TOOLCHAIN DEVELOPMENT AND QUEUE SYSTEM ENHANCED SECURITY COPROCESSOR FOR FPGA-BASED INTERNET OF THINGS (IoT) PROCESSOR

SEE JIN CHUAN

MASTER OF SCIENCE (COMPUTER SCIENCE)

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY
UNIVERSITI TUNKU ABDUL RAHMAN
DECEMBER 2019

TOOLCHAIN DEVELOPMENT AND QUEUE SYSTEM ENHANCED SECURITY COPROCESSOR FOR FPGA-BASED INTERNET OF THINGS (IoT) PROCESSOR

By

SEE JIN CHUAN

A dissertation submitted to the Department of Computer and Communication Technology,
Faculty of Information and Communication Technology,
Universiti Tunku Abdul Rahman,
in partial fulfillment of the requirements for the degree of
Master of Science (Computer Science)
December 2019

ABSTRACT

TOOLCHAIN DEVELOPMENT AND QUEUE SYSTEM ENHANCED SECURITY COPROCESSOR FOR FPGA-BASED INTERNET OF THINGS (IoT) PROCESSOR

See Jin Chuan

Internet of Things (IoT) is developing by leaps and bounds in recent years, which opens up many interesting applications that potentially revolutionize our daily life. Many IoT processors and sensor node designs are being proposed in recent years for various applications, including those designed based on microcontroller, ASIC and FPGA. A recently proposed FPGA based IoT processor, RISC32, is one of the notable examples that provide flexible configurability to meet the needs in IoT applications. However, it does not come with compilation toolchain that support high level language, which increases the code development time. On top of that, one of the main reasons that limit the widespread adoption of IoT in many fields, is the lack of security feature. For instance, failure to provide data confidentiality could cause information leak and bring losses to the users. Unfortunately, RISC32 does not support encryption capability in hardware. In view of that, this research work aims to improve RISC32 in two aspects: providing

compilation toolchain in C language and introduce hardware core to perform encryption.

The most commonly used encryption scheme, Advanced Encryption System (AES) was used in this research work. While AES could be effectively implemented in software, the performance is slow, at the same time affecting the energy efficiency and responsiveness of IoT sensor node. This research work implemented AES as a coprocessor to RISC32 to speed up the encryption process. Experimental result shows at least 200x speed-up and ~99% energy reduction achieved by the AES coprocessor, compared to the software implementation. However, the RISC32 processor has to wait for AES core to complete the encryption before proceeding with other operations, due to data dependency. Hence, a novel Queue System is proposed to overlap the encryption operation with sampling of data, which follows the typical IoT software pattern. Further 1.48x speed-up and ~19% energy reduction was achieved with the introduction of Queue System. To enable rapid IoT application development on RISC32, this research work also delivers a compilation toolchain for RISC32 based on retargetable compiler framework, LLVM. By utilizing the existing MIPS Backend, the LLVM is extended to support code generation for RISC32. The compilation toolchain enables development option using C language on RISC32, where it was previously restricted to slow and error prone assembly language development option.

The achievement obtained in this research work is beneficial to IoT applications, which emphasize on performance and energy consumption. The

proposed Queue System can be used by other processor architectures to efficiently integrate with another block cipher coprocessor. On the other hand, the developed LLVM compilation toolchain can also allow easy extension of additional coprocessors to the RISC32 IoT processor.

ACKNOWLEDMENT

First, I would like to thank the university for funding this project. This project was funded under Universiti Tunku Abdul Rahman Research Fund (UTARRF) with the grant number IPSR/RMC/UTARRF/2016-C2/L04. Next, I would like to thank both of my supervisors Dr. Lee Wai Kong and Mr. Mok Kai Ming for their patience and guidance. Without them, I would have not been able to complete this dissertation. Also, I would like to thank all postgraduate friends that I have met during my master's degree. Although not all us working in the same domain, we share our knowledge and experience among us from time to time. Finally, I would like to thank my family for their support and encouragement. Their support and encouragement are the ones that keeps me motivated to continue and pursue my master's degree.

APPROVAL SHEET

This dissertation entitled "**TOOLCHAIN DEVELOPMENT AND QUEUE SYSTEM ENHANCED SECURITY COPROCESSOR FOR FPGA-BASED INTERNET OF THINGS (IoT) PROCESSOR"** was prepared by SEE JIN CHUAN and submitted as partial fulfillment of the requirements for the degree of Master of Science (Computer Science) at Universiti Tunku Abdul Rahman.

Approved by:

_____

(Dr. Lee Wai Kong) Date: 24th December 2019
Supervisor
Department of Computer and Communication Technology
Faculty of Information and Communication Technology
Universiti Tunku Abdul Rahman

_____

(Mr. Mok Kai Ming) Date: 24th December 2019
Co-supervisor
Department of Computer and Communication Technology
Faculty of Information and Communication Technology
Universiti Tunku Abdul Rahman

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

Date: 24th December 2019

SUBMISSION OF DISSERTATION

It is hereby certified that **SEE JIN CHUAN** (ID No: **17ACM05149**) has completed this dissertation entitled "**TOOLCHAIN DEVELOPMENT AND QUEUE SYSTEM ENHANCED SECURITY COPROCESSOR FOR FPGA-BASED INTERNET OF THINGS (IoT) PROCESSOR**" under the supervision of **Dr. Lee Wai Kong** (Supervisor) from the Department of Computer and Communication Technology, Faculty of Information and Communication Technology , and **Mr. Mok Kai Ming** (Co-Supervisor) from the Department of Computer and Communication Technology, Faculty of Information and Communication Technology.

I understand that University will upload softcopy of my dissertation in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

_____

(**SEE JIN CHUAN**)

DECLARATION

I hereby declare that the dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

_____

(SEE JIN CHUAN)

Date: 24th December 2019

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

ADC	Analog-to-Digital Converter
ADL	Architecture Description Language
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
BLE	Bluetooth Low Energy
BRAM	Block RAM
BUFG	Global Buffer
CBC	Cipher Block Chaining
CP0	Coprocessor 0
CP1	Coprocessor 1
CP2	Coprocessor 2
CTR	Counter
DAG	Directed-Acyclic-Graph
ECB	Electronic Codebook
EX	Execution
MEM	Memory
FF	Flip-Flop
FIPS	Federal Information Processing Standards
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GCC	GNU Compiler Collection

GPIO	General-Purpose Input Output
GPP	General Purpose Processor
I/O	Input/Output
ID	Instruction Decode
IDE	Integrated Development Environment
IF	Instruction Fetch
IoT	Internet of Things
IR	Intermediate Representation
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
IV	Initialization Vector
LUT	Look-Up Table
LUTRAM	Look-Up Table Random Access Memory
NIST	National Institute of Standards and Technology
OS	Operating System
RAM	Random Access Memory
RTL	Register-Transfer Level
SoC	System-on-Chip
SPI	Serial Peripheral Interface
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver Transmitter
WB	Write Back

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

Developing a processor targeting IoT applications is challenging as different IoT applications require microcontroller of different capabilities. The selection of processor to implement sensor nodes varies between low-end or mid-end microcontroller. For example, sensor nodes for environment monitoring only needs low-end processor as the data sampling rate is low; whereas surveillance system for smart city requires high-end processor to deal with image processing. Due to the wide range of performance requirement between each IoT application, manufacturer needs to select a new processor for almost every different application. The company will have to bear huge operating cost since this indicates the need to maintain engineering team with different firmware skill sets for different project. Hence, customizable IoT processor was previously developed to resolve this issue (Kiat, 2018). This dissertation is an extension to the previous work (Kiat, 2018) which focuses on two aspects: development of a new compilation toolchain and novel technique to integrate an AES cryptographic coprocessor with improved performance.

The customizable IoT processor (RISC32) introduces new hardware architecture, so a new compilation toolchain is required to enable firmware development in C language. Without toolchain, developing firmware at

assembly level is error prone, at the same time requires extensive understanding towards the underlying architecture. With compilation toolchain, it can greatly reduce the development time. The company is also at disadvantage due to the competitive market and constant emergence of new product. For example, a compiler takes in source code developed in high-level language and generates equivalent assembly language. Output of the compiler is then assembled into machine code to be readily executed on target processor by assembler. This shows the importance of toolchain as it aids development progress by abstracting out most of the details of the underlying architecture. This project aims to develop the toolchain that can convert C code to binary executables for the RIC32 IoT processor.

Since the primary function of the toolchain is to provide binary executables from user source code, this makes the compiler the core component of a toolchain. However, developing a compiler from scratch requires substantial efforts. A typical compiler consists basic structure as shown in Figure 1.1.

Figure 1.1: Basic Compiler Structure
Source: Lattner, n.d.

A basic compiler can be partitioned into three mains parts, namely the frontend, optimizer and the backend. The frontend performs analysis on the high-level-languages (e.g.: C, C++, FORTAN and etc.) and converts it to different representation known as intermediate representation (IR) for further transformation. Next, optimizer will analyse and optimize the IR to generate shorter and more efficient code. Output from optimizer is then further transformed and synthesized to machine code of the target processor by backend. Designing a compiler is time consuming, and requires great deal of knowledge and experience in algorithm study. Hence, developing a compiler from scratch is not an ideal case most of the time.

To resolve this issue, one of the alternatives available is retargetable compilers, which has similar structure to a basic compiler, but is designed to be customizable. For instance, retargetable compilers can be extended to support compilation of multiple target machine instruction's set. This feature makes cross-compilation possible. To support a new target machine, only new backend needs to be developed and paired with the existing framework of the retargetable compiler. This flexibility is convenient when compared to normal compilers that are targeted for specific machine, which requires a new compiler to be developed whenever a new processor is introduced.

The development toolchain is critical to deliver fast prototyping of IoT applications to cope with its rising demands. However, the rising demand for IoT applications also raises security concerns. This is because data transfer happens between interconnected IoT devices all the time. For example, food

factories use bio-sensors to monitor condition of raw food materials in large refrigerators (Xu et al., 2014). Household consumers' electrical usage collected by smart meters are transferred periodically in smart grid applications (Alahakoon and Yu, 2016). These IoT applications transfer sensitive data's that could be misused, thus threatening the user's privacy. For instance, the deterioration of raw food material should be kept confidential to maintain the reputation of the food factories (Xu et al., 2014). Electrical usage patterns of household users could be analysed to determine whether a house is currently vacant (Valerio, 2016). These scenarios could lead to financial losses of the users, which shows the need for security feature in IoT applications. Hence, security is an important criterion to be fulfilled in IoT application-based processor.

When it comes to fulfilling IoT security, it refers to providing several main features such as confidentiality, integrity, and availability (Humayed et al., 2017; Tomić and McCann, 2017). This project aims to provide confidentiality at current phase of research. Confidentiality refers to maintain the secrecy of transferred data. This could secure the data from attack such as eavesdropping, which ensures that the data could not be known even it is intercepted during transmission. The confidentiality of data can be protected through encryption schemes. With encryption, the data to be transferred (plaintext) will go through series of transformation using a secret key. The resultant output is cipher text, which is the original data scrambled into random and meaningless form. Without knowing the encryption algorithm or

secret key used, one could not retrieve the original content of the scrambled data. Hence, confidentiality of data is guaranteed.

Encryption algorithm while available in abundance, have to be carefully selected to cater for resource constrained IoT processor. In cases where IoT processor is developed for remote sensing purpose, power would be the main concern. The implementation of encryption algorithm should not burden the power consumption of IoT processor. This is to ensure the longevity of the power source (battery) for remote senor node. The choice of encryption should also provide reasonable performance (processing speed) and strength (security level) in spite of the power constrain. Among the abundant encryption algorithm, Advanced Encryption Standard (AES) is one of the most popular and recognized by international standards such as International Organization for Standardization (ISO/IEC-18033-3, 2005) and National Institute of Standards and Technology (FIPS, 2009). It was also adopted as part of the Transport Layer Security (TLS) protocol (Rescorla, 2018), which is used as secure communication protocol among internet-enabled devices. Hence, this project aims to implement AES to provide confidentiality feature in the RISC32. New instructions are created in the compilation toolchain to support the AES cryptographic coprocessor. At the same time, a Queue System is proposed to improve the execution speed of AES cryptographic coprocessor in RISC32 IoT processor.

1.2 Problem Statement

RISC32 (Kiat, 2018), a MIPS-ISA compatible processor, was designed as a customizable IoT sensor node. However, the firmware development option is currently limited to assembly programming. This is inconvenient as instruction set is machine specific. Unlike assembly language, high-level language such as C can be used for any machine, provided with the aid of compilation toolchain. Development is also hindered as most of the standard libraries are offered in high-level languages. Hence, a development toolchain is required to provide rapid development opportunity on RISC32.

RISC32 is designed for IoT purposes, but it still lacks basic security feature, which is one of the main concerns in IoT. Advanced Encryption Standard (AES) is selected in this project as it is an industrial standard block cipher. Although AES can be efficiently implemented in software, the speed performance is still too slow for IoT application. Prolonged execution of AES software might consume high energy, which is non-ideal for low power IoT applications. To address this issue, AES hardware coprocessor could provide better performance compared to software implementation. However, merely adopting an AES coprocessor may not yield optimized performance. When the processor is invoking AES encryption, the AES coprocessor is busy encrypting data but the processor itself is idle (waiting for encryption result). This is a limitation often neglected by other researchers (Wang et al., 2016). As such, this research work aims to fill this gap by proposing some techniques to integrate the AES coprocessor to an IoT processor (RISC32) with more optimized performance

1.3 Objectives

The primary goal of this research is to implement confidentiality feature into the RISC32, to fulfil the security criteria as an IoT processor. The implementation shall be able to encrypt sensor data before sending it out from RISC32. This project also aims to look into the possibility of utilizing the retargetable compiler to establish a compilation toolchain for RISC32. The established compilation toolchain shall be able to compile C code to binary executables that is compatible to RISC32. The objectives of this research are listed in detail as follows:

1) To develop a RISC32 compilation toolchain based on existing retargetable compiler framework, LLVM. The research will focus on extending the LLVM framework to support RISC32 instruction set compilation. The instructions to be extended will include existing instruction set of RISC32 and new instruction that might be introduced due to the integration of AES cryptographic coprocessor.

2) To integrate the AES as cryptographic coprocessor in RISC32. This refers to integrating the cryptographic coprocessor into RISC32 pipeline without affecting the current performance of RISC32. New instructions are created to use the AES coprocessor.

3) To develop a solution to improve the performance of AES cryptographic coprocessor in IoT application. This would require analysis of the software pattern in a typical IoT application with

encryption feature. The derived solution should be able to further speed up the encryption process.

4) To synthesis the RISC32 core with AES cryptographic coprocessor integrated on Xilinx Artix 7 FPGA chip. Experiments will be conducted on the RISC32 with AES coprocessor integrated and the proposed solution that optimize the AES coprocessor performance.

1.4 Contributions

Contributions of this dissertation are as follow:

1) A compilation toolchain that compiles C language program into RISC32 compatible instruction set. This toolchain is able to reduce the firmware development time on RISC32, at the same time enable the use of Operating System (OS) and development of device driver for existing IO module

2) A customizable IoT processor, RISC32 with encryption feature. The integrated AES coprocessor can be used to perform encryption on sensor data to be sent out of RISC32. This fulfils the confidentiality feature required for security criteria of IoT.

3) A solution to optimize the encryption process using integrated AES coprocessor. This solution (Queue System) is catered for typical IoT processing pattern. It ensures an optimal encryption performance through overlapping the encryption and program execution, eventually improving the overall speed performance. At the same time, ensuring optimized dynamic energy consumption without burdening the long-term energy consumption on RISC32.

1.5 Dissertation Organization

The dissertation starts with Chapter 1, to explain the background of the research. In Chapter 2, study is conducted on the existing integration technique of AES into processor and existing compilation technology. Chapter 3 is divided into two parts. The first part is on the toolchain development, where LLVM is extended to support RISC32 code generation. Second part describes the hardware development of this research work, which discusses the integration of AES coprocessor into RISC32. The proposed solution (Queue System) to optimize usage of AES Coprocessor is also discussed here. Chapter 4 is about the performance analysis of RISC32 with AES coprocessor integrated. Detailed assessment on program execution speed and dynamic energy consumption is conducted here. Finally, Chapter 5 concludes the research work and suggest the potential future directions of this research.

CHAPTER 2

LITERATURE REVIEW

2.1 RISC32

RISC32 (Kiat, 2018) is a MIPS Instruction Set Architecture (ISA) compatible 5-stage pipeline 32-bit IoT processor. It is able to decode and execute a subset of MIPS instructions. Figure 2.1 shows the simplified view of RISC32 microarchitecture.

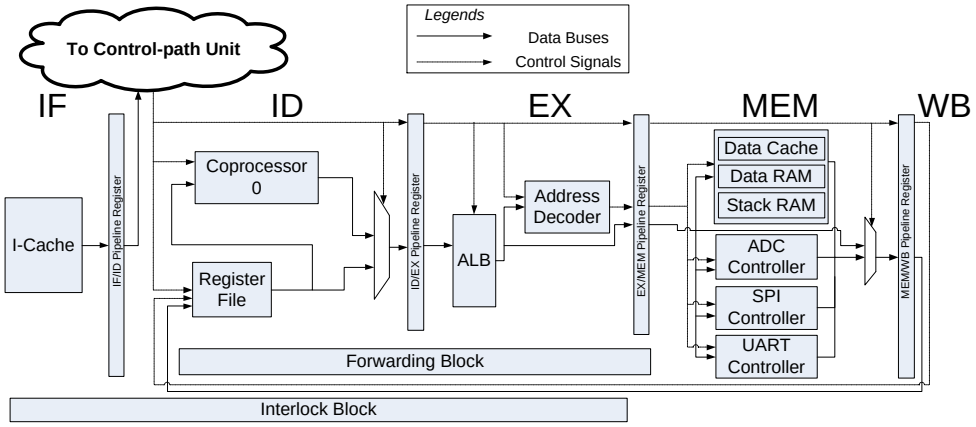


Figure 2.1: Simplified view of RISC32 microarchitecture

In RISC32, a Coprocessor 0 (CP0) is implemented to monitor hardware interrupts caused by the I/O controllers, and also software exceptions such as illegal instructions and arithmetic overflow. RISC32 is also integrated with common I/O controllers to provide common interface suitable for IoT applications. The I/O controllers available are UART, SPI, GPIO and ADC. These interfaces are compatible with wireless communication modules such as Bluetooth Low Energy (BLE), WiFi, ZigBee, etc., which provides the

connectivity feature for IoT applications. GPIO with 32-bit bidirectional I/O pins and 12-bit ADC ports are available from the RISC32 to provide common interface to sensors for data collection. Further details of RISC32 can be found in the dissertation by Kiat (2018).

2.2 Advanced Encryption Standard (AES)

AES is a symmetric block cipher published under the FIPS-197 (2009), a security standard publication by NIST. A typical block cipher requires two inputs: a plaintext/ciphertext, and a secret key. The AES uses the same secret key to perform encryption and decryption, hence it is symmetric. For AES, the operation is performed on a fixed sized data block, hence it is known as block cipher. The block size of AES is fixed at 128 bits (four 32 bits words or 16 bytes in equivalent) for both its input and output. In AES, a series of operation is applied on the input block for a fixed number of rounds to get the final output. The number of rounds is determined by the secret key size being used. There are three secret key sizes specified in FIPS-197, which is 128 bits (AES-128), 192 bits (AES-192) and 256 bits (AES-256). The number of rounds with respect to each key size is as follows: 10 rounds for AES-128, 12 rounds for AES-192 and 14 rounds for AES-256. Despite the difference in number of rounds, the operations to be performed on the input block is the same for each secret key size. The difference in key size, however, determines the strength of the encryption. Larger key size provides higher security level, which implies that it is more difficult to decipher the ciphertext using brute force attack. Another factor that determines the encryption strength of a block cipher is the block cipher mode being used. The mode determines the relationship between the secret key and the input block during the operation. The common modes available are Electronic Code Book (ECB), Cipher Block Chaining (CBC) and Counter (CTR). Other available block cipher modes can be found in NIST SP-800-38A (2007).

2.2.1 Existing AES Hardware Implementation

AES hardware implementation has been actively researched in several aspects, such as reduced hardware footprint (Lu et al., 2018), energy efficient (Hoang et al., 2017) and high throughput (Wang and Ha, 2016). In both work by Lu et al. (2018) and Hoang et al. (2017), their design achieved high energy efficiency, which is ideal for IoT processor with energy constrain. However, both design yields AES hardware with long encryption cycles, which is at least 160 over cycles per encryption. The long encryption cycle could be a performance bottleneck to IoT applications that requires real-time response.

As for the case of high-throughput design (Wang and Ha, 2016), the core design focuses on performing more data encryption by applying pipelined design on their core. The pipelined design divides crucial processing components for each round into several stages. This approach splits the critical path, enabling the design to operate at a higher frequency. This also enables encryption of multiple data to overlap, introducing higher throughput. This however, results in a larger circuit, which would lead to higher energy consumption. The high energy consumption is a concern for IoT processor design that has energy constrain. However, all of the proposed work mentioned above, while showing effort in optimizing the core, does not discuss on how the AES core can be integrated with a main processor efficiently. Only a few works (Soliman and Abozaid, 2011; Anwar et al., 2014; Wang et al., 2016; Yuan et al., 2018) have discussed their design on integrating AES core with a host processor.

2.2.2 AES Integration to Host Processor

This section discusses the works that have presented both their AES implementation and integration to a host processor. A total of 4 literatures (Soliman and Abozaid, 2011; Anwar et al., 2014; Wang et al., 2016; Yuan et al., 2018) will be discussed here.

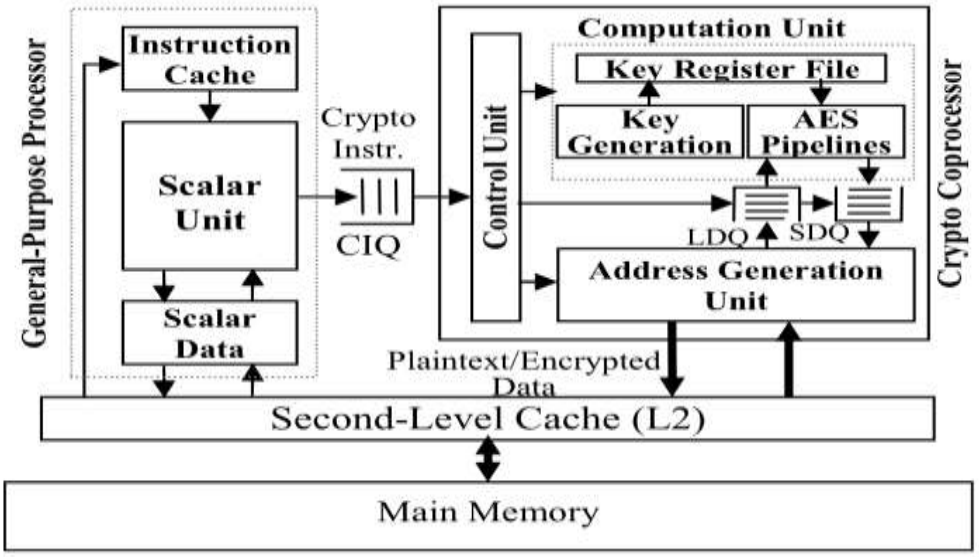


Figure 2.2:Microarchitecture Design of FastCrypto
Source: Soliman and Abozaid, 2011.

Figure 2.2 shows the work by Soliman and Abozaid (2011). In this work, a high throughput AES hardware implementation was proposed. The work is implemented on Xilinx Virtex V FPGA. AES is implemented as a crypto coprocessor, which was integrated into the data-path of general-purpose processor (GPP) by creating specialized data transfer path between the coprocessor and data-path of the GPP. As such, special instructions are created to interact with the coprocessor from the data-path of the GPP. The special instructions are created to provide information in commanding the AES crypto

coprocessor to carry out its task. The special instructions are encoded with information such as starting address to read input data, starting address to store output data, total length of data to be processed and action to be performed on the data (encryption or decryption). Each of the special instructions will first go through decoding stage of the GPP data-path. If a special instruction is detected, it will be dispatched to a specialized queue known as Crypto Instruction Queue (CIQ), waiting to be executed by the crypto coprocessor. The crypto coprocessor will further decode the instruction, extracting addresses and operation to be performed. The address extracted will be used internally by the Address Generation Unit, to generate necessary addresses to read range of data from Second-Level Cache (L2) for processing.

The AES crypto processor (Soliman and Abozaid, 2011) achieve high throughput, by using AES core with pipeline design. Furthermore, the coprocessor is designed with multiple lanes, which indicates existence of multiple AES core in the coprocessor. This design enables large amount of data to be processed within a specific time, hence, achieve high-throughput. However, this indicates the AES crypto coprocessor is large. The large architecture would certainly consume high energy. This high throughput trade-off for high consumption is definitely not suitable for IoT processor with energy constrain.

Furthermore, the AES crypto coprocessor is integrated in such a manner that, the coprocessor is directly interfaced to a L2 Cache. This enables data access by the coprocessor without disrupting the data-path of GPP. This

results in the requirement for a dual-port L2 Cache. Dual-port L2 Cache indicates a larger memory hardware is required, since to create dual-port access, an extra address decoder is required in the existing memory hardware. This could slow down the performance of the memory system. This also introduce a larger memory hardware, which is not beneficial for IoT processor, as it will definitely link to higher energy consumption.

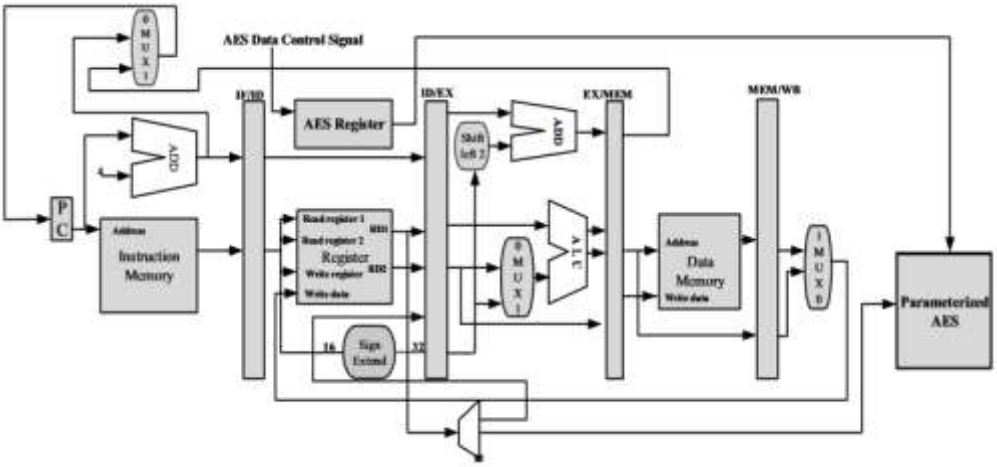


Figure 2.3:Processor microarchitecture with parameterized AES crypto-coprocessor
Source: Anwar et al., 2014.

Figure 2.3 shows the work by Anwar et al. (2014). In this work, a parameterized AES crypto coprocessor was proposed. This work is implemented on Xilinx Virtex 6 FPGA. The AES coprocessor was integrated to a general purpose 5-stage pipeline 32-bit MIPS processor. Specialized data transfer path was created to access the integrated coprocessor from the data-path of the processor. As such, special instructions are designed to interact the coprocessor through the specialized data transfer path. The special instructions, contain starting address for input and output, total length of data

to be processed, and operation to be performed on the data. The special instruction is decoded at the Instruction Decode (ID) stage in the processor, and transferred to coprocessor for execution. A specialized AES memory was introduced, to provide input data for processing by the AES coprocessor. The AES memory also stores the output data by AES coprocessor.

The AES coprocessor introduced by this work is parameterized, which is a pipeline AES architecture with tunable pipeline stages during design time. Although the design is tunable, it is still a pipelined AES core, which consumes large area and power hungry, so it is not suitable for IoT application. Furthermore, this integration method introduces a specialized AES memory. The AES memory introduced is large, which could be slow and might consume high power. This may not be a concern when the AES coprocessor was integrated to a general-purpose MIPS processor. However, it might not suitable for our research work, as our target application is IoT-based, which has power constrain in general.

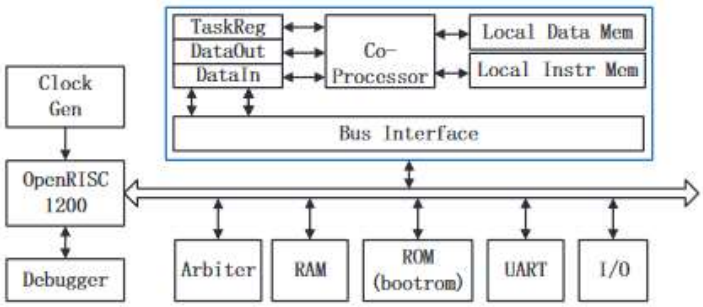


Figure 2.4: OpenRISC1200 interfaced with Crypto-Coprocessor through common bus.
Source: Wang et al., 2016.

Figure 2.4 shows the work by Wang et.al (2016). The work is implemented with Application Specific Integrated Circuit (ASIC) Technology. The AES coprocessor is implemented as an Application Specific Instruction Set Processor (ASIP). This indicates a special set of instruction is designed specifically to command the coprocessor to perform various operation. However, unlike special instruction created in previous work (Soliman and Abozaid, 2011; Anwar et al., 2014), the instruction set is decoded internally by the coprocessor in the case of ASIP. The internal microarchitecture of the AES coprocessor is shown in Figure 2.5.

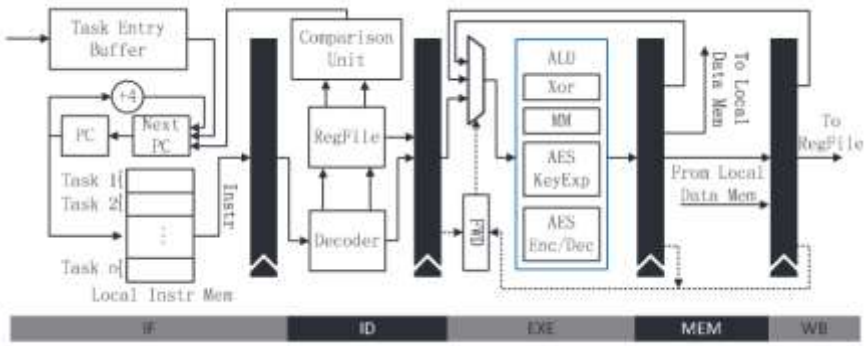


Figure 2.5: Proposed ASIP microarchitecture for AES crypto coprocessor
Source: Wang et al., 2016.

Referring to Figure 2.5, the coprocessor is designed with pipeline stages found in common processor pipeline. As for the interfacing between coprocessor and host processor (OpenRISC1200), it is interfaced through a common shared bus between other peripherals as shown in Figure 2.4. The *TaskReg*, *DataOut* and *DataIn* are designated with dedicated memory address. Accessing these registers can simply be done using load store instructions. The implementation technique used by the author is simple and straightforward. No special instruction or specialized transfer is required as shown in the other work (Soliman and Abozaid, 2011; Anwar et al., 2014) discussed earlier. Only proper assignment of memory address to map and access to the register of coprocessor is required. However, the performance of the coprocessor is largely dependent on the shared bus. This is because all peripheral behaves differently. Between the processor and the peripherals, each of them may have different communication protocol and data transaction pattern. These are all carried out using the shared bus. Peripherals with the slowest performance might be the performance bottleneck for the shared bus. Not to mention, since it is a shared bus, the peripherals are likely assigned with priorities for request of bus usage. In the case where multiple peripherals requested to use the bus, usage is assigned to the peripherals with highest priorities. If the coprocessor is assigned with a lower priority, the performance is likely affected by the peripherals with higher priority than the coprocessor. This shows that although shared bus technique is straightforward, the performance of coprocessor is not guaranteed to be optimal due to the factors mentioned.

Furthermore, the coprocessor implemented by Wang et al. (2016) closely resembles the processor pipeline. The coprocessor proposed may have high performance, but is expected to have large and complex circuitry. While the authors claim the coprocessor is suitable for IoT application, their work is implemented using ASIC technology. ASIC technology has been known for its high-power efficiency but low flexibility. Our work however, adopts FPGA implementation technology. Implementing their coprocessor with our FPGA technology is likely to yield higher power consumption. This indicates that their AES coprocessor is not suitable for our use case of FPGA implementation.

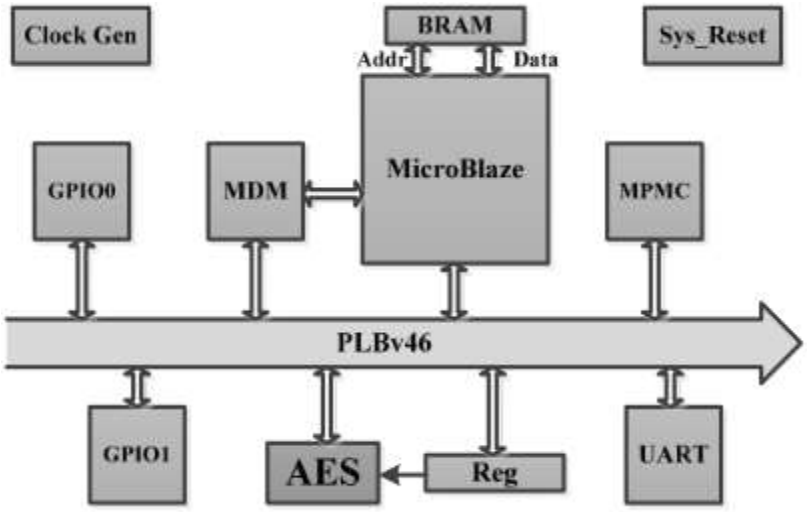


Figure 2.6: AES interfaced with MicroBlaze through PLB share bus
Source: Yuan et al., 2018.

Figure 2.6 shows the work by Yuan et al. (2018). This work proposed a novel implementation of pipelined AES. The work was implemented on Xilinx Spartan 6 FPGA. This work uses the share bus interfacing technique to interface the AES coprocessor with a host processor (MicroBlaze). As

mentioned earlier, the shared bus technique is simple and straightforward. However, performance of the coprocessor is not guaranteed, as it is largely dependent on other interfaced peripherals as well. Furthermore, the AES coprocessor proposed by the author is pipelined design. Pipelined AES core may effectively provide higher throughput, enabling more data encryption to be performed within a short time. The design, however, may yield a larger hardware. Hence, might not be ideal for IoT processor with power consumption concern.

Table 2.1: Summary of Existing Work

	Integration Technique	AES Architecture	Technology	Remarks
Soliman and Abozaid, 2011	Dedicated-Path	Pipelined	FPGA	• Pipelined architecture enables high performance at the expense of larger hardware area. • Larger hardware area consumes higher energy. • Not ideal for IoT use case
Anwar et al., 2014	Dedicated-Path	Pipelined	FPGA	• Pipelined architecture enables high performance at the expense of larger hardware area. • Larger hardware area consumes higher energy. • Not ideal for IoT use case
Wang et al., 2016	Shared-Path	Single-Stage	ASIC	• Shared-Path integration technique is easier to realize • But performance of shared-bus hardware is dependent on overall system bus performance • ASIC implementation is energy efficient but not as flexible when compared to FPGA
Yuen et al., 2018	Shared-Path	Pipelined	FPGA	• Shared-Path integration technique is easier to realize. • But performance of shared-bus hardware is dependent on overall system bus performance

2.3 Existing Toolchain Technology

Toolchain plays important role when developing on a new system. Without toolchain, a developer will go through cumbersome process when developing a new system. For example, developer may only program in assembly language. Then, the developer may need to manually convert each assembly language into machine code of the targeted machine. All of this is error prone and time consuming. Hence, toolchain is usually developed to be distributed as a package with every new system by the vendors. This results in variety of both proprietary and free IDE. Example of proprietary IDE is Visual Studio developed by Microsoft (2019) is the conventional IDE for x86 platform. As for free IDE, µVision® IDE (ARM Limited, 2019) is dedicated for ARM platforms. These IDE's are usually targeted to specific processor family, which indicates compilation for customized machine might not be supported. Extending the IDE's to support new target machine is also impossible as they are usually closed-source, not to mention the risk of spreading into infringement issue. Since modification to vendor distributed software is not an option, alternatives such as automatic software generation using Architecture Description Language (ADL) and retargetable compilers are opted.

