RISC32-E Cryptography Performance Evaluation

By

Teo Sei Hau

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMPUTER ENGINEERING

Faculty of Information and Communication Technology
(Kampar Campus)

Jan 2022

REPORT STATUS DECLARATION FORM

REPORT STATUS DECLARATION FORM

Title: RISC32-E Cryptography Performance Evaluation

Academic Session: Jan 2022

I TEO SEI HAU

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in

Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

(Author's signature)

(Supervisor's signature)

Address:

15-33, Lingkaran Agacia,

Menara Agacia, Bandar Agacia,

31900 Kampar, Perak.

Teoh Shen Khang

Supervisor's name

Date: 05/04/2022

Date: 21 April 2022

FYP THESIS SUBMISSION FORM

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

UNIVERSITY TUNKU ABDUL RAHMAN

Date: 05/04/2022

SUBMISSION OF FINAL YEAR PROJECT

It is hereby certified that Teo Sei Hau (ID No: 18ACB03719) has completed this final year project entitled "RISC32-E Cryptography Performance Evaluation" under the supervision of Mr. Teoh Shen Khang (Supervisor) from the faculty of Information and Communication Technology (FICT).

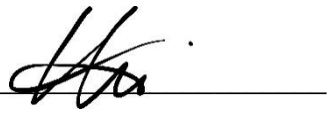
I understand that University will upload softcopy of my final year project in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

(Teo Sei Hau)

DECLARATION OF ORIGINALITY

I declare that this report entitled "RISC32-E Cryptography Performance Evaluation" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : 

Name : Teo Sei Hau

Date : 5/04/2022

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisors, Mr. Mok Kai Ming and Mr. Teoh Shen Khang, who had given me this bright opportunity to engage in an IC design project. It is my first step to establish a career in IC design field. A million thanks to you for your patient and guide me throughout the project, which made me learn a lot of new things.

Besides that, I would like to thank my seniors and classmates for the support and encouragement throughout the courses. Finally, I must say thanks to my family that has supported me and overcomes hardships with me throughout the course with me.

ABSTRACT

This project is about performance evaluation of RISC32, RISC32-E-NQ and RISC32-E-Q for comparison purposes. The main objective of this project is to evaluate the performance of the three architectures mentioned above to have a better understanding how much RISC32-E-Q performance better than another 2 architectures in terms of cryptography operations. In this project, CoreMark is selected as the CPU benchmark due to its simplicity and conformity to the three architectures mentioned. To port CoreMark to the architectures, llvm is set up to convert the c languages source codes into RISC32 understandable machine codes. UART is also configured to obtain the desired output. In short, stimulation result are expected to be delivered at the end of the project.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AES	Advance Encryption Standard
ALB	Arithmetic Logic Block
BLE	Bluetooth Low Energy
BRAM	Block RAM
CP0	Coprocessor 0
CP2	Coprocessor 2
CP2Q	CP2 Queue
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DES	Data Encryption Standard
DTE	Data Terminal Equipment
EEMBC	Embedded Microprocessor Benchmark Consortium
EX	Execution
FF	Flip Flop
FPGA	Field Programmable Gate Array
GPIO	General-Purpose Input/Output
I/O	Input / Output
ID	Instruction Decode
IDE	Integrated Development Environment
IF	Instruction Fetch
IIoT	Industrial Internet of Things
IR	Intermediate Representation
ISA	Instruction Set Architecture
LTS	Long-Term Support
LUT	Look Up Table
MEM	Memory
MIPS	Microprocessor without Interlocked Pipeline Stages
NIST	National Institute of Standard and Technology

PC	Program Counter
RAM	Random Access memory
RISC	Reduced Instruction Set Computer
RTL	Register-transfer Level
RxD	Receive Data
SPI	Serial Peripheral Interface
SWQ	Store Word Queue
T_ASM	Test program of Assembly language
T_C	Test program of C language
T_CP2_NQ	Test program on RISC32-E without queue system
T_CP2_Q	Test program on RISC32-E with queue system
TxD	Transmit Data
UART	Universal Asynchronous Receiver and Transmitter
UTAR	University Tunku Abdul Rahman
WB	Write Back

CHAPTER 1: Introduction

1.1 Background Information

Benchmarking is defined as the processes of measuring a processor against another in one or more aspects of their operations. Benchmarking provides necessary insights to help people understand how the processor is different in terms of some aspects such as performance, power consumption, and others against different processors even though they are not manufacture from the same company. As we know, processor in embedded systems is becoming more complex. As processor complexity increases, a more sophisticated benchmark must be properly exercised and analyse those processors to have a better idea about the performance of the selected processors. This work is going to use CoreMark, a modern, sophisticated benchmark to analyse and compare between the 2 RISC32-E cryptography processor, which are RISC32-E-Q and RISC32-E-NQ.

