ff
sw      x24 , 56(x0)
```

The following command will store 0x7ff which is 2047 in decimal at location 2000 of data memory as the 16 bit of x29 is low.

7.5 Testing:

7.5.1 Simulation Results

The functionality of the integrated system was validated through simulation. Test cases included:

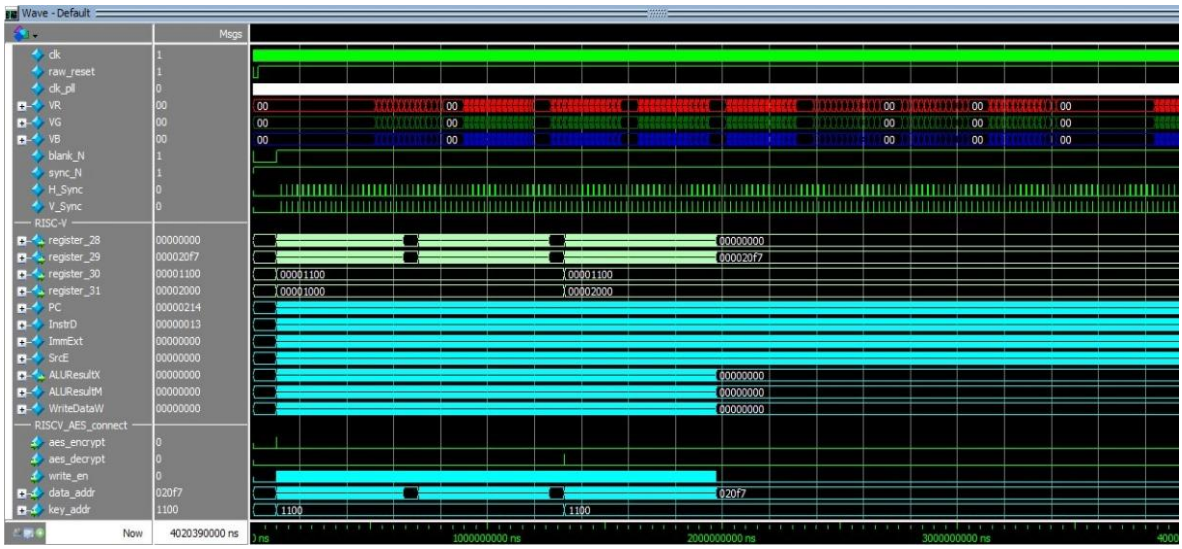


Figure 7.2 Simulation of Overall System

Light Blue colored Signals indicate the RISC -V Signals, the initial RGB signals are for the VGA output. The orange colored signals are for AES, Purple signals indicate VGA Write and Data Memory

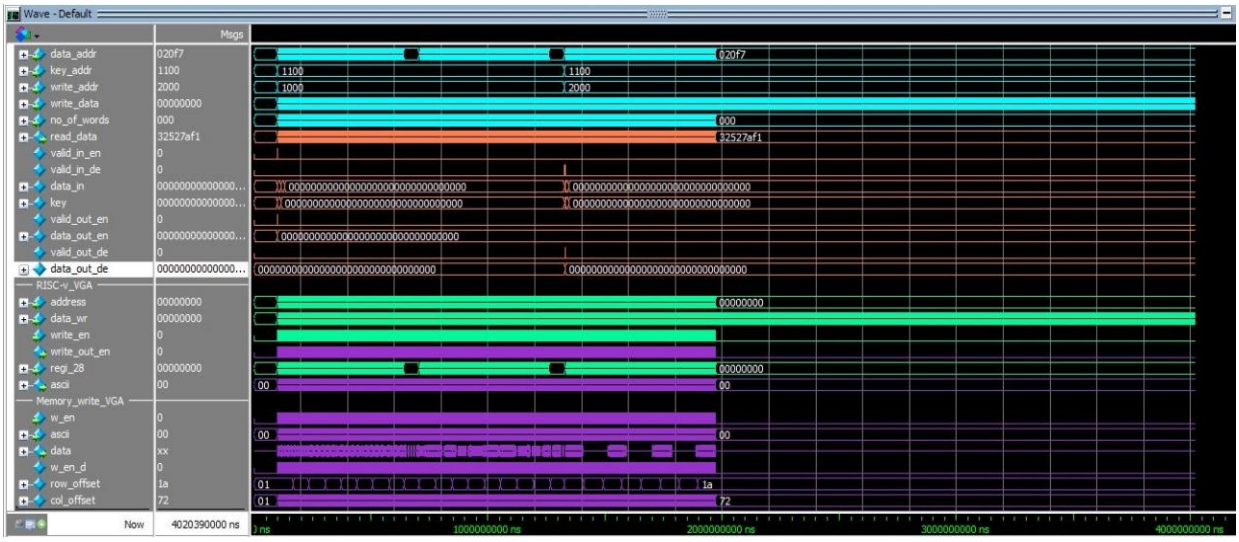


Figure 7.3 RISC-V Commands AES Hardware Accelerator Simulation

Figure 7.4 RISC-V Commands AES Encryption Simulation

This Initial Cypher text provided to AES Hardware Accelerator is Encrypted as seen in the Above simulation. RISC-V through an Assembly Language code commands AES to Encrypt the data provided to it

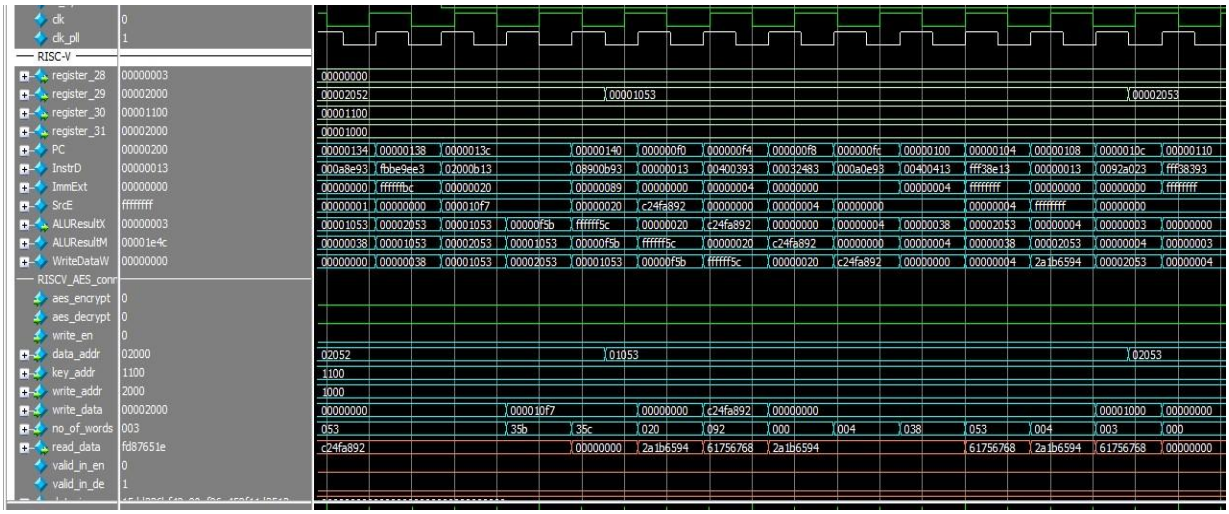


Figure 7.5 AES Encryption in Hardware Accelerator Simulation

The following RISC-V code is used for Commanding AES Encryption

```
# code to Encrypt 62 word placed at 0x1000 and key at 0x1100
# Encrypted data will be stored at 0x1000
addi   x29, x0 , 0x7ff
addi   x30, x0 , 0x7ff
addi   x31, x0 , 0x7ff
addi   x29, x29, 0x2
addi   x30, x30, 0x102
addi   x31, x31, 0x2
addi   x2 , x0 , 0x440   #encrypt 62 words command
addi   x29, x29, 0x7ff   #data addr
addi   x30, x30, 0x7ff   #key addr
addi   x31, x31, 0x7ff   #write addr
nop
nop
nop
sw     x2 , 77(x0)      #AES Encrypt
```

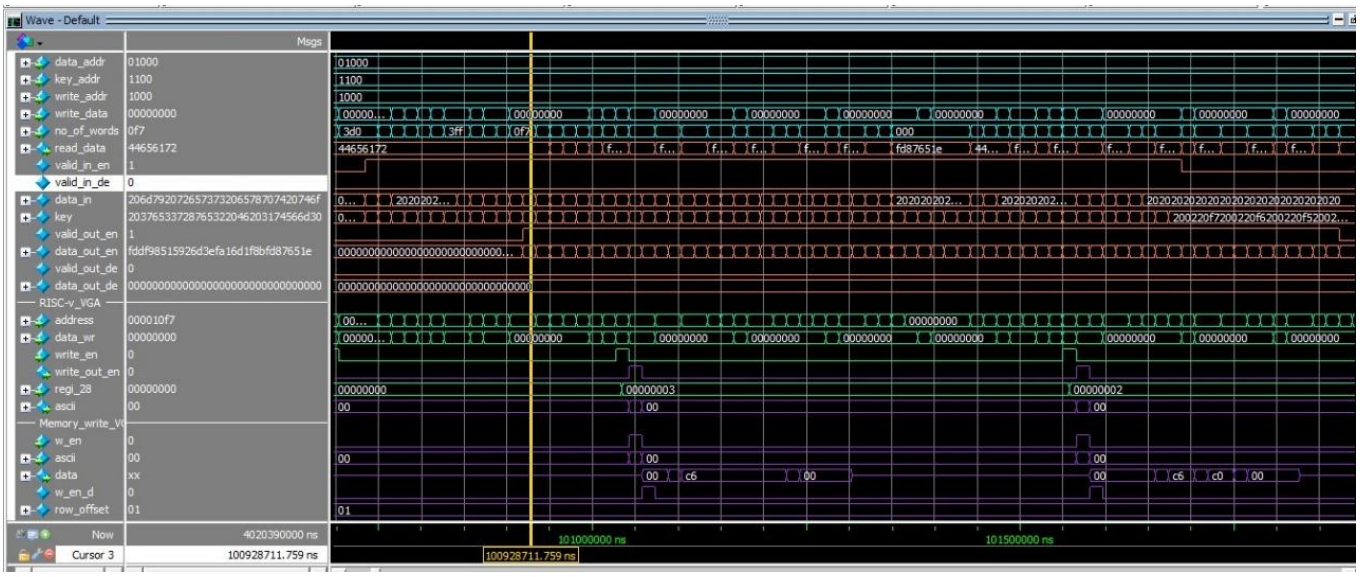


Figure 7.6 Read Data from AES Memory , writing to AES Data Memory Simulation

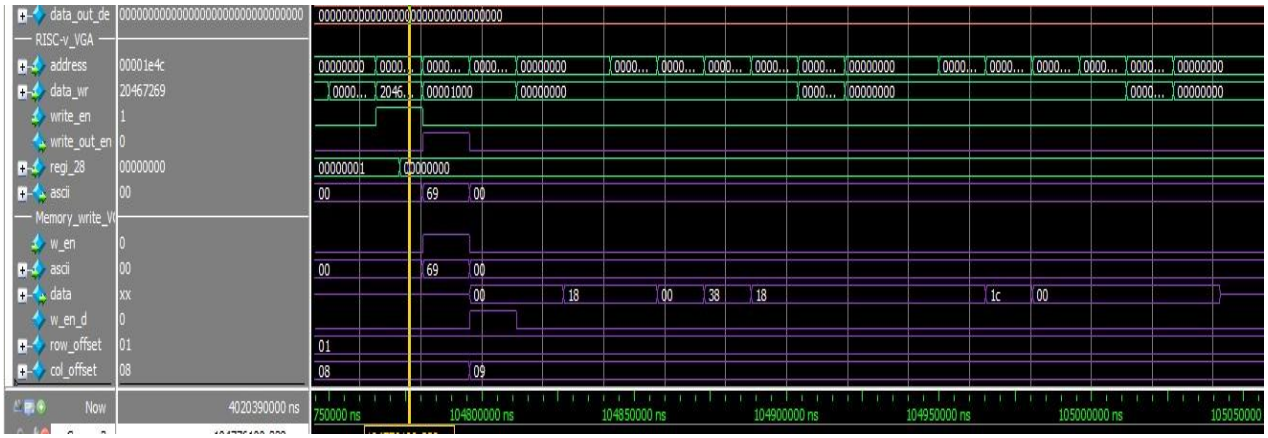


Figure 7.7 Writing data to VGA port from AES Memory

The following Assembly language RISC-V code is used for Reading Data from AES