2.3.1 Architecture Description Language (ADL)

An ADL is a language designed to specify relationship and interaction between each component/block on System-on-Chip's (SOCs). The blocks/components include the processor, peripheral device such as memories, and interfacing circuits of each peripheral. Production-grade software toolchain such as compiler, assembler, linker, debugger and simulator can be synthesized from an ADL (Tomiyama et al., 1999). Based on the paper by Kassem et al. (2012), ADL can be further classified based on their contents and objectives. In terms of contents, the ADL can be categorized into three categories, namely the Behavioural ADL, Structural ADL and Mixed ADL (Kassem et al., 2012). The Behavioural ADL can only describe instruction set instead of structural details of a processor model. Structural ADL on the other hand, can only be used to model structural behaviour such as register transfer level of a processor. As for Mixed ADL, they can achieve both structural and behavioural modelling of a processor. In terms of objectives classification, the ADL can be categorized into 4 categories, namely the Synthesis ADL, Compilation ADL, Validation ADL and Simulation ADL (Kassem et al., 2012; Tomiyama et al., 1999). Synthesis ADL's focuses on describing and designing hardware architectures. Compilation ADL's focuses on code generator design. Validation ADL's is mainly used for embedded processor functional verification purposes. Simulation ADL's generates simulator for hardware generated. Over the years, ADL has been widely used for Application-Specific Instruction set Processor (ASIP) development as presented in several papers (Bejo et al., 2014; Gupta and Pal, 2015; Taglietti et al., 2005). Development of RISC processor using ADL has also been

presented in this paper (Arora et al., 2015). ADL may be efficient in terms of higher abstraction level processor modelling and automatic software toolchain generation, there are trade-offs imposed in exchange for the convenience. As mentioned in this paper (Witte et al., 2005), unoptimized RTL code generation is the bottleneck of ADL generated processor. Hence, various optimization schemes have to be introduced in the ADL compiler to generate optimized RTL code. Due to the lack of power model, evaluation and optimization of power consumption cannot be done on ADL as mentioned here (Yang et al., 2013). Both mentioned disadvantages impact IoT processor design that focuses on low-power and energy efficient design.

2.3.2 Retargetable Compilers

Due to the difficulties of designing a new compiler from scratch, extending support of existing toolchain is another approach to design new toolchain for new machine. As such, retargetable toolchain framework such as GNU Compiler Collection (GCC) and LLVM is often being used. Both frameworks are retargetable, hence they can be extended to support code generation of new instruction set. Since they are open source, both frameworks are becoming robust and quality compilers, due to the growing contribution from their development community.

2.3.3 GCC

The internals of GCC is shown in the Figure 2.7.

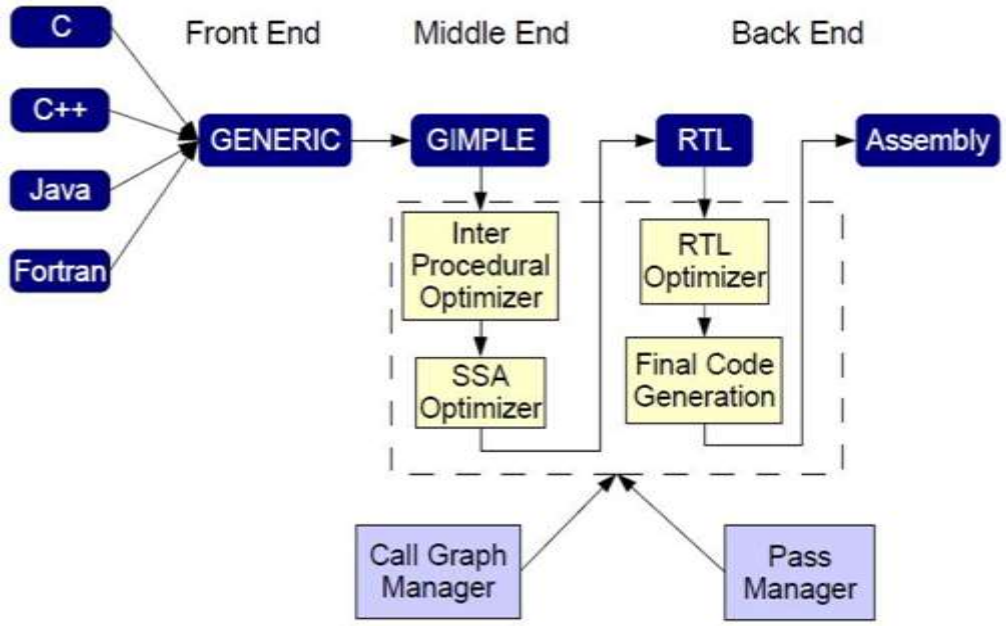


Figure 2.7: GCC Internals
Source: Diego, 2007.

As shown in Figure 2.7, high-level language input to GCC will first be parsed into GENERIC, an intermediate language of GCC. The GENERIC is further transformed to GIMPLE, a tree-based intermediate representation of GCC by the middle end. Transformation of GIMPLE is done in two phases. The GENERIC from frontend will be translated to High GIMPLE, which is target independent. After analysis and transformation by several passes, High GIMPLE will then be lowered to Lower GIMPLE to be constructed as a more target dependent representation. In the backend, GIMPLE is then expanded to Register Transfer Language (RTL) form for instruction matching of target machine assembly to finally generate the machine code. Hence, to support a

new target, most of the effort should be focused on the backend. A number of machine description files will be needed to be specified in the backend. Since GCC is relatively mature, it has been widely adopted for both industrial and academic sectors. In the work by Johann et al. (2016), GCC 4.6.1 is used to setup the toolchain for HF-RISC core. The GCC 4.6.1 is modified to support new flags that correspond to code generation support for different HF-RISC processor configuration.

GCC is also integrated into CooCox CoIDE for cross-compilation of ARM Cortex-M processor family. However, the paper (Campi et al., 2003) suggests that the backend development on instruction patterns file (.md) of machine description was rather difficult due to limited support from GCC. It was also reviewed in this paper (Ghica and Tapus, 2015) that the machine description (.md) was rather difficult to read, enhance, construct and maintain. "*They require specifying instruction patterns using Register Transfer Language (RTL) templates, employing a mechanism which is verbose, repetitive and requires a lot of detail"* (Ghica and Tapus, 2015). The GCC Wiki (Bosscher, 2012) also stated that GCC has grown rather big which induces steep learning curves for new developers.

2.3.4 LLVM

Unlike GCC, LLVM is modular in design. The overview of LLVM Internals is shown in Figure 2.8.

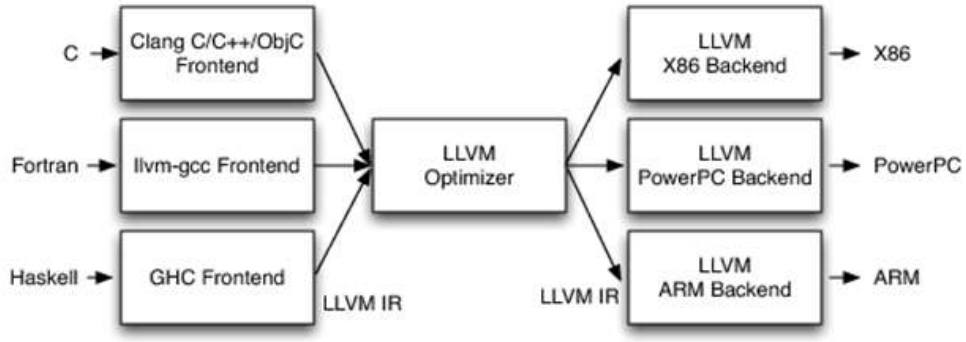


Figure 2.8: LLVM Internals
Source: Lattner, n.d.

Compared to GCC, internals of LLVM is relatively straightforward. As shown in Figure 2.8, the LLVM is made up of three main components. The frontend is responsible for parsing of high-level-language such C, C++ and Java. Each LLVM frontend is exclusive to one language, for example the Clang is the C Frontend while GHC is the Haskell Frontend. The output from frontend is then passed to the LLVM optimizer in the form of LLVM Intermediate Representation (IR). LLVM IR is a RISC assembly like language used by LLVM framework for internal analysis and transformation. It is used to expose lower level information from high level language, but yet target independent. By going through various transformation passes in LLVM optimizer, the final output will be a more optimized LLVM IR. The backend will then generate target specific machine code from the IR. Each backend will be exclusively used for a single target family and are independent of each

other. In the backend, the LLVM IR will be constructed into SelectionDAG, which is a graph-based representation to expose more target dependent information. Mapping of native instruction will be based on the constructed SelectionDAG. Hence, to support a new target machine, information of respective machine is needed. In LLVM backend, the machine description files will be generated by a LLVM utility, namely the TableGen (tblgen) tool. The TableGen will generate records to represent respective target based on their target description (.td) files. The Figure 2.9 shows example of target description generation in LLVM.

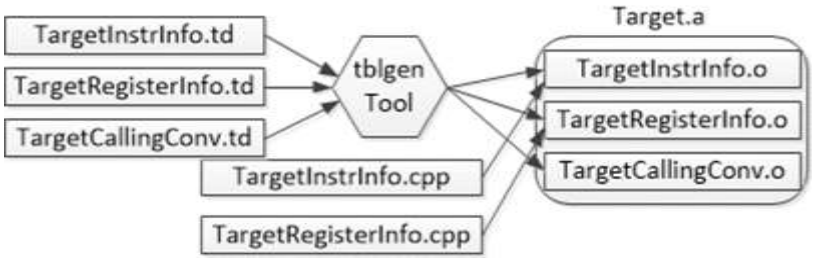


Figure 2.9: Target Description generation using TableGen
Source: Lattner, n.d.

When it comes to maturity, GCC superseded LLVM due to its longer history. However, LLVM is still a popular selection due to its relatively straightforward and easier to understand design. It has been adopted by several works as presented in PicoBlaze Processor (Sýkora, n.d.), RISCO processor (Vilela et al., 2012) ,microMIPS architecture (Kolek et al., 2013) and LEON Processor (Lopez et al., 2015). It is use by Apple Inc. (2017) as their backend for their Xcode IDE, which is used to develop various applications for their Apple products.

2.4 Summary

In this chapter, existing techniques to integrate AES coprocessor to host processor are explored. The common AES integration techniques are dedicated path technique and shared bus integration technique. Dedicated path technique requires introduction of specialized transfer path and new instructions to access the integrated coprocessor. While this technique requires revision to the whole processor architecture, the coprocessor can have a better performance due to its independent transfer path. As for shared bus integration technique, it is relatively simpler and straightforward to realize. However, the performance of integrated coprocessor is dependent on other peripherals that is present on the shared bus.

This chapter also explores existing compilation toolchain technologies. The compilation toolchain is usually distributed along with the processor sold by vendor. However, vendor distributed Integrated Development Environment (IDE) is usually target specific and close source. It cannot be modified for code generation of unsupported target. Hence, alternative such as Architecture Description Language (ADL) and retargetable compilers are often opted. ADL supports automated toolchain generation for the processor modelled using it. However, lack of power modelling in ADL makes it harder to gauge the energy performance of the processor modelled using it. Retargetable compilers are open-source and is retargetable to support compilation for multiple target machine. Example of popular retargetable compilers are GCC and LLVM. GCC is much mature due to its longer history when compared to LLVM.

However, due to the complicated structure of GCC, LLVM gains popularity due to its clear-cut design.

CHAPTER 3

SYSTEM DESIGN

3.1 System Overview: Software

This section describes the proposed software compilation toolchain design for this research work. The proposed software toolchain design is shown in Figure 3.1.

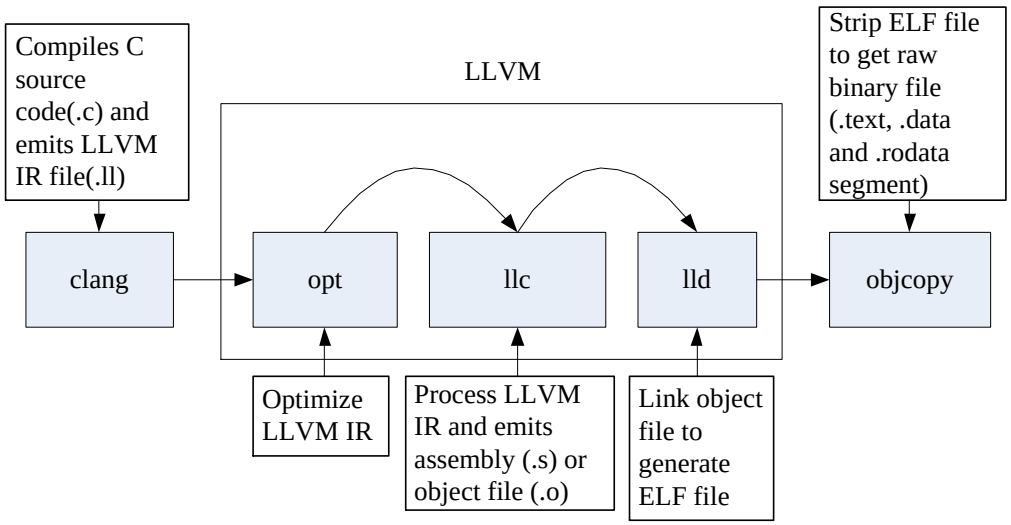


Figure 3.1: Simplified architecture of RISC32 compilation toolchain

The Figure 3.1 shows the simplified architecture of RISC32 compilation toolchain. The toolchain is developed based on LLVM (2013) modular compilation framework, wherein different frontends (process source code) can be freely paired with different backends (emits assembly or object file). Clang is the frontend for LLVM, which processes C language source code and generates LLVM Intermediate Representation (IR). LLVM IR is the

initial input into the LLVM framework, which is further analyzed and transformed into various intermediate forms in between each compilation module.

The initial module that processes the LLVM IR would be *opt*, which is known as LLVM optimizer, responsible to perform various analyses in order to optimize the input LLVM IR. Following this, the *llc* module (static compiler of LLVM) analyzes the LLVM IR further and transform it into various intermediate forms, eventually mapped to the instruction set of the desired target machine (RISC32 for this research work). It can generate output as assembly file or binary object file. The final crucial module would be *lld*, which is the LLVM linker, responsible in performing address calculation for the binary object file generated by static compiler *llc*, and output a final executable object file. As *lld* generates object file in Executable and Linkable Format (ELF), it contains various Operating System (OS) related headers or information sections. This output file is still not suitable to be executed on RISC32, as this research work currently do not intend to host any OS in RISC32 at the moment. Hence, the *objcopy* will be used to extract only the Text Segment (.text) and Data Segment (.data and .rodata) which contains the instructions and data respectively. The binary content extracted from respective sections will be loaded into suitable memory address location based on memory map established for RISC32. The memory map for RISC32 is shown in the Figure 3.2.

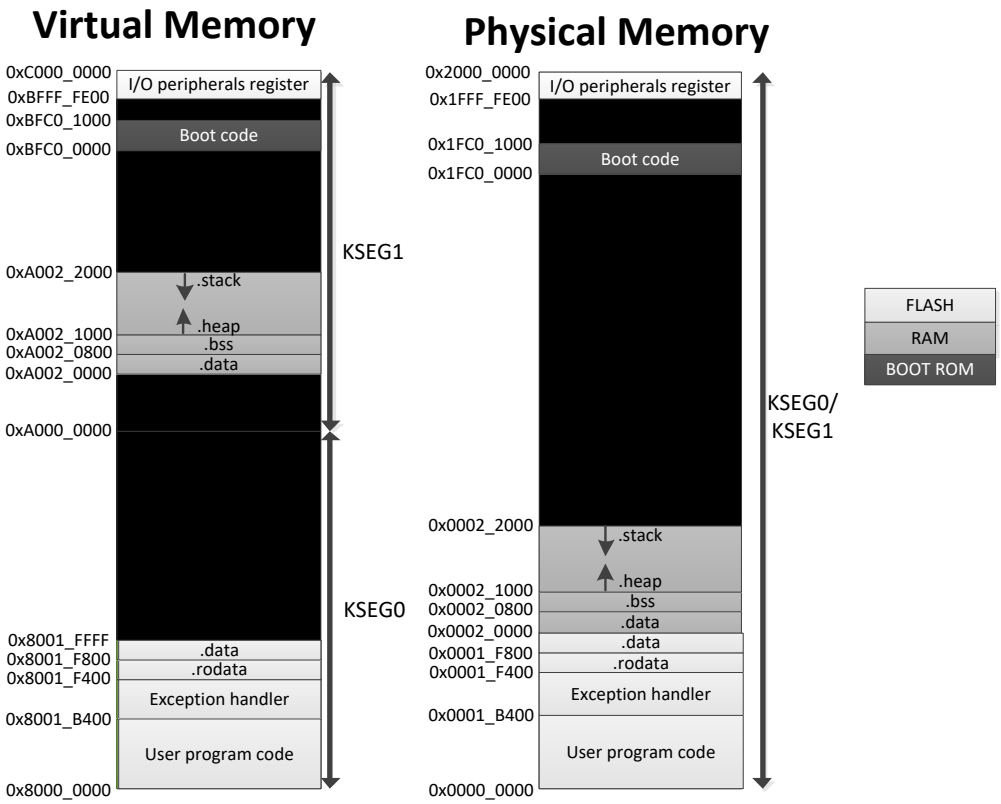


Figure 3.2: Memory map for RISC32
Source: Kiat, 2018.

The following section (Section 3.1.1) discuss the existing RISC32 Instruction Set. Following by, analysis and comparison is performed in Section 3.1.2, on the existing instruction set implemented in MIPS Backend of LLVM. Next, porting of the instruction by category (Section 3.1.3 to 3.1.6) into LLVM is discussed. Finally, the porting of RISC32 Interrupt Service Routine (ISR) programming feature (Section 3.1.7) into LLVM.

3.1.1 RISC32 Instruction Set

RISC32 is a MIPS Instruction Set Architecture (ISA) compatible processor, capable in decoding and executing a subset of the standard MIPS instruction. Part of our research work is to customize the existing LLVM MIPS Backend to compile RISC32 instructions. RISC32 only supports 54 MIPS instructions (shown in Table 3.1) instead of the full-blown MIPS instructions (MIPS, 2016).

Table 3.1: RISC32 Instruction Set

Instruction Class	Instructions
Memory Access	*lw, lwl, lwr, lh, lhu, lb, lbu, sw, swl, swr, sh, sb*
Arithmetic	*add, addu, addi, addiu, sub, subu, mult, multu, mfhi, mflo, mthi, mtlo*
Bitwise	*and, or, xor, nor, sll, srl, sra, andi, ori, xori, lui*
Condition Checking	*slt, sltu, slti, sltiu*
Program Control	*beq, bne, blez, bgtz, j, jal, jr, jalr*
System	*syscall, mtc0, mfc0, eret, mtc2, mfc2, swc2*

3.1.2 Analysis and Comparison of MIPS II vs RISC32 Instruction Set

The current LLVM MIPS Backend supports the MIPS instruction set architecture up to the latest generation (MIPS32 Release 6). A suitable MIPS architecture, MIPS II (Price, 1995), is chosen as the base to support the code generation for RISC32 due to its high similarity in supported instructions. However, not all MIPS II instructions can be executed by RISC32. Some modification is carried out to mask out the unsupported MIPS II instructions, thus, creating a new sub-target in the MIPS Backend for the RISC32 code generation. The new sub-target provides information for *llc*, the LLVM static compiler during the compilation phase to select valid instructions for RISC32.

The following tables (Table 3.2 to Table 3.8) show the comparison between RISC32 and MIPS II instruction set that is currently implemented in MIPS Backend of LLVM. Through the comparisons, the research work determines which instructions that are not supported and should be masked out for code generation of RISC32.

Table 3.2: Comparison for Memory Access Instructions

Instruction	Description	Supported by MIPS II?	Supported by RISC32?	Remarks
lb	Load Byte	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
lbu	Load Byte Unsigned	✓	✓	
lh	Load Halfword	✓	✓	
lhu	Load Halfword Unsigned	✓	✓	
lw	Load Word	✓	✓	
lwl	Load Word Left	✓	✓	
lwr	Load Word Right	✓	✓	
sb	Store Byte	✓	✓	
sh	Store Halfword	✓	✓	
sw	Store Word	✓	✓	
swl	Store Word Left	✓	✓	
swr	Store Word Right	✓	✓	

Table 3.3: Comparison for Arithmetic Instructions

Instruction	Description	Supported by MIPS II?	Supported by RISC32?	Remarks
add	Add Word	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
addu	Add Unsigned Word	✓	✓	
addi	Add Immediate Word	✓	✓	
addiu	Add Immediate Unsigned Word	✓	✓	
sub	Subtract Word	✓	✓	
subu	Subtract Unsigned Word	✓	✓	
mult	Multiply Word	✓	✓	
multu	Multiply Unsigned Word	✓	✓	
div	Divide Word	✓	✗	No divider hardware is implemented in RISC32. Will need to rely on software division.
divu	Divide Unsigned Word	✓	✗	
mthi	Move to HI Register	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
mfhi	Move from HI Register	✓	✓	
mtlo	Move to LO Register	✓	✓	
mflo	Move from LO Register	✓	✓	

Table 3.4: Comparison for Condition Checking Instructions

Instruction	Description	Supported by MIPS II?	Supported by RISC32?	Remarks
slt	Set on Less Than	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
sltu	Set on Less Than Unsigned	✓	✓	
slti	Set on Less Than Immediate	✓	✓	
sltiu	Set on Less Than Immediate Unsigned	✓	✓	

Through the comparison (Table 3.2 and Table 3.4), all the Memory Access and Condition Checking Instructions of MIPS II is supported in RISC32. Hence, the implementation of MIPS II code generation for both categories can be reused. As for Arithmetic Instructions (Table 3.3), the division operation is currently not supported in RISC32. The RISC32 also does not plan to implement those instructions, as it is relatively expensive to implement a hardware module to perform division. Assuming that division operation is required by user program, simple division operation by power 2 divisor can be performed by using logical right shift instructions. For non-power 2 divisor, software division is assumed to be implemented by the user. Hence, the Arithmetic Instruction code generation of MIPS II can be reused as well.

Table 3.5: Comparison for Bitwise Instructions

Instruction	Description	Supported by MIPS II?	Supported by RISC32?	Remarks
and	AND	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
or	OR	✓	✓	
xor	XOR	✓	✓	
nor	NOR	✓	✓	
andi	AND Immediate	✓	✓	
ori	OR Immediate	✓	✓	
xori	XOR Immediate	✓	✓	
lui	Load Upper Immediate	✓	✓	
sll	Shift Word Left Logical	✓	✓	
srl	Shift Word Right Logical	✓	✓	
sra	Shift Word Right Arithmetic	✓	✓	
sllv	Shift Word Left Logical Variable	✓	✕	Not supported in RISC32, but have valid C-syntax that will compile to it. Requires special handling.
srlv	Shift Word Right Logical Variable	✓	✕	
srav	Shift Word Right Arithmetic Variable	✓	✕	

For Bitwise Instructions (Table 3.5), the Shift-by-Variable operations is not supported. These instructions perform shifting based on register operands. This behaviour is quite common, as there might be times where shifting value is not known before program runtime. Hence, special handling for their compilation is discussed in Section 3.1.4.

Table 3.6: Comparison for Program Control Instructions

Instruction	Description	Supported by MIPS II?	Supported by RISC32?	Remarks
beq	Branch on Equal	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
bne	Branch on Not Equal	✓	✓	
blez	Branch on Less Than or Equal to Zero	✓	✓	
bgtz	Branch on Greater Than Zero	✓	✓	
bltz	Branch on Less Than Zero	✓	✗	Not supported in RISC32, but have valid C-syntax that will compile to it. Requires special handling.
bgez	Branch on Greater Than or Equal to Zero	✓	✗	
bltzal	Branch on Less Than Zero and Link	✓	✗	Not supported in RISC32, but does not have valid pattern matching implemented in LLVM for these instructions. Will never be compiled from C code. Hence, no action is needed.
bgezal	Branch on Greater Than or Equal to Zero and Link	✓	✗	
beql	Branch on Equal Likely	✓	✗	
bnel	Branch on Not Equal Likely	✓	✗	
blezl	Branch on Less Than or Equal to Zero Likely	✓	✗	
bgtzl	Branch on Greater Than Zero Likely	✓	✗	
bltzl	Branch on Less Than Zero Likely	✓	✗	
bgezl	Branch on Greater Than or Equal to Zero Likely	✓	✗	
bltzall	Branch on Less Than Zero and Link Likely	✓	✗	
bgezall	Branch on Greater Than or Equal to Zero and Link Likely	✓	✗	
j	Jump	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
jal	Jump and Link	✓	✓	
jr	Jump Register	✓	✓	
jalr	Jump and Link Register	✓	✓	

For Program Control Instructions (Table 3.6), it is shown that a large portion of conditional branch operations in MIPS II is not supported in RISC32. The conditional branch operation shown consists of two variants: Branch and Branch Likely. The Branch variant refers to conditional branch that allows the branch delay slot (instruction scheduled right after conditional branch) instruction to execute regardless of branch taken or not. All the Branch variant is supported in RISC32, except for Branch on Less Than Zero (*bltz*) and Branch on Greater Than or Equal to Zero (*bgez*). These two instructions are exactly opposite of the existing Branch on Greater Than Zero (*bgtz*) and Branch on Less Than or Equal to Zero (*blez*) respectively in RISC32, hence they will not be implemented as hardware. Compilation transformation for the unsupported *bltz* and *bgez* will be discussed in Section 3.1.5. For the Branch Likely variant, this group of instructions refers to conditional branch that nullifies the execution of branch delay slot instructions if branch untaken. These instructions are not required, as Branch Predictor (BP) is implemented in RISC32. The BP performs prediction, eliminating the delay slot for conditional branch. Furthermore, BP will also flush the delay slot instructions from RISC32 pipeline in the case of branch misprediction. Also, these instructions while present in MIPS Backend, no pattern matching implementation was specified for them. This means that the intermediate form will never be matched to these instructions, and will not select or generate the respective group of instructions.

Another group of branch instruction, long branches (*bltzal* and *bgezal*) is also not supported in RISC32. The long branches are typically used for

procedural calls, allowing conditional jump to a particular label address, at the same time, saves the return address to $ra ($31). The long branches are not required in RISC32, as one can pair the conditional branch with unconditional branch (*jal* and *jalr)* to achieve the same effect. Furthermore, the pattern matching is also not implemented for long branches, rendering it not selectable during compilation. Hence, the existing implementation of long branches instruction in MIPS Backend does not affect the code generation for RISC32.

Table 3.7: Comparison for System Instructions

Instruction	Description	Supported by MIPS II?	Supported by RISC32?	Remarks
syscall	System Call	✓	✓	Supported by RISC32. No action is needed
break	Breakpoint	✓	✕	Not supported, but will not compile from C-code. Hence, no action is needed.
eret	Exception Return	✕	✓	Not supported in MIPS II, but is implemented in LLVM MIPS Backend. Does not affect compilation for RISC32, as this instruction does not compile directly from C code. Will be accessed using inline assembly.
mtc0	Move Word to Coprocessor 0	✓	✓	Instructions are supported in both instruction set. No action is needed to be done in LLVM for RISC32 compilation.
mfc0	Move Word from Coprocessor 0	✓	✓	
mtc2	Move Word to Coprocessor 2	✓	✓	
mfc2	Move Word from Coprocessor 2	✓	✓	
lwc2	Load Word from Coprocessor 2	✓	✕	Not supported, but will not compile from C-code. Hence, no action is needed.
swc2	Store Word from Coprocessor 2	✓	✓	Supported by RISC32. No action is needed

The System Instructions comparison (Table 3.7) shows major system level instructions of MIPS II was supported in RISC32. The Breakpoint (*break*) instruction, is a software debugging feature, which allows processor to stop temporarily at a particular point of user program. This feature however, is not supported in RISC32 and shall be considered for future implementation. As for the Exception Return (*eret)*, this instruction is required in RISC32 to return from the kernel mode (exception handler) to user mode. While it is not present in MIPS II, it was implemented in MIPS Backend for the later MIPS generation. This however, does not affect the code generation for RISC32, because the system instructions are typically non-mappable from C-syntax, and they can only be accessed by means of inline assembly. In other words, as long as their instruction format and encoding is implemented in the MIPS Backend, the LLVM assembler could still generate the machine codes for these instructions. For the Coprocessor 2 instructions, Load Word from Coprocessor 2 (*lwc2*) is not implemented for RISC32. This will not affect the RISC32 code generation as well (non-mappable from C), but shall be considered for future implementations. However, to ensure proper usage of the Coprocessor 2 in the user program, CP2 intrinsic functions were implemented for RISC32 sub-target and will be discussed in Section 3.1.6.

Table 3.8: Comparison for Miscellaneous Instructions

Instruction	Description	Supported by MIPS II?	Supported by RISC32?	Remarks
tge	Trap if Greater Than or Equal	✓	✕	Not supported by RISC32, but does not have valid C-syntax. Will not compile from C-code. Hence, no action is needed.
tgeu	Trap if Greater Than or Equal Unsigned	✓	✕	
tlt	Trap if Less Than	✓	✕	
tltu	Trap if Less Than Unsigned	✓	✕	
teq	Trap if Equal	✓	✕	
tne	Trap if Not Equal	✓	✕	
tgei	Trap if Greater Than or Equal Immediate	✓	✕	
tgeiu	Trap if Greater Than or Equal Unsigned Immediate	✓	✕	
tlti	Trap if Less Than Immediate	✓	✕	
tltiu	Trap if Less Than Unsigned Immediate	✓	✕	
teqi	Trap if Equal Immediate	✓	✕	
tnei	Trap if Not Equal Immediate	✓	✕	
ll	Load Linked Word	✓	✕	
sc	Store Conditional Word	✓	✕	
sync	Synchronize Shared Memory	✓	✕	

The Trap, Atomic Load Store and Serialization operations in MIPS II instruction set (Table 3.8) are currently unsupported in RISC32. The Trap operation is a software exception, which triggers the processor to enter kernel mode (exception handler) upon meeting a certain condition during user program execution. However, compilation of C code to Trap instructions are uncommon, as there are no known C-syntax that could be directly mapped to them. As for Atomic (*ll* and *sc*) and Serialization (*sync*) operation, these operations are commonly found in multiprocessor systems. The RISC32 is not a multiprocessor system, hence the user is not expected to implement multiprocessor application with RISC32. In short, there is no concern for these instructions to be compiled even if they are currently implemented in the MIPS Backend for MIPS II.

3.1.3 Implementing RISC32 as a Legal MIPS Sub-target in LLVM

To ensure the LLVM will correctly generate instructions that is compatible to RISC32 instruction set, the LLVM requires "RISC32" to be declared as one of the supported MIPS sub-targets in the MIPS Backend. The Figure 3.3 shows the associated file in declaring a new sub-target for MIPS Backend.

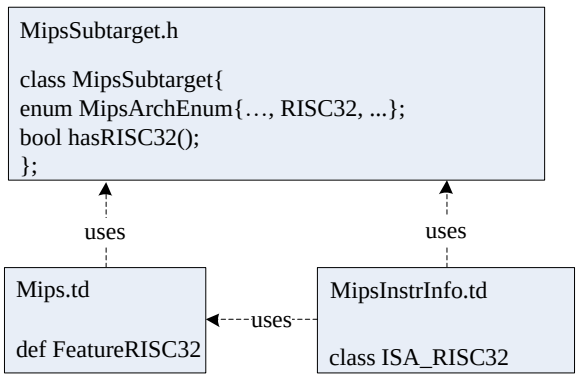


Figure 3.3: Files associated to declare new sub-target in MIPS Backend

The *MipsSubtarget.h* contains a *MipsSubtarget* class. This class contains information of MIPS sub-target supported in the MIPS Backend. The list of supported MIPS sub-target is declared in the class member, *MipsArchEnum* enumerator type. The enumerator assigns integer constant to each of the MIPS Architecture declared with respect to their order. In the previous Section 3.1.2, it has been discussed that, RISC32 supported instructions closely resembles MIPS II Instruction Set. Hence, RISC32 will inherit all of the MIPS II instruction, by introducing a new enumerator element after MIPS II. The declaration is shown in the Figure 3.4.

```
enum MipsArchEnum {
  MipsDefault,
  Mips1, Mips2, RISC32, Mips32, Mips32r2, Mips32r3, Mips32r5, Mips32r6, Mips32Max,
  Mips3, Mips4, Mips5, Mips64, Mips64r2, Mips64r3, Mips64r5, Mips64r6
};
```

Figure 3.4: RISC32 declared after MIPS II in the enumerator *MipsArchEnum*

This enumerator type is used to declare as the class member, *MipsArchVersion* enumerator variable. The *MipsArchVersion* is used in predicate functions, to determine which MIPS Architecture is being requested for each code generation session. Hence, a new predicate function, *hasRISC32()* is created for this purpose as indicated in Figure 3.3.

The next associated file is *Mips.td* file. This file is the main target description (*.td*) file for MIPS Backend, which is also the main reference point for generation of MIPS Backend (also known as target machine library, *Mips.a*). It contains a list of features that is supported in the MIPS Backend. The feature here refers to MIPS Architecture, or advanced architecture instruction set extension such as Digital Signal Processor (DSP) and MIPS SIMD Architecture (MSA). The feature declared here will be used as command line arguments, to be passed in during invocation of compilation to enable or disable a certain feature. As such, a new feature, *FeatureRISC32* was declared to inherit all MIPS II feature.

Another associated file to introduce new sub-target is *MipsInstrInfo.td*. This *.td* file contains every instruction supported by every generation of the MIPS instruction set. This refers to information such as instruction

mnemonics, instruction encoding and pattern to be matched with the SelectionDAG (intermediate form for LLVM code generation). This file also contains instruction membership class, which is used to check if the instruction declared belongs to a particular MIPS Architecture. The Figure 3.5 shows a sample declaration of instructions in *MipsInstrInfo.td*.

```
def MUL    : MMRel, ArithLogicR<"mul", GPR32Opnd, 1, II_MUL, mul>,
             ADD_FM<0x1c, 2>, ISA_MIPS32_NOT_32R6_64R6;

def ADD    : MMRel, StdMMR6Rel, ArithLogicR<"add", GPR32Opnd, 1, II_ADD>,
             ADD_FM<0, 0x20>, ISA_MIPS1;
```

Figure 3.5: Instruction declaration in *MipsInstrInfo.td*

The sample (Figure 3.5) shows the declaration for Multiply Word (*mul*) and Add Word (*add*) instructions. They are shown at the end of each declarations, which are also assigned with an instruction membership class. The *ISA_MIPS32_NOT_32R6_64R6* indicates that the *mul* instruction is introduced in MIPS32 instruction set but removed from both MIPS32 Revision 6 and MIPS64 Revision 6. Similarly, the instruction membership class, *ISA_MIPS1* indicates that the *add* instruction belongs to MIPS I instruction set. The *ISA_MIPS1* also indicates it is available for every MIPS generation, since every subsequent MIPS generation inherits instructions from its previous generation, unless specified, like the case of *mul* instruction. Hence, a new instruction membership class, *ISA_RISC32* was declared. The new instruction membership class can change the instruction availability that is not found in MIPS II but in later generation (*eret*) and to identify new instruction that is exclusive to RISC32 to be introduced in future.

3.1.4 Porting Shift-by-Variable Instructions from MIPS II to RISC32

As presented in the Table 3.5 in earlier section (Section 3.1.2), all of the bitwise Shift-by-Variable instructions in MIPS II instruction set is not supported in RISC32. The Shift-by-Variable instructions is generated by the C code construct in the Figure 3.6.

```
int a = 337;
volatile char b = 6;
unsigned int c = 337;

//Shift-Left-Logical-Variable
a = a << b;

//Shift-Right-Arithmetic-Variable
a = a >> b;

//Shift-Right-Logical-Variable
c = c >> b;
```

Figure 3.6: C code construct for Shift-by-Variable Instructions

The Shifting-by-Variable is generated when a shift operator (<< or >>) is used between variable declared. Shift-Left-Logical-Variable is generated when a left shift operator (<<) and the shift amount is based on a variable declared (variable '*b*' in Figure 3.6). For Shift-Right-Logical-Variable and Shift-Right-Arithmetic-Variable however, their generation is determined by the variable type declared as shown in Figure 3.6. If a variable is declared as *unsigned* variable type, a Shift-Right-Logical Variable will be compiled. Otherwise, a Shift-Right-Arithmetic Variable will be compiled. The variable *b* is declared as *volatile* type, to prevent compiler perform optimization on the shift operations so that Shift-by-Variable instructions can be generated for testing purpose. Without the *volatile* keyword, the compiler will directly generate Shift-by-Immediate instructions with the known shift value.

The Shift-by-Variable instruction syntax are shown in the Table 3.9.