1.1.1 MIPS

MIPS, the short form for Microprocessor without Interlock Pipeline Stages is generally a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies. Its primary implementations are embedded system, eg. Windows CE devices, video game consoles and so on. It also has several optional extensions that are available, such as MIP-3D which added new instructions for improving the 3D graphics applications' performance, MDMX that for accelerating multimedia applications and so on. The opposite of RISC architecture is the Complex Instruction Set Computer (CISC) architecture. CISC architecture is a complex instruction set that often uses a huge number of addressing modes, meaning that processor based on CISC architecture may have variable length of instructions. In contrast to the CISC, RISC architecture is simpler as it uses simplified instruction sets and only few addressing modes. Most of the time, RISC architecture processor execute instructions that in the same length. Therefore, the hardware of processors based on RISC architecture will be less complicated, and this allow the processor to be faster as well as easier to build and test. In trade off, the instructions used per program will be increases as it often required a line of program to be split and execute into multiple instructions. However, execution of more instructions does not mean that increases in

execution time. A general idea that overlap the instructions named pipelining often implement in RISC processors to enhance the throughput and reduce the execution time by instruction, as shown in Figure 1.1.1 F1 [1]. This technique makes the idle time of hardware become minimize and thus the processing speed of a RISC processor can be compared to a more complex CISC processor.

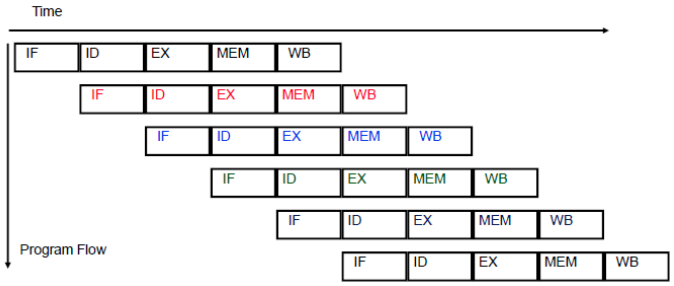


Figure 1.1.1 F1*:* Conventional pipeline execution representation [1]

1.1.2 UART

A universal asynchronous receiver and transmitter (UART) is a hardware device which is computed to regulate a devices' interface to its attached serial devices. UART is meant for asynchronous serial communication purposes, and it allows users to customize the data format and transmission speed. Generally, UART devices often being connected with the RS-232C Data Terminal Equipment (DTE) interface, and the connection allows the devices to communicate to one another by data transmission. Figure 1.1.2 F1 below shows simplified view of devices connection through UART to achieve serial data transmission. [2]

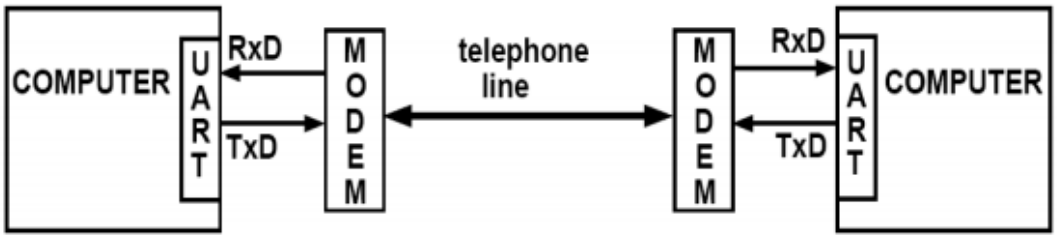


Figure 1.1.2 F1: Serial data transmission [2]

In details, a UART can provided users with functions listed as below:

- For outbound transmission, UART helps to convert the data from host devices, which are usually in byte form, into a single bit stream data and send to the end user device one bit per every clock pulse. This method reduces hardware complexity because it only needs 2 wires.
- Vise versa, the input signal which is in the form of single bit stream will be converted back into the data forms which the host system operates.
- To ensure data transmission correctness, a parity bit is added to the data for outbound transmission and the technique that checks and removes parity bit is implemented for incoming input signals.
- UART will add start and stop bit to the single bit stream data to help guest devices recognize the beginning and ending of a data stream.
- UART also helps to manipulate interrupts from serial devices like a keyboard and a mouse, which have special serial ports.

1.1.3 CoreMark Benchmark

CoreMark is a benchmark certified by Embedded Microprocessor Benchmark Consortium (EEMBC) as a standard benchmark to compute comparison between microprocessors. It is a light-weighted, highly transportable, well understood as well as highly manipulated benchmark to compare embedded processors. CoreMark benchmarking system verified that all computations or instructions were done accurately during execution time, which is capable to debug any errors that may pop up. Generally, CoreMark tries to run a processor with the situations that happened frequently in practically all applications, such as simple codes that use basic algorithms or simple data structures. CoreMark play a role as a performance indicator to allow designers and any other end users have a brief idea about the performance of the processor.

CoreMark implementation is chosen by EEMBC carefully to prevent code elimination during compile time. All of the computations and data values are driven by run-time supplied values, which are not known by the users, to ensure the integrity of the result generated. Besides that, CoreMark also set specific rules to specific the ways to execute the testbench codes and generate outputs, thus preventing inconsistencies and provided a reliable and comparable results between microprocessors. This work is going to use CoreMark as a performance evaluation tool to compare the performance analysis on both RISC32-E architecture. [3]

1.2 Motivation

In this project, 3 microprocessors with different architectures which are RISC32, RISC32-E and RISC32-EQ are going to be evaluate. The processors mentioned above have been developed using Verilog, a hardware description language (HDL) in the Faculty of Information and Communication Technology (FICT) of Universiti Tunku Abdul Rahman (UTAR). In terms of motivation, this project is going to develop and implement of a standard benchmarking system that suites for evaluating the processors. In the benchmarking field, there are many methods out there that are implemented for benchmarking suites for evaluating processor based on different specification and application. However, with some differences in application field, the standards methods and codes needed to suit the needs. So, this project took aims to configure CoreMark benchmarking to obtain the results and data that are comparable under same environment without further arguments.