```
addi   x29, x29, -247
 addi   x12, x12, 0x7ff
 addi   x13, x13, 0x7ff
 addi   x12, x12, 0x2
 add    x20, x12, x13
 addi   x21, x29, 0
 addi   x6 , x0 , 56    #base address last 2 bits 00 for read output memory
loopen0:
  addi   x7 , x0 , 4
  lw     x9 , 0(x6)
  addi   x29, x20, 0
 loopen1:
  addi   x8, x0, 4
  addi   x28, x7 , -1
  nop
  sw     x9 , 0(x5)
  addi   x7, x7, -1
  loopen2:
   addi   x8, x8, -1
   nop
   nop
   bne    x8 , x0 , loopen2
  bne    x7 , x0 , loopen1
 sw     x9 , 0(x6)
 addi   x21, x21, 1
 addi   x20, x20, 1
```

```
addi   x29, x21, 0
bne    x29, x27, loopen0
```

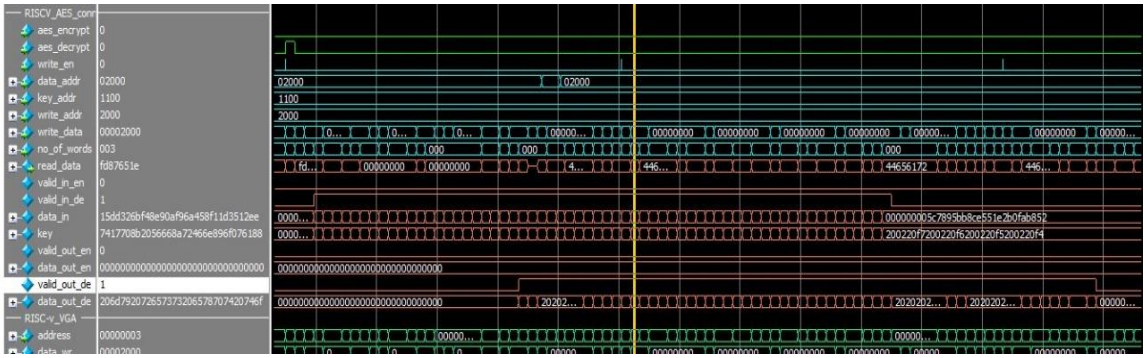


Figure 7.8 AES Decryption in Hardware Accelerator Simulation

The Encrypted text provided to AES Hardware Accelerator is now Decrypted as shown in the Above simulation.

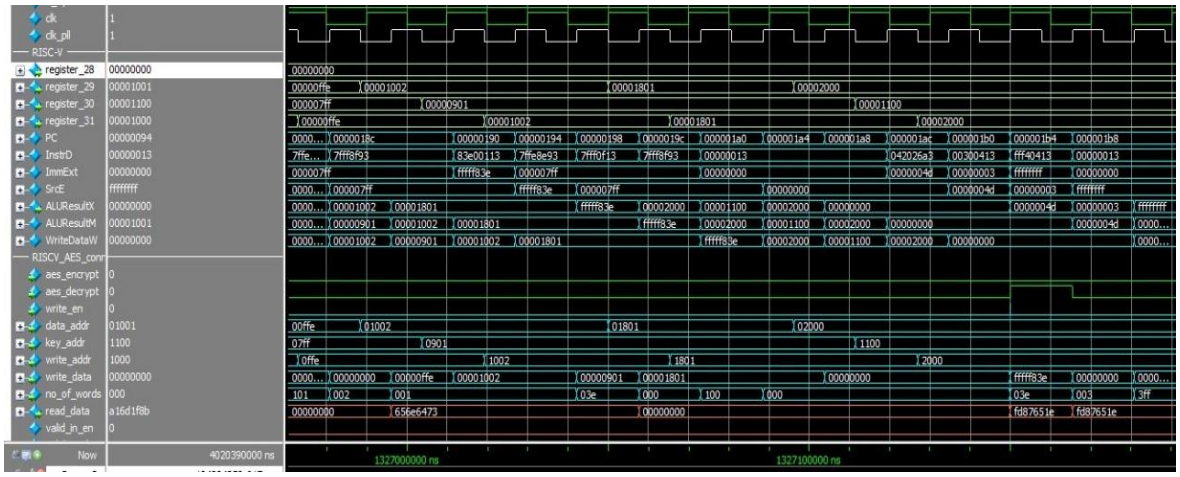


Figure 7.9 RISC-V Commands AES Decryption Simulation

RISC-V through an Assembly Language code commands AES to Decrypt the data provided to it

The following RISC-V code is used for Commanding AES Decryption:

```
addi   x29, x0 , 0x7ff
addi   x30, x0 , 0x7ff
addi   x31, x0 , 0x7ff
```

```
addi   x29, x29, 0x7ff
addi   x31, x31, 0x7ff
addi   x29, x29, 0x004
addi   x30, x30, 0x102
addi   x31, x31, 0x004
addi   x29, x29, 0x7ff
addi   x31, x31, 0x7ff
addi   x2 , x0 , -1986   #decrypt 62 words command
addi   x29, x29, 0x7ff   #data addr
addi   x30, x30, 0x7ff   #key addr
addi   x31, x31, 0x7ff   #write addr
nop
nop
nop
sw     x2 , 77(x0)      #AES decrypt
```

The simulation results collectively validate the successful integration and operation of the secure embedded cryptographic system. Each stage of interaction—from writing data to the AES memory, issuing encryption/decryption commands from the RISC-V processor, and retrieving results—was verified through waveform analysis. Simulations confirm that the AES Hardware Accelerator accurately receives data, key, and control signals from the processor, performs encryption and decryption as instructed, and stores the results at the correct memory locations. Furthermore, the system reliably transfers decrypted output to the VGA display buffer, completing the secure data flow from input to output. These results demonstrate the functional correctness, synchronization, and real-time capability of our hardware-accelerated cryptographic system.

7.5.2 VGA Output Results

The image demonstrates the real-time output from a hardware-based AES encryption and decryption system displayed on a VGA monitor. The screen is divided into three logical sections, each serving a distinct purpose in validating the encryption pipeline.

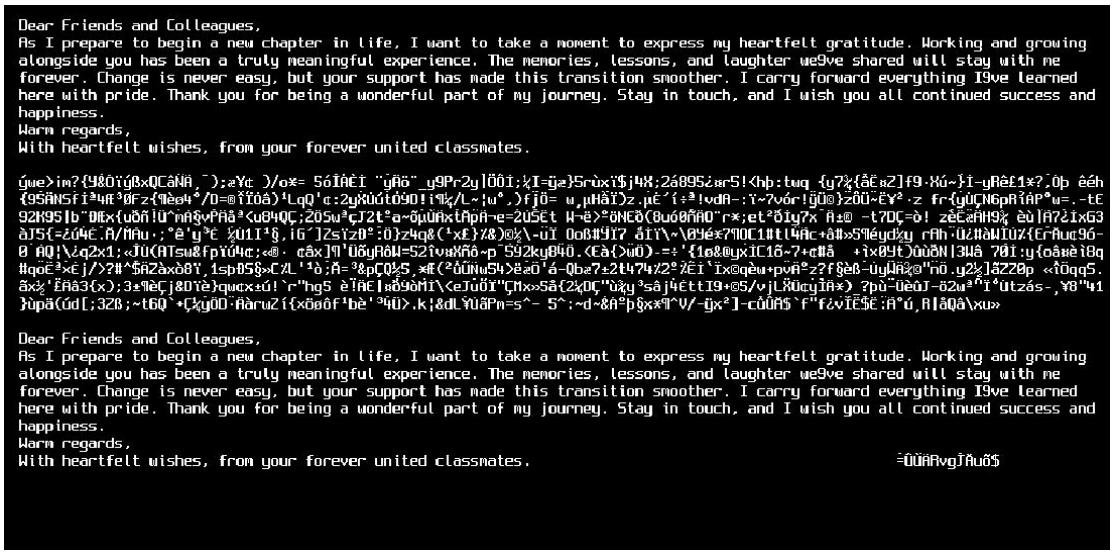


Figure 7.10 VGA Output Verification for AES Encryption and Decryption

1. Plaintext (Original Message)

The first section of the VGA output shows a farewell message written in plain English. This message is the original, unencrypted input provided to the system. It is clearly human-readable and serves as the baseline for verifying the system's encryption and decryption accuracy. The presence of this message in its expected form confirms that the VGA module correctly receives and displays the initial data.

2. Encrypted Text (Ciphertext)

The second section contains a stream of random-looking symbols and characters, representing the AES-encrypted version of the original message. This ciphertext is not human-readable, indicating that the AES encryption algorithm has successfully transformed the plaintext into a secure format. The distortion of content at this stage is

essential, as it ensures that any unauthorized access to this data will yield no meaningful information without the appropriate decryption key.

3. Decrypted Text (Recovered Plaintext)

The third section mirrors the original farewell message displayed in the first section. This confirms that the decryption module has successfully reversed the encryption process and recovered the original plaintext. The exact match with the first section validates the correctness of the AES decryption algorithm and confirms that there has been no data corruption during the encryption/decryption pipeline.

7.6 Comparison and Evaluation:

To evaluate the performance and architectural effectiveness of our proposed system, we conducted a comparison with similar AES acceleration systems found in existing research and industry implementations. The focus of this comparison is to highlight the throughput, latency, resource utilization, and system integration characteristics of each design.

Our system integrates a fully pipelined AES hardware accelerator with a custom 6-stage RISC-V processor and a VGA output module, each operating on separate clock domains provided via dedicated PLLs. The objective is to showcase how our design performs relative to other known implementations in terms of both speed and scalability.

Feature	FAC- V System	Our System
FPGA Platform	Arty / generic low-end FPGA	Cyclone V (target), Agilex/Stratix (future)
RISC-V Core	Rocket or E203 soft-core	6-stage RISC-V @ 200 MHz

Feature	FAC- V System	Our System
AES Accelerator	ISA-integrated, memory-mapped, 11 cycles/block	Fully pipelined, 11 cycles latency, 128 b @ 150 MHz
AES Throughput	≈1.28 Gbps @100 MHz	19.2 Gbps @150 MHz → projected >25 Gbps on faster FPGA
AES Latency	~11 cycles + memory overhead (~19 cycles total)	11 cycles pipeline latency; one output per cycle
Integration Style	Tightly/loosely coupled with CPU	Dedicated HW block; controlled via RISC-V and PLLs
Clock Domains	Single domain (core and AES share clock)	Multi-domain: AES @150 MHz, RISC-V @200 MHz, VGA @65 MHz

Table 7.4 AES System Comparision Table

From the comparison, it is evident that our AES accelerator achieves competitive throughput (19.2 Gbps at 150 MHz) despite being implemented on a mid-range FPGA (Cyclone V). While industry-grade accelerators like Rambus AES-IP-38 and AES-IP-61 offer higher raw throughput, they require significantly more silicon area and power and are often tailored for ASIC implementations. In contrast, our design achieves high performance with optimized pipelining, modular integration with the RISC-V core, and minimal latency.

Additionally, the system's modularity—using separate PLLs for AES, processor, and VGA—allows for scalable performance tuning. Upgrading to higher-performance FPGAs like Agilex or Stratix would further increase clock frequencies and potentially push AES throughput beyond 30 Gbps.

Chapter 8
Conclusion and Future Work

8.1 Conclusion:

This project, titled **"AES Encryption and Decryption on a RISC-V Processor,"** successfully demonstrates the design and implementation of a secure embedded cryptographic system. A custom AES Hardware Accelerator was integrated with a RISC-V processor, enabling real-time encryption and decryption through memory-mapped instructions.

The system was first validated through MATLAB simulations to ensure algorithmic correctness, followed by detailed Verilog-based implementation and testing on the DE1-SoC FPGA. The RISC-V processor effectively controlled the AES module by issuing precise instructions to handle data, key, and output memory locations. Simulation waveforms confirmed correct operation at each stage, while hardware testing verified real-time encryption, decryption, and display output via VGA.

Through this project, we achieved a robust, secure, and modular cryptographic architecture suitable for embedded environments where performance and data security are critical.

8.2 Future Work

Building on this foundation, the system can be further expanded in several practical directions:

Optimized Instruction Set: Adding dedicated RISC-V instructions for cryptographic operations to reduce latency and overhead.

Improved Security: Integrating secure key generation and storage techniques to enhance system resilience.

Scalability: Supporting additional cryptographic standards like RSA or SHA for broader applicability.

On a personal level, this project has laid the technical groundwork for a future **hardware-focused start up**. We plan to build secure, performance-driven embedded solutions for industries such as **IoT, defense electronics, and secure communications**, leveraging open-source hardware platforms and custom accelerators. This project is the first step in transforming technical innovation into scalable, real-world products.

References

[1] Harris, S. L., & Harris, D. M. (2012). *Digital design and computer architecture* (2nd ed., pp. 137–144). Morgan Kaufmann.

[2] Stallings, W. (2017). Chapter 5: Advanced encryption standard. In *Cryptography and network security: Principles and practice* (6th ed.). Pearson.

[3] Chen, Y., & Huang, Z. (2020). Design and evaluation of a RISC-V processor. *IEEE Transactions on Computers*.

[4] Dubach, C., & Qiu, J. (2017). A comparative study of RISC-V and ARM architectures. *Journal of Computer Science and Technology*.

[5] Waterman, A., Lee, Y., & Asanović, K. (2016). The RISC-V instruction set architecture: An overview. *IEEE Micro*.

[6] Jagdale, S. M., Dhole, S. A., & Kale, P. D. (2025, January 30). Security key generation using RSA algorithm.

[7] Nisha, S., & Farik, M. (2017, July). RSA public key cryptography algorithm – A review. *International Journal of Scientific & Technology Research, 6*(7).

[8] Zodpe, H., & Sapkal, A. (2020). An efficient AES implementation using FPGA with enhanced security features. *Journal of King Saud University – Engineering Sciences*.

[9] Grycel, J. T., & Walls, R. J. (n.d.). DRAB-LOCUS: An area-efficient AES architecture for hardware accelerator co-location on FPGAs. *Worcester Polytechnic Institute*.

[10] Paul, R., et al. (n.d.). Design and implementation of real-time AES-128 on RTOS for multiple FPGA communication. *University of Calcutta*.

[11] Katz, J., & Lindell, Y. (2021). *Introduction to modern cryptography*. CRC Press.

[12] National Institute of Standards and Technology (NIST). (2001). *FIPS PUB 197: Advanced encryption standard (AES)*. https://doi.org/10.6028/NIST.FIPS.197

[13] Gomes, T., Sousa, P., Silva, M., Ekpanyapong, M., & Pinto, S. (2022). FAC-V: An FPGA-based AES coprocessor for RISC-V. *Journal of Low Power Electronics and Applications, 12*(50). https://doi.org/10.3390/jlpea12030050

Appendix

Overall System Top Module

```
module system (
        input  logic      clk     ,
        input  logic      raw_reset,
        output logic [6:0] HEX0    ,
        output logic [6:0] HEX1    ,
        output logic [6:0] HEX2    ,
        output logic [6:0] HEX3    ,
        output logic [6:0] HEX4    ,
        output logic [6:0] HEX5    ,
        output logic      clk_pll  ,
        output logic [7:0] VR      ,
        output logic [7:0] VG      ,
        output logic [7:0] VB      ,
        output logic      blank_N  ,
        output logic      sync_N   ,
        output logic      H_Sync   ,
        output logic      V_Sync
);

        logic       reset_pll   ;
        logic [31:0] address    ;
        logic [31:0] data_wr    ;
        logic       write_en    ;
        logic       write_out_en;
        logic [ 7:0] ascii      ;
        logic       w_en_2d     ;
        logic [15:0] wr_addr    ;
        logic [ 7:0] data       ;
        logic       reset       ;
        logic       reset_d     ;
        logic [31:0] regi_28    ;
        logic [31:0] regi_29    ;
        logic [31:0] regi_30    ;
        logic [31:0] regi_31    ;
        logic [31:0] aes_read   ;

        reset_debouncer reset_debouncer (
                .clk     (clk     ),
                .raw_reset(raw_reset),
                .sys_reset(reset    )
        );

        vga_pll vga_pll (
                .refclk  (clk     ),
                .rst     (1'b0    ),
                .outclk_0(clk_pll  ),
                .locked  (reset_pll)
        );

        AES_riscv_connect AES_riscv_connect (
                .clk_processor(clk_pll                                          ),
                .clk_aes     (clk_pll                                           ),
                .reset       (~reset                                            ),
                .aes_encrypt  (data_wr[10] && (address == 77)                                  ),
                .write_en     (write_en && (address != 77) && (address !=
7756)                   ),
                .data_addr    (regi_29[16:0]                                     ),
                .key_addr     (regi_30[15:0]                                     ),
```

```
                .write_addr   (regi_31[15:0]                                    ),
                .write_data   (data_wr                                  ),
                .aes_decrypt  (data_wr[11] && (address == 77)                   ),
                .no_of_words  (((data_wr[10] | data_wr[11]) && (address == 77))?
data_wr[9:0] : address[9:0]),
                .read_data    (aes_read                                 )
        );

        processor_6_stage processor_6_stage (
                .clk       (clk_pll ),
                .reset     (~reset  ),
                .aes_read   (aes_read),
                .ALUResultX (address ),
                .RD2X      (data_wr ),
                .MemWriteX  (write_en),
                .register_28(regi_28 ),
                .register_29(regi_29 ),
                .register_30(regi_30 ),
                .register_31(regi_31 )
        );

        connect_riscv_vga connect_riscv_vga (
                .clk       (clk_pll     ),
                .reset     (~reset      ),
                .address    (address    ),
                .data_wr    (data_wr    ),
                .write_en   (write_en   ),
                .write_out_en(write_out_en),
                .regi_28    (regi_28    ),
                .ascii      (ascii      )
        );

        memory_write_VGA memory_write_VGA (
                .clk   (clk_pll     ),
                .reset  (~reset     ),
                .w_en   (write_out_en),
                .ascii  (ascii      ),
                .w_en_2d(w_en_2d    ),
                .wr_addr(wr_addr    ),
                .data   (data       )
        );
        VGA_port #(
                .H_ACTIVE    (1024),
                .H_SYNC      (136),
                .H_BACK_PORCH (160),
                .H_FRONT_PORCH(24),
                .V_ACTIVE    (768),
                .V_SYNC      (6),
                .V_BACK_PORCH (29),
                .V_FRONT_PORCH(3)
        ) VGA_port (
                .clk   (clk_pll  ),
                .reset  (~reset   ),
                .start  (1'b0     ),
                .wren   (w_en_2d  ),
                .data_wr(data     ),
                .wr_addr(wr_addr  ),
                .VR    (VR       ),
                .VG    (VG       ),
```

```
                .VB     (VB       ),
                .blank_N(blank_N  ),
                .sync_N (sync_N   ),
                .H_Sync (H_Sync   ),
                .V_Sync (V_Sync   )
        );
endmodule
```

RISC-V Assembly code for top level module

```
addi    x29, x0 , 0x7ff
addi    x30, x0 , 0x7ff
addi    x31, x0 , 0x7ff
addi    x29, x29, 0x2
addi    x30, x30, 0x102
addi    x31, x31, 0x2
addi    x2 , x0 , 0x43e   #encrytp 62 words command
addi    x29, x29, 0x7ff   #data addr
addi    x30, x30, 0x7ff   #key addr
addi    x31, x31, 0x7ff   #write addr
nop
nop
nop
sw      x2 , 77(x0)       #AES Encrypt

addi    x12, x0 , 2000
addi    x13, x0 , 2000
addi    x14, x0 , 2000
addi    x15, x0 , 1756
add     x16, x12, x13
add     x17, x14, x15
addi    x5 , x0 , 0
add     x5 , x16, x17     #address 7756

addi    x12, x0 , 0x7ff
addi    x13, x0 , 0x7ff
addi    x12, x12, 0x2
add     x29, x12, x13     #address 0x1000 or 4096
addi    x27, x29 , 247

addi    x6 , x0  , 58     #base address last 2 bits 10 for read data memory

loop0:
   addi    x7, x0, 4
   lw      x9 , 0(x6)
 loop1:
   addi    x8, x0, 4       #01400413
   addi    x28, x7 , -1
   nop
   sw      x9 , 0(x5)      #writing at VGA byte by byte
   addi    x7, x7, -1
   loop2:
     addi    x8, x8, -1
     nop
     nop
     bne     x8 , x0 , loop2
   bne     x7 , x0 , loop1
 addi    x29, x29, 1
 bne     x29, x27, loop0   #Read the real text from data memory and write it to VGA
```