Table 3.9: Shift-by-Variable Instruction Syntax

Instruction Syntax	Description
sllv $rd, $rt, $rs	Shift Left Logical Variable
srlv $rd, $rt, $rs	Shift Right Logical Variable
srav $rd, $rt, $rs	Shift Right Arithmetic Variable

The Shift-by-Variable instructions performs shifting on the operand value in target register *$rt*, based on the shift value specified in source register *$rs*. The shifted result is then updated to destination register, *$rd*. These instructions have the same behaviour as the normal Shift-by-Immediate (*sll, srl and sra*), except the shift value is obtained from register file, instead of encoded in the instruction as immediate value. This also indicates Shift-by-Variable instructions could not be compiled to the normal shift instructions directly, as the shift value might be unknown before program runtime. Hence, to obtain the same effect of Shift-by-Variable using Shift-by-Immediate instructions, the pseudocode in the Algorithm 3.1 is proposed.

ALGORITHM 3.1: SHIFT-BY-VARIABLE TO SHIFT-BY-IMMEDIATE PSEUDO-CODE

Input: Shift-by-Variable Instruction
Output: Expanded routine replaced with Shift-by-Immediate Instruction

1. Read shift-value from Shift-by-Variable source register
2. Read input operand to be shifted from Shift-by-Variable target register
3. while *shift-value != 0* do
    a. Shift 1 bit on input operand using Shift-by-Immediate
    b. Subtract shift-value by 1
4. endwhile
5. Store shift result to destination register of Shift-by-Variable

Algorithm 3.1: Pseudo-code for Shift-by-Variable transformation

To insert the pseudo-code (Algorithm 3.1) presented, understanding of the code generation in LLVM Backend is required. The Figure 3.7 presents a simplified view of code generation in LLVM Backend.

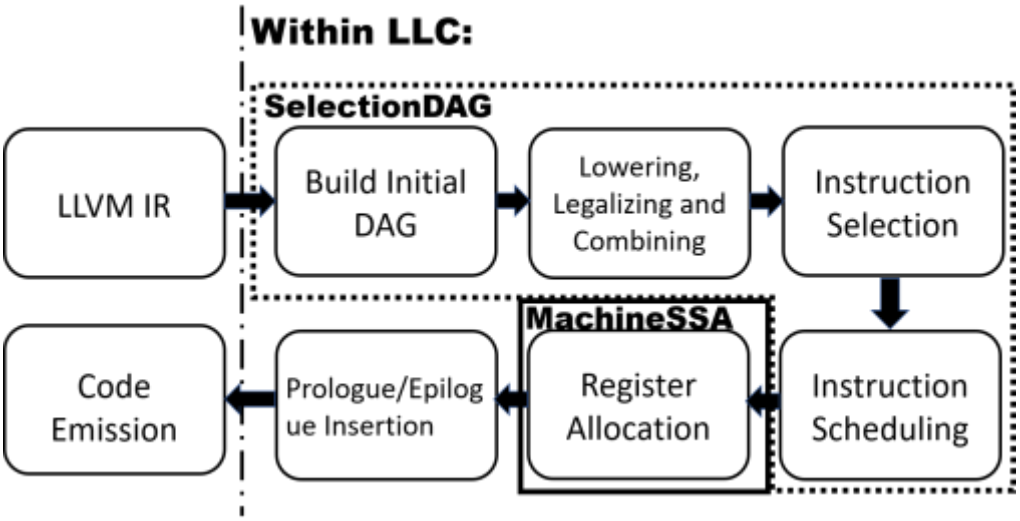


Figure 3.7: Simplified view of code generation in LLVM Backend

The code generation in LLVM relies on its static compiler, *llc* as illustrated in the Figure 3.7. The initial input to *llc* is LLVM Intermediate Representation (IR). LLVM IR is a generic assembly-like language, and is translated from the input source code (Eg: C, C++) by the LLVM frontend. *llc* will then construct an initial Selection Directed-Acyclic-Graph (DAG). The SelectionDAG is a graph-like data structure, where each graph nodes represents a pattern that is mapped from the LLVM IR. The initial SelectionDAG is generic, and goes through Lowering Phase and Legalizing Phase to be transformed into a more target dependent SelectionDAG. The target dependent SelectionDAG will then go through Instruction Selection phase, where each graph node is mapped to an instruction with a matching

pattern. The instructions supported and its respective pattern is specified in the *InstructionInfo.td* files of the respective target machine. Once each graph node is replaced with a matching instruction, the SelectionDAG goes through Instruction Scheduling and is further transformed into Machine Single Static Assignment (SSA) form. The MachineSSA is the final intermediate form in LLVM, where each graph node in SelectionDAG is assigned with an order, and emitted as machine instructions. In MachineSSA form however, the compiler assumes the target machine supports infinite virtual registers. This MachineSSA will then go through Register Allocation Phase, to replace all the virtual registers with limited registers (specified in respective target *RegisterInfo.td* file) in the target machine. Following by, the Prologue and Epilogue Phase will insert starting and ending routine such as stack allocation to the generated machine instructions. Finally, the code emission will generate the desired output by user, either in assembly instruction or binary format.

By observing the process of code generation, the possible way to transform the Shift-by-Variable into Shift-by-Immediate instructions is after Instruction Selection Phase. This is because, during Lowering and Legalizing Phase, the SelectionDAG is still undergoing transformation to be more target dependent. Depending on the transformation and optimization in the phases, the graph node may or may not yield a matching pattern for Shift-by-Variable. Hence, only after the Instruction Selection Phase, a Shift-by-Variable will be mapped to the graph node and determined to need transformation to Shift-by-Immediate. However, after the Instruction Selection Phase, the intermediate representation form is still in SelectionDAG, where no order established

between each graph node. Implementing the Shift-by-Variable transformation while it is still in SelectionDAG may not be able to reflect execution order as required in the pseudocode proposed. Order of the instruction is only established when it is transformed into the MachineSSA form.

This research work utilizes a transformation pass, *processFunctionAfterISel*(), to implement the transformation of Shift-by-Variable to Shift-by Immediate. This transformation pass is an existing implementation in the MIPS Backend, and is placed after Instruction Scheduling Phase but before Register Allocation Phase. Hence, the *processFunctionAfterISel*() processes on the MachineSSA form. However, it is not straightforward to implement the pseudo-code proposed (Algorithm 3.1) in MachineSSA form. A basic understanding is required on a few terminologies such as MachineFunction (MF), MachineBasicBlock (MBB) and Machine Instruction (MI).

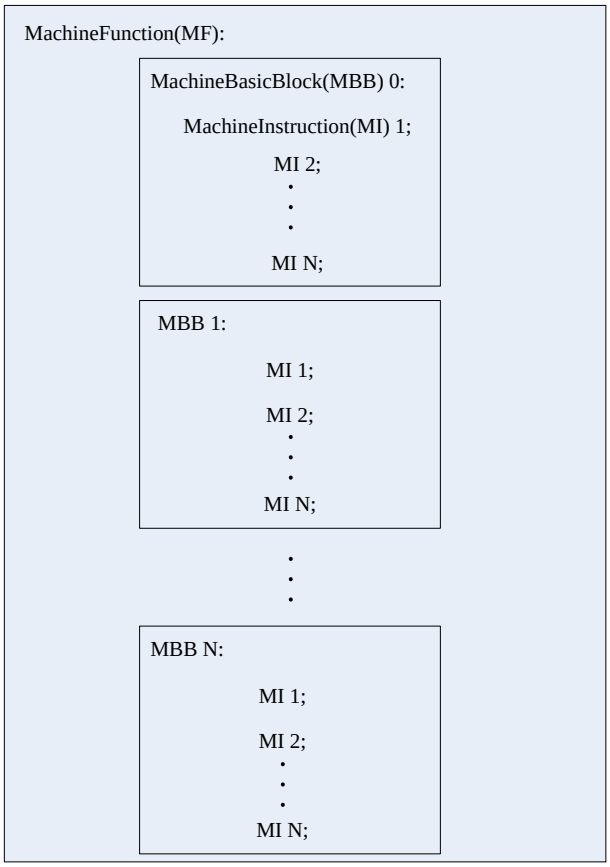


Figure 3.8: Relationship between MF, MBB and MI

In the LLVM Backend, when SelectionDAG is transformed into MachineSSA form, it is expressed in the structure as shown in Figure 3.8. Each user input source code file is interpreted as a MachineFunction (MF). Within each MF, it is made up of several MachineBasicBlock(MBB), where each MBB consists a group of MachineInstructions (MI). The size (number of MI) of each MBB is determined by common control flow structure (loops or if-else statements), function calls or program labels. New MBB is always spawned upon meeting branching instructions or program labels. The MI here refers to the generated target machine assembly instruction. They are expressed in SSA form, hence the name MachineSSA. These MI are translated

from SelectionDAG that has undergone Instruction Selection and Instruction Scheduling Phase.

As mentioned earlier, this research work utilize the transformation pass, *processFunctionAfterISel*() to perform the expand the Shift-by-Variable instruction into a series of instructions accompanied by Shift-by-Immediate instruction. The *processFunctionAfterISel*() takes in MBB(s) and scans through every MI present in it. Upon Shift-by-Variable instructions detected, and compilation for RISC32 Sub-target (using predicate function in Section 3.1.3) is detected, the following routine in Table 3.10 is expected with *srlv* as example.

Table 3.10: Expected Routine for *srlv*

Original Instruction	Expanded Instruction
srlv $1,$2,$3	*andi $3, $3, 0x1f* *srlv: beq $3, $0, end* *sub $3, $3, 1* *srl $2, $2, 1* *j srlv* *end: addu $1, $2, $0*

However, *processFunctionAfterISel*() processes MBB(s) in MachineSSA form. Table 3.11 shows the MachineSSA routine to be inserted to generate the expanded instruction.

Table 3.11: MachineSSA form for *srlv* expansion routine

Original Instruction	Original Expanded Form	MachineSSA Form
srlv $1,$2,$3	*andi $3, $3, 0x1f* *srlv: beq $3, $0, end* *sub $3, $3, 1* *srl $2, $2, 1* *j srlv* *end: addu $1, $2, $0*	*andi %C1, %C, 0x1f* *cond: %B1 = PHI(%B , %B2)* *%C2 = PHI(%C1,%C3)* *beq %C2, $0, end* *srlv: sub %C3, %C2, 1* *srl %B2, %B1, 1* *j cond* *end: addu %A, %B1, $0*

The MachineSSA form illustrated in Table 3.11 is inserted using *BuildMI*(), an existing function implemented in the MI class of LLVM library. It is shown that, the MachineSSA form introduces various virtual registers (*%A, %B1, %C3*…) and new MBBs (*cond*, *srlv*, *end*). The virtual register *%A* is mapped to the destination register (*$1*), that stores the shifted result for *srlv* instruction. The virtual registers *%B, %B1,* and *%B2* are mapped to the target register (*$2*), which contains the input operand to be shifted by *srlv* instruction. Finally, the virtual registers *%C, %C1, %C2,* and *%C3* are mapped to source register (*$3*) of *srlv* instruction, which contains the shift amount. It should be noted that, none of the virtual register naming was repeated. This is to conform to the rules of SSA form, where each register can only be assigned once. Also, each register must have been assigned or declared earlier before their usage. For cases where repeated assignment to an SSA register is required, a special decision structure in SSA form, *PHI*() functions is used instead. This function allows repeated assignment to an SSA register in the cases where source of SSA register originates from several MBBs (MBB before *cond* and *srlv* MBB). These *PHI*() functions will be removed after the Register Allocation Phase.

The Table 3.12, Table 3.13 and Table 3.14 shows the changes in generated assembly code for the respective Shift-by-Variable instructions after the implementation of RISC32 Sub-target. These tables are the compilation output with respect to the C code in Figure 3.6. It can be seen that each Shift-by-Variable instruction has been replaced with their Shift-by-Immediate variant instructions according to the proposed pseudo-code in Algorithm 3.1. The proposed routine shifts bit-by-bit and subtract the extracted shift amount after every loop. The shifting operation is performed until the shift amount equals to zero. The routine will then branch out, and transfer the final shifted value into the destination register as indicated in the original Shift-by-Variable instruction.

Table 3.12: Shift-Left-Logical-Variable compiled using RISC32 Sub-target

Before (Compiled using MIPS II)	After (Compiled using RISC32)

addiu $1, $zero, 337
sw $1, 4($fp)
addiu $1, $zero, 6
j $BB0_1
sb $1, 0($fp)
$BB0_1:
 	
lw $1, 4($fp)
lb $2, 0($fp)
sllv $1, $1, $2
	addiu $1, $zero, 337
	sw $1, 4($fp)
	addiu $1, $zero, 6
	sb $1, 0($fp)
	j $BB0_1
	$BB0_1:
	
	lw $2, 4($fp)
	lb $1, 0($fp)
	andi $3, $1, 31
	$BB0_2:
	
	beqz $3, $BB0_5
	# BB#3:
	addi $3, $3, -1
	sll $2, $2, 1
	# BB#4:
	j $BB0_2
	$BB0_5:
	addu $1, $zero, $2

Table 3.13: Shift-Right-Arithmetic-Variable compiled using RISC32 Sub-target

Before (Compiled using MIPS II)	After (Compiled using RISC32)

addiu $1, $zero, 337
sw $1, 4($fp)
addiu $1, $zero, 6
j $BB0_1
sb $1, 0($fp)
$BB0_1:
 	
lw $1, 4($fp)
lb $2, 0($fp)
srav $1, $1, $2
	addiu $1, $zero, 337
	sw $1, 4($fp)
	addiu $1, $zero, 6
	sb $1, 0($fp)
	j $BB0_1
	$BB0_1:
	
	lw $2, 4($fp)
	lb $1, 0($fp)
	andi $3, $1, 31
	$BB0_2:
	
	beqz $3, $BB0_5
	# BB#3:
	addi $3, $3, -1
	sra $2, $2, 1
	# BB#4:
	j $BB0_2
	$BB0_5:
	addu $1, $zero, $2

Table 3.14: Shift-Right-Logical-Variable compiled using RISC32 Sub-target

Before (Compiled using MIPS II)	After (Compiled using RISC32)

addiu $1, $zero, 337
sw $1, 4($fp)
addiu $1, $zero, 6
j $BB0_1
sb $1, 0($fp)
$BB0_1:
 	
lw $1, 4($fp)
lb $2, 0($fp)
srlv $1, $1, $2
	addiu $1, $zero, 337
	sw $1, 4($fp)
	addiu $1, $zero, 6
	sb $1, 0($fp)
	j $BB0_1
	$BB0_1:
	
	lw $2, 4($fp)
	lb $1, 0($fp)
	andi $3, $1, 31
	$BB0_2:
	
	beqz $3, $BB0_5
	# BB#3:
	addi $3, $3, -1
	srl $2, $2, 1
	# BB#4:
	j $BB0_2
	$BB0_5:
	addu $1, $zero, $2

3.1.5 Porting Branch on Conditional Instructions from MIPS II to RISC32

As discussed in Section 3.1.2, the RISC32 will only require implementation to handle compilation for Branch on Less Than Zero (*bltz*) and Branch on Greater Than or Equal to Zero (*bgez*). Other branch variant (Conditional Branch Likely, Long Branch and Long Branch Likely) does not requires handling, as compilation from C code to these instructions was not possible with their existing implementation in MIPS LLVM Backend. The Figure 3.9 and Figure 3.10 shows their respective C construct.

```
char a = 55;
char b = 13;
volatile char c = -8;

//bgez
if(c < 0){
    c = a+b;
}
```

Figure 3.9: C construct for *bgez* instruction

```
char a = 13;
char b = 55;
volatile char c = 87;

//bltz
if(c >= 0){
    b = c - a;
}
```

Figure 3.10: C construct for *bltz* instruction

It is shown in both Figure 3.9 and Figure 3.10 that the condition of the if-construct determines the compiled Conditional Branch instruction. However, the Conditional Branch instruction is always compiled with the

reversed condition with respect to their C code. For instance, the "*c < 0"* in Figure 3.9, while interpreted as branch on *c* less than zero, it is expected to be compiled to *bgez.* The compilation of reverse branching condition ensures the execution order of the code as interpreted from their C language semantics. If a reverse condition is detected, it will branch away, to a further point of the program. Otherwise, the consecutive code segment right after the reverse branching instruction is executed, which fulfils the execution condition in their C language semantics.

Table 3.15: Instruction syntax for *bltz* and *bgez*

Instruction Syntax	Description
bltz $rs, offset	Branch on Less Than Zero
bgez $rs, offset	Branch on Greater Than or Equal to Zero

The Table 3.15 illustrates the instruction syntax for both *bltz* and *bgez* instruction. Both instructions perform comparison of source register (*$rs*) against register zero (*$0*), and branch to the 16-bit offset if branching condition is matched. They can both be replaced with a combination of *beq, bgtz* and *j* instructions. The expected routine for both *bltz* and *bgez* is shown in the Table 3.16.

Table 3.16: Expected Routine and equivalent MachineSSA form for *bltz* and *bgez*

Original Instruction	Expanded Instruction	MachineSSA form
bltz $rs, br_label	*bltz: bgtz $rs, exit* *beq $rs, $zero, exit* *j br_label* *exit:*	*bltz: bgtz %rs, exit* *beq %rs, $zero, exit* *j br_label* *exit:*
bgez $rs, br_label	*bgez: bgtz $rs, br_label* *beq, $rs, $zero, br_label* *exit:*	*bgez: bgtz %rs, br_label* *beq, %rs, $zero, br_label* *exit:*

The labels (*exit* and *br_label*) shown in Table 3.16 corresponds to the 16-bit branch offset. These labels are replaced with calculated PC-relative offset during the Linking Phase. The *bltz* performs branching upon source register (*$rs*) less than register zero (*$0*). It can be interpreted as: the *$rs* must not be greater than zero and must not equal to zero as well. Hence, the proposed expanded routine shown for *bltz* (1st row of Table 3.16) achieve *$rs* must not be greater than zero by branching away to the *exit* label using *bgtz*. The *beq* branching away to *exit* label achieves *$rs* must not be equal to zero. If both conditions (*bgtz* and *beq*) are not met, the final unconditional branch (*j*) will take place and branch to the intended *bltz* offset. The *bgez* performs branching upon *$rs* greater or equal to *$0*. This can be interpreted as: the *$rs* is either greater than zero or equal to zero. The proposed expansion routine for *bgez* (2nd row of Table 3.16) achieve *$rs* greater than zero using *bgtz* instruction and branch to the intended branch offset. *beq* is used to achieve *$rs* equals to zero. Subsequent instructions will be executed if neither *bgtz* nor *beq* is taken, which fulfils the branch not taken condition for *bgez*.

Similar to Shift-by-Variable transformation, the *bltz* and *bgez* will only be transformed at the *processFunctionAfterISel*() transformation pass. The pass will iterate over every MI of MBB(s) passed in to find out every *bltz* and *bgez* instruction. If RISC32 Sub-target is requested, every instance of both instructions will be replaced with the equivalent MachineSSA form of the proposed routine (Table 3.16). It should be noted as well, the proposed MachineSSA form does not require complicated virtual register renaming as

presented in Section 3.1.4, as there was no assignment statement for branching.

The Table 3.17 and Table 3.18 shows the compilation of *bgez* (Figure 3.9) and *bltz* (Figure 3.10) for RISC32 as suggested in Table 3.16. It should be noted that, for the compilation before RISC32 is implemented in LLVM, both conditional branches (*beq, bgtz, bgez*, etc…) and unconditional branches (*j, jr, jal*, etc…) will have a NOP inserted right after them. This is to fill the delay slots of the branching instructions, to prevent the subsequent instruction after from executing. For RISC32 however, the implementation of Branch Predictor (BP) eliminates the need for delay slots. Hence, no NOP insertion is required for RISC32 compilation. It should be noted as well, the proposed solution in Table 3.16 uses *beq* to compare the source register with the *$zero* register. The illustrated compilation however, uses *beqz* instruction instead. This does not affect the behaviour of the code, as *beqz* is the alias instruction for *beq* instructions that compares to *$zero* register. They both have the same instruction encoding and behaves in the same manner.

Table 3.17: Branch on Greater or Equal to Zero compiled using RISC32 Sub-target

Before (Compiled using MIPS II)	After (Compiled using RISC32)

addiu $1, $zero, 55
sb $1, 12($fp)
addiu $1, $zero, 13
sb $1, 8($fp)
addiu $1, $zero, 248
sb $1, 4($fp)
lb $1, 4($fp)
bgez $1, $BB0_3
nop
# BB#1: 	
j $BB0_2
nop
$BB0_2:
lbu $1, 12($fp)
lbu $2, 8($fp)
addu $1, $1, $2
j $BB0_3
sb $1, 4($fp)
$BB0_3:
	addiu $1, $zero, 55
	sb $1, 12($fp)
	addiu $1, $zero, 13
	sb $1, 8($fp)
	addiu $1, $zero, 248
	sb $1, 4($fp)
	lb $2, 4($fp)
	bgtz $2, $BB0_4
	# BB#1:
	beqz $2, $BB0_4
	# BB#2:
	j $BB0_3
	$BB0_3:
	lbu $1, 12($fp)
	lbu $2, 8($fp)
	addu $1, $1, $2
	sb $1, 4($fp)
	j $BB0_4
	$BB0_4:

Table 3.18: Branch on Less Than Zero compiled using RISC32 Sub-target

Before (Compiled using MIPS II)	After (Compiled using RISC32)

addiu $1, $zero, 13
sb $1, 12($fp)
addiu $1, $zero, 55
sb $1, 8($fp)
addiu $1, $zero, 87
sb $1, 4($fp)
lb $1, 4($fp)
bltz $1, $BB0_3
nop
# BB#1: 	
j $BB0_2
nop
$BB0_2:
lbu $1, 4($fp)
lbu $2, 12($fp)
subu $1, $1, $2
j $BB0_3
sb $1, 8($fp)
$BB0_3:
	addiu $1, $zero, 13
	sb $1, 12($fp)
	addiu $1, $zero, 55
	sb $1, 8($fp)
	addiu $1, $zero, 87
	sb $1, 4($fp)
	lb $2, 4($fp)
	bgtz $2, $BB0_3
	# BB#1:
	beqz $2, $BB0_3
	# BB#2:
	j $BB0_5
	$BB0_3:
	j $BB0_4
	$BB0_4:
	lbu $1, 4($fp)
	lbu $2, 12($fp)
	subu $1, $1, $2
	sb $1, 8($fp)
	j $BB0_5
	$BB0_5:

3.1.6 Implementation of CP2 Intrinsic Functions in LLVM for RISC32

The CP2 is the AES Coprocessor integrated into RISC32 to provide encryption functionality. The introduction of CP2 introduces three instructions, namely the Move to Coprocessor 2 (*mtc2*), Move from Coprocessor 2 (*mfc2*) and Store Word from Coprocessor 2 (*swc2*). The CP2 integration into RISC32 and its instructions will be discussed under Section 3.2. In the previous Section 3.1.2, it has been discussed that the system instructions are typically non-mappable from C-syntax. This includes the CP2 instructions as well. They can only be accessed using inline assembly programming. The inline assembly programming feature is by default supported in LLVM, hence will not be discussed here. However, programming with inline assembly for CP2 use case is not recommended. This is to prevent user from creating a sequence of assembly code that is not compatible with the CP2 execution behavior. This research work introduces the implementation of new intrinsic function that maps to a specific sequence of assembly instructions which is compatible to the underlying CP2 hardware.

The intrinsic function is a special C function that maps to a specific sequence of instruction defined by the compiler. The frontend (clang) translates C intrinsic function and emits an equivalent LLVM IR intrinsic function call to *llc*. *llc* then transforms the LLVM IR and maps it to a specific sequence of instructions accordingly. The instruction sequence is specified in a custom function inserter implemented by respective target machine backend. Hence, implementation of new intrinsic function requires extension to both the frontend and backend as shown in the Figure 3.11.

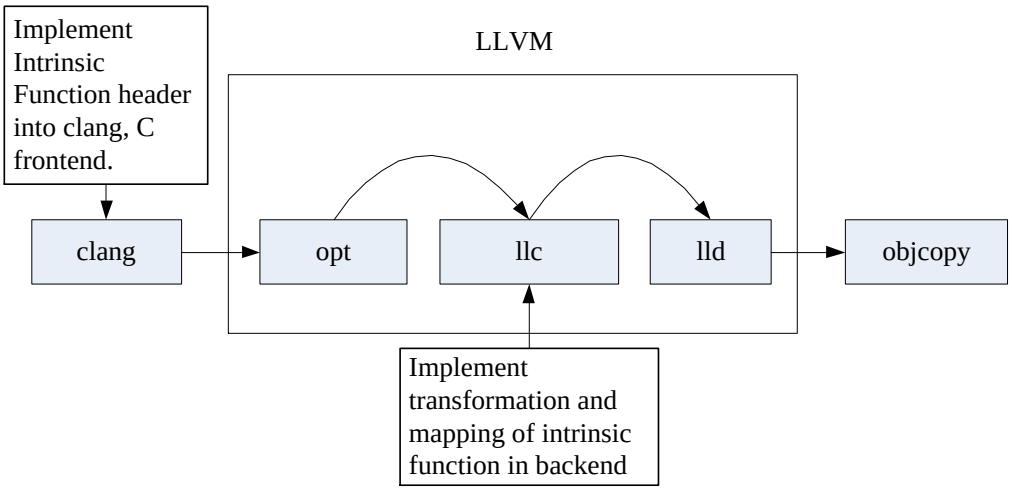


Figure 3.11: Overview of intrinsic porting in LLVM

At the frontend, clang, valid intrinsic function prototypes are declared in each target machines' respective *BuiltinsXX.def* file, where *XX* specifies the target machine. For example, *BuiltinsMips.def* is specific to MIPS target machine only. For RISC32, two new intrinsic function was implemented. The new intrinsic function header is shown in Table 3.19.

Table 3.19: CP2 Intrinsic Function Header

CP2 Key Expansion Intrinsic Function Header: *__builtin_risc32_aes128_keyinit (uint32_t* key)*	(1)
CP2 Encryption Intrinsic Function Header: *__builtin_risc32_aes128_enc (uint32_t*plaintext, volatile uint32_t* ciphertext)*	(2)

Function (1) is for invoking key expansion. Input argument required is the base address (*uint32_t* key*) for the 128-bit secret key. As there is no 128-bit data type in C, the secret key is split into array of four words (32-bit each),

which is represented using integer data type in C. This intrinsic function is designed to load secret key from RISC32 data memory and pass it to CP2 for key expansion. The expanded round keys will be stored in CP2 and used during encryption call.

Function (2) is for plaintext encryption. Two arguments are required: base address (*uint32_t * plaintext*) for the 128-bit plaintext and base address (*volatile uint32_t* ciphertext*) to store 128-bit ciphertext. Similarly, both plaintext and ciphertext are stored in array of four words. This intrinsic function is designed to read plaintext from the base address (*plaintext*) and pass it to the CP2 for encryption. Once the encryption completes, ciphertext from CP2 will be stored into the memory location specified by the base address, *ciphertext*.

After declaring the new function prototype, its code generation implementation is needed. Through code generation, the function argument from the source code will be extracted to construct a proper function call to be mapped to LLVM IR. These implementations are done under target specific intrinsic function expression emitter method which is labeled as *EmitXXBuiltinExpr*(), where *XX* specifies the target machine. They all are implemented under the code generation of clang frontend, *CGBuiltin.cpp* file. For RISC32, the *EmitMipsBuitlinExpr*(), is used instead. This function will detect the intrinsic function header (1) and (2) that has been defined in *BuiltinsMips.def* and emit equivalent LLVM IR intrinsic function call.

With the code generation for the newly implemented intrinsic function, its equivalent LLVM IR pattern is needed. This LLVM IR pattern is required for initial construction of SelectionDAG in *llc*, which will then further be transformed by target specific transformation routine to yield a target specific SelectionDAG. The new LLVM IR pattern is implemented under the target specific intrinsic *.td* file, for example *IntrinsicMips.td*. The frontend will utilize the record of the new pattern to map the function call construct from code generation phase to its equivalent intrinsic function LLVM IR construct. For CP2 intrinsic function (1) and (2), it will be emitted as the LLVM IR as illustrated in Table 3.20

Table 3.20: LLVM IR for CP2 Intrinsic Function

CP2 Key Expansion Intrinsic LLVM IR: *void @llvm.mips.__builtin_risc32_aes128_keyinit(i32*)*	(1)
CP2 Encryption Intrinsic LLVM IR: *void @llvm.mips.__builtin_risc32_aes128_enc(i32*, i32*)*	(2)

With the LLVM IR in Table 3.20, an equivalent Instruction SelectionDAG (ISD) node will be generated during Lowering Phase. This ISD node will be mapped using pattern matching by recognizing the ISD input pattern during Instruction Selection Phase of *llc*. However, since the intrinsic functions are not exactly an instruction on its own, it is implemented as pseudo instruction node.

```
class AES128_KEYINIT_DESC:
  PseudoSE<(outs), (ins PtrRC:$key_ptr),
           [(int_risc32_aes128_keyinit iPTR:$key_ptr)]>;

class AES128_ENC_DESC:
  PseudoSE<(outs), (ins PtrRC:$data_ptr, PtrRC:$cipher_ptr),
           [(int_risc32_aes128_enc iPTR:$data_ptr, iPTR:$cipher_ptr)]>;

let usesCustomInserter = 1 in {
  def AES128_KEYINIT         : AES128_KEYINIT_DESC;
  def AES128_ENC             : AES128_ENC_DESC;
}
```

Figure 3.12: CP2 Intrinsic Function pseudo instruction node in *MipsInstrInfo.td*

The pseudo instructions nodes declared (*AES128_KEYINIT* and *AES128_ENC*) shown in Figure 3.12 utilizes the *usesCustomInserter* flag. With the flag set, a Custom Inserter Function, *EmitInstrWithCustomInserter*() is call upon during Expand Instruction Selection Pseudo Pass. This pass is responsible to expand pseudo instructions generated after Instruction Selection Phase. Within the Custom Inserter Function, all pseudo instructions are specifically translated to a suitable instruction, or in the cases of intrinsic function, it will be mapped to a specific sequence of instructions.

Table 3.21: CP2 Key Expansion Routine

Instruction	Comment
lw $rt0, 0($rs)	Load 128-bit secret key (four words) from memory
lw $rt1, 4($rs)	
lw $rt2, 8($rs)	
lw $rt3, 12($rs)	
mtc2 $rt0, $0	Move loaded 128-bit secret key (four words) into secret key register ($0~$3) of CP2
mtc2 $rt1, $1	
mtc2 $rt2, $2	
mtc2 $rt3, $3	
addi $rt4, $zero,0x1	Prepare key expansion command
mtc2 $rt4, $12	Move key expansion command to command register ($12) of CP2 and start key expansion
nop	Insert *NOPS* to wait for key expansion to complete. Key expansion requires 15 clock cycles to complete, hence 15 *NOPS* is inserted.
nop	
..	

Table 3.22: CP2 Encryption Routine

Instruction	Comment
lw $rt0, 0($rs)	Load 128-bit plaintext (four words) from memory
lw $rt1, 4($rs)	
lw $rt2, 8($rs)	
lw $rt3, 12($rs)	
mtc2 $rt0, $4	Move loaded 128-bit plaintext in terms of (four words) into secret key register (*$0~$3*) of CP2
mtc2 $rt1, $5	
mtc2 $rt2, $6	
mtc2 $rt3, $7	
addi $rt4, $zero,0x2	Prepare encrypt plaintext command
mtc2 $rt4, $12	Move key expansion command to command register (*$12*) of CP2 and start encryption
nop	Insert *NOPS* to wait for encryption to complete. CP2 encryption requires 55 clock cycles to complete, hence 55 *NOPS* is inserted.
nop	
..	
swc2 $8,0($rs)	Read encrypted 128-bit cipher text (four words) from cipher text register (*$8~$11*) of CP2
swc2 $9,4($rs)	
swc2 $10,8($rs)	
swc2 $11,12($rs)	

The Table 3.21 and Table 3.22 shows expected sequence of instructions generated for both CP2 key expansion intrinsic Function (1) and CP2 encryption intrinsic Function (2) respectively. For Table 3.21, 15 *NOPS* are inserted to wait for all round keys to be generated. The *NOPS* inserted will have insignificant impact on overall program execution performance, as the secret key is assumed to be the same for consecutive batch of data (*N* byte in Section 3.2.4) of plaintext. A new secret key is only required when the sensor node sends another *N* byte of data. Hence, key expansion should only be executed once for the same consecutive blocks of plaintext. As for Table 3.22, the 55 *NOPS* is only inserted if the Queue System (Section 3.2.6) for CP2 is not implemented. With Queue System implemented, the *NOPS* can be removed, which will result a shorter instruction sequence.

3.1.7 LLVM Compilation of Interrupt Service Routine (ISR) for RISC32

The Interrupt Service Routine (ISR) is a common programming feature for microcontroller and IoT sensor nodes. It is typically used to serve hardware I/O interrupts, where I/O transaction between the sensor node and outer world is usually slow and could happen anytime. With ISR programming support, the processor does not need to constantly poll for an I/O action to happen and proceed with another task. The processor could even go into sleep mode when no operation is required, to preserve its limited energy supply, and waken again when interrupt event is triggered.

In LLVM, the ISR programming feature is supported by default. The Table 3.23 shows an example of writing an ISR in C code using the LLVM C frontend, Clang.

Table 3.23: Sample ISR using Clang

<table>
<tr><th>C code</th><th>Compiled Assembly</th><th>Comment</th></tr>
<tr><td rowspan="2">unsigned char a = 0;

__attribute((interrupt))
void isr0()
{
a++;
}</td><td>mfc0 $26, $13, 0
mfc0 $27, $14, 0
sw $27, 4($sp)
mfc0 $27, $12, 0
ext $26, $26, 10, 6
sw $27, 0($sp)
ins $27, $26, 10, 6
ins $27, $zero, 1, 4
ins $27, $zero, 29, 1
mtc0 $27, $12, 0</td><td>Exception handler Prologue in LLVM</td></tr>
<tr><td>addiu $sp, $sp, -16
sw $2, 12($sp)
sw $1, 8($sp)
lui $1, %hi(a)
lbu $2, %lo(a)($1)
addiu $2, $2, 1
sb $2, %lo(a)($1)
lw $1, 8($sp)
lw $2, 12($sp)</td><td>a++;</td></tr>
</table>

Continued from Table 3.23

	di *ehb* *lw* *$27, 4($sp)* *mtc0* *$27, $14, 0* *lw* *$27, 0($sp)* *mtc0* *$27, $12, 0* *addiu* *$sp, $sp, 16* *eret*	Exception Handler Epilogue in LLVM Exception Return

Each ISR is identified by Clang specific C programming attribute feature, *__attribute((interrupt))*. With the *interrupt* attribute, function body implemented under it will be compiled with Exception Handler Prologue, Epilogue and Exception Return (*eret*). The Exception Handler Prologue reads, process and records the Coprocessor 0 (CP0) registers (*Cause*, *Status* and *EPC*) before proceeding to carry out the ISR function body. The Exception Handler Epilogue updates the CP0 registers after serving the ISR. Exception Return allows the kernel mode (Exception Handling) to return to the user program. Due to this convention, each ISR currently will have a separate instance of Exception Handler Prologue and Epilogue compiled together. This also indicates that each ISR is expected to have a separate program address. The hardware will need to keep track the entry address or have a fixed entry address for each ISR.

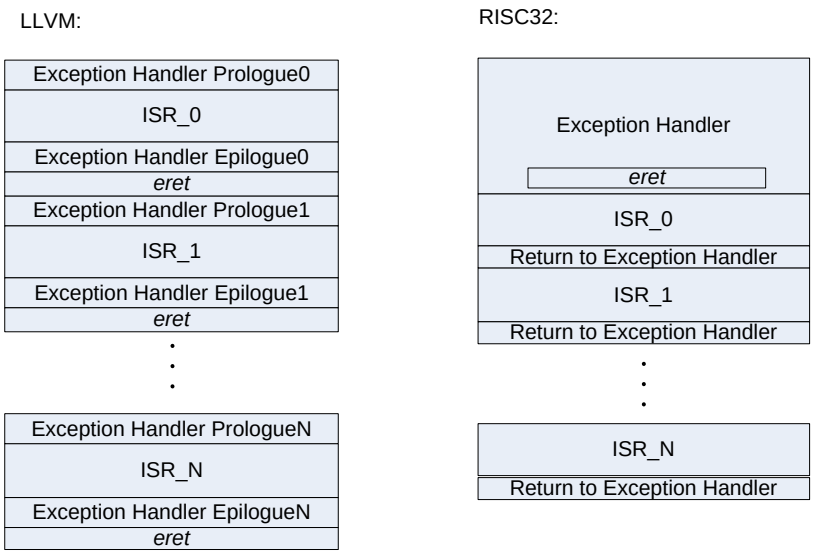


Figure 3.13: ISR Convention Comparison between LLVM and RISC32

In RISC32, only a single instance of Exception Handler will always be referenced and the starting address for it is always fixed at 0x8001_B400 (Section 3.1, Figure 3.2). The RISC32 Exception Handler will always read, record and process CP0 registers, perform checking for software exceptions or hardware interrupt sources, and then branch to their respective ISR. At the end of each ISR, it will branch back to the same Exception Handler again, to update CP0 registers and perform Exception Return (*eret*). Due to this convention in RISC32, the hardware does not need to keep track the entry address or have a fixed entry address for each ISR. However, the RISC32 Exception Handler will have to be compiled together with all the ISR. This is to resolve the label address of every ISR in the Exception Handler by utilizing the Linking Phase of code compilation.