1.3 Problem Statement

In the previous work done by See et al., performance evaluation is done to compare the performance including processing speed and power consumption of the proposed technique which is the implementation of queue system in RISC32-E processor (implemented as RISC32-E-Q architecture) with some other existing implementations such as RISC32 and RISC32-E. The experiment done clearly showed that the newest implementation with the queue system has a significant improve in terms of throughput and energy consumption as illustrated in Figure 1.3 F1 and Figure 1.3 F2. However, the

performance evaluation done previously is not based on some standard for example CoreMark and the MiBench and is only a functional test program developed to get a brief comparison between the RISC32-E-Q and some existing implementations. In other word, it is not reliable enough in the view of standard in performance evaluation.[4]

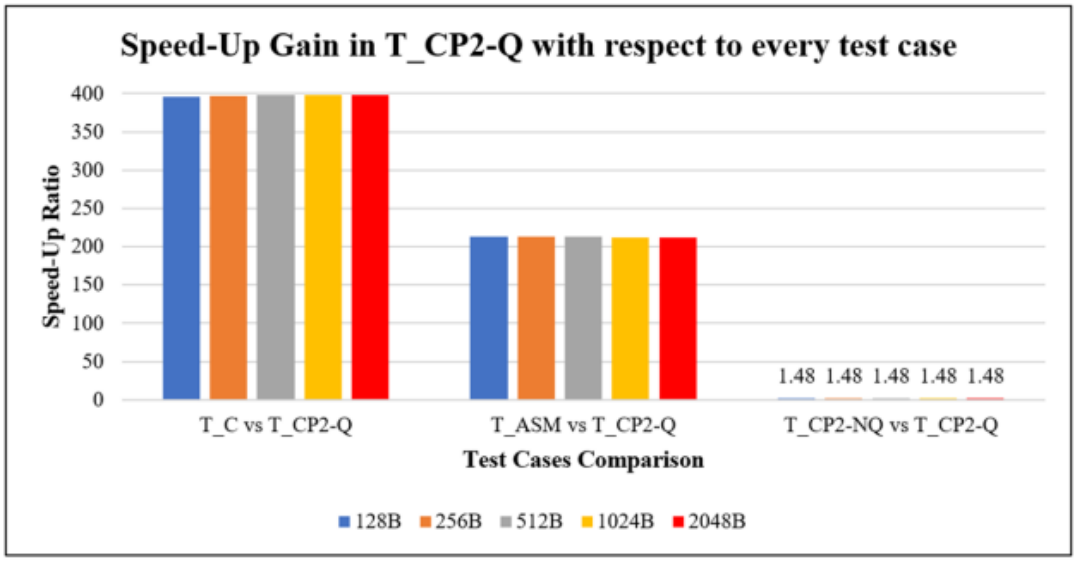


Figure 1.3 F1: Comparison of speed-up ratio between T_CP2-Q (RISC32-E-Q) compared with other test cases [4]

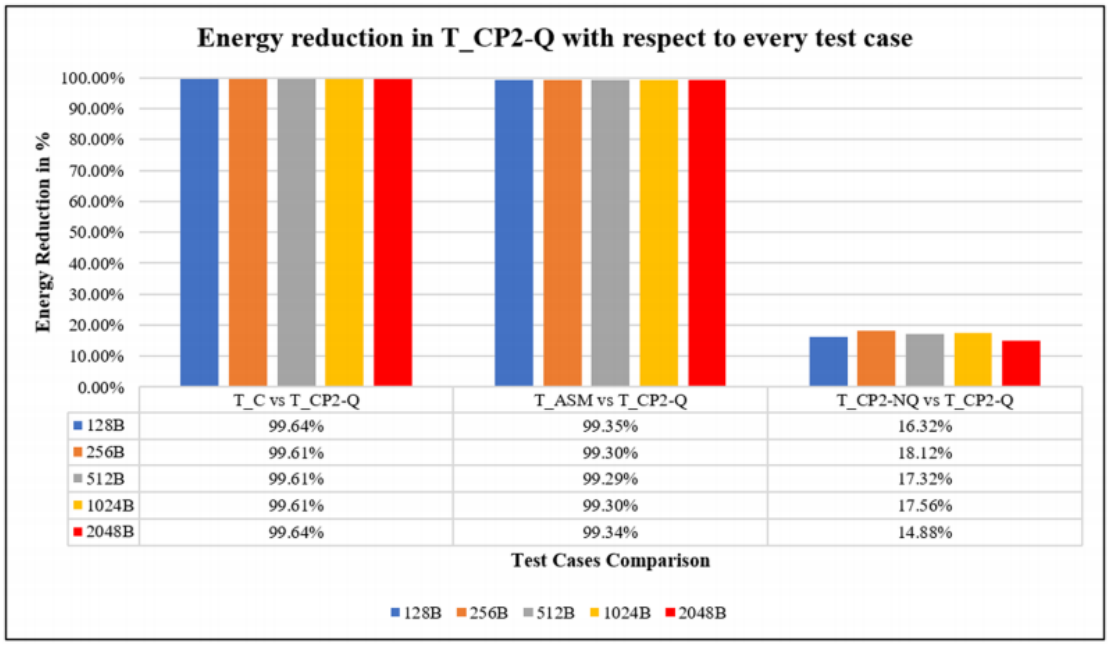


Figure 1.3 F2: Comparison of energy consumption between T-CP2-Q (RISC32-E-Q) compared with other test cases [4]

1.4 Project Scope and Direction

This project will primarily focus on configure reliable benchmarking codes and implements it to the three RISC32 architectures mentioned. The reliable benchmarking chosen in this project is the CoreMark benchmarking certified by EEMBC due to its simplicity and standardization. The benchmarking codes provided will be studied to know the theories behind the benchmarking system and the ways to port it to the architectures.