```
 addi    x22, x0, 0x20
 addi    x23, x0 , 137
 ext0loop0:
  addi    x8, x0, 4      #01400413
  addi    x23, x23 , -1
  nop
  sw     x22, 0(x5)     #writing at VGA byte by byte
  ext0loop1:
   addi    x8, x8, -1
   nop
   nop
   bne     x8 , x0 , ext0loop1
  bne     x23 , x0 , ext0loop0

 addi    x29, x29, -247
 addi    x12, x12, 0x7ff
 addi    x13, x13, 0x7ff
 addi    x12, x12, 0x2
 add     x20, x12, x13
 addi    x21, x29, 0
 addi    x6 , x0 , 56     #base address last 2 bits 00 for read output memory

loopen0:
   addi    x7 , x0 , 4
   lw      x9 , 0(x6)
   addi    x29, x20, 0
 loopen1:
   addi    x8, x0, 4
   addi    x28, x7 , -1
   nop
   sw      x9 , 0(x5)
   addi    x7, x7, -1
   loopen2:
    addi    x8, x8, -1
    nop
    nop
    bne     x8 , x0 , loopen2
   bne     x7 , x0 , loopen1
 sw      x9 , 0(x6)
 addi    x21, x21, 1
 addi    x20, x20, 1
 addi    x29, x21, 0
 bne     x29, x27, loopen0

 addi    x22, x0, 0x20
 addi    x23, x0 , 137
 ext0loop2:
  addi    x8, x0, 4      #01400413
  addi    x23, x23 , -1
  nop
  sw     x22, 0(x5)     #writing at VGA byte by byte
  ext0loop3:
   addi    x8, x8, -1
   nop
   nop
   bne     x8 , x0 , ext0loop3
  bne     x23 , x0 , ext0loop2

addi    x29, x0 , 0x7ff
```

```
addi    x30, x0 , 0x7ff
addi    x31, x0 , 0x7ff
addi    x29, x29, 0x7ff
addi    x31, x31, 0x7ff
addi    x29, x29, 0x004
addi    x30, x30, 0x102
addi    x31, x31, 0x004
addi    x29, x29, 0x7ff
addi    x31, x31, 0x7ff
addi    x2 , x0 , -1986   #decrypt 62 words command
addi    x29, x29, 0x7ff   #data addr
addi    x30, x30, 0x7ff   #key addr
addi    x31, x31, 0x7ff   #write addr
nop
nop
nop
sw      x2 , 77(x0)       #AES decrypt

addi    x8, x0, 3
delay:
 addi    x8, x8, -1
 nop
 nop
 bne     x8 , x0 , delay

addi    x12, x0 , 0x7ff
addi    x13, x0 , 0x7ff
addi    x12, x12, 0x2
add     x29, x12, x13     #address 0x1000 or 4096
add     x29, x29, x29
addi    x27, x29 , 247

addi    x6 , x0  , 56     #base address last 2 bits 10 for read data memory

loopde0:
   addi    x7, x0, 4
   lw      x9 , 0(x6)
 loopde1:
   addi    x8, x0, 5       #01400413
   addi    x28, x7 , -1
   nop
   sw      x9 , 0(x5)      #writing at VGA byte by byte
   addi    x7, x7, -1
   loopde2:
     addi    x8, x8, -1
     nop
     nop
     bne     x8 , x0 , loopde2
   bne     x7 , x0 , loopde1
 addi    x29, x29, 1
 bne     x29, x27, loopde0   #Read the real text from data memory and write it to VGA

end:
beq     x17, x17, end    #01188063
```

AES and RISC-V interconnecting Module

```
module AES_riscv_connect (
          input  logic       clk_processor,
          input  logic       clk_aes     ,
          input  logic       reset       ,
          input  logic       aes_encrypt  ,
```

```
        input  logic       aes_decrypt  ,
        input  logic       write_en     ,
        input  logic [16:0] data_addr    ,
        input  logic [15:0] key_addr     ,
        input  logic [15:0] write_addr   ,
        input  logic [31:0] write_data   ,
        input  logic [ 9:0] no_of_words  ,
        output logic [31:0] read_data
);

        logic [ 13:0] aes_data_addr ;
        logic [ 13:0] aes_writ_addr ;
        logic [ 13:0] aes_key_addr  ;

        logic [13:0]      write_addr_AES;
        logic [13:0]      write_addr_KEY;
        logic [25:0][13:0] write_addr_OUT;

        logic [127:0] data_in      ;
        logic [127:0] key          ;
        logic         valid_in_en  ;
        logic         valid_in_de  ;

        logic [ 31:0] read_data_0  ;
        logic [ 31:0] read_data_1  ;
        logic [ 31:0] read_data_2  ;

        logic [127:0] data_out_en  ;
        logic [127:0] data_out_de  ;
        logic         valid_out    ;

        logic         aes_encrypt_1d;
        logic         aes_encrypt_2d;
        logic         aes_encrypt_3d;

        logic         aes_decrypt_1d;
        logic         aes_decrypt_2d;
        logic         aes_decrypt_3d;

        logic [ 9:0]  count_input  ;
        logic [ 9:0]  count_done   ;

        assign read_data = (no_of_words[1])? read_data_2 : ((no_of_words[0])?
read_data_1 : read_data_0);

// Recieve Signals from Processor in clock processor
        always_ff @(posedge clk_processor) begin
                if(reset) begin
                        aes_encrypt_1d <= 0;
                        aes_decrypt_1d <= 0;
                end
                else begin
                        aes_encrypt_1d <= aes_encrypt;
                        aes_decrypt_1d <= aes_decrypt;
                end
 end

        always_ff @(posedge clk_aes) begin
                if(reset)
```

```
                count_done <= 0;
        else if(aes_encrypt_1d | aes_decrypt_1d)
                count_done <= 1;
        else
                count_done <= count_done + 1;
end


        always_ff @(posedge clk_processor) begin
                if(reset)
                        count_input <= 0;
                else if(aes_encrypt | aes_decrypt)
                        count_input <= no_of_words;
end

        always_ff @(posedge clk_processor) begin
                if(reset) begin
                        aes_data_addr  <= 0;
                        aes_writ_addr  <= 0;
                        aes_key_addr   <= 0;
                end
                else if(aes_encrypt | aes_decrypt) begin
                        aes_data_addr  <= data_addr [15:2];
                        aes_writ_addr  <= write_addr[15:2];
                        aes_key_addr   <= key_addr  [15:2];
                end
        end

        always_ff @(posedge clk_aes) begin
                if(reset) begin
                        aes_encrypt_2d <= 0;
                        aes_encrypt_3d <= 0;
                        aes_decrypt_2d <= 0;
                        aes_decrypt_3d <= 0;
                end
                else begin
                        aes_encrypt_2d <= aes_encrypt_1d;
                        aes_encrypt_3d <= aes_encrypt_2d;
                        aes_decrypt_2d <= aes_decrypt_1d;
                        aes_decrypt_3d <= aes_decrypt_2d;
                end
        end

        always_ff @(posedge clk_aes) begin
                if(reset) begin
                        write_addr_OUT <= 0;
                end
                else if(aes_encrypt_1d | aes_decrypt_1d) begin
                        write_addr_OUT[0]    <= aes_writ_addr      ;
                        write_addr_OUT[25:1] <= write_addr_OUT[24:0];
                end
                else if((count_done < count_input) && (count_done != 0)) begin
                        write_addr_OUT[0]    <= write_addr_OUT[0] + 1;
                        write_addr_OUT[25:1] <= write_addr_OUT[24:0];
                end
                else begin
                        write_addr_OUT[25:1] <= write_addr_OUT[24:0];
                        write_addr_OUT[0]    <= 0;
                end
        end

        always_ff @(posedge clk_aes) begin
                if(reset) begin
```

```
                write_addr_AES <= 0;
                write_addr_KEY <= 0;
        end
        else if(aes_encrypt_1d | aes_decrypt_1d) begin
                write_addr_AES <= aes_data_addr;
                write_addr_KEY <= aes_key_addr;
        end
        else if((count_done < count_input) && (count_done != 0)) begin
                write_addr_AES <= write_addr_AES + 1;
                write_addr_KEY <= write_addr_KEY + 1;
        end
end

always_ff @(posedge clk_aes) begin
        if(reset)
                valid_in_en <= 0;
        else if(aes_encrypt_2d)
                valid_in_en <= 1;
        else if((count_done <= count_input) && (count_done != 0) && valid_in_en)
                valid_in_en <= 1;
        else begin
                valid_in_en <= 0;
        end
end

always_ff @(posedge clk_aes) begin
        if(reset)
                valid_in_de <= 0;
        else if(aes_decrypt_2d)
                valid_in_de <= 1;
        else if((count_done <= count_input) && (count_done != 0) && valid_in_de)
                valid_in_de <= 1;
        else begin
                valid_in_de <= 0;
        end
end

AES_DATA_mem AES_KEY_mem (
        .address_a(data_addr[15:0]        ),
        .address_b(write_addr_KEY         ),
        .clock_a  (clk_processor          ),
        .clock_b  (clk_aes                ),
        .data_a   (write_data             ),
        .data_b   (128'b0                 ),
        .wren_a   (write_en && data_addr[16]),
        .wren_b   (1'b0                   ),
        .q_a      (read_data_1            ),
        .q_b      (key                    )
);

AES_DATA_mem AES_DATA_mem (
        .address_a(data_addr[15:0]        ),
        .address_b(write_addr_AES         ),
        .clock_a  (clk_processor          ),
        .clock_b  (clk_aes                ),
        .data_a   (write_data             ),
        .data_b   (128'b0                 ),
        .wren_a   (write_en && ~data_addr[16]),
        .wren_b   (1'b0                   ),
```

```
            .q_a     (read_data_2            ),
            .q_b     (data_in                )
        );