Through the comparison in Figure 3.13, this research work is required to:

1) Eliminate the Exception Handler Prologue and Epilogue generation by LLVM if current ISR is generated for RISC32
2) Replace the *eret* at the end of each ISR with instruction to return to the RISC32 Exception Handler
3) Specify the MIPS register usage convention for RISC32 ISR

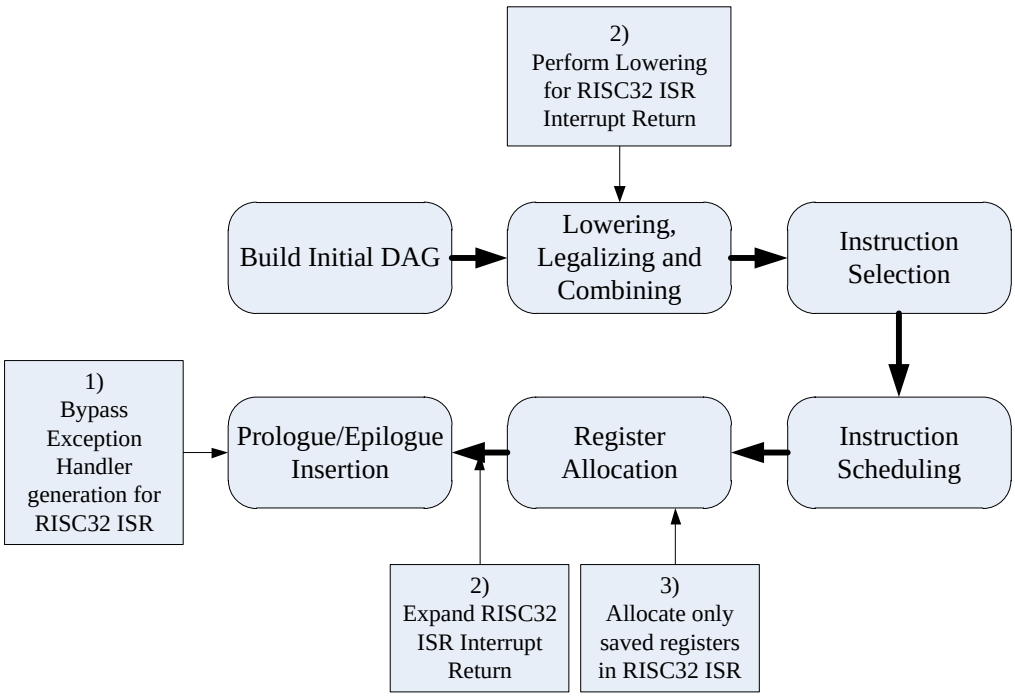


Figure 3.14: Overview of RISC32 ISR Porting

The Figure 3.14 presents the overview of implementation required to the code generation process of LLVM for RISC32 ISR Porting. To prevent the generation of Exception Handler Prologue and Epilogue of existing LLVM implementation for RISC32, modification to the Prologue and Epilogue

Insertion is required. The Prologue and Epilogue Phase of MIPS Backend is implemented under *MipsSEFrameLowering.cpp*. This phase is responsible to perform stack allocation, and insert special starting and ending code if the *interrupt* attribute is detected. By utilizing the RISC32 Sub-target predicate function (discussed in Section 3.1.3, Figure 3.3), whenever the *interrupt* attribute is detected, the generation of both Exception Handler Prologue and Epilogue will be skipped.

The *eret* as mentioned in Section 3.1.2, is categorized as System Instructions. Hence, this instruction cannot be compiled from C code directly. In the existing implementation, *eret* is implemented as a pseudo type MIPS Instruction SelectionDAG (ISD) graph node. This pseudo MIPS ISD is inserted during Lowering Phase as a return graph node when the *interrupt* attribute is detected. Unlike the implementation of CP2 intrinsic node in Section 3.1.6, the *eret* code generation does not takes place in the Custom Inserter method. It is only inserted during the Post Register Allocation (RA) Phase, by detecting the *eret* pseudo MIPS ISD and insert it as an MI. This research work will utilize the same approach, by first declaring a new interrupt return node for RISC32 and expand the new return node during Post RA Phase.

```
def MipsINTRet : SDNode<"MipsISD::INTRet", SDTNone,
                        [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

def INTRet : PseudoSE<(outs), (ins), [(MipsINTRet)]>;
```

Figure 3.15: MIPS Interrupt Return ISD Node declaration for RISC32 in *MipsInstrInfo.td*

The created pseudo *INTRet* MIPS ISD (Figure 3.15) node is inserted during the Lowering Phase, if the *interrupt* attribute and RISC32 Sub-target is detected. During Post RA Phase, this *INTRet* node will be replaced as *j POP_STACK*. The *POP_STACK* is a program label in the RISC32 Exception Handler (Figure 3.16). It is the return address when RISC32 Exception Handler branch to respective software or hardware ISR.

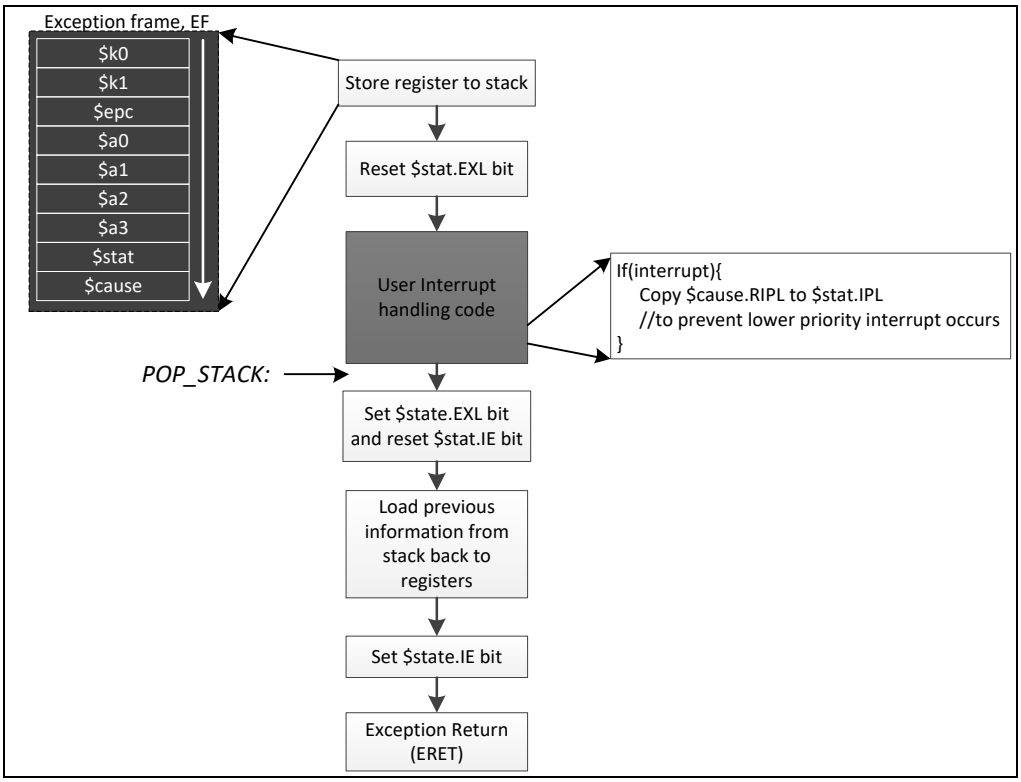


Figure 3.16: RISC32 Exception Handler Flow
Source: Kiat, 2018.

By referring to Figure 3.16, only a limited set of registers is saved (Exception Frame) upon entering RISC32 Exception Handler. To specify the limited register sets for ISR compilation, modification is made to the *MipsRegisterInfo.td*. This file specifies the register size, available register set and their usage convention. The information is used during Register Allocation Phase, where virtual registers of MachineSSA form is replaced with the supported register files of respective target machine.

```
def GPR32 : GPR32Class<[i32]>{
  let AltOrders = [(add (sub GPR32, GPR32), A0, A1, A2, A3, K0, K1)];
  let AltOrderSelect =[{
    return (MF.getFunction()->hasFnAttribute("interrupt")&& MF.getSubtarget<MipsSubtarget>().hasRISC32());
  }];
}
```

Figure 3.17: Alternate List of Allocable Register File for RISC32 ISR

The Figure 3.17 shows the register class definition for MIPS. This definition is for the default MIPS 32 General Purpose Register (GPR), which allows the 32 registers file allocable during Register Allocation Phase. Due to the construction of RISC32 Exception Handler, the Exception Frame (Figure 3.16) only saves the *$k0, $k1, $a0, $a1, $a2* and *$a3* register. Hence, the *AltOrders* specifies the available registers for RISC32 during the ISR compilation. The *AltOrderSelect* specifies the condition to be fulfilled in order to allocate the register file with the alternate list.

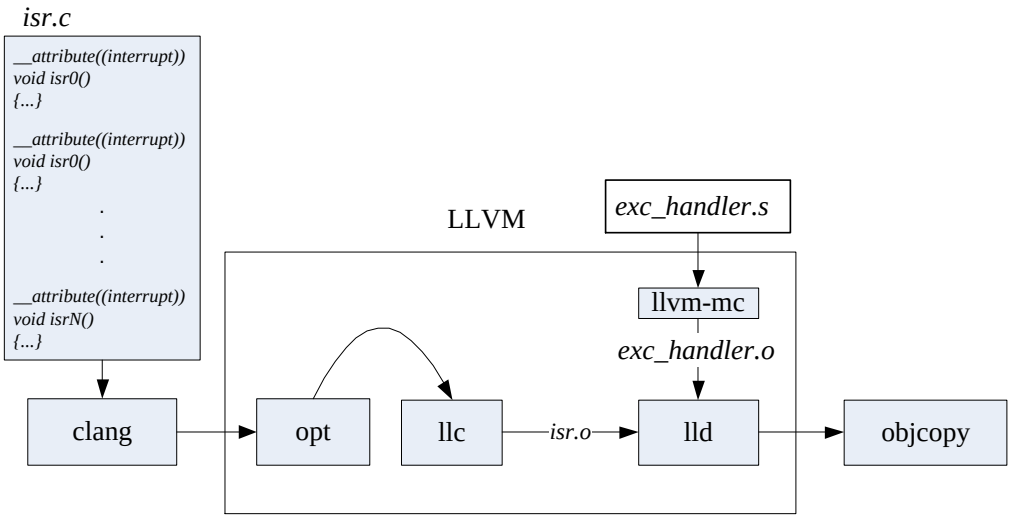


Figure 3.18: Compilation of RISC32 ISR using LLVM

Finally, to compile the RISC32 ISR, the ISR programming will utilize the *interrupt* attribute as illustrated in Figure 3.18. The ISR C source code (*isr.c*) will be translated and compiled until it yields an object file, *isr.o* by the *llc*. The RISC32 Exception Handler (*exc_handler.s*) is handcrafted using RISC32 assembly language, and assembled using the *llvm-mc*, a machine code assembler in LLVM, to yield *exc_handler.o* object file. This *llvm-mc* is also part of the *llc*, and is responsible to generate the *isr.o* object file as well. With both *exc_handler.o* and *isr.o* obtained, they will be passed into the LLVM linker, *lld* to resolve all ISR label address in the RISC32 Exception Handler. The *lld* is also responsible to link the object files (*exc_handler.o* and *isr.o)* and generate the final output in the form as illustrated in Figure 3.13 earlier. This final output will be an executable, ready to be flashed into RISC32 memory at the address 0x8001_B400, and executed whenever software or hardware exception is raised in RISC32.

3.2 System Overview: Hardware

This section will describe the proposed hardware design for this research work. This research work aims to provide confidentiality feature in the IoT processor, RISC32. The confidentiality here refers to providing secrecy to the data being transferred. This can be achieved by encrypting the data before it is transferred out of the processor. As such, the Advanced Encryption Standard (AES) encryption core is selected to be integrated as cryptographic coprocessor into RISC32. However, there are several techniques to integrate a new coprocessor into a host processor. The techniques can be categorized as dedicated path integration and share memory path integration. In this research work, dedicated path is selected to integrate the new AES core in order to ensure a better performance of the integrated AES core. In contrast, if the shared memory path technique is being used, performance of the AES core will be dependent on the traffic condition in shared transfer path. Since the integration technique selected will introduce a new transfer path for the AES core, new special instructions are required to activate and utilize this new transfer path.

While the AES core implemented in hardware performs encryption faster than its software implementation, the AES core is expected to have a processing latency. The processing latency here refers to time taken for the AES core to complete its encryption of a single input (128 bits or 16 bytes in equivalent). For all block cipher algorithms, the input data (plaintext) goes through a series of transformation iteratively to obtain an encrypted output (ciphertext). The processing latency is caused by the number of iterations

required. Common approach used to compensate the processing latency is by implementing AES architecture with higher throughput. However, architecture with higher throughput usually involves pipelined architecture. These pipelined architectures will still have the processing latency issue, but it allowed more encryptions to be performed at the same time. This may improve the encryption core performance, with the trade-off of larger hardware.

Since IoT sensor nodes are operating on battery power supply, it may not be a good idea to implement a power-hungry encryption hardware. As such, instead of having high throughput AES core, single stage AES core is selected for this research work. Further software analysis was made on a typical IoT application processing pattern to seek potential solutions to compensate the processing latency issue present in the AES core. The analysis results in a solution, which is inspired by the Tomasulo Algorithm (Hennessy and Patterson, 2011). A Queue System was designed to perform dynamic scheduling, which is optimized for the processing pattern of the typical IoT application.

The following section (Section 3.2.1) will first discuss on how integration of the AES core will be realized in RISC32 pipeline. Following by, the design of new instruction (Section 3.2.2) and the AES core used in this research work (Section 3.2.3). Next, analysis is made on the software pattern (Section 3.2.4) to eventually derive the solution, Queue System. Finally, the design of Queue System is discussed (Section 3.2.6).

3.2.1 Placement of the AES Coprocessor

The selected technique to integrate the AES core is dedicated path. As such, a new transfer path is required for the AES core. However, the new transfer path requires careful design consideration to prevent disruption to the work load balance in the processor pipeline. In this research work, the target RISC32 IoT processor is a 5-stage pipeline processor. Figure 3.19 revisits the architecture of RISC32 as discussed in Chapter 2.

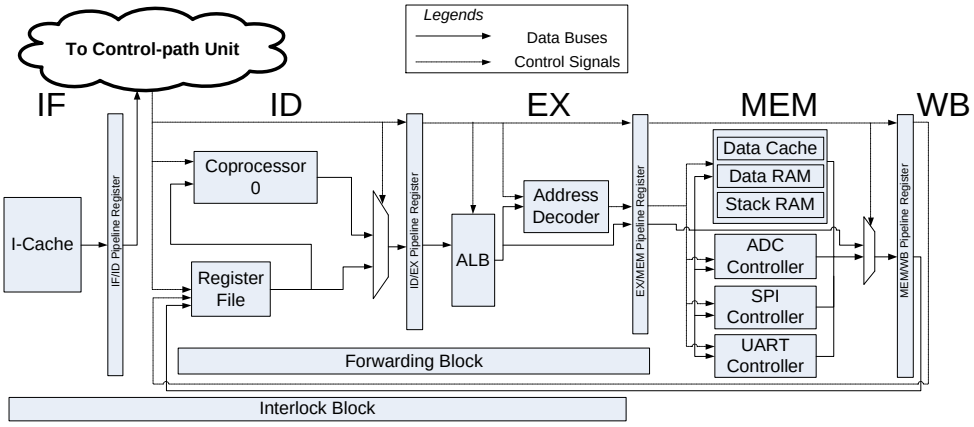


Figure 3.19: Simplified view of RISC32 microarchitecture revisited

The 5 stages of RISC32 pipeline consists of Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write Back (WB). The IF stage performs instruction fetching from the instruction memory. The instruction fetched is transferred into ID stage, where in ID stage, the instruction fetched is decoded. The decoding here refers to actions such as Control-Path Unit assessing instruction opcodes to generate control signals, operands is fetched from Register File, and immediate value extraction from instruction. The information generated is further processed in

the EX stage. At EX stage, the desired operation decoded from the instructions is carried out by the Arithmetic Logic Block (ALB). The Address Decoder generates the necessary control signals to activate the memory modules at MEM stage. At the MEM stage, load store operation is carried out. Data is either loaded from or stored into the requested data memory location (Data RAM, Data Cache or Stack RAM). Other Input/Output (I/O) modules integrated with the shared memory technique is also situated at MEM stage. If load store operation was not requested by the current instruction, the data from EX stage will resume its transfer to WB stage. Similarly, loaded data from the requested memory address also resumes its transfer to WB stage. The WB stage will update the Register File with the data it obtained from the MEM stage.

To determine the placement of the AES core, workload in each pipeline stage is analysed. The Table 3.24 shows the longest timing delay for each stage in RISC32.

Table 3.24: Longest Timing Delay for Each Stage in RISC32

Pipeline Stage	IF	ID	EX	MEM	WB
Timing delay	14.537ns	13.309ns	14.668ns	17.830ns	2.556ns

**These values are derived from the post-synthesis static timing analysis of the RISC32 using the Xilinx Vivado HLx 2017.2 IDE*

The timing delay indicates the workload incurred to each pipeline stage. Performance of the processor is determined by the stage with the longest timing delay of all. From Table 3.24, it is indicated that the longest

delay path among all stages is situated in MEM stage. Apart from WB stage, the IF and ID stage has a lighter workload when compared to MEM stage and EX stage. However, the AES core is not suitable to be placed in the IF stage, because the current instruction is not yet decoded at this stage. For ID stage, important information is already extracted from the instruction. This makes ID stage a potential selection. In EX stage, most of the desired operation by the instruction is carried out by the ALB. All desired computation at EX stage has to be completed within one cycle. Hence, the ALB is expected to have a fairly large combinational logic in order to achieve the one cycle constraint. This in turn introduces a relatively long delay path to EX stage. The EX stage can be a viable selection, provided, the integrated AES Core would not introduce another longer delay path than the existing one's in MEM stage.

In MEM stage, it is reserved for shared memory integration technique. The module (data memory and I/O) here is expected to have a common interface so that the common controls and data bus is shared among each other. This reduces the logic needed to create a new transfer path whenever a new module is attached to MEM stage. However, since this research work focuses on the dedicated path approach, the integrated AES core will have a different interface than the modules at MEM stage. Furthermore, the timing analysis in Table 3.24 indicates that, MEM stage currently has the highest workload among all stages. Introducing the AES Core to MEM stage could lengthen the existing longest delay path and further reduce the overall performance of RISC32. Hence, MEM stage will not be considered. As for the WB stage, it only performs updates to the Register File. However, only

minimal information of the instruction is carried over to the WB stage. Introducing AES core could further complicate the transfer logic at WB stage. Furthermore, since RISC32 is a 5-stage pipeline processor, all instruction takes exactly 5 clock cycles to complete. No stages can be bypassed without going through all the 5 stages. Introducing new transfer path to WB stage would certainly affect all of the previous stages as well. Hence, WB stage would not be considered as well.

In summary, the viable stages for inserting the AES core are ID stage and EX stage. For this research work, ID stage is selected because it possesses the necessary information extracted from the instruction. The necessary operands and control signals are ready to be transferred into AES core by this stage. Moreover, it has been indicated in Table 3.24, ID stage currently has a lighter workload than the EX stages. Introducing AES Core into EX stage might lengthen the existing EX stage delay path, further reducing the overall performance of RISC32 due to workload imbalance.

3.2.2 New Instructions for AES Coprocessor

To utilize the AES core, new instructions are designed to activate and transfer the data along the new transfer path into the AES core during ID stage.

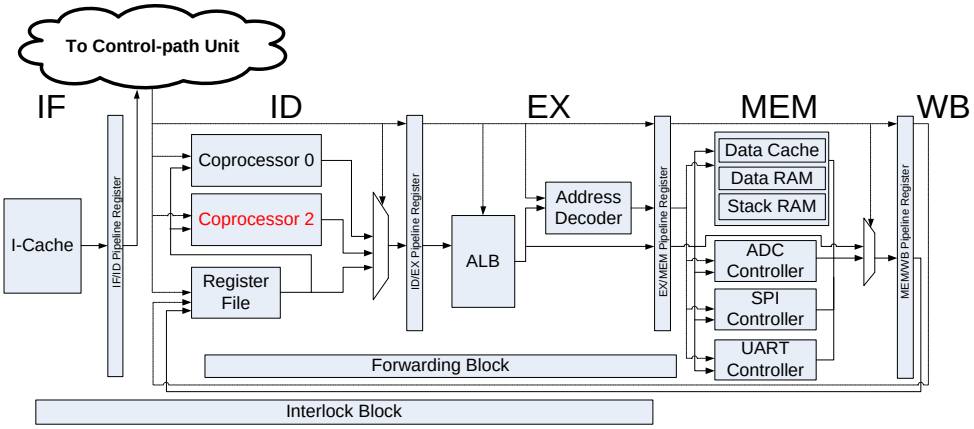


Figure 3.20: RISC32 Microarchitecture with Coprocessor 2 (CP2)

Figure 3.20 shows the RISC32 pipeline connected with AES Coprocessor, which was assigned as Coprocessor 2 (CP2). Coprocessor 0 (CP0) has been assigned for interrupt controller, which monitors various software and hardware interrupt flags in RISC32. Coprocessor 1 (CP1) is reserved for Floating-Point Unit (FPU), which is responsible to perform all floating points computations. CP1 is not implemented in this research work, but it is reserved for future expansion when the needs arise. The new instructions created for CP2 are Move from Coprocessor 2 (*mfc2*) and Move to Coprocessor 2 (*mtc2*).

opcode[31:26]	rs[25:21]	rt[20:16]	rd[15:11]	shamt[10:6]	funct[5:0]
010010	00000	rt	rd	00000	000000

mfc2 $rt ,$rd #Regfile($rt) ← CP2($rd)

Figure 3.21: Move from Coprocessor 2 (*mfc2*) R-Type Instruction Encoding and Syntax

The Figure 3.21 shows the instruction encoding and syntax for Move from Coprocessor 2 (*mfc2*). This instruction moves a 32-bit data into the Register File of RISC32 specified by the *$rt* in the instruction. The *$rd* specifies the location of data to read from, which is the register file in CP2. The registers available for reading in CP2 is shown in Figure 3.22.

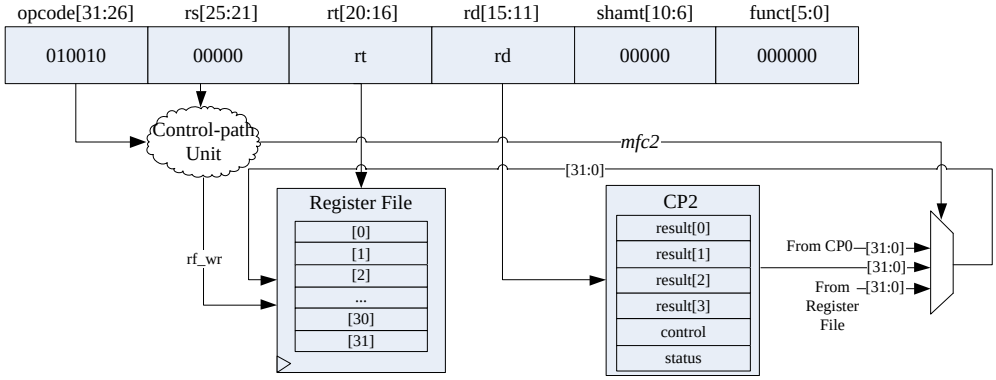


Figure 3.22: *mfc2* implemented using Register Addressing Mode

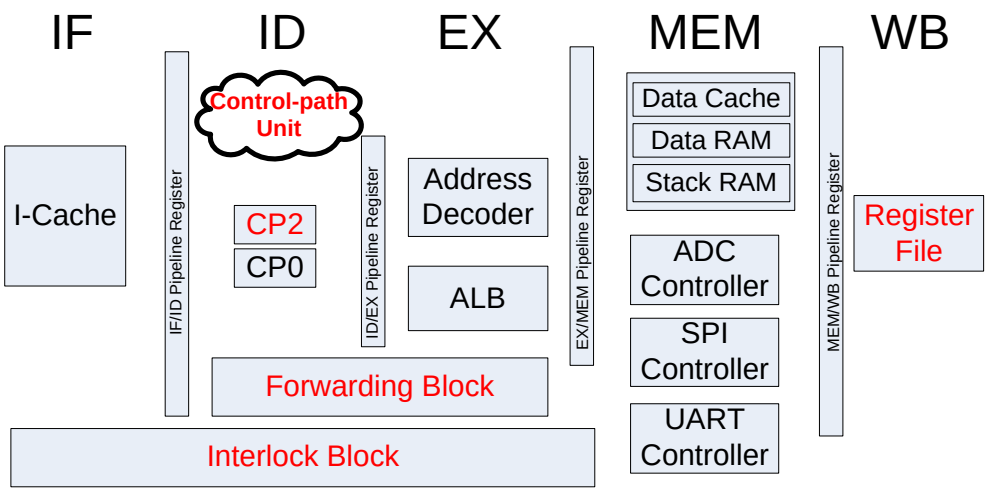


Figure 3.23: Logical view of *mfc2* execution

The Figure 3.23 shows the logical view of the execution of *mfc2*. The *mfc2* is first decoded by Control-path Unit and executed by CP2 at ID stage. The data obtained from CP2 will propagate along the pipeline registers, carried along to EX and MEM stage with no operation performed on it, until it reaches WB stage. The data from CP2 is then updated into the Register File at WB stage.

The Forwarding Block is connected to resolve data hazard related to *mfc2* instructions. The *mfc2* related data hazard arises when the status from CP2 (CP2 status register is read) is used by other instructions but has not reached RISC32 Register File. Another potential scenario for data hazard in *mfc2* occurs when the data (ciphertext) from CP2 has not reached RISC32 Register File, but *sw* instruction is requesting for the ciphertext. Table 3.25 lists all the potential data hazard related to *mfc2*.

Table 3.25: Potential *mfc2* related data hazard

No.	Instructions	Scenario
1	*mfc2 $8, $13* *and $7, $8, $7*	Reading status register of CP2 to check CP2 status. Can be resolved by forwarding CP2 data from EX stage to ID stage.
2	*mfc2 $8, $13* *andi $7, $8, 0x1*	
3	*mfc2 $8, $13* *beq $7, $8, 500*	
4	*mfc2 $8, $13* *nop* *and $7, $8, $7*	Reading status register of CP2 to check CP2 status. Can be resolved by forwarding CP2 data from MEM stage to ID stage.
5	*mfc2 $8, $13* *nop* *andi $7, $8, 0x1*	
6	*mfc2 $8, $13* *nop* *beq $7, $8, 500*	
7	*mfc2 $8, $8* *sw $8, 0($7)*	Store requested before data (ciphertext) of CP2 reaches RISC32 Register File. Can be resolved by forwarding CP2 data from EX stage to ID stage.
8	*mfc2 $8, $8* *nop* *sw $8, 0($7)*	Store requested before data (ciphertext) of CP2 reaches RISC32 Register File. Can be resolved by forwarding CP2 data from MEM stage to ID stage.

opcode[31:26]	rs[25:21]	rt[20:16]	rd[15:11]	shamt[10:6]	funct[5:0]
010010	00100	rt	rd	00000	000000

mtc2 $rt ,$rd #Regfile($rt) → CP2($rd)

Figure 3.24: Move to Coprocessor 2 (*mtc2*) R-Type Instruction Encoding and Syntax

The Figure 3.24 shows the instruction encoding and syntax for Move to Coprocessor 2 (*mtc2*). This instruction reads a 32-bit data from RISC32 Register File, which is specified by *$rt*. The data being read is then written into the CP2 register file, which is specified by *$rd*. The available register for writing in CP2 is shown in the Figure 3.25.

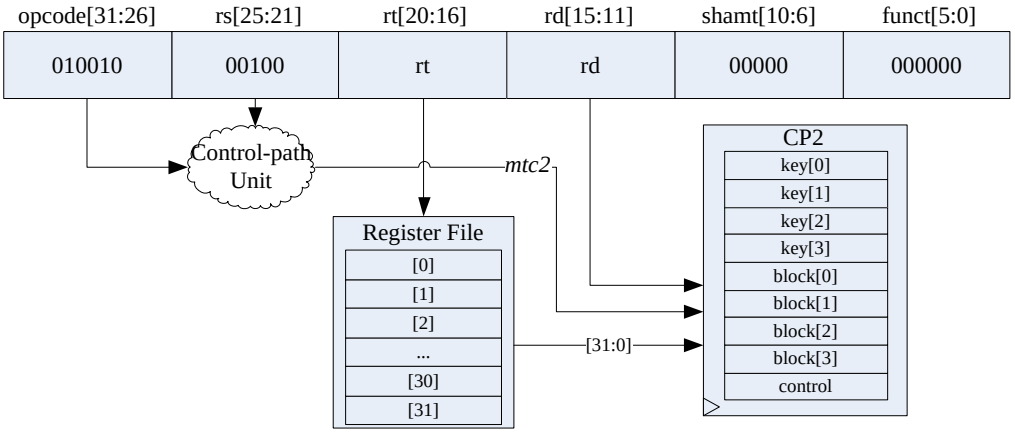


Figure 3.25: *mtc2* implemented using Register Addressing Mode

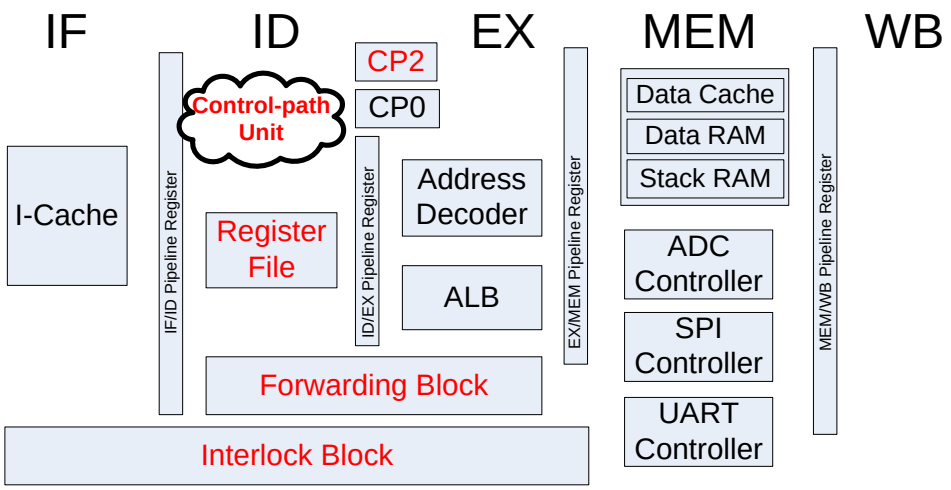


Figure 3.26: Logical view of *mtc2* execution

The Figure 3.26 shows the logical view of *mtc2* execution. It is shown that *mtc2* is decoded in ID stage and its operand is obtained from the Register File during ID stage. This data is then transferred into the CP2 register file during EX stage. Hence, reading the CP2 is in the ID stage whereas writing into CP2 occurs in the EX stage. Note that this does not incur longer delay path to EX stage, because the function of *mtc2* is to transfer data and commands to CP2. It is a write instruction, wherein no computation is involved. Furthermore, the computation path (more details in Section 3.2.3) of CP2 does not yield output within one cycle as compared to the ALB in the EX stage. The output can only be obtained using *mfc2*, which is just a read instruction that does not perform any computation as well. Hence, the alignment of writing into CP2 to EX stage will not affect the overall performance of EX stage.

Similar to *mfc2*, potential data hazard exists for *mtc2* instruction as well. The scenario for data hazard in *mtc2* is considered as general condition related to data hazard in Register File, which is extensively analysed in prior work (Kiat et al., 2017). Another potential data hazard for *mtc2* instruction is load use hazard, which requires Interlock Block to resolve. Table 3.26 lists the possible data hazard condition for *mtc2* instructions.

Table 3.26: Potential *mtc2* related data hazard

No.	Instructions	Scenario
1	*add $8, $8, $7* *mtc2 $8, $12*	*mtc2* require data computed by previous instruction. Can be resolved by forwarding from EX stage to ID stage
2	*addi $8, $8, 0x1* *mtc2 $8, $12*	
3	*lw $8, 0($7)* *mtc2 $8, $0*	Load Use hazard. *mtc2* required data that is currently loaded from data memory. Require Interlock Block to stall pipeline cycle. Then loaded data is forwarded from MEM stage to ID stage by Forwarding Block

3.2.3 CP2 Overview

The AES core is assigned to Coprocessor 2 in RISC32. The AES core used in the research work is an open source single stage AES core designed by Strömbergson (2014). It is a single stage rolled AES architecture; hence, computation is performed iteratively (55 clock cycle) with respect to the number of iterations established for the standard AES algorithm. In this work, the AES core by Strömbergson (2014) is redesigned to perform encryption with 128-bit secret key. Other secret key sizes (192-bit and 256-bit) are not supported currently, as the 128-bit is sufficient to provide a strong security in IoT applications. The AES core will also perform only encryption; decryption is not implemented as RISC32 is targeted for sensor node applications. The decryption is expected to be performed by the gateway device, which collects all the data transmitted from all the sensor nodes in the wireless sensor network. Figure 3.27 shows the top-level interface for CP2 block. Table 3.27 contains the pin description for CP2 block interface. Table 3.28 contains list of CP2 register file and its respective usage.

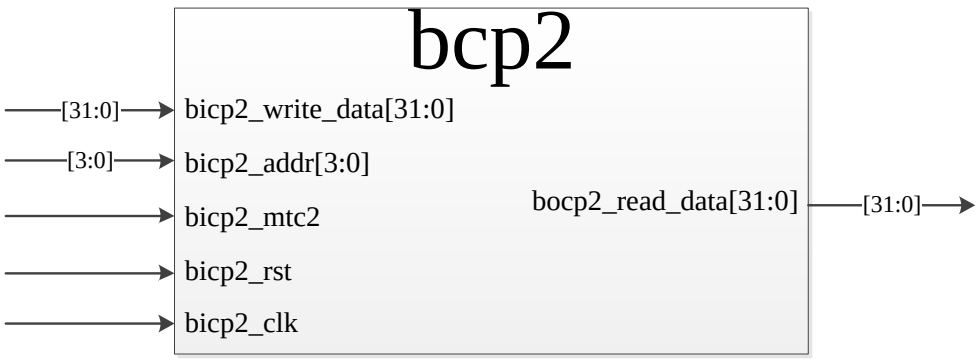


Figure 3.27: Top-Level Interface for CP2 Block

Table 3.27: Pin Description for CP2 Block Interface

Pin Name: bocp2_read_data[31:0] Pin Direction: Output Pin Size: 32 bits Source → Destination: CP2 Block → RISC32 Pipeline Pin Function: Output port for data read from the CP2 register file specified by *mfc2* instruction
Pin Name: bicp2_write_data[31:0] Pin Direction: Input Pin Size: 32 bits Source → Destination: Register File → CP2 Block Pin Function: Input port for data and command to be written into CP2 register file specified by *mtc2* instruction
Pin Name: bicp2_addr[3:0] Pin Direction: Input Pin Size: 4 bits Source → Destination: Data-path Unit → CP2 Block Pin Function: Input port for address of CP2 register file to be accessed by *mtc2* and *mfc2* instruction
Pin Name: bicp2_mtc2 Pin Direction: Input Pin Size: 1 bit Source → Destination: Control-path Unit → CP2 Block Pin Function: Input signal to write data from bicp2_write data into CP2 register file. 0: Do not write into CP2 register file 1: Write into CP2 register file
Pin Name: bicp2_rst Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Reset → CP2 Block Pin Function: 0: Do not reset CP2 1: Reset CP2
Pin Name: bicp2_clk Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Clock → CP2 Block Pin Function: Clock Source

Table 3.28: CP2 register file and their conventions

Register Number	Instruction Encoding	Register Name	Usage
0	0000	key[0]	Stores bit 127 to bit 96 of the128-bit secret key
1	0001	key[1]	Stores bit 95 to bit 64 of the128-bit secret key
2	0010	key[2]	Stores bit 63 to bit 32 of the128-bit secret key
3	0011	key[3]	Stores bit 31 to bit 0 of the128-bit secret key
4	0100	block[0]	Stores bit 127 to bit 96 of the128-bit plaintext
5	0101	block[1]	Stores bit 95 to bit 64 of the128-bit plaintext
6	0110	block[2]	Stores bit 63 to bit 32 of the128-bit plaintext
7	0111	block[3]	Stores bit 31 to bit 0 of the128-bit plaintext
8	1000	result[0]	Stores bit 127 to bit 96 of the128-bit ciphertext
9	1001	result[1]	Stores bit 95 to bit 64 of the128-bit ciphertext
10	1010	result[2]	Stores bit 63 to bit 32 of the128-bit ciphertext
11	1011	result[3]	Stores bit 31 to bit 0 of the128-bit ciphertext

Continued from Table 3.28

12	1100	control	Stores command to be executed by CP2 core. Available options: 0x00000001: Round Key Generation 0x00000002: Encryption
13	1101	status	Stores status of current CP2 core.