Besides that, configuration of UART and the internal timer within CP0 must be done for the processor to return the results back to personal computer every time it successfully run the codes. In this stage, there will be learning of UART and internal clock configuration to fit the purposes.

Moreover, LLVM will be studied and setup to compile the testbench codes which are originally obtained as C language codes into RISC32 architectures understandable machine codes.

Lastly, a full and complete documentation on this project will be fully developed and maintained. In brief, the benchmarking results are expected to be handed over at the end of the project.

1.5 Project Objectives

The main objectives of this project are:

- Analyze the CoreMark architecture and components for component selection and modification purpose.
- Develop a suitable subset of CoreMark programme for the RISC32.
- LLVM setup and CoreMark compilation / debugging.
- To simulate the CoreMark programme in the various RISC32 versions for comparison.

1.6 Impact, Significance and Contribution

After this project is done, it will provide a complete performance evaluation result on both RISC32-E microprocessor.

In terms of contribution, this project can contribute to have a better idea about the selection of the architecture by providing more reliable comparison results to the hardware modelling of RISC32-E cryptography processor research work later. With the available well-developed and organized result provided, a researcher can easily identify the type of architecture to use for developing their own processor for various purposes such as developing a sensor node to support fast AES encryption in Industrial Internet of Thing application. As a result, the research works that wish to be done by researchers later could be done faster and easier, as well as speed up significantly.

1.7 Project Organization

This report contains 5 chapters with the details of the project listed in the following chapters.

- Chapter 1: Background information that is important for this project is given. The motivation, problem statement, project scope and direction, project objectives are also provided to enhance the readability for the readers.
- Chapter 2: The literature review of RISC32 architectures, RISC32 compilation toolchains, RISC32 instructions set and CoreMark are given.
- Chapter 3: Discussion of proposed methods/approaches in details.
- Chapter 4: Discussion of system design in this project.
- Chapter 5: Discussion of CoreMark implementation throughout the whole project, and show simulation results
- Chapter 6: Conclude what have been achieved in the project, and discussion of future works.

CHAPTER 2: Literature Review

2.1 Overview of RISC32-E architecture

RISC32-E is the integrated version of RISC32 where AES-128 core is integrated to the RISC32 for AES encryption purpose. To integrate the AES-128 core into RISC32 architecture, the suitable stage in the pipeline is required to be selected to maintain the balanced workload among all the stages in RISC32 architecture. The available stages are Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM), and Write Back (WB). According to the work done by See et al. [4], his team had chosen Instruction Decode (ID) stage for the AES-128 core integration. ID stage was chosen because the operands and other information required to execute the operation are readily obtain at this stage. Integration of AES-128 core to Execution (EX) stage may also be a possible solution, but it might cause the complexity of hardware increase as more logic needed to pull all relevant information of an instruction from the ID stage to the EX stage, which is not necessary. The AES-128 core is placed under Coprocessor 2 (CP2) [4]. The simplified view of RISC32-E microarchitecture is shown in Figure 2.1 F1 below.

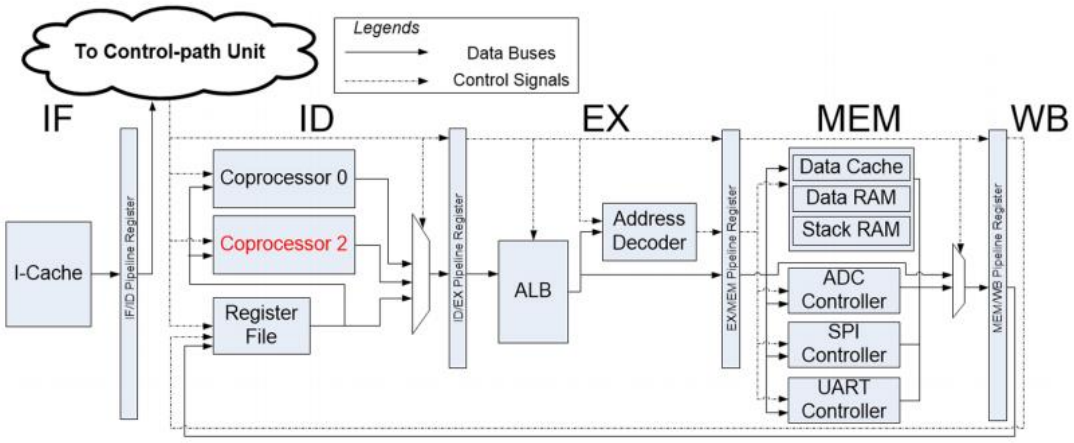


Figure 2.1 F1: Simplified view of RISC32-E microarchitecture [4]