        AES_DATA_mem AES_OUT_mem (
            .address_a(data_addr[15:0]                              ),
            .address_b((valid_out_de)? write_addr_OUT[23] : write_addr_OUT[13]),
            .clock_a  (clk_processor                                ),
            .clock_b  (clk_aes                                      ),
            .data_a   (32'b0                                        ),
            .data_b   ((valid_out_de)? data_out_de : data_out_en    ),
            .wren_a   (1'b0                                         ),
            .wren_b   (valid_out_en | valid_out_de                  ),
            .q_a      (read_data_0                                  ),
            .q_b      (                                             )
        );

        aes_encrypter aes_encrypter (
            .clk     (clk_aes     ),
            .rst     (reset       ),
            .data_in  (data_in     ),
            .key      (key         ),
            .valid_in (valid_in_en ),
            .data_out (data_out_en ),
            .valid_out(valid_out_en)
        );

        aes_decrypter aes_decrypter (
            .clk     (clk_aes     ),
            .rst     (reset       ),
            .data_in  (data_in     ),
            .key      (key         ),
            .valid_in (valid_in_de ),
            .data_out (data_out_de ),
            .valid_out(valid_out_de)
        );

endmodule
```

AES Encrypter

```
 module aes_encrypter (
  input  logic       clk       ,
  input  logic       rst       ,
  input  logic [127:0] data_in    ,
  input  logic [127:0] key        ,
  input  logic       valid_in   ,
  output logic [127:0] data_out   ,
  output logic       valid_out
);

  logic [ 10:0][0:127] round_keys          ;
  logic [127:0]       state_pipeline  [0:10];
  logic [ 10:0]       valid_pipeline       ;

  key_expansion key_exp (
    .clk      (clk      ),
    .rst      (rst      ),
    .cipher_key(key      ),
    .out      (round_keys)
  );

  // Initial AddRoundKey
```

```
always_ff @(posedge clk or posedge rst) begin
 if (rst) begin
   state_pipeline[0] <= 0;
 end
 else if (valid_in) begin
   state_pipeline[0] <= data_in ^ key;
 end
end

// AES Rounds 1-9
genvar i;
generate
 for (i = 0; i < 9; i++) begin : aes_rounds
   logic [127:0] subbyte_stage, shiftrows_stage, mixcolumns_stage;

   sub_byte sub_byte_inst (
     .in (state_pipeline[i]),
     .out(subbyte_stage    )
   );

   shift_rows shift_rows_inst (
     .in (subbyte_stage  ),
     .out(shiftrows_stage)
   );

   mix_column mix_column_inst (
     .in (shiftrows_stage ),
     .out(mixcolumns_stage)
   );

   always_ff @(posedge clk or posedge rst) begin
     if (rst)
       state_pipeline[i+1] <= 0;
     else if (valid_pipeline[i])
       state_pipeline[i+1] <= mixcolumns_stage ^ round_keys[i+1];
   end
 end
endgenerate

// Final Round (no MixColumns)
logic [127:0] subbyte_last, shiftrows_last;
sub_byte sub_byte_last (
  .in (state_pipeline[9]),
  .out(subbyte_last     )
);

shift_rows shift_rows_last (
  .in (subbyte_last  ),
  .out(shiftrows_last)
);

always_ff @(posedge clk or posedge rst) begin
 if (rst)
   valid_pipeline[10:0] <= 0;
 else
   valid_pipeline[10:0] <= {valid_pipeline[9:0], valid_in};
end

always_ff @(posedge clk or posedge rst) begin
```

```
   if (rst) begin
     state_pipeline[10] <= 0;
   end
   else if (valid_pipeline[9]) begin
     state_pipeline[10] <= shiftrows_last ^ round_keys[10];
   end
  end

  always@(posedge  clk) begin
   if(rst) begin
     data_out <= 0;
     valid_out <= 0;
   end
   else if(valid_pipeline[10]) begin
     data_out <= state_pipeline[10];
     valid_out <= 1;
   end
   else begin
     data_out <= 0;
     valid_out <= 0;
   end
  end

endmodule
```

AES Decrypter

```
module aes_decrypter (
  input  logic        clk        ,
  input  logic        rst        ,
  input  logic [127:0] data_in     ,
  input  logic [127:0] key         ,
  input  logic        valid_in    ,
  output logic [127:0] data_out    ,
  output logic         valid_out
);

  // Pipeline registers for AES rounds
  logic [ 10:0][0:127] round_keys          ; // Store 11 round keys
  logic [127:0]        state_pipeline  [0:10];
  logic [ 10:0]        valid_pipeline      ;
  logic                valid_pipeline_d    ;
  logic [ 10:0]        valid_in_store      ;
  logic [ 10:0][0:127] data_in_store       ;

  logic [10:0][0:127] key_storage[19:0];

  always_ff @(posedge clk) begin
   if(rst) begin
     for (int i = 0; i < 20; i++) begin
       key_storage[i] <= 0;
     end
   end
   else begin
     key_storage[0] <= round_keys;
     for (int i = 0; i < 19; i++) begin
       key_storage[i+1] <= key_storage[i];
     end
   end
  end
```

```
 always_ff @(posedge clk) begin
  if(rst) begin
    valid_in_store <= 0;
    data_in_store  <= 0;
  end
  else begin
    valid_in_store <= {valid_in_store[9:0], valid_in};
    data_in_store  <= {data_in_store[9:0], data_in};
  end
 end

 // Key Expansion Module
 key_expansion key_exp (
  .clk      (clk      ),
  .rst      (rst      ),
  .cipher_key(key      ),
  .out      (round_keys)
 );

 // Initial AddRoundKey
 always_ff @(posedge clk or posedge rst) begin
  if (rst) begin
    state_pipeline[0] <= 0;
  end else if (valid_in_store[9]) begin
    state_pipeline[0] <= data_in_store[9] ^ round_keys[10]; //key_storage[10][10][0:127];
  end
 end

 // AES Rounds 1-9
 genvar i;
 generate
  for (i = 0; i < 9; i++) begin : aes_rounds
    logic [127:0] inv_subbyte_stage, inv_shiftrows_stage,
inv_mixcolumns_stage ,inv_addround_stage;

    inv_shift_rows inv_shift_rows_inst (
      .in (state_pipeline[i]  ),
      .out(inv_shiftrows_stage)
    );
    inv_sub_byte inv_sub_byte_inst (
      .in (inv_shiftrows_stage),
      .out(inv_subbyte_stage  )
    );

    always_ff @(posedge clk or posedge rst) begin
      if (rst)
        inv_addround_stage <= 0;
      else if (valid_pipeline[i])
        inv_addround_stage <= inv_subbyte_stage ^ key_storage[(i<<1)+1][9-i];
    end

    inv_mix_column inv_mix_column_inst (
      .in (inv_addround_stage),
      .out(state_pipeline[i+1])
    );
  end
 endgenerate

 // Final Round (no MixColumns)
```

```
 logic [127:0] inv_subbyte_last, inv_shiftrows_last;
 inv_shift_rows inv_shift_rows_last (
  .in (state_pipeline[9] ),
  .out(inv_shiftrows_last)
 );
 inv_sub_byte inv_sub_byte_last (
  .in (inv_shiftrows_last),
  .out(inv_subbyte_last  )
 );

 always_ff @(posedge clk or posedge rst) begin
  if (rst)
    valid_pipeline[10:0] <= 0;
  else
    valid_pipeline[10:0] <= {valid_pipeline[9:0], valid_in_store[9]};
 end

 always_ff @(posedge clk or posedge rst) begin
  if (rst) begin
    state_pipeline[10] <= 0;
  end
  else if (valid_pipeline[9]) begin
    state_pipeline[10] <= inv_subbyte_last ^ key_storage[19][0];
  end
 end

 always@(posedge  clk) begin
  if (rst) begin
    valid_out <= 0;
    data_out  <= 0;
  end
  else if(valid_pipeline[10]) begin
    valid_out <= 1;
    data_out  <= state_pipeline[10];
  end
  else begin
    valid_out <= 0;
    data_out  <= 0;
  end
 end

endmodule
```

Reset Debouncer

```
module reset_debouncer #(parameter reset_hold = 5000, reset_take = 50000) (
 input  logic clk      , // System clock (e.g., 50MHz)
 input  logic raw_reset, // Raw input from KEY0 (active low)
 output logic sys_reset  // Debounced reset output (active high)
);

 logic [ 1:0] current_state;
 logic [ 1:0] next_state   ;
 logic [19:0] count        ;

 always_ff @(posedge clk) begin
  current_state <= next_state;
 end

 always_ff @(*) begin
  case (current_state)
```

```
    2'b00 : begin
     if(raw_reset)        next_state <= 2'b00;
     else                 next_state <= 2'b01;
    end
    2'b01 : begin
     if (count == reset_hold)
       next_state <= 2'b10;
     else
       next_state <= 2'b01;
    end
    2'b10 : begin
     next_state <= 2'b00;
    end
    default : next_state <= 0;
  endcase
 end

 always_ff @(posedge clk) begin
  case (current_state)
    2'b00 : begin
     count     <= 0;
     sys_reset <= 1;
    end
    2'b01 : begin
     sys_reset <= 0;
     if (count == reset_hold)
       count <= 0;
     else
       count <= count + 1;
    end
    2'b10 : begin
     sys_reset <= 1;
     count     <= 0;
    end
    default : begin
     next_state <= 0;
     sys_reset  <= 1;
    end
  endcase
 end

endmodule
```

RISC-V VGA Connect

```
module connect_riscv_vga (
          input  logic       clk        ,
          input  logic       reset      ,
          input  logic [31:0] address     ,
          input  logic [31:0] data_wr     ,
          input  logic       write_en    ,
          input  logic [31:0] regi_28     ,
          output logic       write_out_en,
          output logic [ 7:0] ascii
);
always_ff @(posedge clk) begin
          if(reset) begin
                    write_out_en <= 0;
```

```
                ascii       <= 0;
        end
        else if(write_en && (address == 7756)) begin
                if(regi_28[1:0] == 0) begin
                        write_out_en <= 1;
                        ascii       <= data_wr[7:0];
                end
                else if(regi_28[1:0] == 1) begin
                        write_out_en <= 1;
                        ascii       <= data_wr[15:8];
                end
                else if(regi_28[1:0] == 2) begin
                        write_out_en <= 1;
                        ascii       <= data_wr[23:16];
                end
                else if(regi_28[1:0] == 3) begin
                        write_out_en <= 1;
                        ascii       <= data_wr[31:24];
                end
        end
        else begin
                write_out_en <= 0;
                ascii       <= 0;
        end
end
Endmodule
```

Memory write VGA

```
module memory_write_VGA (
        input  logic       clk    ,
        input  logic       reset  ,
        input  logic       w_en   ,
        input  logic [ 7:0] ascii  ,
        output logic       w_en_2d,
        output logic [15:0] wr_addr,
        output logic [ 7:0] data
);

        genvar          count_0  ;
        logic          w_en_d   ;
        logic      [ 7:0] row_offset; //   (1024 / 8)
        logic      [ 6:0] col_offset; //   (768  /16)
        logic [15:0][ 7:0] f_data   ;
        logic [15:0][ 7:0] f_data_d ;
        logic [15:0][ 7:0] old_data ;
        logic      [ 5:0] count_wr  ;

        always_ff @(posedge clk) begin
                if(reset) begin
                        w_en_d   <= 0;
                        f_data_d <= 0;
                end
                else begin
                        w_en_d   <= w_en  ;
                        f_data_d <= f_data;
                end
        end

        always_ff @(posedge clk) begin
                if(reset) begin
                        w_en_2d  <= 0;
                        count_wr <= 0;
```