Figure 3.28: Status Register (*$13*) layout of CP2

The Figure 3.28 shows the Status Register layout of CP2. The ready (bit 0) indicates if the CP2 core is currently idle. If this bit is HIGH (1), CP2 core is free to perform encryption or key generation; otherwise if it is LOW (0), CP2 is currently busy in performing encryption or key generation. The valid (bit 1) indicates current CP2 core completes its encryption. If this bit is HIGH (1), ciphertext is ready for reading; otherwise if it is LOW (0), it means no ciphertext for reading or the current encryption is not yet completed.

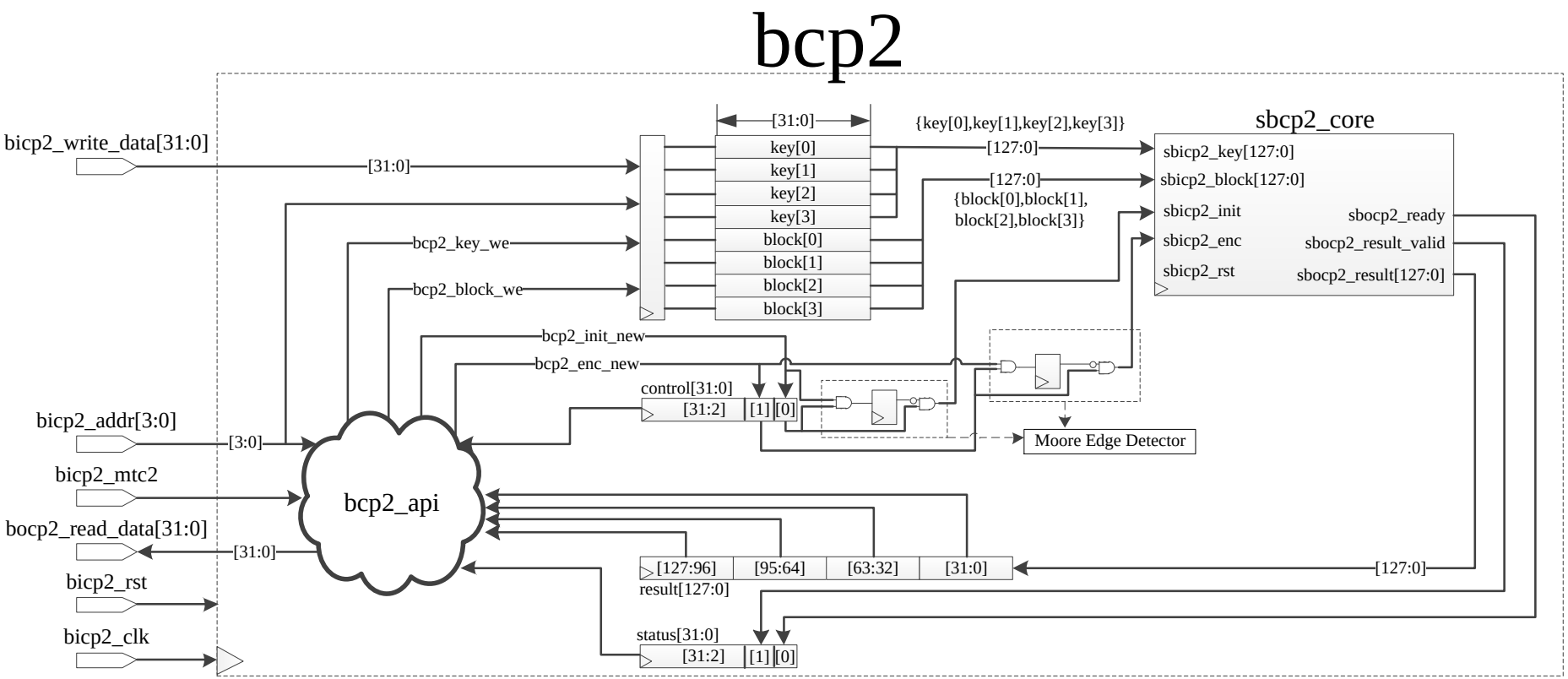


Figure 3.29: The microarchitecture of CP2 Block derived from analysing the AES source code by Strömbergson (2014).

The Figure 3.29 shows the internal microarchitecture of CP2 Block. The top-level architecture of CP2 Block is responsible in registering every input data (secret key and plaintext) and command from the RISC32 pipeline. It is also responsible in storing the output (ciphertext) and status of current CP2 core. Based on the *mtc2* (bicp2_mtc2) input signal and CP2 register file address (bicp2_addr[3:0]), read write operation to the register files of CP2 is decoded and performed by the control logic (bcp2_api). The main operations (round key generation and encryption) of AES is performed by the CP2 Core (sbcp2_core) that is implemented as a sub-block in CP2. Operation to be carried out by it is determined by the command requested in the Control Register of CP2 Block. Figure 3.30 shows the interface for CP2 Core (sbcp2_core). Pin description of the CP2 Core Sub-Block is provided in Table 3.29.

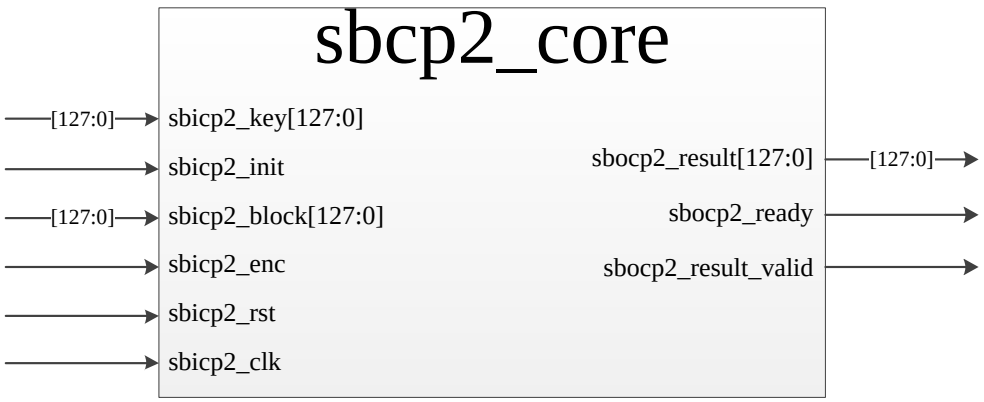


Figure 3.30: CP2 Core Sub-Block interface

Table 3.29: Pin Description for CP2 Core Sub-Block

Pin Name: sbocp2_result[127:0] Pin Direction: Output Pin Size: 128 bits Source→ Destination: CP2 Core Sub-Block → CP2 Block Pin Function: Output port for 128 bits encrypted ciphertext from the CP2 Core Sub-Block
Pin Name: sbocp2_ready Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2 Core Sub-Block → CP2 Block Pin Function: Output status signal to indicate is it currently busy or ready for operation 0: CP2 Core Sub-Block busy. Currently performing encryption or round key generation 1: CP2 Core Sub-Block ready. Request for encryption or round key generation is permitted
Pin Name: sbocp2_result_valid Pin Direction: Output Pin Size:1 bit Source → Destination: CP2 Core Sub-Block → CP2 Block Pin Function: Output status signal to indicate current ciphertext ready for reading 0: Ciphertext not valid or no ciphertext for reading. Encryption is still ongoing in CP2 Core Sub-Block 1: Ciphertext valid. Encryption has been completed by CP2 Core Sub-Block
Pin Name: sbicp2_key[127:0] Pin Direction: Input Pin Size: 128 bits Source → Destination: CP2 Block → CP2 Core Sub-Block Pin Function: Input port for 128-bit secret key for round key generation
Pin Name: sbicp2_init Pin Direction: Input Pin Size: 1 bit Source → Destination: CP2 Block → CP2 Core Sub-Block Pin Function: Input control signal to request for round key generation based on Control Register (*$12*) of CP2 register file 0: No request for round key generation 1: Request for round key generation
Pin Name: sbicp2_block[127:0] Pin Direction: Input Pin Size: 128 bits Source → Destination: CP2 Block → CP2 Core Sub-Block Pin Function: Input port for 128-bit plaintext for encryption
Pin Name: sbicp2_enc Pin Direction: Input Pin Size: 1 bit Source → Destination: CP2 Block → CP2 Core Sub-Block Pin Function: Input control signal to request for encryption based on Control Register (*$12*) of CP2 register file 0: No request for encryption 1: Request for encryption
Pin Name: sbicp2_rst Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Reset → CP2 Core Sub-Block Pin Function: Reset Signal for CP2 Core Sub-Block
Pin Name: sbicp2_clk Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Clock → CP2 Core Sub-Block Pin Function: Clock Source for CP2 Core Sub-Block

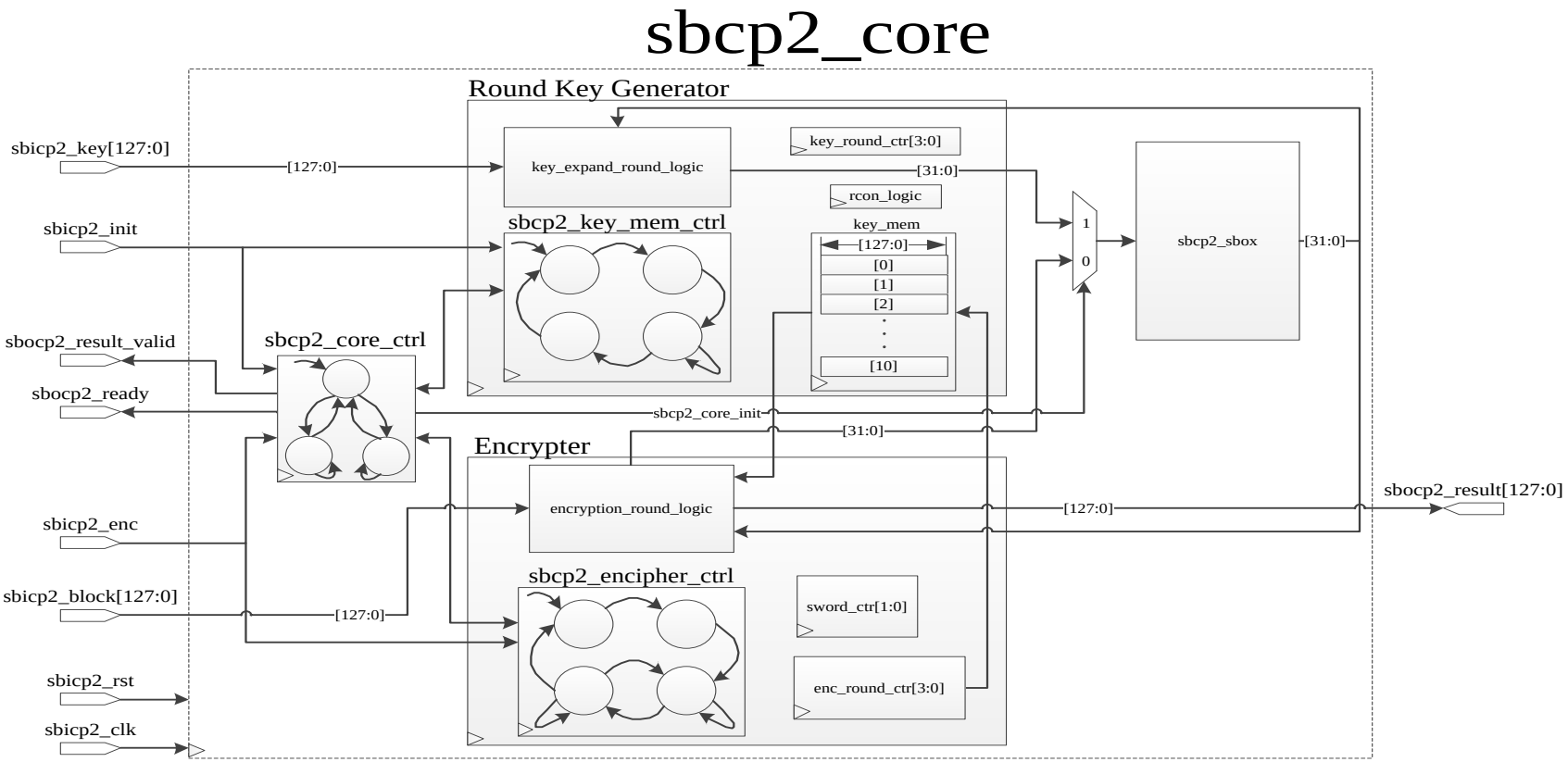


Figure 3.31: Internal microarchitecture of CP2 Core Sub-Block

The Figure 3.31 shows the internal microarchitecture of CP2 Core Sub-Block. The sub-block consists 4 main regions: CP2 Core Finite State Machine (sbcp2_core_ctrl), Round Key Generator (Section 3.2.3.1), Encrypter (Section 3.2.3.2) and Substitution Box (sbcp2_sbox). The main function of CP2 Core Finite State Machine (FSM) is to synchronize activities between Round Key Generator and Encrypter. Also, since there is only single 32-bit Substitution Box, the CP2 Core FSM is also responsible to control share usage of Substitution Box between Round Key Generator and Encrypter. Figure 3.32 shows the states of CP2 Core FSM. The state descriptions and corresponding output is shown in Table 3.30 and Table 3.31.

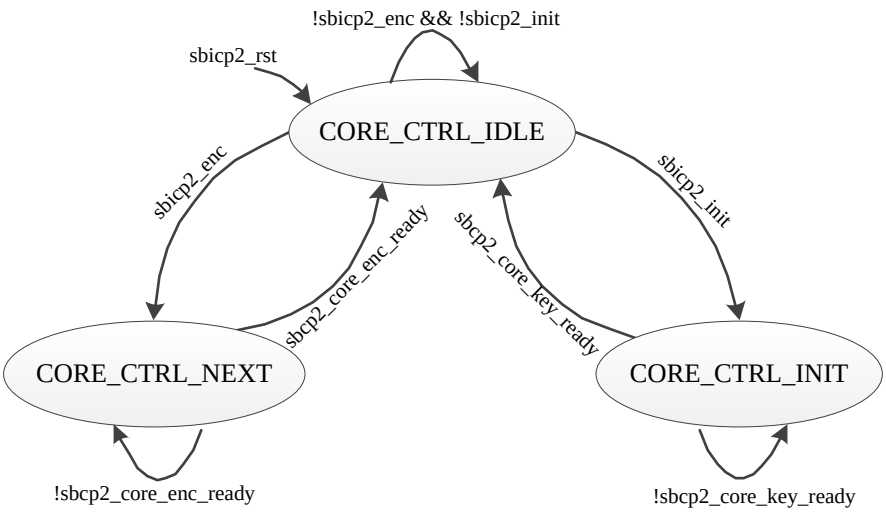


Figure 3.32: CP2 Core FSM state diagram

Table 3.30: CP2 Core FSM state description

State Name	Description
CORE_CTRL_IDLE	No Operation
CORE_CTRL_INIT	Performing round key generation
CORE_CTRL_NEXT	Performing encryption

Table 3.31: CP2 Core FSM state corresponding output

State Name	Corresponding Output

CORE_CTRL_IDLE	When sbicp2_init is HIGH
	sbcp2_core_init = 1'b1;
	sbcp2_core_ready_new = 1'b0;
	sbcp2_core_ready_we = 1'b1;
	sbcp2_core_result_valid_new = 1'b0;
	sbcp2_core_result_valid_we = 1'b1;
	sbcp2_core_ctrl_new = CORE_CTRL_INIT;
	sbcp2_core_ctrl_we = 1'b1;
	
	When sbicp2_enc is HIGH
	sbcp2_core_init = 1'b0;
	sbcp2_core_ready_new = 1'b0;
	sbcp2_core_ready_we = 1'b1;
	sbcp2_core_result_valid_new = 1'b0;
	sbcp2_core_result_valid_we = 1'b1;
	sbcp2_core_ctrl_new = CORE_CTRL_NEXT;
	sbcp2_core_ctrl_we = 1'b1;

CORE_CTRL_INIT	sbcp2_core_init = 1'b1;
	
	When sbcp2_core_key_ready is HIGH
	sbcp2_core_ready_new = 1'b1;
	sbcp2_core_ready_we = 1'b1;
	sbcp2_core_ctrl_new = CORE_CTRL_IDLE;
	sbcp2_core_ctrl_we = 1'b1;

CORE_CTRL_NEXT	sbcp2_core_init = 1'b0;
	
	When sbcp2_core_enc_ready is HIGH
	sbcp2_core_ready_new = 1'b1;
	sbcp2_core_ready_we = 1'b1;
	sbcp2_core_result_valid_new = 1'b1;
	sbcp2_core_result_valid_we = 1'b1;
	sbcp2_core_ctrl_new = CORE_CTRL_IDLE;
	sbcp2_core_ctrl_we = 1'b1;

3.2.3.1 Round Key Generator

The Round Key Generator generates round key based on a 128-bit secret key input. Since AES-128 supported by the CP2 core executes for 10 rounds, a total of 11 round keys will be generated. The round keys generated is stored in a dedicated key memory in the Round Key Generator. These round keys are retrieved by Encrypter when there is an encryption requested. It should be noted that, round key generation and encryption operation cannot be performed simultaneously. Hence, round key generation must always be performed before encryption.

The standard procedure to use CP2 core for round key generation is as follows:

1) Load 128-bit secret key from RISC32 data memory into RISC32 Register File using four Load Word (*lw*) instructions.
2) Move 128-bit secret key from RISC32 Register File into key registers (*$0* to *$3*) of CP2 register file, using four Move to Coprocessor 2 (*mtc2)* instructions.
3) Set round key generation command, *0x00000001* into the RISC32 Register File using Add Immediate (*addi*) instruction.
4) Move round key generation command from RISC32 Register File into Control Register (*$12*) of CP2 register file using *mtc2* to start round key generation.
5) Wait 15 clock cycles for round key generation to complete.

Round Key Generator

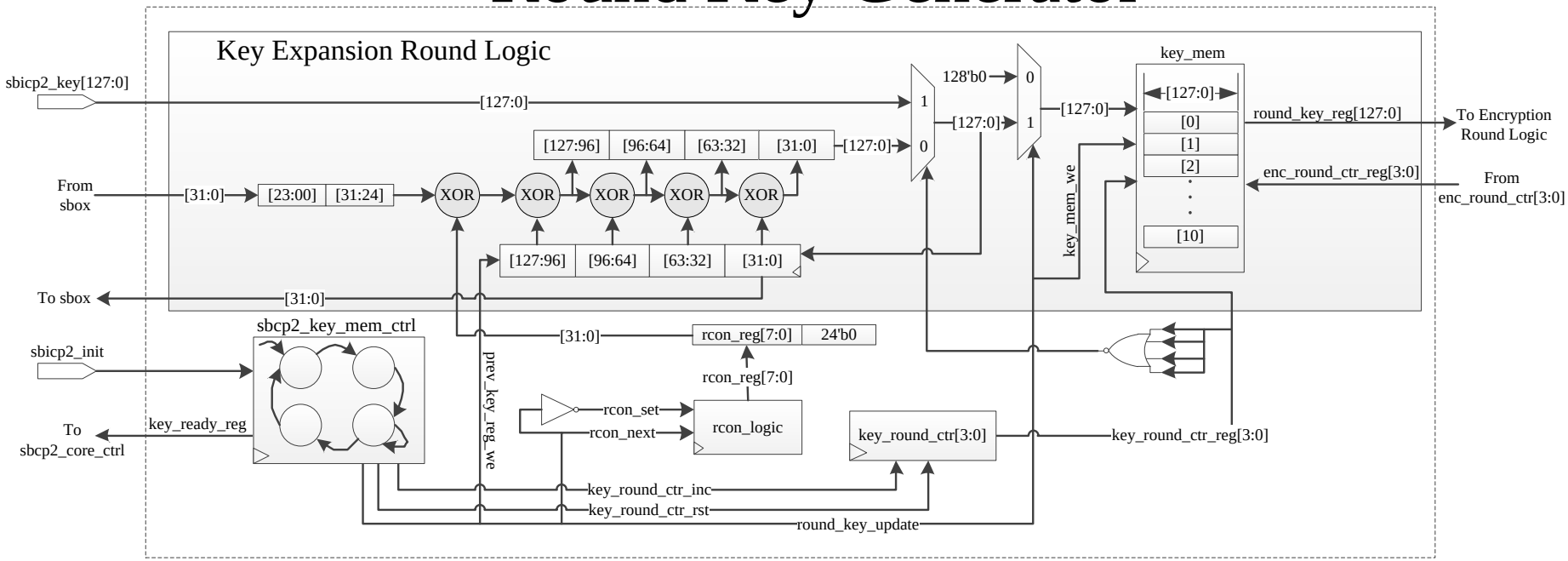


Figure 3.33: Microarchitecture for Round Key Generator

Figure 3.33 illustrates the microarchitecture for Round Key Generator. The Round Key Generator consists of Round Key Generator FSM (sbcp2_key_mem_ctrl), Key Expansion Logic, Round Constant Generator (rcon_logic), Key Expansion Round Counter (key_round_ctr[3:0]) and Key Memory (key_mem). The Round Key Generator FSM generates necessary control signals to control the whole round key expansion operation. The Key Expansion Logic contains logic to be performed on every round of key expansion operation. The Key Expansion Round Counter is a 4-bit up counter that counts from 0 to 10, which corresponds to the rounds of round key generation. The Round Constant Generator generates round constant to be used for each round of the round key expansion operation. Key memory is used to store round keys generated. A total of 11 round keys (128 bits each) is generated upon every round key generation. The Figure 3.34 shows the state diagram for Round Key Generator FSM. Table 3.32 and Table 3.33 provides its state description and state output respectively.

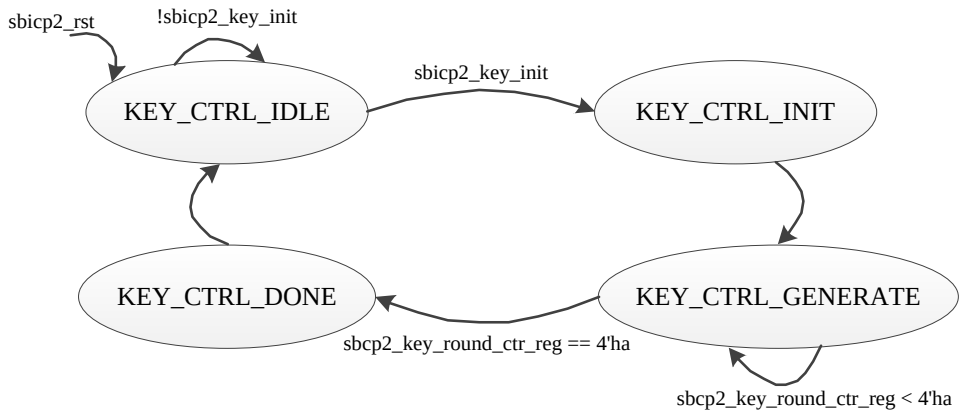


Figure 3.34: Round Key Generator FSM state diagram

Table 3.32: State Description for Round Key Generator FSM

State Name	Description
KEY_CTRL_IDLE	No Operation
KEY_CTRL_INIT	Initialize first round key
KEY_CTRL_GENERATE	Generate next 10 round keys for 10 rounds
KEY_CTRL_DONE	Round key generation completes

Table 3.33: State Output for Round Key Generator FSM

State Name	Corresponding Output

KEY_CTRL_IDLE	When sbicp2_init is HIGH
	sbcp2_key_ready_new = 1'b0;
	sbcp2_key_ready_we = 1'b1;
	sbcp2_key_mem_ctrl_new = KEY_CTRL_INIT;
	sbcp2_key_mem_ctrl_we = 1'b1;

KEY_CTRL_INIT	sbcp2_key_round_ctr_rst = 1'b1;
	sbcp2_key_mem_ctrl_new= KEY_CTRL_GENERATE;
	sbcp2_key_mem_ctrl_we = 1'b1;

KEY_CTRL_GENERATE	sbcp2_key_round_ctr_inc = 1'b1;
	sbcp2_round_key_update = 1'b1;
	
	When sbcp2_key_round_ctr_reg equals 10
	sbcp2_key_mem_ctrl_new = KEY_CTRL_DONE;
	sbcp2_key_mem_ctrl_we = 1'b1;

KEY_CTRL_DONE	sbcp2_key_ready_new = 1'b1;
	sbcp2_key_ready_we = 1'b1;
	sbcp2_key_mem_ctrl_new = KEY_CTRL_IDLE;
	sbcp2_key_mem_ctrl_we = 1'b1;

The round key generation in CP2 Core takes 15 clock cycles to complete. This is contributed by:

1) *mtc2* transfer round key generation command (*0x00000001*) into CP2. This command is only registered at next clock cycle→ 1 clock cycle.
2) The Round Key Generator FSM takes 13 clock cycles to complete all states.

3) The CP2 Core FSM only generates sbcp2_core_ready signal at next clock cycle after Round Key Generator FSM reaches KEY_CTRL_DONE state. → 1 clock cycle.

The Algorithm 3.2 illustrates the round key expansion algorithm implemented for CP2 core with respect to the specification published in NIST FIPS-197(2009).

ALGORITHM 3.2: ROUND KEY EXPANSION ALGORITHM

Input: 128-bit Secret Key
Output: key_mem[0] to key_mem[10] as 11 Round Keys

1. while *round!=10* do
2. if *round == 0*
    a. Store Input as key_mem[0]
    b. Assign Input as previous_key
    c. Calculate next 8-bit rcon
3. else
    a. Assign previous_key[127:96] as word[0]
    b. Assign previous_key[95:65] as word[1]
    c. Assign previous_key[64:32] as word[2]
    d. Assign previous_key[31:0] as word[3]
    e. Assign word[3] to Substitution Box
    f. Assign Substitution Box output to word[3]
    g. Zero-pad 8-bit rcon up to 32-bit
    h. Assign word[3] XOR 32-bit rcon to word[3]
    i. Assign word[3] XOR word[0] to word[0]
    j. Assign word[0] XOR word[1] to word[1]
    k. Assign word[1] XOR word[2] to word[2]
    l. Assign word[2] XOR word[3] to word[3]
    m. Concatenate all word to form 128-bit new_round_key
    n. Store new_round_key to key_mem[round]
    o. Assign new_round_key as previous_key
    p. Calculate next 8-bit rcon
4. endif
5. Increment *round* by 1
6. endwhile

Algorithm 3.2: Round Key Expansion Algorithm of CP2 Core derived from AES Source Code by Strömbergson (2014)

3.2.3.2 Encrypter

The Encrypter performs encryption operation for the CP2 Core. The Encrypter takes 128 bits input plaintext and perform a series of operations on the input for a fixed number of iterations. The resultant 128 bits output between each operation is known as *state*. The series of operations are *AddRoundKey(), ShiftRows(), MixColumns()* and *SubByte()*. The *AddRoundKey()* operation perform simple XOR operation between *state* and the round key. The round key is retrieved from the key memory of the Round Key Generator. In *ShiftRows()* and *MixColumns()*, the operation is performed with the *state* in four words form. The four words *state* is rearranged and assume the form of a 4x4 matrix. The *ShiftRows()* operation perform left cyclic byte shifting on each rows. Number of bytes to be shifted is with respect to the row number. For instance, the first row does not perform any rotation, while at the fourth row, three bytes is rotated to the left and vice versa. The *MixColumns()*, perform a 4x1 matrix multiplication between a fixed polynomial matrix and each column of the 4x4 matrix. Detailed discussion of the fixed polynomial matrix can be found from the NIST FIPS-197 (2009) document. The *SubByte()* operation performs byte substitution between all bytes of the *state* and the Substitution Box (refer to Section 3.2.3.3), which is also known as Sbox. All of the operations mentioned above is performed in every round for 10 rounds.

The standard procedure to use CP2 core for encryption is as follows:

1) Load 128-bit plaintext from RISC32 data memory into RISC32 Register File using four Load Word (*lw*) instructions.
2) Move 128-bit plaintext from RISC32 Register File using four Move to Coprocessor 2 (*mtc2)* instructions into plaintext registers (*$4* to *$7*) of CP2 register file.
3) Set encryption command, *0x00000002* using Add Immediate (*addi*) instruction into RISC32 Register File.
4) Move encryption command from RISC32 Register File into Control Register (*$12*) of CP2 register file using *mtc2* to start encryption.
5) Wait 55 clock cycles for encryption to complete.
6) Move 128-bit ciphertext from ciphertext registers (*$8* to *$11*) of CP2 register file to RISC32 Register File using four Move from Coprocessor 2 (*mfc2*) instructions.
7) Store 128-bit ciphertext from RISC32 Register File into data memory using Store Word (*sw*) instructions.

Encrypter

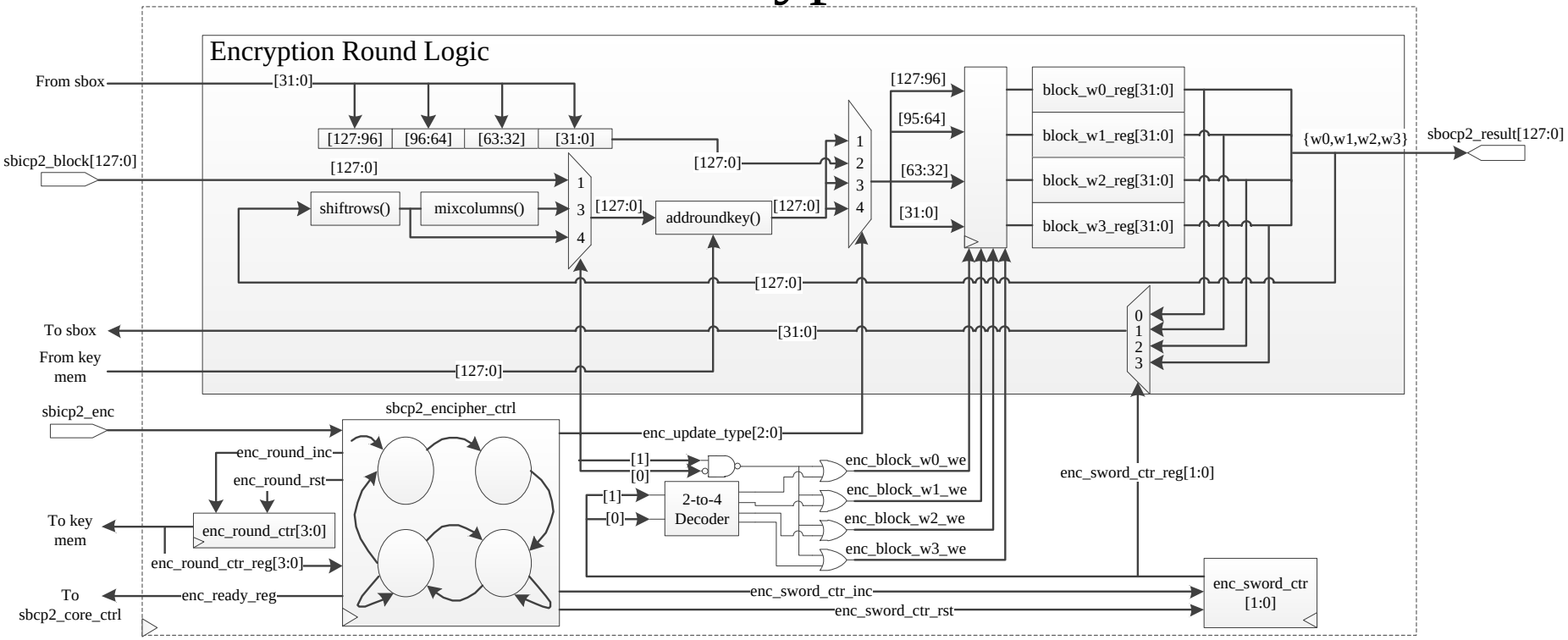


Figure 3.35: Microarchitecture for Encrypter

The Figure 3.35 illustrates the microarchitecture of Encrypter. The Encrypter consists of Encrypter FSM (sbcp2_encipher_ctrl), Encryption Round Logic, Encryption Round Counter (enc_round_ctr[3:0]) and Encryption Substitution Word Counter (enc_sword_ctr[1:0]). The Encrypter FSM generates control logic to control Encryption Round Logic, Encryption Round Counter and Encryption Substitution Word Counter. The Encryption Round Logic performs the *AddRoundkey()*, *ShiftRows()*, *SubByte()* and *MixColumns()* operation. The Encryption Round Counter is a 4-bit up counter that counts from 0 to 10, which corresponds to the rounds of encryption. The counter value is also used for reading the round keys from the round key memory. The Encryption Substitution Word Counter is a 2-bit up counter to track the word position of the current *state* to be substituted by Sbox. The Figure 3.36 shows the state diagram of Encrypter FSM. Table 3.34 and Table 3.35 provides its state description and corresponding output for each state.

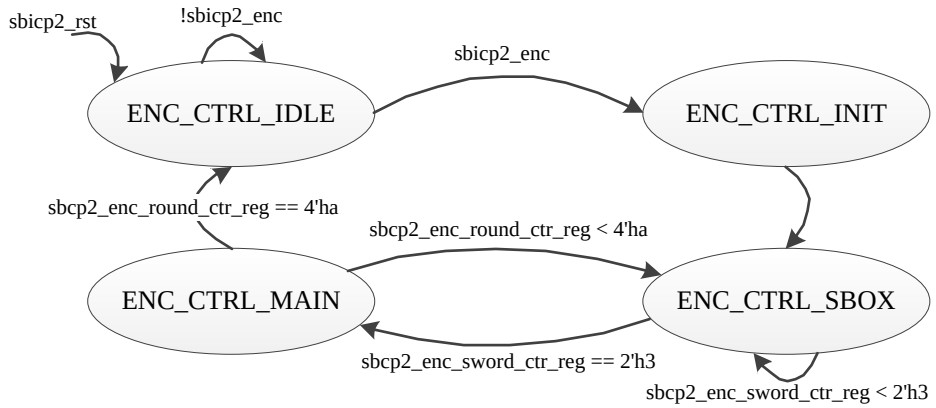


Figure 3.36: Encrypter FSM state diagram

Table 3.34: State Description for Encrypter FSM

State Name	Description
ENC_CTRL_IDLE	No Operation
ENC_CTRL_INIT	Initialize first *state*
ENC_CTRL_SBOX	Perform *SubByte()* operation on every word of *state*
ENC_CTRL_MAIN	Perform *ShiftRows()* , *MixColumns()* , and *AddRoundKey()*

Table 3.35: State Output for Encrypter FSM

State Name	Corresponding Output

ENC_CTRL_IDLE	When sbicp2_ enc is HIGH
	sbcp2_enc_round_ctr_rst = 1'b1;
	sbcp2_enc_ready_new = 1'b0;
	sbcp2_enc_ready_we = 1'b1;
	sbcp2_enc_ctrl_new = ENC_CTRL_INIT;
	sbcp2_enc_ctrl_we = 1'b1;

ENC_CTRL_INIT	sbcp2_enc_round_ctr_inc = 1'b1;
	sbcp2_enc_sword_ctr_rst = 1'b1;
	sbcp2_enc_update_type = ENC_INIT_UPDATE;
	sbcp2_enc_ctrl_new = ENC_CTRL_SBOX;
	sbcp2_enc_ctrl_we = 1'b1;

ENC_CTRL_SBOX	sbcp2_enc_sword_ctr_inc = 1'b1;
	sbcp2_enc_update_type = ENC_SBOX_UPDATE;
	
	When sbcp2_enc_sword_ctr_reg equals 3
	sbcp2_enc_ctrl_new = ENC_CTRL_MAIN;
	sbcp2_enc_ctrl_we = 1'b1;

ENC_CTRL_MAIN	sbcp2_enc_sword_ctr_rst = 1'b1;
	sbcp2_enc_round_ctr_inc = 1'b1;
	
	When sbcp2_enc_round_ctr_reg less than 10
	sbcp2_enc_update_type = ENC_MAIN_UPDATE;
	sbcp2_enc_ctrl_new = ENC_CTRL_SBOX;
	sbcp2_enc_ctrl_we = 1'b1;
	
	When sbcp2_enc_round_ctr_reg equals 10
	sbcp2_enc_update_type = ENC_FINAL_UPDATE;
	sbcp2_enc_ready_new = 1'b1;
	sbcp2_enc_ready_we = 1'b1;
	sbcp2_enc_ctrl_new = ENC_CTRL_IDLE;
	sbcp2_enc_ctrl_we = 1'b1;

The encryption in CP2 Core takes 55 clock cycles to complete. This is contributed by:

1) *mtc2* transfer encryption command (*0x00000002*) into CP2. This command is only registered at next clock cycle→ 1 clock cycle
2) The Encrypter FSM takes 53 clock cycles to complete all states
3) The CP2 Core FSM only generates sbcp2_core_ready signal and sbcp2_core_result_valid at next clock cycle after Encrypter FSM return to the ENC_CTRL_IDLE state. → 1 clock cycle

The Algorithm 3.3 illustrates the encryption algorithm implemented in CP2 core with respect to the specification published in NIST FIPS-197(2009).