2.2 RISC32-E-Q

In the conventional architecture (RISC32-E), encryption of N byte of data required a total of T clock cycles, where N representing the size of data in terms of byte. N value can be 64 Bytes, 128 Bytes, 256 Bytes, 512 Bytes, or 1024 Bytes. The relationship between T and N is shown in formula below:

$$T = (N / 16) * (18\ CP2\ read\ write\ instructions + 55\ cycles\ computational\ time)$$

For every 16 bytes encryption, the user program needs to execute a total of 18 read write instructions for the encryption purpose and wait for another 55 NOPS (No Operations) instructions for the Coprocessor 2 (CP2) to generate a valid ciphertext. The problem comes with the 55 NOPS instructions, the processor need to be wait for 55 clock cycles for the generation of valid ciphertext output in the processor pipeline, by executing the 55 NOPS instructions. Try to consider the case when N = 1024B, the T will 4672 cycles, where 3520 cycles (75.34%) are spent idle by executing the NOPS instructions to wait for the encryption to be completed! This is not efficient enough, and so, See et al. proposed a new solution by implementing a queue system so that the data acquisition and encryption process can be execute concurrently, as illustrate in Figure 2.2 F2 below, to maximize the utilization of the processor and thus increase the throughput. [4]

Figure 2.2 F1: Data processing pattern of RISC32-E (without queue system) [4]

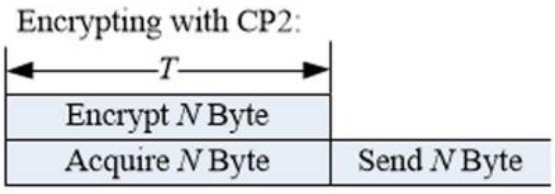


Figure 2.2 F2: Data processing pattern with encryption and data acquisition processor execute concurrently. [4]

To perform the overlapping execution technique, a queue system consisting of 2 new hardware queue, CP2 Queue (CP2Q) and Store Word Queue (SWQ) were introduced in the system so that the processor is able to handle two different types of operations (encryption and acquisition) concurrently. CP2Q stores incoming CP2 related instructions and data when CP2 is busy executing the existing encryption operation, while SWQ will stores computed store address until the CP2 output a valid cyphertext. Also because of the 2 new hardware queues, a new instruction, swc2 (store word to coprocessor 2) was created to adopt the new hardware. Figure 2.2 F3 below shows the simplified view of RISC32-E microarchitecture. [4]

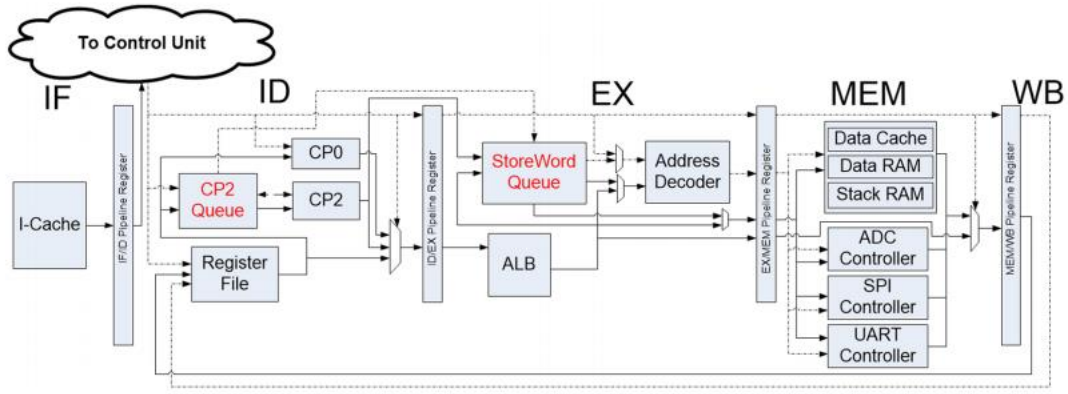


Figure 2.2 F3: Simplified view of RISC32-E pipeline with CP2 and queue system. [4]

2.3 RISC32 Compilation Toolchain

Figure 2.3 F1 below shows the structural view of RISC32-E compilation toolchain architecture. The compilation toolchain is designed according to LLVM19 modular compilation framework. The aimed of using modular compilation framework is to freely pair different frontends (process source code) with different backends (emits assembly or object file). Clang, the frontend compiler that compile C language family including C, C++, Objective C/C++ and others, is used as the frontend for LLVM, which compiles C language source codes and outputs LLVM intermediate representation (IR), which is a portable, high-level assembly codes. LLVM IR acts as the first input into the LLVM structure, it will pass through opt (optimizer) and llc, the

static compiler to further analysed and transformed into other intermediate forms such as assembly (.s) and object (.o) before link operation.