```
                end
                else if((count_wr < 16) && (count_wr != 0))begin
                  w_en_2d <= 1;
                  count_wr <= count_wr + 1;
                end
                else if(w_en) begin
                        w_en_2d <= 1;
                        count_wr <= 1;
                end
                else begin
                        w_en_2d  <= 0;
                        count_wr <= 0;
                end
        end

        always_ff @(posedge clk) begin
                if(reset) begin
                        col_offset <= 1;    // 1000
                end
                else begin
                        if(w_en)
                                col_offset <= col_offset + 1;
                        else if((col_offset == 126) && (!w_en & !w_en_2d))
                                col_offset <= 1;
                end
        end

        always_ff @(posedge clk) begin
                if(reset) begin
                        row_offset <= 1;    // 1000
                end
                else begin
                        if((col_offset == 126) && (!w_en & !w_en_2d))
                         row_offset <= row_offset + 1;
                end
        end

        generate
                for (count_0 = 0; count_0 < 16; count_0++) begin : vga_memory_name
                        vgamem vgamem (
                                .address({ascii,count_0[3:0]}  ),
                                .clock  (clk                ),
                                .data   (8'b0               ),
                                .wren   (1'b0               ),
                                .q      (old_data[count_0[3:0]])
                        );
                end
        endgenerate

always_ff @(posedge clk) begin
        if (reset) begin
   wr_addr <= 0;
        end
        else begin
           wr_addr <= ((row_offset<<4)<<7) + (col_offset) + (count_wr<<7);
        end
        end

assign f_data = (w_en_d)? old_data : f_data_d;
```

```
  assign data = (f_data[count_wr-1] != 0)? (f_data[count_wr-1]) : 8'b0;

endmodule
```

VGA Port

```
module VGA_port (
        input  logic       clk     ,
        input  logic       raw_reset,
        output logic [ 7:0] VR      ,
        output logic [ 7:0] VG      ,
        output logic [ 7:0] VB      ,
        output logic       blank_N  ,
        output logic       sync_N   ,
        output logic       H_Sync   ,
        output logic       V_Sync
);

        logic reset;

        parameter
                H_ACTIVE     = 1024,
                H_SYNC       = 136 ,
                H_BACK_PORCH  = 160 ,
                H_FRONT_PORCH = 24  ,
                V_ACTIVE     = 768 ,
                V_SYNC       = 6   ,
                V_BACK_PORCH  = 29  ,
                V_FRONT_PORCH = 3   ;

        parameter
                IDLE = 3'd0,
                VBP  = 3'd1,
                HBP  = 3'd2,
                HA   = 3'd3,
                HFP  = 3'd4,
                HS   = 3'd5,
                VFP  = 3'd6,
                VS   = 3'd7;


reset_debouncer #(
        .DEBOUNCE_TIME(500000),
        .RESET_ACTIVE_HIGH(1)
        ) i_reset_debouncer (
        .clk     (clk     ),
        .raw_reset(raw_reset),
        .sys_reset(reset    )
);


        logic [18:0] address   ;
        logic [ 7:0] data      ;
        logic [ 2:0] D         ;
        logic [ 2:0] Q         ;
        logic [ 8:0] horz_count;
        logic [ 8:0] vert_count;
        logic [10:0] horz_activ;
        logic [10:0] vert_activ;

  assign sync_N = 1'b1;
```

```
 always@(posedge clk) begin
  if(~reset)
    blank_N <= 1'b0;
  else
    blank_N <= 1'b1;
 end

        always@(posedge clk) begin
                if(~reset) begin
                        horz_activ <= 11'd0;
                        vert_activ <= 11'd0;
                        horz_count <= 9'd0 ;
                        vert_count <= 9'd0 ;
                end
                else if(vert_count == V_SYNC + V_FRONT_PORCH + H_BACK_PORCH) begin
                        horz_activ <= 11'd0       ;
                        horz_count <= 9'd0        ;
                        vert_count <= 9'd0        ;
                        vert_activ <= vert_activ + 1;
                end
                else begin
                        if((Q == VBP) | (Q == VFP) | ((Q == VS)))     vert_count <= vert_count
+ 1;
                        if((Q == HBP) | (Q == HFP) | (Q == HS))       horz_count <=
horz_count + 1;
                        if(Q == HA)                             horz_activ <= horz_activ + 1;
                        if(1 == V_SYNC + V_FRONT_PORCH + V_BACK_PORCH) vert_activ <=
vert_activ + 1;
                end
        end

        always_comb begin
                case(Q)
                        IDLE :                                          D <= VBP ;
                        VBP  : if(vert_count == V_BACK_PORCH)                   D <= HBP ;
                        else                                            D <= VBP ;
                        HBP  : if(horz_count == H_BACK_PORCH)                   D <= HA  ;
                        else                                            D <= HBP ;
                        HA   : if(horz_activ == H_ACTIVE)                       D <= HFP ;
                        else                                            D <= HA  ;
                        HFP  : if(horz_count == H_FRONT_PORCH + H_BACK_PORCH)       D
<= HS  ;
                        else                                            D <= HFP ;
                        HS   : if(horz_count == H_SYNC + H_FRONT_PORCH +
H_BACK_PORCH) D <= VFP ;
                        else                                            D <= HS  ;
                        VFP  : if(vert_count == V_FRONT_PORCH + V_BACK_PORCH)       D
<= VS  ;
                        else                                            D <= VFP ;
                        VS   : if(vert_count == V_SYNC + V_FRONT_PORCH + V_BACK_PORCH)
                                        if(vert_activ == V_ACTIVE)                      D <= IDLE;
                                  else                                  D <= VBP ;
                                else                                    D <= VS  ;
                        default :                                         D <= IDLE;
                endcase
        end

        always@(posedge clk) begin
                if(~reset)
```

```
                Q <= 3'b000;
            else
                Q <= D    ;
        end

        always_comb begin
            case(Q)
                IDLE : begin
                    H_Sync <= 1'b0;
                    V_Sync <= 1'b0;
                end
                VBP : begin
                    H_Sync <= 1'b0;
                    V_Sync <= 1'b0;
                end
                HBP : begin
                    H_Sync <= 1'b0;
                    V_Sync <= 1'b0;
                end
                HA : begin
                    H_Sync <= 1'b0;
                    V_Sync <= 1'b0;
                end
                HFP : begin
                    V_Sync <= 1'b0;
                    H_Sync <= 1'b0;
                end
                HS : begin
                    H_Sync <= 1'b1;
                    V_Sync <= 1'b0;
                end
                VFP : begin
                    H_Sync <= 1'b0;
                    V_Sync <= 1'b0;
                end
                VS : begin
                    V_Sync <= 1'b1;
                    H_Sync <= 1'b0;
                end
                default : {H_Sync,V_Sync} = 2'd0;
            endcase
        end

        assign address = (vert_activ<<10) + horz_activ;

        dispmem dispmem (
            .address_a(address[18:3]),
            .address_b(16'b0       ),
            .clock    (clk         ),
            .data_a   (8'b0        ),
            .data_b   (8'b0        ),
            .wren_a   (1'b0        ),
            .wren_b   (1'b0        ),
            .q_a      (data        ),
            .q_b      (            )
        );

always_ff @(posedge clk) begin
        if(data[address[2:0]] == 1) begin
            VR <= 8'hff;
            VG <= 8'hff;
            VB <= 8'hff;
        end
```

```
        else begin
                VR <= 0;
                VG <= 0;
                VB <= 0;
        end
 end

endmodule
```

6-Stage Pipelined Processor

```
module PPWH(clk,reset);
input clk,reset;

wire [31:0]PCPlus4,PCPlus41,PCTargetE,PCNext,PCNext1,PC,Instr,InstrX,InstrF;
reg PCSrc;

wire [31:0]InstrD,InstrL,InstrK,InstrY,PCD,PCPlus4D,WriteDataW,RD1,RD2,ImmExt;
wire Branch,MemWrite,ALUSrc,RegWrite,Jump,RegWriteW,FF,FD,FE;
wire [1:0]ResultSrc, ImmSrc;
wire [2:0]ALUControl;
wire [4:0]WriteAddW;

wire [31:0]RD1E,RD2E,PCE,ImmExtE,PCPlus4E,SrcE,ALUResultX,A32,B32;
wire [4:0]WriteAddE,rs1,rs2,IS1,IS2;
wire [1:0]ResultSrcE,ResultSrcX,A,B;
wire [2:0]ALUControlE;
wire
RegWriteE,MemWriteE,RegWriteE1,MemWriteE1,JumpE,JumpE1,BranchE,BranchE1,ALUSrc
E;

wire [31:0]RD2X,SrcX,A32X,PCPlus4X,C,C1,PCX,ImmExtX;
wire [4:0]WriteAddX;
wire [2:0]ALUControlX,funct3X;
wire RegWriteX,MemWriteX,JumpX,BranchX,ALUSrcX,Zero;

wire [31:0]ALUResultM,RD2M,PCPlus4M,ReadDataM;
wire [1:0]ResultSrcM;
wire [4:0]WriteAddM;
wire RegWriteM,MemWriteM,SS11,s,STF,STD,STE,STF1;

wire [31:0]ReadDataW,ALUResultW,PCPlus4W;
wire [1:0]ResultSrcW;

wire [63:0]E[15:0];
wire [63:0]F[15:0];

Mux2by1 #(32) pcmux(PCPlus4, PCTargetE, PCSrc, PCNext);

FF321 #(32) ff0(clk, reset, STF, PCNext, PC);
FF32 #(1) ff001(clk, reset, STF , STF1);

                assign Instr = (STF1)? Instr : InstrF ;
    InstructionMemory IM(PCNext,clk, InstrF,reset);
    Adder a0(PC, 32'd4 , PCPlus4);

FF321 #(107) ff1(clk, reset, STD, {Instr,PC,PCPlus4,Instr[19:15],Instr[24:20],FF},
{InstrL,PCD,PCPlus4D,IS1,IS2,SS11});
```

```
                Mux2by1 #(32) ins1(InstrL,32'h00000013, SS11, InstrD);
    Controller3 c(InstrD[6:0], InstrD[14:12], InstrD[30],ResultSrc, ImmSrc, MemWrite,
            Branch,ALUSrc, RegWrite, Jump, ALUControl,FD,InstrD[25]);
    Register rff(clk, RegWriteW, InstrD[19:15], InstrD[24:20],WriteAddW, WriteDataW,
RD1 ,RD2);
                Extend ext(InstrD[31:7], ImmSrc, ImmExt);
    PP1 HU(ResultSrcX[0],WriteAddX,RegWriteX,rs1,rs2,FF,FD,FE,STF,STD,STE,PCSrc);