ALGORITHM 3.3: ENCRYPTION ALGORITHM

Input: 128-bit Plaintext, Eleven (11) 128-bit Rounds Keys
Output: Final *state* as 128-bit Ciphertext

1. while *round* ≤ *10* do
2. Read key_mem[round]
3. if *round == 0*
    a. Assign *AddRoundKey*(Input, key_mem[round]) as *state*
    b. Increment *round* by 1
4. else
    a. Assign *state*[127:96] as word[0]
    b. Assign *state*[95:64] as word[1]
    c. Assign *state*[63:32] as word[2]
    d. Assign *state*[31:0] as word[3]
    e. Reset *sword*
    f. while *sword* ≤ *3* do
        i. Assign *SubByte*(word[*sword*]) as word[*sword*]
        ii. Increment *sword* by 1
    g. endwhile
    h. Concatenate all word and assign to *state*
    i. Assign *ShiftRows*(*state*) as *state*
    j. if round *!= 10* then
        i. Assign *MixColumns*(*state*) as *state*
    k. endif
    l. Assign *AddRoundKey*(*state*, key_mem[round]) as *state*
    m. Increment *round* by 1
5. endif
6. endwhile

Algorithm 3.3: Encryption Algorithm of CP2 Core derived from AES Source Code by Strömbergson (2014)

3.2.3.3 Substitution Box (Sbox)

The Substitution Box (Sbox) is a substitution table used in both round key generation and encryption operation. The values in Sbox is predefined and implemented as an 8-bit x 256 lookup-table. The Sbox performs substitution by using the one-byte input as offset to retrieve value from the lookup-table. Details of the calculation for each value is not discussed in this research work, but can be found from the NIST FIPS-197 (2009) document. The Table 3.36 shows the values in Sbox.

Table 3.36: SBox Table
Source: NIST FIPS-197, 2009

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

**The x and y represent rows and columns position. Eg: Input = {0x3b}, then Output of sbox = {0xe2} and vice versa*

In CP2 Core, the Sbox is implemented as four 8-bit x 256 ROM. Each of these ROM can only output one byte every clock cycle. Hence, CP2 Core performs 4 parallel substitution on every clock cycle to output a complete 32-bit word. However, substitution transformation is present in both round key generation and encryption operation. Due to this reason and the maximum

output of Sbox (32-bit substitution per clock cycle) in CP2, both operations cannot be executed in parallel and have to share the Sbox. The design of Sbox also affects encryption operation, where each round requires *SubByte()* to be performed on the 128-bit *state*. Since Sbox can only output 32-bit every clock cycle, the *SubByte()* operation takes 4 clock cycles to complete, which contributes a longer clock cycle for the encryption operation in CP2 Core. However, this Sbox design has a smaller hardware consumption, as it only implements four 8-bit x 256 ROM (32-bit) instead of 16 (128-bit). The smaller hardware consumption is beneficial in terms of power consumption for IoT implementation purposes. Figure 3.37 shows the internal structure of Sbox for CP2 Core.

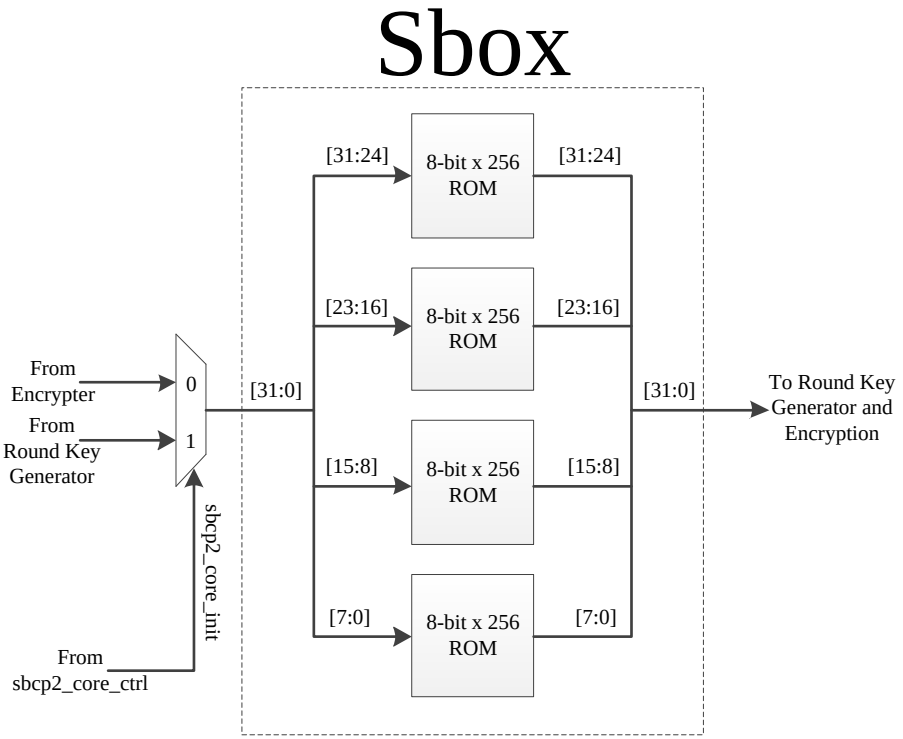


Figure 3.37: Internal structure of Sbox in CP2 Core

3.2.4 Software Pattern Analysis for CP2

The primary aim of RISC32 (Kiat, 2018) was to develop a processor for implementing IoT sensor nodes. With the integration of CP2, this research works aims to provide security feature to RISC32. This security feature can ensure confidentiality of the sensor data, which will be transferred out of the sensor node periodically. Figure 3.38 shows a typical software pattern of the IoT applications in sensor nodes.

Acquire *N* Byte (Eg: ADC)	Process *N* Byte (Eg: Encrypt with AES)	Send *N* Byte (Eg:UART,SPI)

Figure 3.38: Typical IoT application in sensor nodes

Referring to Figure 3.38, the typical software pattern performs three tasks, namely Data Acquisition, Data Processing and Data Sending. The Data Acquisition refers to collecting data through various sensors. Common sensors available are humidity sensor, temperature sensor, infrared sensor and vibration sensor. These sensors are usually interfaced to Analog-to-Digital Converter (ADC), so that the analog output by the sensors are converted to digital values for processing. Data Processing refers to performing computation on the collected sensor data. For example, calculation (e.g. average, minimum/maximum and etc.) may need to be performed on the ADC data to yield meaningful representation of the sensor data. Action of Data Processing varies between each IoT applications. In the context of this research work, Data Processing is assumed to be AES Encryption. After Data Processing, Data Sending sends processed data out to gateway and cloud

server, so that it can be analysed to derive meaningful patterns. The sending can usually be done through common I/O modules such as WiFi, Bluetooth Low Energy (BLE) or ZigBee modules. These I/O modules are usually interfaced with I/O interface such as UART, SPI and I2C.

All the task mentioned in previous paragraph revolves around specific data size, which is labelled as *N* bytes. The typical size for *N* ranges from 64B, 128B, 256B, 512B and 1024B. Sizes of *N* could differ between applications. This research work recommends *N* between the range of 256 to 1024 for IoT applications that constantly monitors and send large data size. The recommended range could reduce the need for frequent sending. Smaller data size would indicate the need to frequent sending, which increases activity of IoT processor. This could cause higher power consumption, which is not ideal for sensor nodes with energy constrain. Moreover, the proposed sizes can also fit into the RAM of targeted FPGA and other common microcontrollers.

However, the CP2 currently integrated into RISC32 cannot fully utilize the software pattern presented in Figure 3.38. Figure 3.39 shows the software pattern incorporated with encryption using CP2.

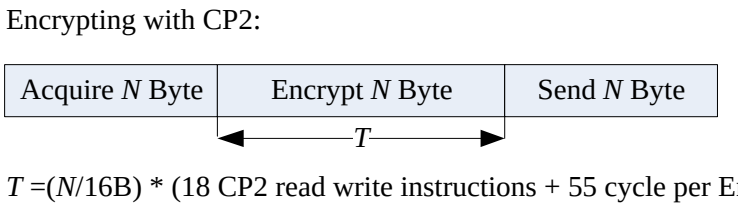


Figure 3.39: Data processing pattern with CP2 encryption

Referring to Figure 3.39, the encryption of *N* byte of data requires total *T* clock cycles. *T* is determined from the total number of instructions required to prepare and read the sensor data from CP2, setting command for CP2, together with 55 clock cycles of encryption computation. The Table 3.37 shows the instructions required to use CP2 for encryption.

Table 3.37: Encryption Routine for CP2 Excluding Data Acquisition

Instruction	Comment
lw $rt0, 0($rs)	Load 128-bit plaintext (four words) from memory into RISC32 Register File
lw $rt1, 4($rs)	
lw $rt2, 8($rs)	
lw $rt3, 12($rs)	
mtc2 $rt0, $4	Move loaded 128-bit plaintext (four words) into secret key register (*$0* to *$3*) of CP2
mtc2 $rt1, $5	
mtc2 $rt2, $6	
mtc2 $rt3, $7	
addi $rt4, $zero,0x2	Prepare encrypt plaintext command
mtc2 $rt4, $12	Move key expansion command to command register (*$12*) of CP2 and start encryption
nop	Insert *NOPS* to wait for encryption to complete. CP2 encryption requires 55 clock cycles to complete, hence 55 *NOPS* is inserted.
nop	
..	
mfc2 $rt0, $8	Read encrypted 128-bit cipher text (four words) from cipher text register (*$8* to *$11*) of CP2 and write into RISC32 Register File
mfc2 $rt1, $9	
mfc2 $rt2, $10	
mfc2 $rt3, $11	
sw $rt0,0($rs)	Store 128-bit cipher text (four words) from RISC32 Register File into data memory
sw $rt1,4($rs)	
sw $rt2,8($rs)	
sw $rt3,12($rs)	

A total of 18 instructions are required to start and read CP2 data, with additional 55 NOP instructions to wait for CP2 to output the valid ciphertext. Hardware pipeline stalling is another solution where the pipeline stages are held whenever there is a read to the CP2 register file while CP2 is still encrypting. Both solutions are not efficient, as it creates 55 cycles of idle time in the processor pipeline. Consider the case where $N = 1024$, then T is 4672

clock cycles, wherein 3520 cycles (75%) are spent idle for waiting the encryption to complete!

To utilize the idle time caused by CP2 encryption, a more effective way would be to overlap the data acquisition and encryption, which is illustrated in the Figure 3.40.

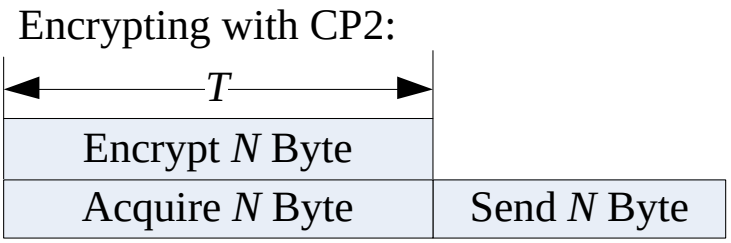


Figure 3.40: Data processing pattern with encryption and data acquisition overlapped

The data processing pattern (Figure 3.40), however, is highly dependent on the AES encryption mode used. The common AES encryption modes available are Electronic Code Book (ECB), Cipher Block Chaining (CBC) and Counter (CTR).

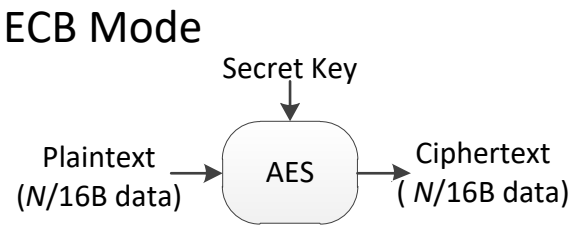


Figure 3.41: Electronic Code Book (ECB) AES Encryption Mode

The ECB encryption mode (Figure 3.41) performs encryption directly on the input plaintext. The plaintext here refers to the sensor data collected during Data Acquisition task. This shows that the ECB has dependency on the sensor data. Data Acquisition needs to take place before encryption can be started. Hence, ECB mode could not fit the proposed processing pattern (Figure 3.40) as Data Acquisition and encryption could not be overlapped due to the dependency.

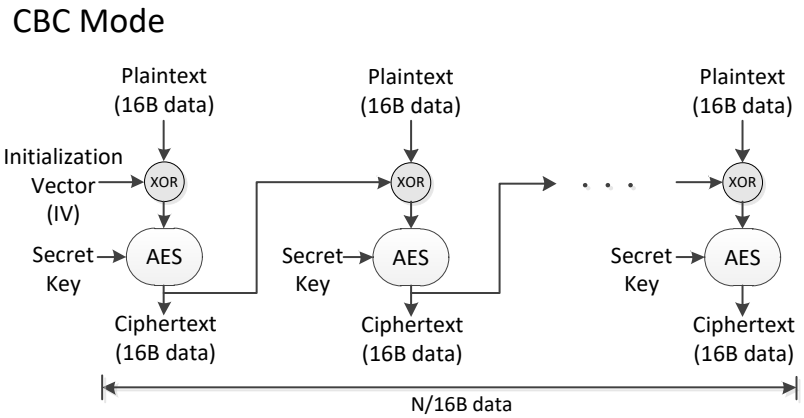


Figure 3.42: Cipher Block Chaining (CBC) AES Encryption Mode

The CBC encryption mode is illustrated in the Figure 3.42. For the first encryption among all data (*N* Byte), the input plaintext will be XORed with an equivalent size (16 Byte) Initialization Vector (IV). The consecutive encryption is dependent on previous encryption, where their input is the XOR of respective plaintext and previous ciphertext. This dependency of CBC on previous ciphertext does fit to the consecutive encryption for all data (*N* Byte) as illustrated in Figure 3.40. However, the CBC also has dependency for the plaintext. Data Acquisition has to be performed first to obtain plaintext (sensor

data) before the first encryption can be started. This does not fit the proposed software pattern (Figure 3.40), as encryption and Data Acquisition is expected to be performed together independently.

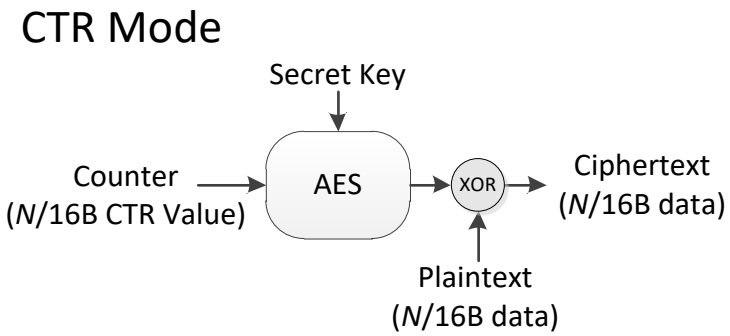


Figure 3.43: Counter (CTR) AES Encryption Mode

The Counter (CTR) AES Encryption Mode (Figure 3.43) has a distinctive difference than the two (ECB and CBC) encryption mode mentioned earlier. Instead of performing encryption directly on the plaintext, the encryption is first applied on a counter. The counter is constructed from a random Initialization Vector and an initial counter value to form a 128-bit value. The counter is incremented between each encryption for consecutive series of data (*N* Byte). The encrypted counter value is then XORed with the respective plaintext to yield ciphertext. The process of CTR mode shows that encryption can be executed independently without needing the plaintext beforehand. The encryption can be executed first. The idle time (55 clock cycles per encryption using CP2) can be used to perform Data Acquisition of *N* byte. By the time both encryption and Data Acquisition completes, simple XOR operation will be carried out to obtain the final ciphertext. This

encryption mode certainly fits the proposed software pattern (Figure 3.40). Hence, the Figure 3.44 shows the software pattern from Figure 3.40 fitted with CTR AES Encryption Mode.

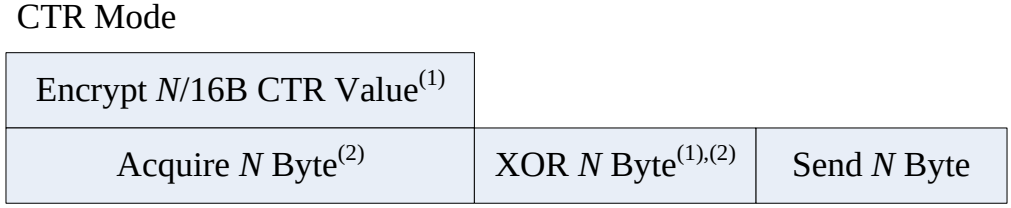
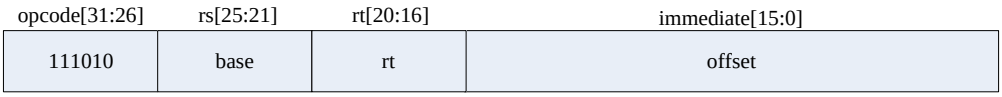


Figure 3.44: Data processing pattern with encryption and data acquisition overlapped in CTR Mode

The RISC32 integrated with CP2 at the current stage however, are not catered for data processing as shown in Figure 3.44. The proposed data processing pattern (Figure 3.44) requires RISC32 to carry out two separate tasks in parallel together. While the CP2 can execute independently from other functional unit of RISC32, the CP2 integrated are single stage architecture. This means the CP2 can only process a single plaintext at a time. The CP2 cannot be used before the encryption completes. This is to prevent overwriting the intermediate data that would yield ciphertext by the end of encryption. Furthermore, the RISC32 currently can only dispatch one instruction at a time. To obtain the effect of executing two different tasks together in Figure 3.44, RISC32 is required to dispatch two different set of instructions for two different tasks. As such, a Queue System (Section 3.2.6) was proposed to realize the data processing pattern in Figure 3.44. The Queue system will schedule the encryption task so that it could be executed alongside with Data Acquisition tasks. Additionally, a new instruction, Store Word Coprocessor 2

(*swc2*) (Section 3.2.5) is introduced, to resolve potential register file dependency issue present with the Queue System.

3.2.5 Store Word from Coprocessor 2 (*swc2*)



swc2 $rt ,offset($rs) #Memory[base+offset] ← CP2($rt)

Figure 3.45: Store Word from Coprocessor 2 (*swc2*) I-Type Instruction Encoding and Syntax

Figure 3.45 shows the instruction encoding and syntax for Store Word from Coprocessor 2 (*swc2*). This instruction has a similar behaviour as *mfc2*, which also reads from CP2 register file. This instruction reads 32 bits data from CP2 register file specified by the *$rt* in the instruction. Unlike *mfc2*, *swc2* writes to the memory address specified, instead of the Register File in RISC32. The memory address, with *$rs* providing the base address, and memory offset as immediate value from bit 15 to bit 0 is encoded into the *swc2* instruction. The addressing mode of *swc2* is shown in Figure 3.46.

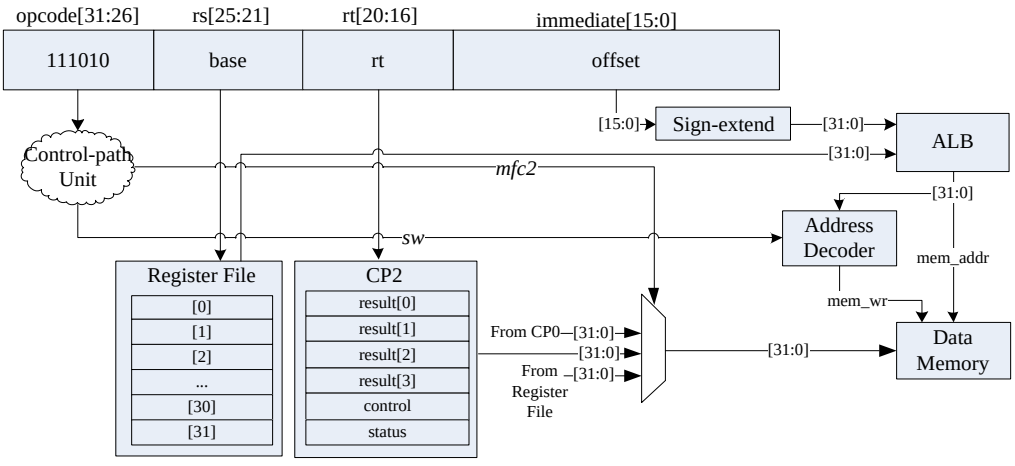


Figure 3.46: *swc2* implemented using Base Addressing Mode

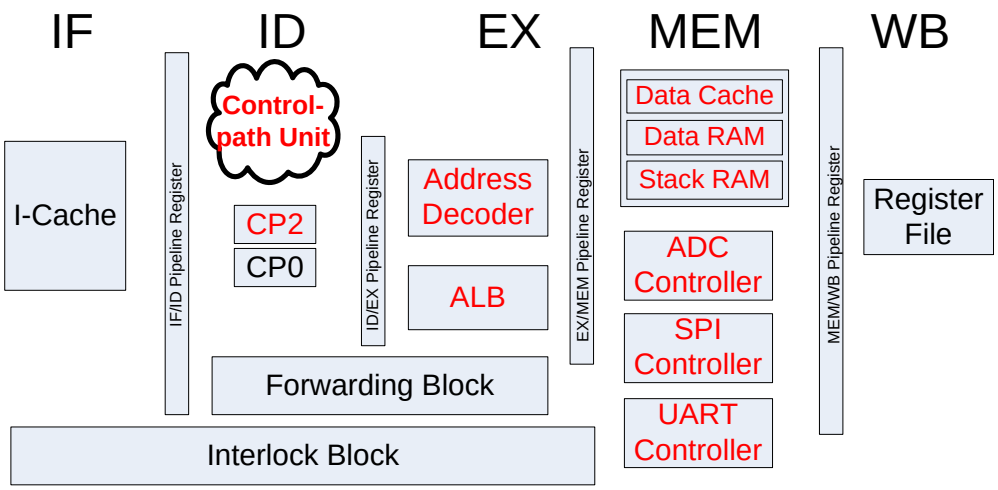


Figure 3.47: Logical view of *swc2* execution

The Figure 3.47 shows the logical view of *swc2* execution. As shown in Figure 3.47, the *swc2* is decoded by Control-path Unit and executed by CP2 at ID stage. The data obtained from CP2 at this stage is carried along EX through pipeline register until it reaches MEM stage. At EX stage, the memory address for *swc2* is calculated using ALB. The calculated address is decoded by the Address Decoder to activate the desired memory module (Data RAM, Stack RAM, Data Cache or I/O modules). At MEM stage, the data is written into the desired memory location, based on the activated memory module decoded during EX stage.

Unlike *mfc2*, *swc2* does not have data hazard issue. This is because the data fetched during ID stage is directly stored into the memory location at MEM stage. Since both actions are performed within the execution of the same instruction, data hazard will not happen.

The *swc2* is introduced to resolve potential scheduling issue with the Queue System. Before *swc2* is introduced, the ciphertext can only be read from CP2 using *mfc2*. The ciphertext should not be stored in the limited Register File of RISC32 due to its large size, which is 128 bits. Since the *mfc2* can only write back to Register File of RISC32, *mfc2* is required to pair with *sw* to store the ciphertext into data memory. Table 3.38 shows the required instructions to store a complete 128-bit ciphertext into data memory.

Table 3.38: Storage of ciphertext from CP2 using *mfc2-sw* pair

Potential Instruction Pattern 1		Potential Instruction Pattern 2	
mfc2 $rt0, $8	#Read result[0] from CP2	*mfc2 $rt0, $8*	#Read result[0] from CP2
mfc2 $rt1, $9	#Read result[1] from CP2	*sw $rt0,0($rs)*	#Store result[0]
mfc2 $rt2, $10	#Read result[2] from CP2	*mfc2 $rt1, $9*	#Read result[1] from CP2
mfc2 $rt3, $11	#Read result[3] from CP2	*sw $rt1,4($rs)*	#Store result[1]
sw $rt0,0($rs)	#Store result[0]	*mfc2 $rt2, $10*	#Read result[2] from CP2
sw $rt1,4($rs)	#Store result[1]	*sw $rt2,8($rs)*	#Store result[2]
sw $rt2,8($rs)	#Store result[2]	*mfc2 $rt3, $11*	#Read result[3] from CP2
sw $rt3,12($rs)	#Store result[3]	*sw $rt3,12($rs)*	#Store result[3]

It is shown that for both instruction pattern (Table 3.38), the *sw* and *mfc2* has a weak link. There is not enough information to determine a strong relationship between them, except that both instructions use the same *$rt*. However, determining the *sw* and *mfc2* based on their *$rt* has a potential risk. If there is an exception raised during the execution of the instruction pattern illustrated (Table 3.38), the exception handler or interrupt service routine (ISR) might have a *sw* instruction that has the matching *$rt* to preceding *mfc2*. The Queue System might not be able to queue the correct *sw* if such case arises. With *swc2* however, Queue System will only need to detect *swc2* instead. Total instructions required to store ciphertext is also reduced.

3.2.6 Overview of the Queue System

In Section 3.2.4, analysis was performed on the typical software pattern of IoT applications, where an overlapping data processing pattern is proposed (Figure 3.44). This section discusses the Queue System proposed in this research work to implement the overlapping pattern. The proposed hardware solution is inspired by the Tomasulo Algorithm (Hennessy and Patterson, 2011). The Tomasulo Algorithm is a common dynamic scheduling technique found in Floating Point Units (FPU) to handle long computation cycle of floating-point operations. In Tomasulo Algorithm, dynamic scheduling is realized by reservation stations. The reservation stations constantly monitor the status (busy or idle) of its respective functional units (Eg: Floating-Point Adder, Floating-Point Multiplier), determining whether the functional units are ready to execute an instruction. In cases where functional units are busy, the reservation stations will hold any incoming instructions until the functional units are idle again. The reservation stations also have the ability to resolve data dependency issue between instructions.

The Queue System proposed have similar function to the reservation stations. It consists of two new hardware, namely the Coprocessor 2 Queue (CP2Q) and Store Word Queue (SWQ). The CP2Q (Section 3.2.6.1) is responsible to monitor the status of CP2 and scheduling of CP2 related instruction. The SWQ (Section 3.2.6.2) is controlled by CP2Q, to schedule store instructions that is awaiting output from CP2 due to its long encryption clock cycle.

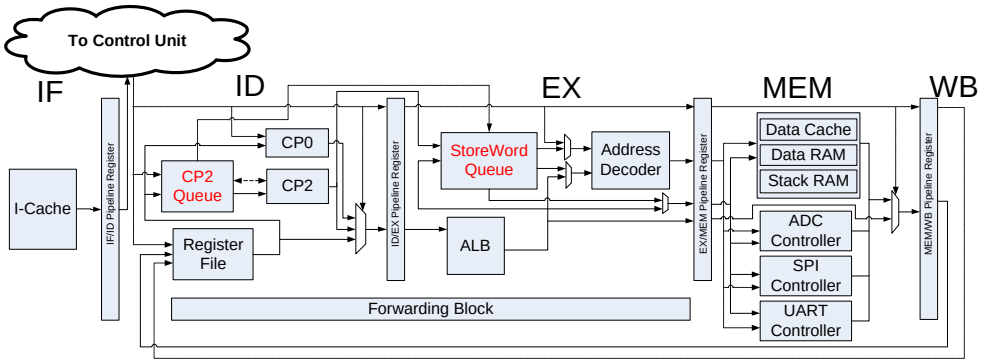


Figure 3.48: RISC32 with CP2 and Queue System

The Figure 3.48 shows the RISC32 with CP2 and Queue System. The Queue System composes of CP2 Queue (CP2Q) and Store Word Queue (SWQ). The CP2Q and SWQ work together with CP2 to queue and execute the CP2 instructions in the pipeline. All queuing and re-insertion of CP2 instructions into the processor pipeline, is completely hidden from the software. The user can write program with the pattern shown in Figure 3.49, while the underlying hardware queue will reschedule the execution of the CP2 instructions as proposed at the end of Section 3.2.4 (Figure 3.44).

Encrypt *N*/16B CTR Value	Acquire *N* Byte	XOR *N* Byte	Send *N* Byte

Figure 3.49: Serial processing pattern in CTR mode

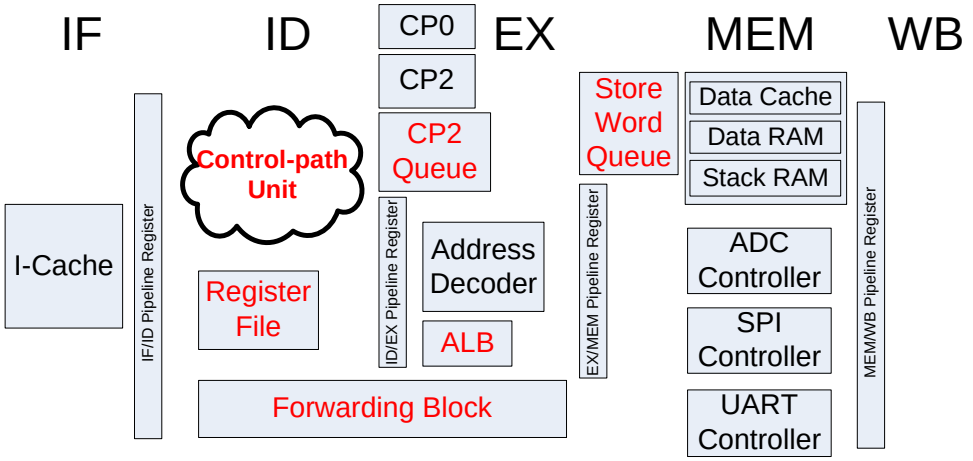


Figure 3.50: Logical view of Queue System execution when CP2 is busy

The Figure 3.50 shows the Queue System execution when CP2 is busy. Every decoded instruction at ID stage will go through CP2Q before proceeding to EX stage. When CP2 is busy, the CP2 Queue (CP2Q) will determine if current instruction at ID stage should be queued. This refers to *mtc2* and *swc2*. Any other instruction that is not related will bypass CP2Q and proceed as usual. It should be noted as well, the Queue System will allow CP2 instruction to bypass for direct execution when CP2 is idle.

When an *mtc2* instruction is detected by CP2Q, the decoded *mtc2* instruction is stored into CP2Q. CP2 register file address and decoded *mtc2* control signal is stored into CP2Q Instruction RAM. Fetched operand for *mtc2* is stored into the CP2Q Data RAM. In cases where data hazard occurred for *mtc2*, the operand will be forwarded to the CP2Q by the Forwarding Block. Otherwise, the operand comes directly from the Register File. When *swc2* is detected by the CP2Q, the decoded *swc2* control signal and CP2 register file address will be stored into the CP2Q Instruction RAM. With the detection of

swc2 in CP2Q, the address encoded with *swc2* will be recorded in SWQ as well. This action is triggered by CP2Q, where a control signal (bcp2Q_swQ_sw_wr) will be generated by CP2Q at the next cycle and transmitted to EX stage. This control signal will instruct SWQ to store the calculated address by ALB into SWQ Address RAM.

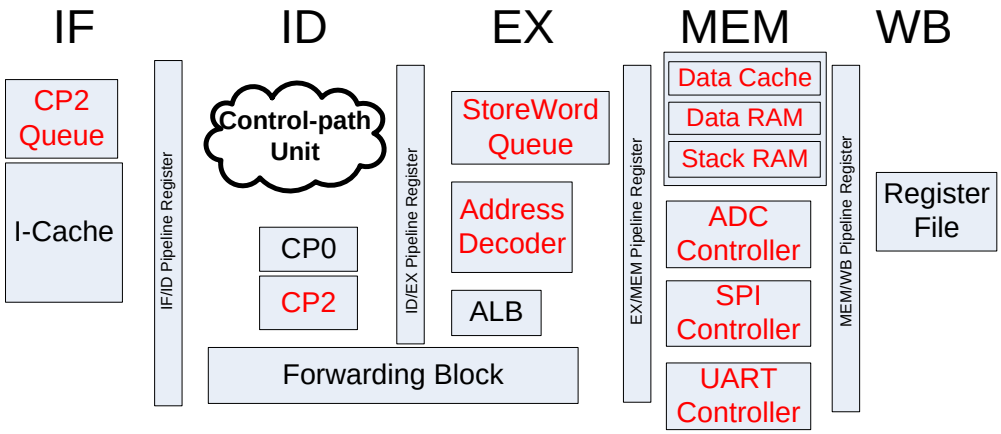


Figure 3.51: Logical view of Queue System execution when CP2 is ready

Figure 3.51 shows the Queue System execution when CP2 is ready. The illustrated execution only applies if CP2 instruction is previously queued in the Queue System. If CP2 is idle and no unexecuted CP2 instruction were present in the Queue System, the CP2 instructions (*mtc2*, *mfc2*, and *swc2*) are allowed to bypass Queue System and executed by CP2 directly.

When CP2Q detects that CP2 is ready (encryption completes), the CP2Q will fetch decoded CP2 instruction (*mtc2* or *swc2* control signal) and CP2 register file address from CP2Q Instruction RAM. In the case of *mtc2*, operand is fetched from the CP2Q Data RAM as well. The information for

mtc2 instruction will be carried over to ID stage and executed by CP2 as discussed in Section 3.2.2. In the case of *swc2* fetched, the action to read ciphertext from CP2 will be executed at ID stage, and updated into SWQ Data RAM at EX stage. The SWQ will keep track of the number of words read from CP2, until it accumulated 4 words (32-bit) from CP2, which is equivalent to 128-bit ciphertext. These data from CP2 is stored into SWQ Data RAM. When a complete ciphertext is being read, the address for previously queued *swc2* instructions will be fetched from SWQ Address RAM. Ciphertext accumulated is also fetched from SWQ Data RAM again. These actions (reading from SWQ Address and Data RAM) will trigger SWQ to generate a stall signal (bswQ_pipe_stall), eventually stalls the IF and ID stage of RISC32 pipeline. At the same time, the stall signal also flushes the EX stage. These series of actions are performed to allow *swc2* instruction to be reinserted into EX stage from SWQ. The ciphertext will then be stored into the desired memory location at the MEM stage. The stall signal (bswQ_pipe_stall) will be deactivated as soon as the 128-bit ciphertext is stored into the memory, where it is hold for four clock cycle (equivalent to four *swc2* instructions). From this point onward, the pipeline will resume its operation from the instruction that was previously stalled in ID stage.

3.2.6.1 Coprocessor 2 Queue (CP2Q) Design

The Coprocessor 2 Queue (CP2Q) is responsible in keeping track of the CP2 related instructions and the status of CP2. Based on the status signal from RISC32 pipeline and CP2, the CP2Q determines whether to execute detected CP2 related instructions. The Figure 3.52 shows the CP2Q Block Top Level interface. Pin description for the CP2Q Block is provided at Table 3.39.