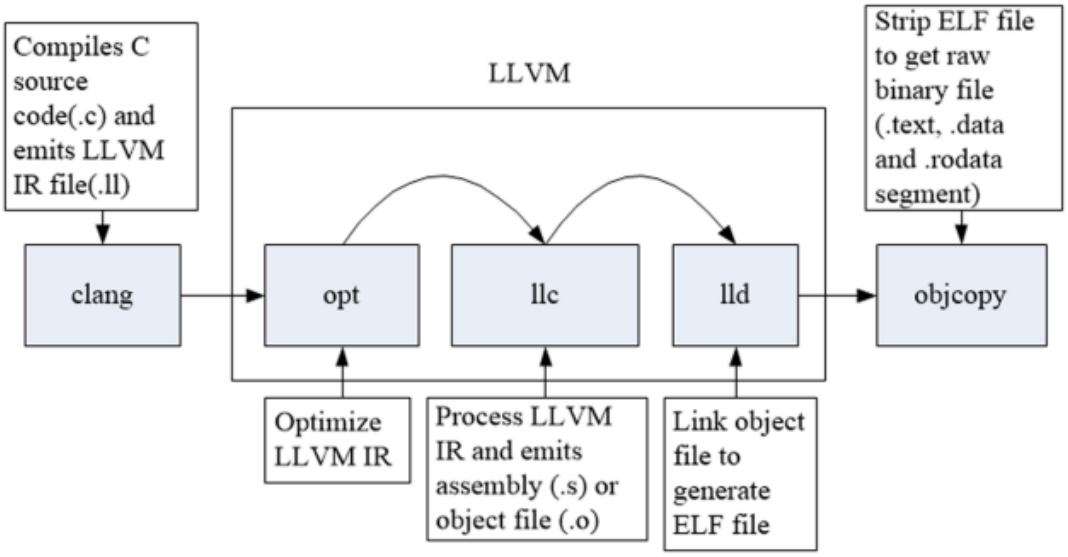


Figure 2.3 F1: Structural view of RISC32-E compilation toolchain [4]

The first module inside the LLVM is the LLVM optimizer, the opt, which plays a role to perform analysis and thus optimize the input LLVM IR. Next, the llc module, which is the static compiler of LLVM, takes roles to transforms the LLVM IR into various intermediate forms such as assembly files and binary object files to maps to the instruction set of the RISC32-E processor. After that, the assembly files and binary object files will pass to the last module in LLVM architecture, which is the LLVM linker, lld. The linker functions are to perform address calculation, link together and produce a final executable file in .elf format. The executable file further passes to objcopy module to extract the .Text and .Data Segment that contained the instructions and data only and eliminate the operating system related headers and information sections as the code is not running on any operating system. Now, the binary content is readily to be extracted and load into suitable memory address based of the respective sections and memory map established in RISC32-E processor. [4]

2.4 RISC32-E Instruction Set

RISC32-E is a MIPS Instruction Set Architecture (ISA) compatible processor. It can decode and execute a subset of the standard MIPS instruction as in green card shown in Appendix A. In the previous work done by See et al., his team had customized the existing LLVM MIPS backend to compile for 54 MIPS instruction as shown in Table 2.4 T1 below, but not all the standard MIPS instructions.

They had chosen MIPS II as the base architecture to support the binary content generation for RISC32-E processors as both MIPS II and RISC32-E has high similarities in supporting subset of MIPS instructions. Some modifications have also been done to eliminate out some unsupported MIPS II instructions that cannot be executed in RISC32-E processors to create a new sub target in the MIPS backend that provided information for llc, the LLVM static compiler.

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

Table 2.4 T1: RISC32 instruction set [4]

2.5 RISC32 Memory Map

A memory map contains the complete information on how the memory is structured in a processor. In RISC32, there is a total of 4GB memory space as it uses 32-bits data width. Memory mapping defines the way of how this 4GB address space will be used.

Generally, implementation of memory address space of RISC32 can be categorized into 2 types: virtual memory and physical memory. Virtual address, which is the address corresponding to virtual memory, are used by the CPU for instructions and data accessing, while physical address will be used to allocate with physical memory such as Flash memory, Data RAM, Stack RAM, boot ROM and I/O registers. Figure 2.5 F1

below shows the virtual to physical memory mapping for all the five segments which are kernel user segment (kuseg), kernel segment 0 (kseg0), kernel segment 1 (kseg1), kernel segment 2 (kseg2) and kernel segment 3 (kseg3). [5]

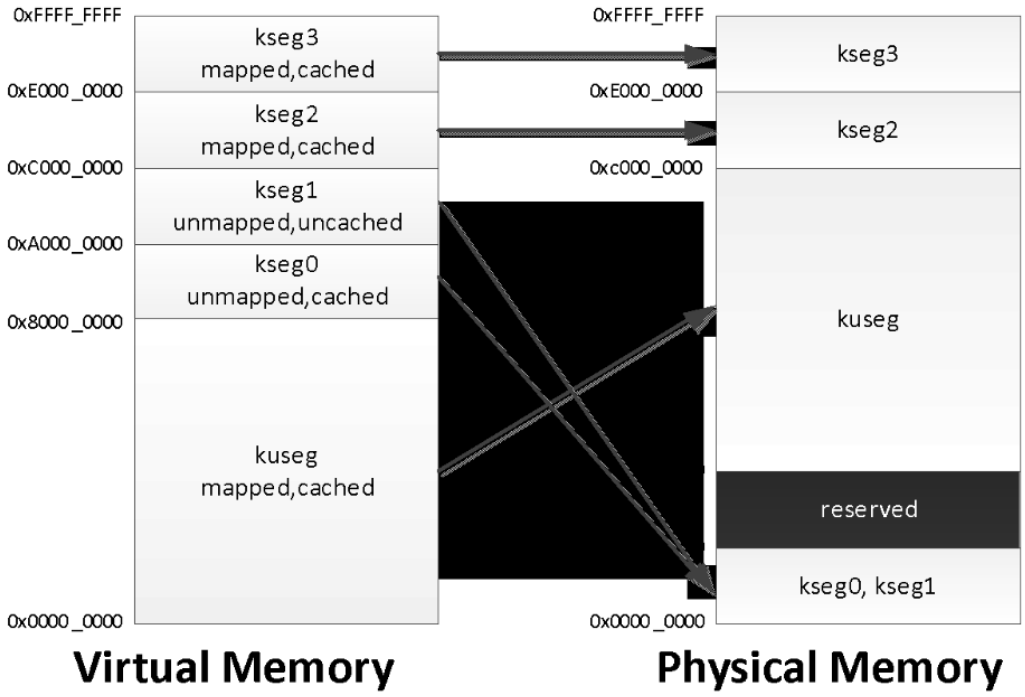


Figure 2.5 F1: Virtual to physical memory mapping [5]