FF321 #(217) ff2(clk, reset, STE,
{RegWrite,ResultSrc,MemWrite,Jump,Branch,ALUControl,ALUSrc,RD1,RD2,
            PCD,InstrD[11:7],ImmExt,PCPlus4D,InstrD[19:15], InstrD[24:20],InstrD},
{RegWriteE,ResultSrcE,MemWriteE,JumpE,BranchE,

ALUControlE,ALUSrcE,RD1E,RD2E,PCE,WriteAddE,ImmExtE,PCPlus4E,rs1,rs2,InstrK});


                Mux3by1 #(32) muxA(RD1E, ALUResultM, WriteDataW, A, A32);
                Mux3by1 #(32) muxB(RD2E, ALUResultM, WriteDataW, B, B32);
                Mux2by1 #(32) srcbmux(B32, ImmExtE, ALUSrcE, SrcE);
                ForwardingUnit
fdu(WriteAddM,WriteAddW,rs1,rs2,RegWriteM,RegWriteW,A,B);
                Mux2by1 #(4)
Flush(4'b0000,{BranchE,JumpE,RegWriteE,MemWriteE},FE,{BranchE1,JumpE1,RegWriteE1,M
emWriteE1});


FF32 #(209) ff3(clk, reset,
{RegWriteE1,ResultSrcE,MemWriteE1,A32,SrcE,B32,WriteAddE,PCPlus4E,ALUControlE,InstrK
[14:12],BranchE1,JumpE1,PCE,ImmExtE},

{RegWriteX,ResultSrcX,MemWriteX,A32X,SrcX,RD2X,WriteAddX,PCPlus4X,ALUControlX,funct
3X,BranchX,JumpX,PCX,ImmExtX});

                ALU2 alu(A32X, SrcX, ALUControlX, ALUResultX, Zero,funct3X);
                Adder a1(PCX,ImmExtX, PCTargetE);
                MWCS(clk,reset,A32X,SrcX,1'b0,C);
    always@(*) begin
                  if((BranchX === 1'bx)|(Zero === 1'bx)|(JumpX === 1'bx))
                     PCSrc = 0;
                  else
              PCSrc = (BranchX & Zero) | JumpX ;
                end

FF32 #(106) ff4(clk, reset,
{RegWriteX,ResultSrcX,MemWriteX,ALUResultX,RD2X,WriteAddX,PCPlus4X,funct3X[0]},
            {RegWriteM,ResultSrcM,MemWriteM,ALUResultM,RD2M,WriteAddM,PCPlus4M,s});

             DataMemory DM(clk, ALUResultX, RD2X, MemWriteX, ReadDataM);


FF32 #(104) ff5(clk, reset,
{RegWriteM,ResultSrcM,ALUResultM,ReadDataM,WriteAddM,PCPlus4M},
            {RegWriteW,ResultSrcW,ALUResultW,ReadDataW,WriteAddW,PCPlus4W});

    mux4by1 #(32) resultmux(ALUResultW, ReadDataW, PCPlus4W, C, ResultSrcW,
WriteDataW);

Endmodule
```

3-Stage Multiplier

```
module MWCS(clk,reset,A,B,s,O);
input clk,reset,s;
input [31:0]A,B;
output [31:0]O;

wire [31:0]D[31:0];
wire [33:0]C0[9:0];
wire [33:0]S0[9:0];
wire [33:0]S0f;

wire [39:0]C1[5:0];
wire [39:0]S1[5:0];
wire [39:0]C1f[5:0];
wire [39:0]S1f[5:0];
wire [35:0]C16,C16f;
wire [35:0]S16,S16f;

wire [57:0]C2[3:0];
wire [57:0]S2[3:0];
wire [37:0]C24,C24f;
wire [37:0]S24;

wire [58:0]C3[1:0];
wire [58:0]S3[1:0];
wire [62:0]C32;
wire [62:0]S32;

wire [59:0]S40,C40;
wire [63:0]S41,C41;

wire [60:0]S5,C5;

wire [65:0]S6,C6;

wire [66:0]S70,C70;
wire [66:0]S71,C71;

wire [63:0]sum;

generate
	genvar j,k;
	for(j=0; j<32; j=j+1) begin : AND_ROW
		for(k=0; k<32; k=k+1) begin : AND_Col
			and a0(D[j][k],A[k],B[j]);
		end
	end
endgenerate

CSA #(34) c0({2'b00,D[0]},{1'b0,D[1],1'b0},{D[2],2'b00},S0[0],C0[0]);
CSA #(34) c1({2'b00,D[3]},{1'b0,D[4],1'b0},{D[5],2'b00},S0[1],C0[1]);
CSA #(34) c2({2'b00,D[6]},{1'b0,D[7],1'b0},{D[8],2'b00},S0[2],C0[2]);
CSA #(34) c3({2'b00,D[9]},{1'b0,D[10],1'b0},{D[11],2'b00},S0[3],C0[3]);
CSA #(34) c4({2'b00,D[12]},{1'b0,D[13],1'b0},{D[14],2'b00},S0[4],C0[4]);
CSA #(34) c5({2'b00,D[15]},{1'b0,D[16],1'b0},{D[17],2'b00},S0[5],C0[5]);
CSA #(34) c6({2'b00,D[18]},{1'b0,D[19],1'b0},{D[20],2'b00},S0[6],C0[6]);
CSA #(34) c7({2'b00,D[21]},{1'b0,D[22],1'b0},{D[23],2'b00},S0[7],C0[7]);
CSA #(34) c8({2'b00,D[24]},{1'b0,D[25],1'b0},{D[26],2'b00},S0[8],C0[8]);
CSA #(34) c9({2'b00,D[27]},{1'b0,D[28],1'b0},{D[29],2'b00},S0[9],C0[9]);
```

```
CSA #(40) c10({6'b0,S0[0]},{3'b0,S0[1],3'b0},{S0[2],6'b0},S1[0],C1[0]);
CSA #(40) c11({6'b0,S0[3]},{3'b0,S0[4],3'b0},{S0[5],6'b0},S1[1],C1[1]);
CSA #(40) c12({6'b0,S0[6]},{3'b0,S0[7],3'b0},{S0[8],6'b0},S1[2],C1[2]);
CSA #(40) c13({6'b0,C0[0]},{3'b0,C0[1],3'b0},{C0[2],6'b0},S1[3],C1[3]);
CSA #(40) c14({6'b0,C0[3]},{3'b0,C0[4],3'b0},{C0[5],6'b0},S1[4],C1[4]);
CSA #(40) c15({6'b0,C0[6]},{3'b0,C0[7],3'b0},{C0[8],6'b0},S1[5],C1[5]);
CSA #(36) c16({2'b0,C0[9]},{1'b0,D[30],3'b0},{D[31],4'b0},S16,C16);

FF32 #(586)
ff1(clk,rset,{S1[0],S1[1],C1[0],C1[1],S1[2],S1[3],C1[2],C1[3],S1[4],S1[5],C1[4],C1[5],S16,C16,S
0[9]}
      ,{S1f[0],S1f[1],C1f[0],C1f[1],S1f[2],S1f[3],C1f[2],C1f[3],S1f[4],S1f[5],C1f[4],C1f[5],S16f,C1
6f,S0f});

CSA #(58) c17({18'b0,S1f[0]},{9'b0,S1f[1],9'b0},{S1f[2],18'b0},S2[0],C2[0]);
CSA #(58) c18({18'b0,S1f[3]},{9'b0,S1f[4],9'b0},{S1f[5],18'b0},S2[1],C2[1]);
CSA #(58) c19({18'b0,C1f[0]},{9'b0,C1f[1],9'b0},{C1f[2],18'b0},S2[2],C2[2]);
CSA #(58) c20({18'b0,C1f[3]},{9'b0,C1f[4],9'b0},{C1f[5],18'b0},S2[3],C2[3]);
CSA #(38) c21({4'b0,S0f},{1'b0,S16f,1'b0},{C16f,2'b0},S24,C24);

CSA #(59) c22({1'b0,S2[0]},{S2[1],1'b0},{S2[2],1'b0},S3[0],C3[0]);
CSA #(59) c23({1'b0,C2[0]},{C2[1],1'b0},{C2[2],1'b0},S3[1],C3[1]);
CSA #(63) c24({5'b0,S2[3]},{4'b0,C2[3],1'b0},{S24,25'b0},S32,C32);

CSA #(60) c25({1'b0,S3[0]},{S3[1],1'b0},{C3[0],1'b0},S40,C40);
CSA #(64) c26({1'b0,S32},{C32,1'b0},{C24,26'b0},S41,C41);

CSA #(61) c27({1'b0,S40},{C40,1'b0},{C3[1],2'b0},S5,C5);

CSA #(66) c28({4'b0,S5},{S41,2'b0},{3'b0,C5,1'b0},S6,C6);

CSA #(67) c29({1'b0,S6},{C41,3'b0},{C6,26'b0},S70,C70);

FF32 #(134) ff2(clk,rset,{S70,C70},{S71,C71});

assign sum = S71[63:0] + C71[63:0];
assign O = (s)? sum[63:32] : sum[31:0];

endmodule
```

CSA

```
module CSA #(parameter WIDTH = 8) (X,Y,Z,S,C);
input [WIDTH-1:0]X,Y,Z;
output [WIDTH-1:0]C,S;

generate
  genvar i;
        for(i=0; i<WIDTH ; i=i+1) begin : Add
          assign S[i] = X[i] ^ Y[i] ^ Z[i];
                assign C[i] = (X[i] & Y[i]) | (X[i] & Z[i]) | (Z[i] & Y[i]);
        end
endgenerate

endmodule
```

4-Stage Multiplier

```
module Mult(clk,reset,A,B,C,s);
input clk,reset,s;
input [31:0]A,B;
output [31:0]C;
```

```
wire s1,s2,s3,s4;
wire [63:0]D[31:0];
wire [63:0]d[31:0];
wire [63:0]E[15:0];
wire [63:0]F[7:0];
wire [63:0]X[7:0];
wire [63:0]Y[3:0];
wire [63:0]Z[1:0];
wire [63:0]z[1:0];
wire [63:0]sum;

                        generate
                         genvar i;
                         for (i = 0; i < 32; i=i+1) begin: PARTIAL_PRODUCTS
                                Mux2by1 #(64) pcmux(64'b0,{{{32{1'b0}},B} << i},
A[i], D[i]);
                         end
                        endgenerate

generate
 genvar k;
 for (k = 0; k < 32; k=k+1) begin: PARTIAL_Store
    FF32 #(64) ff1(clk,reset,D[k],d[k]);
 end
endgenerate
FF32 #(1) f(clk,reset,s1,s2);

                        generate
                         genvar j;
                         for (j = 0; j < 16; j=j+1) begin: ADD_PARTIALS
                                assign E[j] = d[j] + d[31-j];
                         end
                        endgenerate

                        generate
                         genvar l;
                         for (l = 0; l < 8; l=l+1) begin: ADD2_PARTIALS
                                assign X[l] = E[l] + E[15-l];
                         end
                        endgenerate