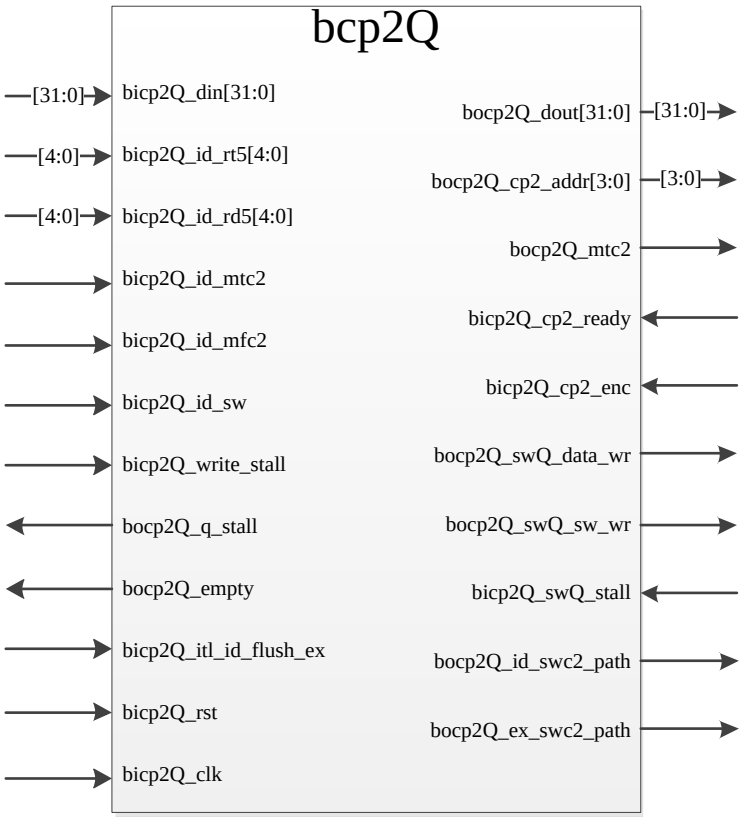


Figure 3.52: Top-level Interface for CP2Q Block

Table 3.39: Pin Description for CP2Q Block Interface

Pin Name: bocp2Q_dout[31:0] Pin Direction: Output Pin Size: 32 bits Source→ Destination: CP2Q Block → CP2 Block Pin Function: Output port for 32-bit data to be processed by CP2
Pin Name: bocp2Q_cp2_addr[3:0] Pin Direction: Output Pin Size: 4 bits Source → Destination: CP2Q Block → CP2 Block Pin Function: Output port for 4-bit CP2 register file address
Pin Name: bocp2Q_mtc2 Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2Q Block → CP2 Block Pin Function: Output control signal for CP2 to perform *mtc2* instructions 0: *mtc2* instruction not requested 1: *mtc2* instruction requested
Pin Name: bocp2Q_swQ_data_wr Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2Q Block → SWQ Block Pin Function: Output control signal to SWQ to store CP2 output 0: Do not store CP2 ciphertext output 1: Store CP2 ciphertext output
Pin Name: bocp2Q_swQ_sw_wr Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2Q Block → SWQ Block Pin Function: Output control signal to SWQ to store calculated address for *swc2* instruction 0: Do not store address calculated by Arithmetic Logic Block 1: Store address calculated by Arithmetic Logic Block
Pin Name: bocp2Q_id_swc2_path Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2Q Block → RISC32 Pipeline Pin Function: Output control signal to RISC32 pipeline ID stage to select read source as CP2 output 0: *swc2* is queued by CP2Q. Do not select CP2 output as read source 1: *swc2* bypassed CP2Q. Select CP2 output as read source
Pin Name: bocp2Q_ex_swc2_path Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2Q Block → SWQ Block Pin Function: Output control signal to RISC32 pipeline EX stage to allow CP2 output from ID stage bypass SWQ 0: *swc2* is queued by CP2Q. SWQ wait for control signal from CP2Q 1: *swc2* bypassed CP2Q. SWQ ignore control signal from CP2Q
Pin Name: bocp2Q_q_stall Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2Q Block → Interlock Block Pin Function: Output control signal to Interlock Block to indicate currently CP2Q is full 0: CP2Q is not full. RISC32 pipeline can continue fetch new instruction 1: CP2Q is full. RISC32 pipeline should be stalled to prevent fetching of new instruction

Continued from Table 3.39

Pin Name: bocp2Q_empty Pin Direction: Output Pin Size: 1 bit Source → Destination: CP2Q Block → Programmable Interrupt Controller Pin Function: Output control signal to Programmable Interrupt Controller to indicate currently CP2Q is empty 0: CP2Q is not empty 1: CP2Q is empty. Trigger CP0 if Queue System Interrupt is enabled
Pin Name: bicp2Q_cp2_ready Pin Direction: Input Pin Size: 1 bit Source → Destination: CP2 Block → CP2Q Block Pin Function: Input status signal to indicate current status of CP2 0: CP2 is busy. Do not dispatch instruction from CP2Q and queue incoming CP2 related instructions 1: CP2 is idle. Dispatch instruction from CP2Q if queue is not empty
Pin Name: bicp2Q_cp2_enc Pin Direction: Input Pin Size: 1 bit Source → Destination: CP2 Block → CP2Q Block Pin Function: Input control signal to indicate if encryption command (*0x2*) is currently requested in CP2 0: No encryption requested. Dispatch instruction from CP2Q if queue is not empty 1: Encryption requested. Do not dispatch instruction from CP2Q and queue incoming CP2 related instructions
Pin Name: bicp2Q_swQ_stall Pin Direction: Input Pin Size: 1 bit Source → Destination: SWQ Block → CP2Q Block Pin Function: Input control signal to indicate if SWQ is performing *swc2* reinserting into RISC32 pipeline and CP2Q Block should be stalled 0: No *swc2* reinserting by SWQ. CP2Q operate as usual 1: SWQ is reinserting *swc2*. Stall writing operation for CP2Q.
Pin Name: bicp2Q_din[31:0] Pin Direction: Input Pin Size: 32 bits Source → Destination: RISC32 Register File → CP2Q Block Pin Function: Input port for operand fetched by decoding *mtc2* instructions
Pin Name: bicp2Q_id_rt5[4:0] Pin Direction: Input Pin Size: 5 bits Source → Destination: RISC32 Pipeline → CP2Q Block Pin Function: Input port for CP2 register file address for *swc2* instruction
Pin Name: bicp2Q_id_rd5[4:0] Pin Direction: Input Pin Size: 5 bits Source → Destination: RISC32 Pipeline → CP2Q Block Pin Function: Input port for CP2 register file address for *mtc2* or *mfc2* instruction
Pin Name: bicp2Q_id_mtc2 Pin Direction: Input Pin Size: 1 bit Source → Destination: Control-path Unit → CP2Q Block Pin Function: Input control signal from Control-path Unit when *mtc2* instruction is decoded 0: No *mtc2* decoded by Control-path Unit 1: *mtc2* decoded by Control-path Unit

Continued from Table 3.39

Pin Name: bicp2Q_id_mfc2 Pin Direction: Input Pin Size: 1 bit Source → Destination: Control-path Unit → CP2Q Block Pin Function: Input control signal from Control-path Unit when *mfc2* or *swc2* is decoded 0: No *mfc2* or *swc2* instruction decoded Control-path Unit 1: *mfc2* or *swc2* instruction decoded by Control-path Unit
Pin Name: bicp2Q_write_stall Pin Direction: Input Pin Size: 1 bit Source → Destination: RISC32 Pipeline → CP2Q Block Pin Function: Input status signal to prevent CP2Q Block to queue any incoming CP2 instructions when instruction cache miss occurred 0: No global stall signal detected. CP2Q Block operates as usual 1: Global stall signal detected. Stall CP2Q Block from queueing any incoming instruction
Pin Name: bicp2Q_itl_id_flush_ex Pin Direction: Input Pin Size: 1 bit Source → Destination: Interlock block → CP2Q Block Pin Function: Input status signal to prevent CP2Q Block to queue any incoming CP2 instructions when load use data hazard occurred 0: No load use hazard detected by Interlock Block. CP2Q operates as usual 1: Load use hazard detected by Interlock Block. Stall CP2Q Block from queueing any incoming instructions
Pin Name: bicp2Q_rst Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Reset → CP2Q Block Pin Function: Reset signal for CP2Q Block
Pin Name: bicp2Q_clk Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Clock → CP2Q Block Clock Source for CP2Q Block

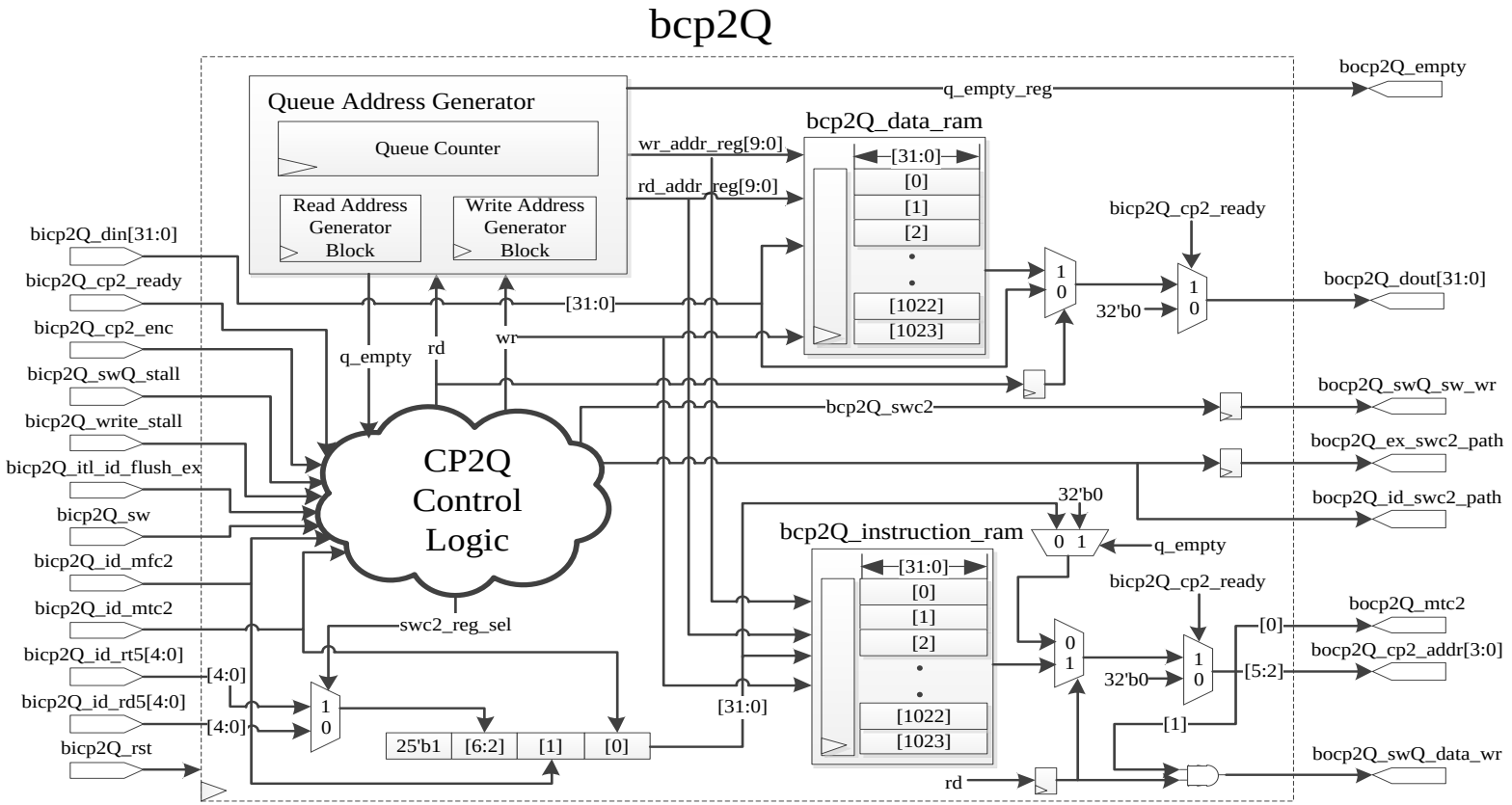


Figure 3.53: Microarchitecture of CP2Q Block

The Figure 3.53 illustrates the microarchitecture of CP2Q Block. The main components in CP2Q Block are CP2Q Data RAM (bcp2Q_data_ram), CP2Q Instruction RAM (bcp2Q_instruction_ram), Queue Address Generator and CP2Q Control Logic.

The CP2Q Instruction RAM stores CP2 instructions decoded form, which is the *mtc2* control signal, *mfc2* control signal and CP2 register file address. The CP2Q Data RAM stores operand for *mtc2* instructions. Both CP2Q Instruction RAM and Data RAM is implemented with Block RAM technology available on Digilent Nexys 4 DDR Artix-7 FPGA Board, and has a total of 1024 word locations. The number 1024 is estimated by calculating the worst-case scenario of maximum unexecuted encryption task based on the recommend range of *N* byte during software pattern analysis in Section 3.2.4. With *swc2* implemented, an encryption request consists of four *mtc2* for plaintext transfer, one *mtc2* for encryption command transfer and four *swc2* for ciphertext reading. By assuming the worst-case, the largest N = 1024B is selected. The 1024B is arranged into a total of 64 encryption request (16B per encryption). This is equivalent to a maximum of 576 CP2 instructions to be expected for the encryption task. Since the Block RAM comes in the size of 32-bit x 512 and 32-bit x1024, the location of CP2Q Instruction RAM and Data RAM is determined to be 1024 to meet the worst-case scenario.

The Queue Address Generator is responsible to generate the read and write addresses for both CP2Q Data RAM and Instruction RAM. It also keeps track the number of instructions currently queued in the CP2Q Data RAM and

Instruction RAM. The CP2Q Control Logic assess the input status signals from CP2, SWQ and RISC32 pipeline to generate control signals for read write operation in CP2Q. All main components in CP2Q works together to form a First-In-First-Out (FIFO) queue. This ensures the queued CP2 instruction to be executed In-Order with respect to their sequence in the user program. It should be noted as well, the CP2Q Instruction RAM and Data RAM has separate read port and write port. This allows the CP2Q to dispatch older instruction from the head of the FIFO if the CP2 is ready, at the same time, queue any incoming instruction by appending at the end of FIFO. Figure 3.54 shows the algorithm flowchart for CP2Q Control Logic.

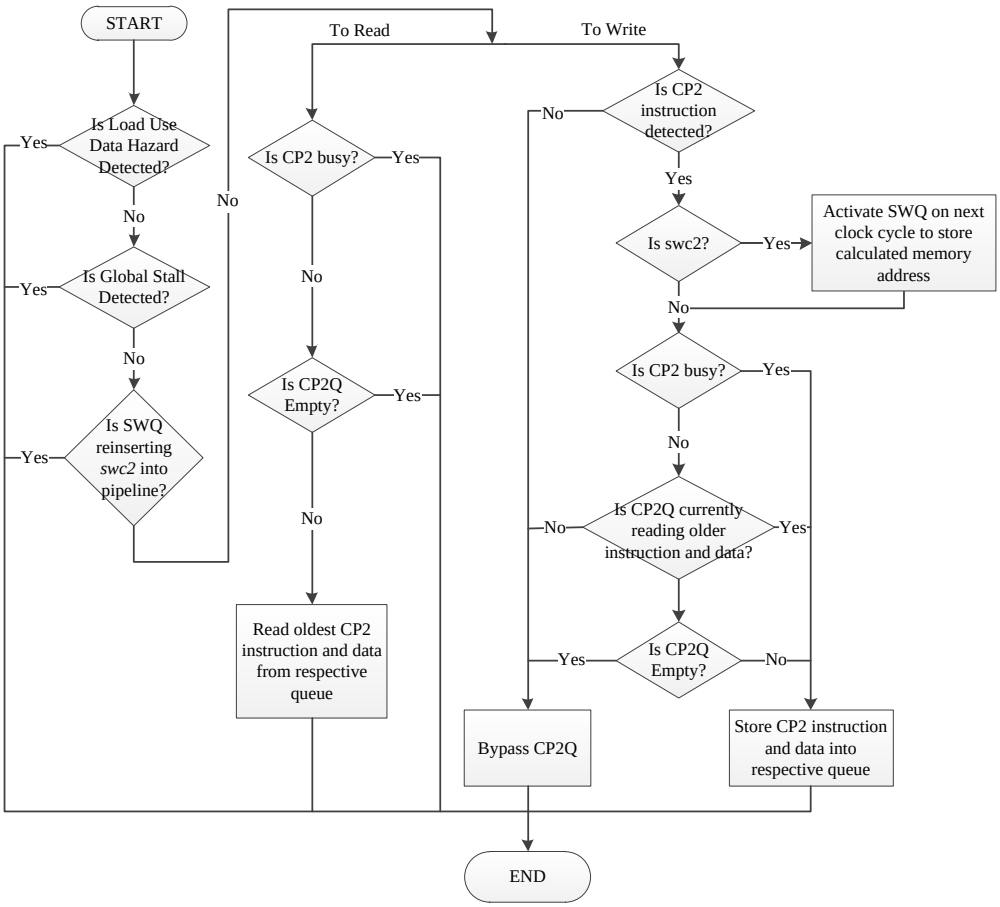


Figure 3.54: Algorithm Flowchart for CP2Q Control Logic

3.2.6.2 Store Word Queue (SWQ) Design

The Store Word Queue (SWQ) is responsible in storing address and reading data for *swc2* instruction that is previously queued in CP2Q. Based on the control signals generated from CP2Q, the SWQ determines whether to queue the *swc2* instruction from RISC32 pipeline. The execution of queued *swc2* from SWQ is also controlled by CP2Q. Figure 3.55 shows the top-level interface for SWQ Block. Pin description for top-level interface of SWQ Block is listed in Table 3.40.

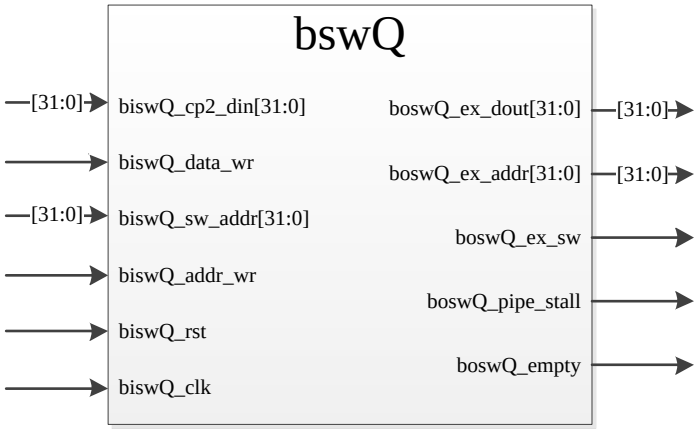


Figure 3.55: Top-Level Interface for SWQ Block

Table 3.40: Pin Description for SWQ Block Interface

Pin Name: boswQ_ex_dout[31:0] Pin Direction: Output Pin Size: 32 bits Source→ Destination: SWQ Block → RISC32 EX stage Pin Function: Output port for 32-bit data to be stored into specified memory location
Pin Name: boswQ_ex_addr[31:0]] Pin Direction: Output Pin Size: 32 bits Source → Destination: SWQ Block → Address Decoder → RISC32 EX Stage Pin Function: Output port for 32-bit address to store output data of CP2. It is also decoded by Address Decoder to select between RISC32 Data RAM and integrated I/O module.

Continued from Table 3.40

Pin Name: boswQ_ex_sw Pin Direction: Output Pin Size: 1 bit Source → Destination: SWQ Block → Address Decoder Pin Function: Output control signal activated when reinserting *swc2* instructions into RISC32 pipeline. Required for Address Decoder to activate memory module at MEM stage for store data operation 0: No *swc2* instruction reinsertion. 1: *swc2* instruction is reinserting into RISC32 pipeline
Pin Name: boswQ_pipe_stall Pin Direction: Output Pin Size: 1 bit Source → Destination: SWQ Block → RISC32 Pipeline Pin Function: Output status signal activated when reinserting *swc2* instructions. This signal stalls the IF and ID stage to prevent new instruction entering pipeline. The CP2Q is also stalled, to prevent any execution of previously queued instructions. The EX stage is flushed, to allow *swc2* to reuse the existing *sw* transfer path. 0: No *swc2* instruction reinsertion. 1: *swc2* instruction is reinserting into RISC32 pipeline
Pin Name: boswQ_empty Pin Direction: Output Pin Size: 1 bit Source → Destination: SWQ Block → Programmable Interrupt Controller Pin Function: Output status signal to indicate SWQ is currently empty. 0: SWQ is not empty. 1: SWQ is empty. Trigger CP0 to raise exception if Queue System Interrupt is enabled
Pin Name: biswQ_cp2_din[31:0] Pin Direction: Input Pin Size: 32 bits Source → Destination: CP2 Block → SWQ Block Pin Function: Input port for ciphertext output from CP2 Block. To be stored into memory module
Pin Name: biswQ_data_wr Pin Direction: Input Pin Size: 1 bit Source → Destination: CP2Q Block → SWQ Block Pin Function: Input control signal to store current ciphertext output from CP2 Block. 0: Do not store output from CP2 Block 1: Store current output from CP2 Block
Pin Name: biswQ_sw_addr[31:0] Pin Direction: Input Pin Size: 32 bits Source → Destination: Arithmetic Logic Block → SWQ Block Pin Function: Input port for memory address of *swc2* queued in CP2Q Block. To be used when *swc2* is reinserted back into RISC32 pipeline
Pin Name: biswQ_addr_wr Pin Direction: Input Pin Size: 1 bit Source → Destination: CP2Q Block → SWQ Block Pin Function: Input control signal to store calculated output from Arithmetic Logic Block as memory address encoded with *swc2* instruction 0: Do not store output from Arithmetic Logic Block 1: Store current output from Arithmetic Logic Block

Continued from Table 3.40

Pin Name: biswQ_rst Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Reset → SWQ Block Pin Function: Reset Signal for SWQ Block
Pin Name: biswQ_clk Pin Direction: Input Pin Size: 1 bit Source → Destination: Global Clock → SWQ Block Pin Function: Clock Source for SWQ Block

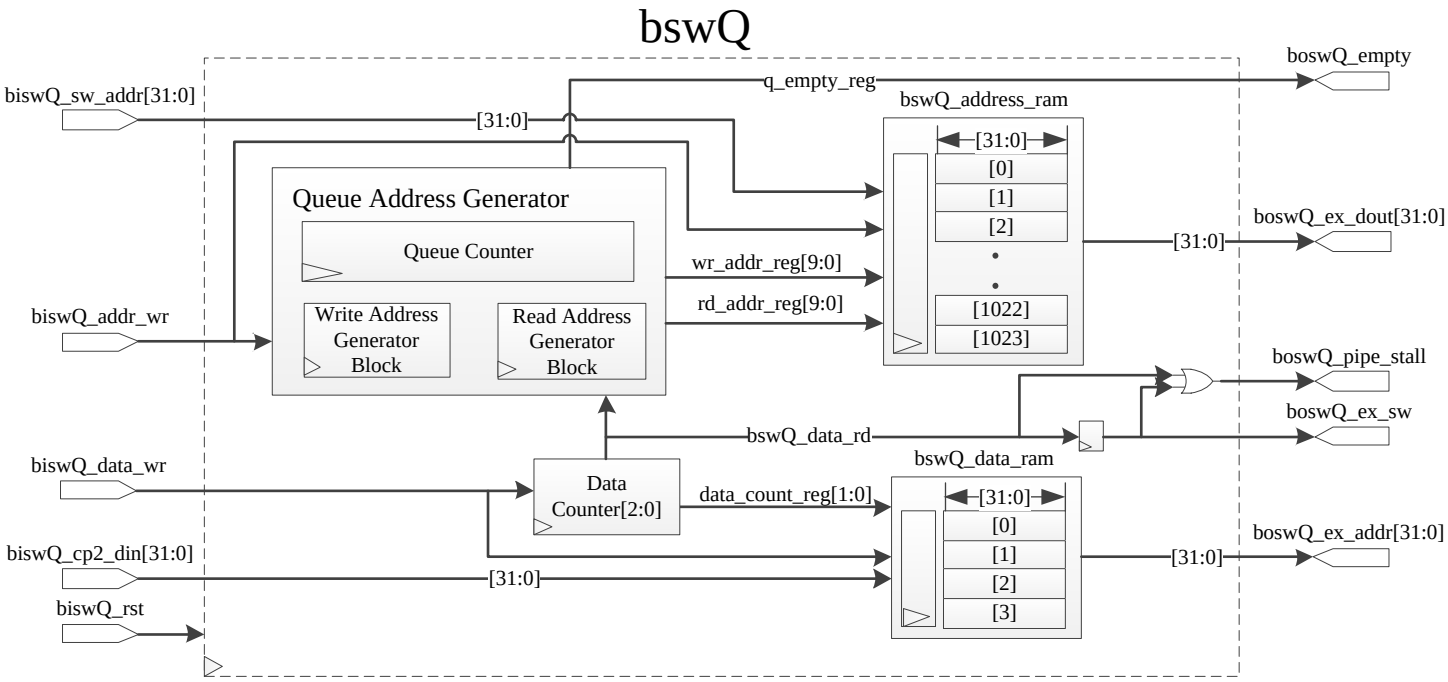


Figure 3.56: Microarchitecture of SWQ Block

Figure 3.56 shows the microarchitecture for SWQ Block. The SWQ Block contains four main components, which is the SWQ Address RAM, SWQ Data RAM, Queue Address Generator and Data Counter. The SWQ Address RAM stores the memory address of the *swc2* instruction if it is queued in CP2Q. The SWQ Address RAM is implemented using Block RAM technology, and has a total location of 1024 in word size. The number of locations corresponds to the size implemented for CP2Q Instruction RAM and CP2Q Data RAM. The SWQ Data RAM stores the ciphertext output from CP2. Unlike SWQ Address RAM, the SWQ Data RAM is implemented with Distributed RAM technology, and only has four locations in total. Each of these locations are in word size, and all four locations will be concatenated to form 128-bit in total, which is the size of ciphertext from CP2. Since the number of locations for SWQ Data RAM is limited, the SWQ has to stall CP2Q when it has accumulated a complete ciphertext. This is to prevent CP2Q from executing any previously queued *swc2* and overwrite the SWQ Data RAM. The Queue Address Generator for SWQ generates read and write address for SWQ Address RAM. It also keeps track the number of address currently queued. The Data Counter keeps track the number of data currently accumulated in SWQ Data RAM. The Data Counter also serves as the read and write address for SWQ Data RAM.

The SWQ however, does not have Control Logic that is present in CP2Q. This is because the main controls (biswQ_data_wr and biswQ_addr_wr) for SWQ has been handled by the CP2Q Control Logic. The internal operation of SWQ is presented as flowchart in the Figure 3.57.

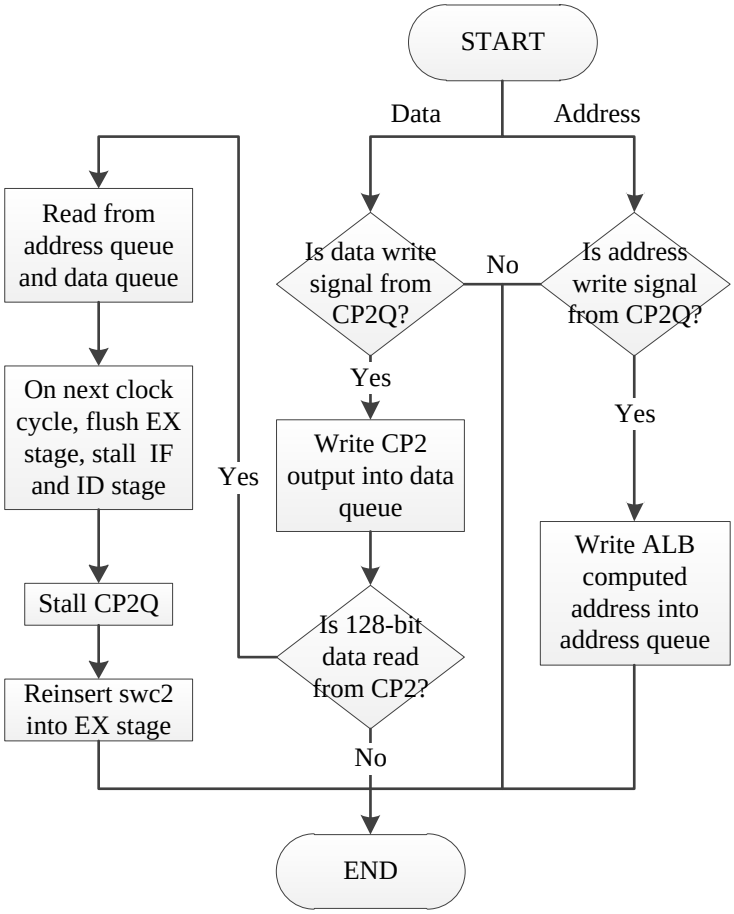


Figure 3.57: Internal Operation of SWQ

3.3 Summary

This chapter has discussed the compiler development to realize the RISC32 toolchain. The RISC32 toolchain is established around the LLVM retargetable compiler. Currently, MIPS was one of the supported backend in LLVM. While RISC32 was designed to be MIPS-ISA compatible, the supported MIPS instruction set in LLVM is up to the latest MIPS generation, which has much more instruction than RISC32 could support. As such, a suitable sub-target, MIPS II in the MIPS Backend of LLVM has been selected, to narrow down the instruction set available for code generation. However, the RISC32 and MIPS II instruction set is not completely compatible. Special transformation routine was discussed to convert the unsupported instructions in MIPS II into RISC32 equivalent instructions. The CP2 intrinsic function was also implemented, to allow compatible code generation that fits the usage of integrated CP2 and Queue System in RISC32. Interrupt Service Routine (ISR) programming feature was implemented to allow programming of interrupt-based applications, which is commonly used to realize I/O transactions between IoT sensor node and its surrounding IoT devices.

This chapter has also discussed the CP2 integration into RISC32. The CP2 integrated is an AES-128 Coprocessor. With the introduction of CP2, 2 new instructions, namely Move to Coprocessor 2 (*mtc2*) and Move from Coprocessor 2 (*mfc2*) has been created to allow data transfer between the RISC32 pipeline and CP2 register file. However, the CP2 requires 55 clock cycle to perform encryption on its 16-byte/128-bit plaintext input. After analysing the typical IoT pattern, a solution was derived to utilize the long

encryption clock cycle. This solution utilizes the Counter (CTR) AES encryption mode and the idea of dynamic scheduling, to allow both Data Acquisition and Data Processing task to execute in parallel. A Queue System was proposed to realize the dynamic scheduling idea. Additionally, Store Word from Coprocessor 2 (*swc2*) instruction was introduced to resolve the potential data hazard faced by the Queue System.

CHAPTER 4

SYSTEM VERIFICATION

The developed RISC32 integrated with CP2 and Queue System is synthesized and implemented onto the Xilinx Artix-7 XC7A100T FPGA in Digilent Nexys 4 DDR board using Xilinx Vivado HLx 2017.2 IDE. All C test programs developed are compiled using the customized RISC32 Toolchain developed in this work, which is based on LLVM version 5.0. The RISC32 Toolchain is installed on a host computer with Ubuntu 16.04 LTS Operating System. The Figure 4.1 shows the RISC32 microarchitecture implemented with CP2 and Queue System. Table 4.1 shows the FPGA resource consumption for microarchitecture of RISC32 with CP2 and Queue System. Resource overhead between each microarchitecture is calculated in Table 4.2.

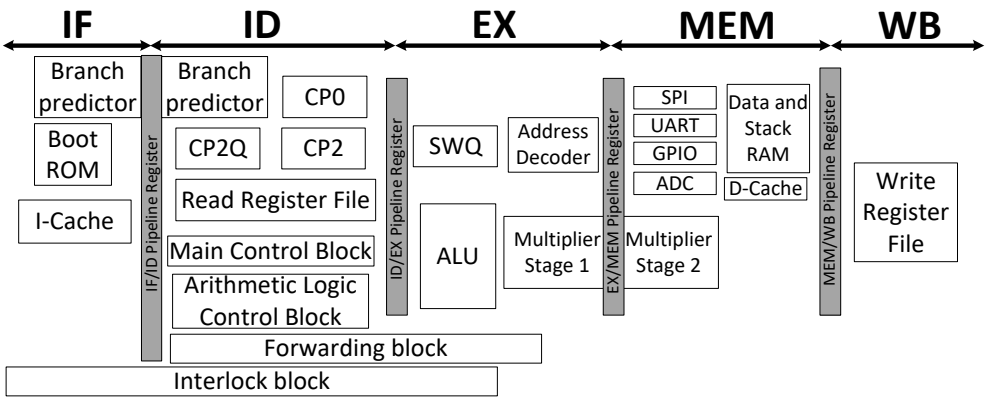


Figure 4.1: RISC32 Microarchitecture Components with CP2 and Queue System

Table 4.1: FPGA Resource Usage for RISC32 with CP2 and Queue System

FPGA Resources	Microarchitecture		
	RISC32[1]	RISC32_CP2-NQ[2]	RISC32_CP2-Q[3]
LUT	6046	7573	7849
LUTRAM	311	311	343
FF	2574	4661	4745
BRAM	3.50	3.50	5.00
IO	49	49	50
BUFG	2	2	2

**Note:*

1. *Original RISC32 without CP2 and Queue System*
2. *RISC32 implemented with CP2 but without Queue System*
3. *RISC32 implemented with CP2 and Queue System*

Table 4.2: FPGA Resource overhead comparison

FPGA Resources	Microarchitecture		
	RISC32 to RISC32_CP2-NQ	RISC32 to RISC32_CP2-Q	RISC32_CP2-NQ to RISC32_CP2-Q
LUT	25.25%	29.82%	3.64%
LUTRAM	0.00%	10.29%	10.29%
FF	81.08%	84.34%	1.80%
BRAM	0.00%	42.86%	42.86%
IO	0.00%	2.04%	2.04%
BUFG	0.00%	0.00%	0.00%

**Resource overhead = ((improved implementation / existing implementation)- existing implementation) * 100%*

Table 4.3: Longest Timing Delay for Each Stage for Different RISC32 Microarchitecture

Microarchitecture	Pipeline Stage				
	IF	ID	EX	MEM	WB
RISC32	14.537ns	13.309ns	14.668ns	17.830ns	2.556ns
RISC32_CP2-NQ	14.287ns	14.347ns	14.486ns	17.945ns	2.763ns
RISC32_CP2-Q	13.986ns	15.655ns	16.436ns	18.541ns	2.936ns

From the timing analysis result in Table 4.3, it could be seen in RISC32_CP2-Q microarchitecture, the longest timing delay is now 18.541 ns. However, this timing is still within the minimum clock period requirement of RISC32, which is 20 ns. This shows that integrating the CP2 and Queue System does not impose a huge effect on the overall RISC32 performance.

4.1 Functional Verification

4.1.1 RISC32 Toolchain Compilation Verification

The RISC32 Toolchain was established using the retargetable LLVM compiler. As the MIPS backend was currently implemented in LLVM, it could be used for code compilation for RISC32. However, the MIPS backend implemented supports more instructions than the existing RISC32 instruction set has. As such among the legal sub-target supported by the MIPS backend, MIPS II was selected as a base for RISC32 code compilation due to their high similarity in instruction set. The following efforts was made on LLVM to use the MIPS II for RISC32 code compilation:

1) Analysis and comparison between the instructions supported by MIPS II and RISC32 instruction set. Discussion on action to be taken for each instruction has been discussed in Chapter 3, Section 3.1.2.

2) Through the analysis, several instructions in MIPS II instruction set which were not supported by the RISC32 are also implemented. The affected instruction groups are Shift-by-Variable and Branch on Conditional. The porting to support compilation for both instruction groups in RISC32 was discussed and verified in Section 3.1.4 and Section 3.1.5 of Chapter 3.

3) With the introduction of CP2 core to RISC32, new instructions (*mtc2*, *mfc2* and *swc2*) were implemented to support data transaction between

the RISC32 pipeline and CP2 Core. These instructions were implemented by default in existing MIPS backend of LLVM. However, to ensure a compatible routine to be compiled for the proper usage of the CP2 core and Queue System, intrinsic functions were implemented. Test programs were developed using these intrinsic functions.

4) The Interrupt Service Routine (ISR) compilation is currently supported in LLVM. However, the compiled ISR output does not conform to the ISR convention of RISC32. Detailed discussion on the ISR compilation for both LLVM and RISC32 can be found in Chapter 3, Section 3.1.7.

With all the implementations performed above, the LLVM is now ready to generate compatible code to be executed on RISC32. All of the C test programs in this Chapter 4 is compiled using this RISC32 Toolchain.