From Figure 2.5 F1, it is clear to see that kseg0 and kseg1 are sharing the same physical memory, but they are having different virtual memory. kseg0 and kseg1 is the only two segments that can be used in implementation, or in other word, kseg0 and kseg1 are used to store information such as user program code, exception handler code, normal data, stack data, heap data and boot loader code. Figure 2.5 F2 below shows the memory allocation on kseg0 and kseg1. [5]

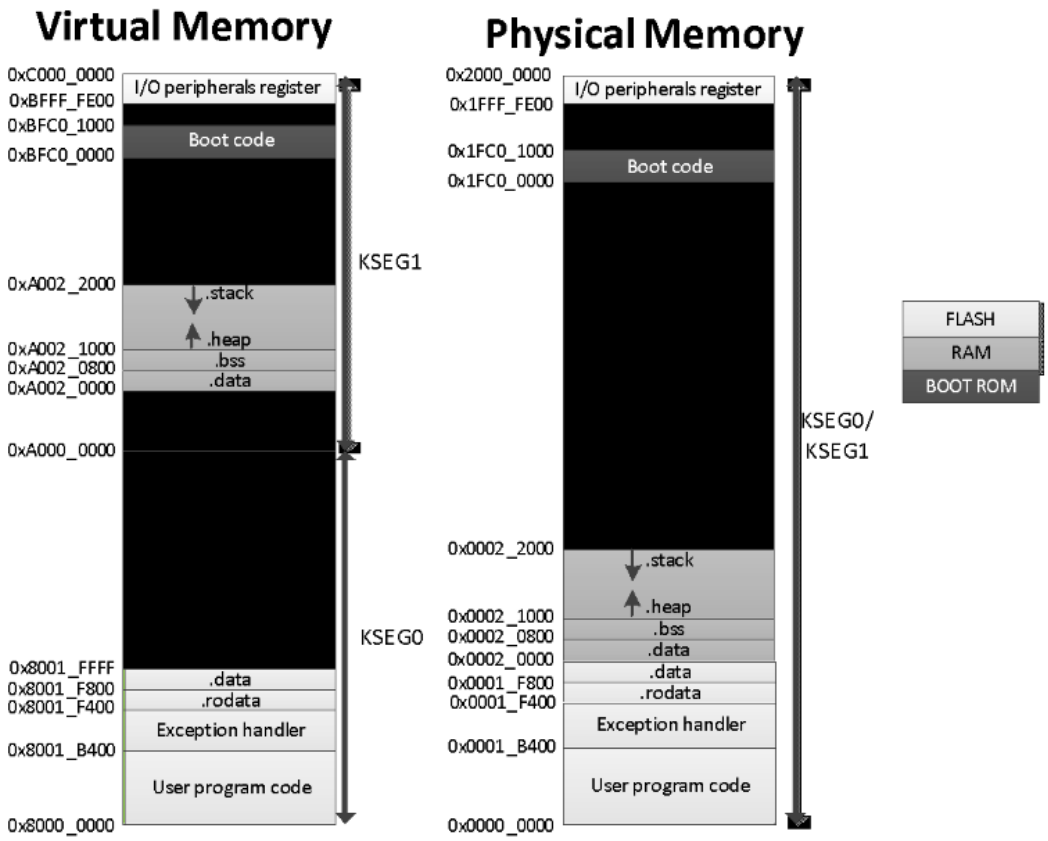


Figure 2.5 F2: Memory allocation on kseg0 and kseg1 [5]

2.6 CoreMark

In embedded world, CoreMark is typically one of the most ordinary benchmarks chosen by developers to test for the microcontrollers and CPUs (central processing unit) performance. CoreMark is favourite due to its simplicity, light-weighted, easy to understand as well as easy to port over a design, and most importantly, it is free. CoreMark is released in 2009 by Shay Gal-On, and it takes purposes to replace the old-fashioned benchmark, Dhrystone, which claims to be having some unavoidable problematic aspects. For example, Dhrystone benchmark put more focus on a compiler's ability to optimize the execution size as well as workload, rather than the abilities and capabilities of a CPU. So, the performance of a CPU is highly dependent to the capabilities of compiler when we test it using Dhrystone, but it is not logic. However, the improved testbench, CoreMark avoid the same problems as seen in

Dhrystone, at the same time, CoreMark inherit the good sides of Dhrystone such as simplicity, highly portable and easy to understand. Unlike Dhrystone, CoreMark also improved by specified the running iterations and reporting rules, made it generate more reliable results by eliminating consistencies when we compare the magic number, CoreMark/Mhz with other processors from different architectures. (Levy, 2011)