FF32 #(513) ff2(clk,reset,{X[0],X[1],X[2],X[3],X[4],X[5],X[6],X[7],s2},
                {F[0],F[1],F[2],F[3],F[4],F[5],F[6],F[7],s3});

                        generate
                         genvar m;
                         for (m = 0; m < 4; m=m+1) begin: ADD3_PARTIALS
                                assign Y[m] = F[m] + F[7-m];
                         end
                        endgenerate

                        generate
                         genvar n;
                         for (n = 0; n < 2; n=n+1) begin: ADD4_PARTIALS
                                assign Z[n] = Y[n] + Y[3-n];
                         end
                        endgenerate
```

```
FF32 #(129) ff3(clk,reset,{Z[0],Z[1],s3},{z[0],z[1],s4});

                                assign sum = z[0] + z[1];

assign C = (s4)? sum[31:0] : sum[63:32];

endmodule
```

Hazard unit for 6-Stage

```
module PP1 (ResultSrcX,WriteAddX,RegWriteX,rs1E,rs2E,FF,FD,FE,STF,STD,STE,PCSrc);

input ResultSrcX,RegWriteX;
input [4:0]WriteAddX,rs1E,rs2E;
output reg FF,FD,FE,STF,STD,STE;
input PCSrc;

always@(*) begin
  if( ((WriteAddX == rs1E) | (WriteAddX == rs2E)) & (ResultSrcX | RegWriteX) &
(WriteAddX != 5'd0))
            FE = 1'b0;
          else if (PCSrc)
            FE = 1'b0;
          else
            FE = 1'b1;
end

always@(*) begin
  if(((WriteAddX == rs1E) | (WriteAddX == rs2E)) & ResultSrcX & (WriteAddX != 5'd0))
            FD = 1'b0;
          else if (PCSrc)
            FD = 1'b0;
          else
            FD = 1'b1;
end

always@(*) begin
  if(PCSrc)
            FF = 1'b1;
          else
            FF = 1'b0;
end

always@(*) begin
  if(((WriteAddX == rs1E) | (WriteAddX == rs2E)) & ResultSrcX & (WriteAddX != 5'd0))
            {STF,STD,STE} <= 3'b110;
  else if(((WriteAddX == rs1E) | (WriteAddX == rs2E)) & RegWriteX & (WriteAddX != 5'd0))
            {STF,STD,STE} <= 3'b111;
          else {STF,STD,STE} <= 3'b000;
end
endmodule
```

5-Stage Pipeline Processor

```
module PipelinedProcessorwithoutHazard(clk,reset);
input clk,reset;

wire [31:0]PCPlus4,PCPlus41,PCTargetE,PCNext,PCNext1,PC,Instr,InstrX;
reg PCSrc;
```

```
wire [31:0]InstrD,InstrL,InstrK,InstrY,PCD,PCPlus4D,WriteDataW,RD1,RD2,ImmExt;
wire Branch,MemWrite,ALUSrc,RegWrite,Jump,RegWriteW,SS1,SS2;
wire [1:0]ResultSrc, ImmSrc;
wire [2:0]ALUControl;
wire [4:0]WriteAddW;

wire [31:0]RD1E,RD2E,PCE,ImmExtE,PCPlus4E,SrcE,ALUResultE,A32,B32;
wire [4:0]WriteAddE,rs1,rs2,IS1,IS2;
wire [1:0]ResultSrcE,A,B;
wire [2:0]ALUControlE,funct3;
wire RegWriteE,MemWriteE,JumpE,BranchE,ALUSrcE,Zero;

wire [31:0]ALUResultM,RD2M,PCPlus4M,ReadDataM;
wire [1:0]ResultSrcM;
wire [4:0]WriteAddM;
wire RegWriteM,MemWriteM,SS11;

wire [31:0]ReadDataW,ALUResultW,PCPlus4W;
wire [1:0]ResultSrcW;

Mux2by1 #(32) pcmux(PCPlus4, PCTargetE, PCSrc, PCNext1);

FF32 #(32) ff0(clk, reset, PCNext1, PC);

    Mux2by1 #(32) stall2(PC, PCNext1, SS1, PCNext);
    InstructionMemory IM(PCNext,clk, InstrX,reset);
                Mux2by1 #(32) stall1(32'h00000013, InstrX, !PCSrc, Instr);
    Adder a0(PC, 32'd4 , PCPlus41);
                Mux2by1 #(32) st2(PC, PCPlus41, SS1, PCPlus4);


FF32 #(107) ff1(clk, reset, {Instr,PC,PCPlus4,Instr[19:15],Instr[24:20],SS1},
{InstrL,PCD,PCPlus4D,IS1,IS2,SS11});


    Mux2by1 #(32) ins(InstrL,InstrK, !SS11, InstrY);
                Mux2by1 #(32) ins1(InstrY,32'h00000013, PCSrc, InstrD);
    Controller2 c(InstrD[6:0], InstrD[14:12], InstrD[30],ResultSrc, ImmSrc, MemWrite,
            Branch,ALUSrc, RegWrite, Jump, ALUControl,SS2);
    Register1 rff(clk, RegWriteW, InstrD[19:15], InstrD[24:20],WriteAddW, WriteDataW,
RD1 ,RD2);
                Extend ext(InstrD[31:7], ImmSrc, ImmExt);
    HazardUnit HU(ResultSrcE[0],WriteAddE,IS1,IS2,SS1,SS2,PCSrc);



FF32 #(220) ff2(clk, reset,
{RegWrite,ResultSrc,MemWrite,Jump,Branch,ALUControl,ALUSrc,RD1,RD2,
        PCD,InstrD[11:7],ImmExt,PCPlus4D,InstrD[19:15],
InstrD[24:20],InstrD,InstrD[14:12]}, {RegWriteE,ResultSrcE,MemWriteE,JumpE,BranchE,

ALUControlE,ALUSrcE,RD1E,RD2E,PCE,WriteAddE,ImmExtE,PCPlus4E,rs1,rs2,InstrK,funct3});



                Mux3by1 #(32) muxA(RD1E, ALUResultM, WriteDataW, A, A32);
                Mux3by1 #(32) muxB(RD2E, ALUResultM, WriteDataW, B, B32);
                Mux2by1 #(32) srcbmux(B32, ImmExtE, ALUSrcE, SrcE);
    ALU2 alu(A32, SrcE, ALUControlE, ALUResultE, Zero,funct3);
    Adder a1(PCE,ImmExtE, PCTargetE);
                ForwardingUnit
fdu(WriteAddM,WriteAddW,rs1,rs2,RegWriteM,RegWriteW,A,B);
    always@(*) begin
```

```
                if((BranchE === 1'bx)|(Zero === 1'bx)|(JumpE === 1'bx))
                   PCSrc = 0;
                else
            PCSrc = (BranchE & Zero) | JumpE ;
               end


FF32 #(105) ff3(clk, reset,
{RegWriteE,ResultSrcE,MemWriteE,ALUResultE,B32,WriteAddE,PCPlus4E},
          {RegWriteM,ResultSrcM,MemWriteM,ALUResultM,RD2M,WriteAddM,PCPlus4M});


           DataMemory DM(clk, ALUResultE, B32, MemWriteE, ReadDataM);


FF32 #(104) ff4(clk, reset,
{RegWriteM,ResultSrcM,ALUResultM,ReadDataM,WriteAddM,PCPlus4M},
          {RegWriteW,ResultSrcW,ALUResultW,ReadDataW,WriteAddW,PCPlus4W});

    Mux3by1 #(32) resultmux(ALUResultW, ReadDataW, PCPlus4W, ResultSrcW,
WriteDataW);

Endmodule
```

HazardUnit

```
module HazardUnit (RegWriteE,WriteAddE,rs1,rs2,SS1,SS2,PCSrc);

input RegWriteE;
input [4:0]WriteAddE,rs1,rs2;
output reg SS1,SS2;
input PCSrc;

always@(*) begin
  if(((WriteAddE == rs1) | (WriteAddE == rs2)) & RegWriteE)
            SS2 = 1'b0;
          else
            SS2 = 1'b1;

  if (PCSrc)
            SS2 = 1'b1;
end

always@(*) begin
  if(((WriteAddE == rs1) | (WriteAddE == rs2)) & RegWriteE)
            SS1 = 1'b0;
          else
            SS1 = 1'b1;
end

endmodule
```

Registers

```
module register_up (
          input  logic       clk        , // Clock
          input  logic       reset      , // reset
          input  logic       write_enable, // Reset
          input  logic [ 4:0] read_addr1  , // Read address 1
          input  logic [ 4:0] read_addr2  ,
          input  logic [ 4:0] write_addr  , // Write address
          input  logic [31:0] write_data  , // Read address 2
          output logic [31:0] read_data1  , // Read data 1
          output logic [31:0] read_data2  , // Read data 2
          output logic [31:0] register_28 ,
          output logic [31:0] register_29 ,
```

```
        output logic [31:0] register_30 ,
        output logic [31:0] register_31
);

 logic [31:0] register[31:0];
 logic [31:0] reg_temp[31:0];

 always@(negedge clk) begin
        if(reset) begin
                read_data1  <= 0;
                        read_data2  <= 0;
                        register_28 <= 0;
                        register_29 <= 0;
                        register_30 <= 0;
                        register_31 <= 0;
        end
        else begin
                        read_data1  <= (read_addr1 == 5'b00000)? 32'b0 :
register[read_addr1];
                 read_data2  <= (read_addr2 == 5'b00000)? 32'b0 : register[read_addr2];
                 register_28 <= register[28]                    ;
                 register_29 <= register[29]                    ;
                 register_30 <= register[30]                    ;
                 register_31 <= register[31]                    ;
        end
 end

 always@(posedge clk) begin
                for (int i = 0; i < 32; i=i+1) begin
                        reg_temp[i] <= register[i];
        end
 end

 always@(*) begin
                for (int i = 0; i < 32; i=i+1) begin
                if (reset) begin
                                register[i] = 0;
                end
                else if(write_enable && (write_addr != 5'b00000) && (i == write_addr))
begin
                   register[i] = write_data;
                end
                else begin
                        register[i] = reg_temp[i];
                end
          end
 end

endmodule
```

Forwarding Unit

```
module ForwardingUnit(rdM,rdW,rs1,rs2,RegWriteM,RegWriteW,A,B);
input [4:0]rdM,rdW,rs1,rs2;
input RegWriteM,RegWriteW;
output reg [1:0]A,B;

always@(*) begin
  if ((rdM == rs1) & !(rdM == 5'b00000) & RegWriteM)
            A = 2'b01;
        else      if ((rdW == rs1) & !(rdW == 5'b00000) & RegWriteW)
          A = 2'b10;
        else
```

```
            A = 2'b00;
end

always@(*) begin
  if ((rdM == rs2) & !(rdM == 5'b00000) & RegWriteM)
            B = 2'b01;
          else      if ((rdW == rs2) & !(rdW == 5'b00000) & RegWriteW)
            B = 2'b10;
          else
            B = 2'b00;
end

endmodule
```