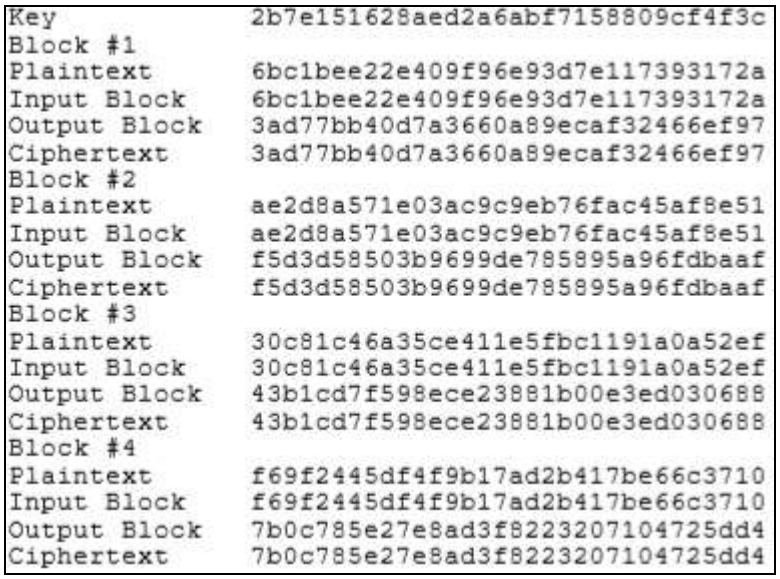
4.1.2 Coprocessor 2 (CP2) and Queue System Verification

The CP2 core integrated into RISC32 IoT processor performs AES-128 Encryption. A C test program was developed to perform encryption using the CP2 core. The C test program was written using the CP2 intrinsic functions as discussed in Chapter 3, Section 3.1.6. The test program is setup as follows:

1) Initialize control register (UARTCR) of UART Controller with baud rate of 9600.
2) Perform secret key expansion using CP2 key expansion intrinsic function, *__builtin_risc32_aes128_keyinit ()*.
3) Load 128-bit of input test vector plaintext into a 4 word 128-bit array.
4) Perform AES-128 Encryption using CP2 encryption intrinsic function, *__builtin_risc32_aes128_enc ()*.
5) Prepare address for next 128-bit of input test vector plaintext.
6) Repeat Step 3 to 5 until all test vector is encrypted
7) Transmit all encrypted ciphertext through UART to host computer
8) Repeat from Step 3 until Step 7

The test program first sets up the UART control register, to allow the transmission of the encrypted ciphertext. Detailed information of the RISC32 UART Controller can be found in the work by Kiat (2018). The test program will then prepare the round keys, before starting encryption on the input test

vectors. The secret key and input vectors are shown in the Figure 4.2. The encrypted value is then sent back to a host computer to verify their correctness, which is compared against the test vector provided for AES-128 in the NIST SP800-38 (2007) document. The sample ciphertext output is provided in Figure 4.2.



Key	2b7e151628aed2a6abf7158809cf4f3c
Block #1	
Plaintext	6bc1bee22e409f96e93d7e117393172a
Input Block	6bc1bee22e409f96e93d7e117393172a
Output Block	3ad77bb40d7a3660a89ecaf32466ef97
Ciphertext	3ad77bb40d7a3660a89ecaf32466ef97
Block #2	
Plaintext	ae2d8a571e03ac9c9eb76fac45af8e51
Input Block	ae2d8a571e03ac9c9eb76fac45af8e51
Output Block	f5d3d58503b9699de785895a96fdbaaf
Ciphertext	f5d3d58503b9699de785895a96fdbaaf
Block #3	
Plaintext	30c81c46a35ce411e5fbc1191a0a52ef
Input Block	30c81c46a35ce411e5fbc1191a0a52ef
Output Block	43b1cd7f598ece23881b00e3ed030688
Ciphertext	43b1cd7f598ece23881b00e3ed030688
Block #4	
Plaintext	f69f2445df4f9b17ad2b417be66c3710
Input Block	f69f2445df4f9b17ad2b417be66c3710
Output Block	7b0c785e27e8ad3f8223207104725dd4
Ciphertext	7b0c785e27e8ad3f8223207104725dd4

Figure 4.2: AES-128 Test Vector
Source: NIST SP800-38, 2007

The same test program is used to test both the CP2 core encryption and also the Queue System proposed. For CP2 core verification only, the CP2 encryption intrinsic function is compiled with 55 *NOPS* to wait for CP2 to complete its encryption. The main purpose is to test the CP2 instructions (*mtc2*, *mfc2* and *swc2*). If these instructions can be carried out successfully, the encryption should be performed successfully and yield the correct ciphertext output as compared to Figure 4.2. As for Queue System, the CP2

encryption intrinsic function will be compiled without the 55 *NOPS* as discussed in Chapter 3, Section 3.1.6. This will induce the Queue System to queue up the CP2 instructions when the CP2 is busy. To verify the functionality of the Queue System, the final ciphertext output stored into the data memory of RISC32 will be checked if it matches the expected ciphertext in Figure 4.2. If the output ciphertext is matched, this indicates the CP2Q and SWQ of the Queue System had successfully rescheduled the execution of the CP2 encryption task while the CP2 is busy. The Figure 4.3 shows the received ciphertext through UART transmission. The ciphertext is read from the data memory of RISC32.

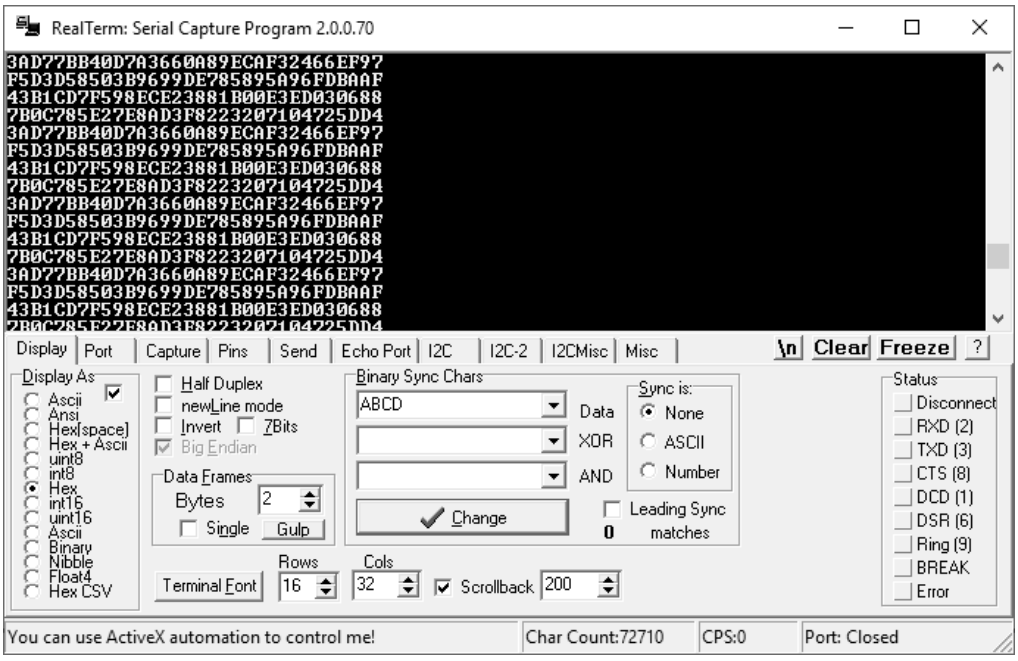


Figure 4.3: Ciphertext received from UART on the host computer. Data is displayed using RealTerm

4.2 Performance Analysis

To evaluate the effectiveness of the proposed Queue System, three test programs were developed. The evaluation metrics that are being assessed are execution time and energy consumption of each test program. The test combinations are shown in Table 4.4

Table 4.4: Test Combination for Performance Analysis

Test Program	Hardware Architecture	Test Case
tiny-AES-C (kokke, 2014)	RISC32	T_C
Assembly AES	RISC32	T_ASM
AES using CP2 instructions	RISC32_CP2-NQ	T_CP2-NQ
AES using CP2 instructions	RISC32_CP2-Q	T_CP2-Q

Figure 4.4: Test Program Software Pattern

The test programs are developed with the software pattern shown in Figure 4.4. Note that test program for T_C and T_ASM implements AES encryption in software (C and assembly language). T_CP2-NQ is assessed on RISC32 with CP2 core but without Queue System. T_CP2-Q is performed on RISC32 with CP2 Core and incorporates proposed Queue System to overlap data acquisition and AES encryption for better speed and energy performance. Both T_CP2-NQ and T_CP2-Q executes the same software program, but they achieved varied performance due to different hardware architecture.

4.2.1 Timing Performance

The total execution clock cycle (C.C) count for each test case is shown in Table 4.5. This is measured by executing the data processing part (Encrypt, Data Acquisition and XOR) of the test program. The data sizes (*N*) used in the experiment are 256 Byte, 512 Byte and 1024 Byte.

Table 4.5: Data Processing Execution Time (C.C) For Each Test Case

Test Case	Data Size (Byte)		
	256	512	1024
T_C	663674	1326346	2651690
T_ASM	355741	708989	1415485
T_CP2	2471	4935	9863
T_CP2-Q	1671	3335	6663

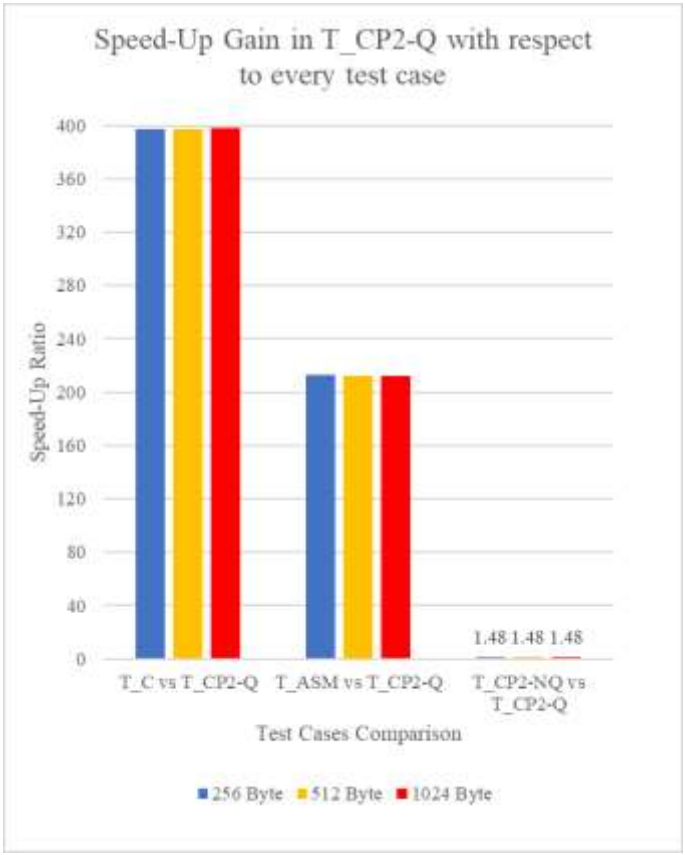


Figure 4.5: Speed-Up achieved in T_CP2-Q compared to other test cases

The experimental results shown in Figure 4.5 are the speed-up ratio achieved in T_CP2-Q against other test cases. The speed-up ratio refers to the ratio of total clock cycle count between original implementation and improved implementation. Note that result for T_CP2-Q shows that it is much faster than the software implementation in T_C and T_ASM, with more than 200x speed up. This is not surprising, as software implementation of AES is fundamentally slower than hardware. In software AES, the 128-bit operation has to be broken down to multiple serial 32-bit operation due to the maximum data size supported in RISC32, which is 32-bit only. Furthermore, a single substitution box in AES is a 256 Byte lookup-table. It is impractical to implement multiple substitution box on limited memory of IoT sensor node. Hence, common practice of software AES only implements a single substitution box. Every substitution process of 32-bit operation have to be further spilt down into 4 load store operation. Executing 4 parallel load store instruction in software is not possible, as there is only single core in RISC32. Also, there are no single load store instruction that supports parallel 4-byte load store operation between random memory location in RISC32.

In the case of hardware implemented AES, more parallel operations can be performed on a single transformation AES round. While the AES operations are fundamentally performed on words (32-bit), the 128-bit operation can be broken into four parallel 32-bit operation in the hardware. Hence, the opportunity to perform parallel operations in hardware implementation could achieve better performance compared to software implementation. Comparing between hardware implementations

(RISC32_CP2-NQ vs RISC32_CP2-Q), a significant improvement (1.48x speed up) is observed for all data sizes. This speed-up is achieved is because there is no data dependency between data acquisition and encryption in CTR mode. Hence, both tasks can be effectively overlapped. It can be concluded that, the proposed Queue System in RISC32_CP2-Q of test case T_CP2-Q, can reorder the execution sequence of CP2 instructions in the program, at the same time effectively overlaps the encryption task with other processing task (data acquisition and XOR), eventually achieve good speed performance against RISC32_CP2-NQ in test case T_CP2-NQ.

4.2.2 Energy Consumption

The energy measurement is obtained by monitoring the current drawn during the execution of the data processing (Encrypt, Data Acquisition and XOR) part of the test program. The current drawn measurement is derived from the voltage difference across a 0.01 Ω shunt-resistor connected serially between Digilent Nexys 4 Artix-7 FPGA Board and a 1.0V power supply. The voltage difference across the shunt-resistor is amplified with instrumentation amplifier, INA215 configured with 75 amplifier gain.

For measurement purpose, the test program was written to continuously loop the data processing activity (Encrypt, Data Acquisition and XOR). To identify the starting and ending of the data processing activity, the test program sets a GPIO output pin to HIGH upon starting the encryption (T1). Upon completion of the encryption of final CTR value, the GPIO output pin is toggle to LOW (T2) so that it can be set HIGH again upon the start of new encryption. The region between the two rising-edges of the waveform marks the region for data processing activity used to measure the energy consumption. Both the amplified voltage difference and GPIO output pin are monitored using Tektronix TBS1202B-EDU Oscilloscope at Channel 1 (voltage across 0.01 Ω shunt-resistor) and Channel 2 (GPIO trigger) respectively. Figure 4.6 shows the screenshot from oscilloscope during the measurement for T_CP2-Q with data size of 256 Byte. Voltage readings along the two rising-edges are recorded to calculate the energy consumption for each test case.

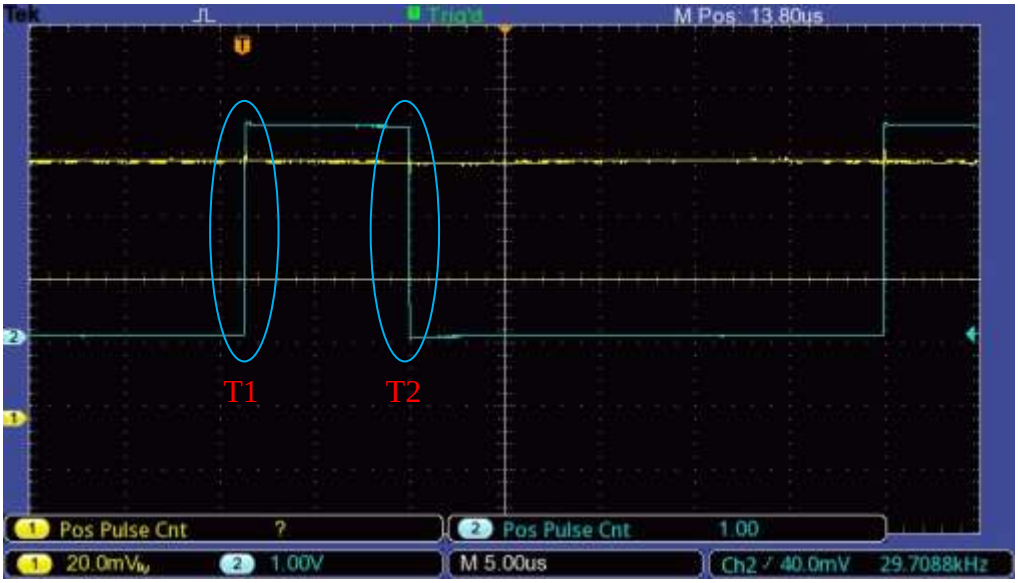


Figure 4.6: Screenshot during energy measurement for T_CP2-Q

To obtain the current drawn at each time instance, the formula $I = V / R / 75$ is being used, where R refers to the 0.01 Ω shunt-resistor and 75 is the amplifier gain. With the current drawn (I) obtained, power is calculated using $P = V * I$, where V refers to the power supply voltage (1.0V). Energy is then calculated using the formula $E = P * t$, where t refers to the time interval between two measurement data. Since the time is in discretized form, $E = \sum_{t=0}^{N} P \times t$, where N is the total time points within the measurement region. The energy consumption measured for each test case is shown in Table 4.6.

Table 4.6: Data Processing Energy Consumption (mJ) For Each Test Case

Test Case	Data Size (Byte)		
	256	512	1024
T_C	0.9051mJ	1.7966mJ	3.5933mJ
T_ASM	0.5031mJ	0.9909mJ	2.0080mJ
T_CP2-NQ	0.0046mJ	0.0089mJ	0.0178mJ
T_CP2-Q	0.0037mJ	0.0073mJ	0.0145mJ

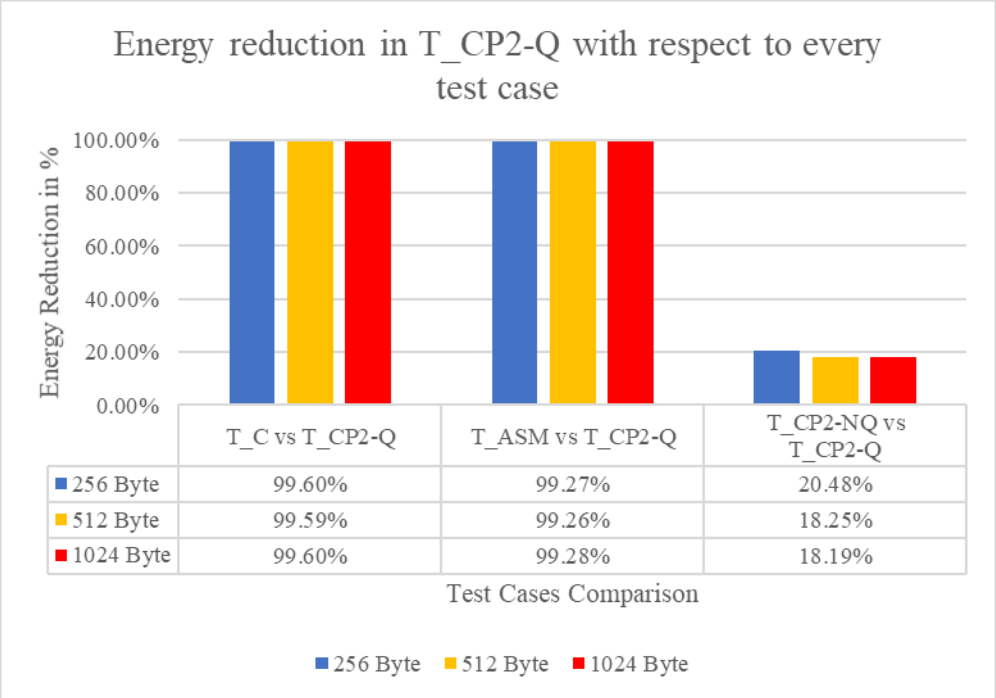


Figure 4.7: Energy reduction achieved in T_CP2-Q compared to other test cases

Figure 4.7 shows the energy reduction achieved in T_CP2-Q when compare to other test cases. Note that result of T_CP2-Q shows ~99% energy reduction against software implementation in test case T_C and T_ASM. Although RISC32_CP2-Q used in T_CP2-Q does have additional hardware consumption (resulting in more static power) as shown in Table 4.2, it also reduces the data processing time significantly, eventually reduce the energy consumption when compared to software implementation. Comparing both hardware implementations (RISC32_CP2-NQ vs RISC32_CP2-Q), the averaged energy reduction for all data size is ~19%. This shows that the proposed Queue System (in RISC32_CP2-Q) not only has faster performance than conventional hardware implementation without Queue System (RISC32_CP2-NQ), but also better in terms of energy efficiency.

4.3 Summary

In this chapter, the RISC32 Toolchain and RISC32 integrated with CP2 and Queue System has been verified its functionality. The RISC32 Toolchain is able to transform unsupported instructions to RISC32 compatible instruction sets. The Queue System proposed successfully rescheduled the encryption task when CP2 is busy, to ensure encryption could be correctly performed in an orderly fashion. The RISC32 integrated with CP2 and Queue System was assessed for its performance in terms of execution time and energy reduction by testing with a typical IoT software pattern. With the Queue System, a 1.48x speed-up and ~19% of energy reduction was achieved. This is an important achievement for IoT applications, which stresses on low latency communication and energy efficiency.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this research work, a C compilation toolchain was developed to support RISC32 (Kiat, 2018) IoT processor. This helps in reducing the code development time, as the programmer no longer need to code in assembly language. This also provides rapid development opportunity on RISC32 with the enabling of standard libraries usage that is usually delivered in high-level language. On top of that, a hardware AES core was integrated into the RISC32, which provides confidentiality, allowing the data transmitted within IoT network in encrypted form. A queue system was proposed to further optimize the speed performance of data encryption.

In summary, this dissertation has provided answers to the following research challenges:

1) RISC32 is an IoT processor that lacks security feature. Cryptography algorithms can be implemented in software, but could achieve better performance if implemented as hardware. Since energy efficiency is an important criterion in IoT processor, how should the cryptography core integrated to achieve performance without significantly increasing the energy consumption of IoT processor?

- The AES cryptography core was selected for this research work. Dedicated-path integration technique was opted to ensure the integrated AES core does not face performance cap that is present in shared-path technique. Due to the energy constrain, AES core with single-stage rolled architecture and smaller hardware consumption was selected. As Counter Mode is being used in encryption, only AES-128 encryption and round key generation are implemented. The decryption circuit is not required as most of the IoT applications do not requires sensor node to decrypt data; it only sends data to the gateway device. Most importantly, these decisions could further reduce the hardware resource required for implementation, hence does not introduce significant energy consumption to IoT sensor node.

2) The AES core encryption operation is expected to have data processing latency. This data processing latency was due to the single-stage rolled architecture. The data processing latency indicates that the processor has to wait for the encryption to complete before proceeding to other task, rendering significant idle time. Can this idle time be utilized to perform other tasks, at the same time, improving the speed performance of the overall program?

- The AES core is implemented as Coprocessor 2 (CP2) in RISC32. The CP2 requires 55 clock cycles to perform one block of encryption (128-bit). Considering the software pattern of a typical

IoT applications, RISC32 remains idle for at least 75% of the time during the encryption with CP2. As such, a solution is proposed to overlap the sensor data sampling and encryption task. Overlapping of both tasks is only possible if Counter (CTR) encryption mode is being used. Hence, Queue System is being proposed to realize this overlapping mechanism, which is realized by the hardware. This Queue System will queue the encryption task when CP2 is busy, and re-execute the pending encryption task when the CP2 is ready. This frees up the RISC32 pipeline from waiting the output from CP2 for 55 clock cycle. At the same time, it allows the sensor data sampling to execute while the CP2 performing encryption, overlapping the encryption and data sampling, eventually reducing the overall program execution time.

3) The retargetable compiler framework, LLVM was selected to develop the RISC32 compilation toolchain. The LLVM currently supports code generation for MIPS target machine. However, is the MIPS backend of LLVM completely compatible for RISC32 code generation?

- The MIPS instruction set is an incremental instruction set, where instructions from previous generation MIPS is inherited by the newer MIPS instruction set. The current MIPS backend of LLVM supports up to the latest generation MIPS instruction set. However, the RISC32 only supports a subset of these implemented instructions. By comparing the implemented instructions with

RISC32 instruction set, a suitable sub-target, MIPS II instruction set was selected as a base for RISC32 code generation. However, MIPS II is not completely compatible with RISC32 instruction set. As such, transformation routine was derived to convert the unsupported MIPS II instruction to RISC32 equivalent instructions. To support the compilation for CP2 instructions of RISC32, intrinsic functions were implemented, to generate compatible software routine to ensure proper usage of the CP2 and Queue System. The existing Interrupt Service Routine (ISR) programming in LLVM was also modified to conform to RISC32 ISR programming convention.

4) How was the performance of RISC32 with the Queue System and compilation toolchain?

- The RISC32 integrated with CP2 and Queue System was synthesized and implemented on the Xilinx Artix 7 FPGA Chip on Digilent Nexys DDR4 Development board. Test programs were developed in C language, compiled using the RISC32 toolchain and successfully executed on the RISC32. The assessed performance metrics is program execution speed and energy consumption. The RISC32 with CP2 achieved at least 200x speed-up in terms of program execution when compared to software solution. With the introduction of Queue System to the existing RISC32 with CP2, a further 1.48x speed-up was achieved. In terms

of energy consumption, a reduction of ~99% was shown when comparing hardware solution to the software encryption. A further ~19% energy reduction was shown when RISC32 with CP2 was introduced with Queue System. These result shows, by overlapping the encryption and data sampling task, better performance can be achieved in terms of execution speed at the expense of extra hardware implementation. At the same time, the improved timing performance achieves better energy efficiency. To sum up, these achievements will further aid the RISC32 IoT processor to be implemented as an energy efficient, yet secure IoT sensor node.

In summary, this research work had accomplished all the planned objectives. This greatly improved the capability of RISC32 IoT processor, since it can be programmed through C language, at the same time equipped with AES core with Queue System to perform data encryption.

5.2 Future Work

The established RISC32 toolchain supports code compilation using high-level language (C). The next potential direction is to develop device libraries for the existing I/O controllers (ADC, UART, SPI and GPIO controller) in RISC32. These device libraries will be useful to interface with the common I/O modules such as WiFi, ZigBee and Bluetooth Low Energy (BLE) in IoT applications. With the toolchain, standard benchmarking suite such as CoreMark and Dhrystone can also be compiled and benchmarked on the RISC32 IoT processor. The benchmarking result obtained should be compared with existing IoT processor on the market, to observe the further improvement required on RISC32.

While this research work has been focusing on energy reduction on the RISC32 core alone, the energy consumption during I/O transmission was not explored. The energy consumption pattern during I/O should be studied, as I/O transaction is fundamentally slow. If high energy consumption was contributed by slow I/O transaction, it is a concerning factor that relates to the longevity of the energy source on IoT sensor nodes. Effective communication protocols should be explored to reduce the energy consumption toll contributed by slow I/O transaction to minimum.

In recent years, light-weight cryptosystems have been actively researched, to introduce low-power cryptography core design, and provide decent security level. However, these light-weight cryptosystems are still under review for NIST standardization (2019). As such, AES core still

remains as the widely recognized cryptography standards in the industry. In future, with the standardization finalized, these light-weight cryptosystems should be explored and considered as encryption core to be integrated into RISC32.

LIST OF PUBLICATIONS

1. See, J.C., Lee, W.K., Mok, K.M. and Goh, H.G., 2018. Development of LLVM compilation toolchain for IoT processor targeting wireless measurement applications. In: *2017 IEEE International Conference on Smart Instrumentation, Measurement and Applications, ICSIMA 2017*. pp.1–4.

2. See, J.C., Lee, W.K., Mok, K.M. and Goh, H.G., 2019. RISC32-E: FPGA-based sensor node with queue system to support fast encryption in industrial Internet of Things applications. [Currently under review. Submitted to Wiley International Journal of Circuit Theory and Applications.]

BIBLIOGRAPHY

MIPS, 2016. *MIPS32 ® Architecture For Programmers Volume II : The MIPS32 ® Instruction Set*. [online] Available at: <https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf>.

NIST, 2019. *Lightweight Cryptography | CSRC*. [online] Available at: <https://csrc.nist.gov/projects/lightweight-cryptography> [Accessed 20 Jul. 2019].

Alahakoon, D. and Yu, X., 2016. Smart Electricity Meter Data Intelligence for Future Energy Systems: A Survey. *IEEE Transactions on Industrial Informatics*, 12(1), pp.425–436.

Anwar, H., Daneshtalab, M., Ebrahimi, M., Plosila, J., Tenhunen, H., Dytckov, S. and Beltrame, G., 2014. Parameterized AES-based crypto processor for FPGAs. In: *Proceedings - 2014 17th Euromicro Conference on Digital System Design, DSD 2014*. pp.465–472.

Apple Inc, 2017. *Xcode IDE*. [online] Apple Developer. Available at: <https://developer.apple.com/xcode/features/>.

ARM Limited, 2019. *µVision IDE - Keil*. [online] Available at: <http://www2.keil.com/mdk5/uvision/>.

Arora, H., Gupta, A., Singhai, R. and Purwar, D., 2015. *Design Space Exploration of RISC Architectures using retargetability*. *2015 International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA)*, .

Bejo, A., Li, D., Isshiki, T. and Kunieda, H., 2014. A Method of Software Development Tool and Hardware Generation for ASIP with a Co-processor based on the Derivative ASIP Approach. *Journal of Information Processing*, [online] 22(2), pp.131–141. Available at: <http://ci.nii.ac.jp/naid/130003394456/en/>.

Bosscher, S., 2012. *Modular GCC*. [online] Available at: <https://gcc.gnu.org/wiki/ModularGCC> [Accessed 11 Jul. 2017].

Campi, F., Cappelli, A., Guerrieri, R., Lodi, A., Toma, M., Rosa, A. La, Lavagno, L., Passerone, C. and Canegallo, R., 2003. *A reconfigurable processor architecture and software development environment for embedded systems*. *Proceedings International Parallel and Distributed Processing Symposium*, .

Diego, N., 2007. *GCC Internals*. Available at: <www.airs.com/dnovillo/200711-GCC-Internals/200711-GCC-Internals-1-condensed.pdf>.

Dworkin, M.J., 2007. NIST Special Publication 800-38: Recommendation for Block Cipher Modes of Operation. [online] Available at: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>.

FIPS, P., 2009. 197, Advanced Encryption Standard (AES), National Institute of Standards and Technology, US Department of Commerce, November 2001. *Link in: http://csrc. nist. gov/publications/fips/fips197/fips-197. pdf*.

Ghica, L. and Tapus, N., 2015. *Optimized retargetable compiler for embedded processors - GCC vs LLVM*. *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, .

Gupta, A. and Pal, A., 2015, January. Accelerating SVM on ultra low power ASIP for high throughput Streaming Applications. In *2015 28th International Conference on VLSI Design* (pp. 517-522). IEEE.

Hennessy, J.L. and Patterson, D.A., 2011. *Computer Architecture: A Quantitative Approach*. *Elsevier*.

Hoang, V.-P., Dao, V.-L. and Pham, C.-K., 2017. Design of ultra-low power AES encryption cores with silicon demonstration in SOTB CMOS process. *Electronics Letters*, [online] 53(23), pp.1512–1514. Available at: <http://digital-library.theiet.org/content/journals/10.1049/el.2017.2151>.

Humayed, A., Lin, J., Li, F. and Luo, B., 2017. Cyber-Physical Systems Security - A Survey. *IEEE Internet of Things Journal*, 4(6), pp.1802–1831.

ISO/IEC-18033-3, 2005. Information technology - Security techniques - Encryption algorithms-Part 3: Block ciphers. *INTERNATIONAL STANDARD ISO / IEC*.

Johann, S.F., Moreira, M.T., Calazans, N.L. V and Hessel, F.P., 2016. The HF-RISC processor: Performance assessment. *LASCAS 2016 - 7th IEEE Latin American Symposium on Circuits and Systems, R9 IEEE CASS Flagship Conference*, pp.95–98.

Kassem, R., Briday, M., Béchennec, J.-L., Savaton, G. and Trinquet, Y., 2012. Harmless, a hardware architecture description language dedicated to real-time embedded system simulation. *Journal of Systems Architecture*, [online] 58(8), pp.318–337. Available at: <http://www.sciencedirect.com/science/article/pii/S1383762112000410>.

Kiat, W.P., 2018. *The design of an fpga-based processor with reconfigurable processor execution structure for Internet of Things (IoT) applications*. Master Dissertation, Universiti Tunku Abdul Rahman (UTAR).

Kiat, W.P., Mok, K.M., Lee, W.K., Goh, H.G. and Andonovic, I., 2017. A comprehensive analysis on data hazard for RISC32 5-stage pipeline processor. In: *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. [online] pp.154–159. Available at: <http://ieeexplore.ieee.org/document/7929670/>.

kokke, 2014. *tiny-AES-C*. [online] Available at: <https://github.com/kokke/tiny-AES-c>.

Kolek, J., Jovanović, Z., Šljivić, N. and Narančić, D., 2013. *Adding microMIPS backend to the LLVM compiler infrastructure*. *2013 21st Telecommunications Forum Telfor (TELFOR)* .

Lattner, C., n.d. *The Architecture of Open Source Applications:LLVM*. *The Architecture of Open Source Applications*. Available at: <http://www.aosabook.org/en/llvm.html>.

LLVM, P., 2013. *The LLVM Compiler Infrastructure*. [online] Available at: <http://llvm.org>.

Lopez Trescastro, J., Vassev, E., Hellstrom, D. and Cederman, D., 2015. An LLVM Backend for LEON Processors. In: *DASIA 2015-DAta Systems in Aerospace*.

Lu, M., Fan, A., Xu, J. and Shan, W., 2018. A Compact, Lightweight and Low-Cost 8-Bit Datapath AES Circuit for IoT Applications in 28nm CMOS. *Proceedings - 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications and 12th IEEE International Conference on Big Data Science and Engineering, Trustcom/BigDataSE 2018*, pp.1464–1469.

Microsoft, 2019. *Visual Studio IDE, Code Editor, VSTS, & App Center - Visual Studio*. [online] Available at: <https://visualstudio.microsoft.com/>.

Price, C., 1995. MIPS IV Instruction Set. *Memory*. [online] Available at: <http://www.weblearn.hs-bremen.de/risse/RST/docs/MIPS/mips-isa.pdf>.

Rescorla, E., 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*.
Soliman, M.I. and Abozaid, G.Y., 2011. FPGA implementation and performance evaluation of a high throughput crypto coprocessor. *Journal of Parallel and Distributed Computing*, [online] 71(8), pp.1075–1084. Available at: <http://dx.doi.org/10.1016/j.jpdc.2011.04.006>.

Strömbergson, J., 2014. *secworks/aes*. [online] Available at: <https://github.com/secworks/aes>.

Sýkora, J., n.d. *LLVM-Based C Compiler for the PicoBlaze Processor*.

Taglietti, L., Filho, J.O.C., Casarotto, D.C., Furtado, O.J. V and Santos, L.C. V, 2005. Automatic ADL-Based Assembler Generation for ASIP Programming Support. *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pp.262–268.

Tomić, I. and McCann, J.A., 2017. A Survey of Potential Security Issues in Existing Wireless Sensor Network Protocols. *IEEE Internet of Things Journal*, 4(6), pp.1910–1923.

Tomiyama, H., Halambi, A., Grun, P., Dutt, N. and Nicolau, A., 1999. Architecture description languages for systems-on-chip design. In: *Asia Pacific Conference on Chip Design Language*.

Valerio, P., 2016. Is the IoT a Tech Bubble for Cities?: With more cities joining the smart city revolution and investing in sensors and other IoT devices, the risk of a new tech bubble is rising. *IEEE Consumer Electronics Magazine*, 5(1), pp.61–62.

Vilela, G., Correa, E. and Kreutz, M., 2012. A LLVM based development environment for embedded systems software targeting the RISCO processor. *Brazilian Symposium on Computing System Engineering, SBESC*, pp.77–82.

Wang, W., Han, J., Xie, Z., Huang, S. and Zeng, X., 2016. Cryptographic coprocessor design for iot sensor nodes. In: *ISOCC 2016 - International SoC Design Conference: Smart SoC for Intelligent Things*. pp.37–38.

Wang, Y. and Ha, Y., 2016. High throughput and resource efficient AES encryption/decryption for SANs. *Proceedings - IEEE International Symposium on Circuits and Systems*, 2016-July, pp.1166–1169.

Witte, E.M., Chattopadhyay, A., Schliebusch, O., Kammler, D., Leupers, R., Ascheid, G. and Meyr, H., 2005. Applying resource sharing algorithms to ADL-driven automatic ASIP Implementation. *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2005, pp.193–199.

Xu, L. Da, He, W. and Li, S., 2014. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, 10(4), pp.2233–2243.

Yang, L., Ni, X., Tan, Y. and Liu, H., 2013. ADL and High Performance Processor Design. [online] pp.67–74. Available at: <http://link.springer.com/10.1007/978-3-642-35898-2%7B_%7D8>.

Yuan, Y., Yang, Y., Wu, L. and Zhang, X., 2018. A High Performance Encryption System Based on AES Algorithm with Novel Hardware Implementation. *2018 IEEE International Conference on Electron Devices and Solid State Circuits, EDSSC 2018*, pp.1–2.