Besides, CoreMark is preferable in this project as it is free. It is not practical to be use if CoreMark is behind a paywall, as it's simply not going to be used widely in the industry. CoreMark is also a easy benchmarking tool to port and run over different processors, the port functions are all defined under core_portme.c file in neat and easily understandable manner. In short, CoreMark is a popular benchmark for embedded system as it is free, easy to port over processors, and have organization background to evolve the system. Table 2.6 T1 below show comparison between some popular benchmark such as Linpack, Dhrystone, SPEC CPU, CoreMark and MLPerf 0.5. (Patterson , 2019)

	CoreMark	**Dhrystone**	**SPEC CPU**	**Linpack**	**MLPerf 0.5**
Year	2009	1984	1989	1977	2018
Target	Embedded	Systems programming	Unix Server	Supercomputer	Servers
Quatity reputation	Low	Low	High	Low	High
Free of charge	✓	✓	✘ ($1000)	✓	✓
Easy to port	✓	✓	✓	✓	✘
Organization to evolve benchmark	✓	✘	✓	✓	✓
Single summary score	Speed ratio	Speed ratio	Geometric mean of speed ratio	FLOPS/sec	Weighted mean ratio + Std. Dev.

Table 2.6 T1: Comparison between CoreMark, Dhrystone, SPEC CPU, Linpack and MLPerf 0.5

In term of testing a processor performance, CoreMark categorized the abilities of a processor into 3 main categories: the ability to handle list items, the ability of process matrix, and the ability to process state machine. As a result, the testbench is trying to implement the algorithms which targeted the categories mentioned, which are list processing, matrix processing and state machine processing. Figure 2.6 F1 below show the simplified view of architecture of CoreMark.

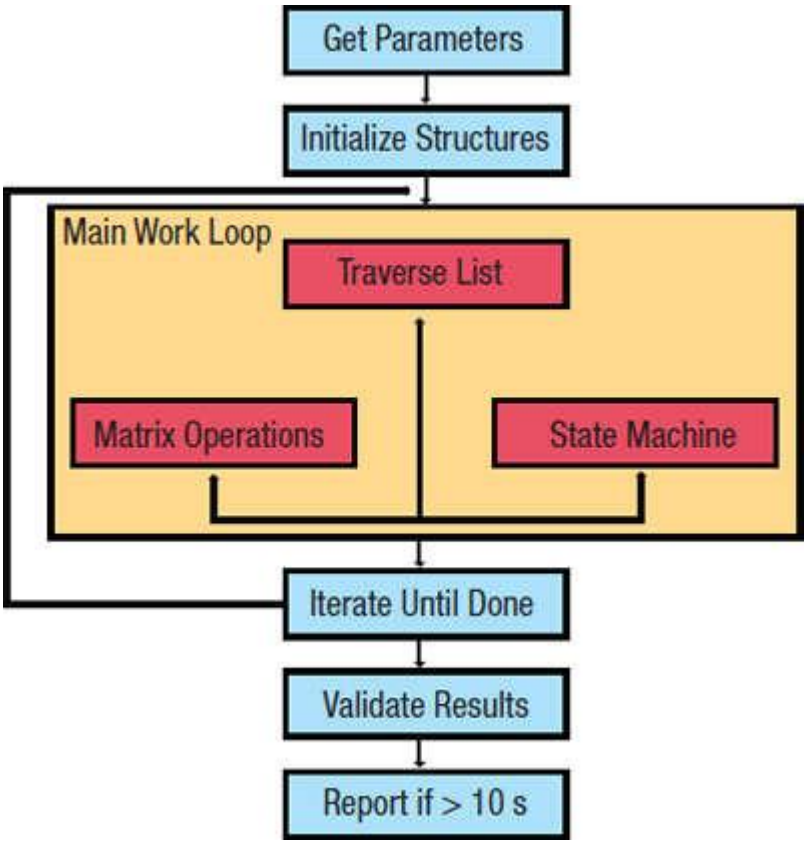


Figure 2.6 F1: Simplified view of CoreMark architecture [3]

2.6.1 CoreMark List Processing

List processing comprises of algorithms such as searching, reversing, and sorting of a list item in line with various parameters that support by the content of the list data objects. List processing of CoreMark will operates on pointers and it is often specified as non-commercial memory patterns. In the aspect of testing the core of a CPU, CoreMark list processing algorithm plays a vital role in testing how brisky data is wore to scan through the list. In order to make sure the operation performed is correct, CoreMark also carry out a 16 bits cyclic redundancy check (CRC) based on computed data that is contained in the objects of the list. The CRC calculation is additionally

placed in the time portion of CoreMark as CRC is a common function employed in various embedded system.

To start with the list processing function, CoreMark will first split the available data space into 2 subspaces, the first one contains the list data, while the second one contains the header objects point towards the data. The purpose of splitting the data is to simulate the common embedded system function, where the data is stored in an exceeding buffer while the pointers towards the object containing data is store inside a list (or sometimes ring buffers). The sample c codes of the list_data structure defined as below (noted that the data16 items are initialized based on seeds value which is non determinant during compile time):

```
typedef struct list_data_s {
ee_s16 data16;
ee_s16 idx;
} list_data;
```

Each of the data16 item consists of two 8-bit parts, where the upper 8 bits containing the original value for the lower 8 bits. The definition of data contained at the lower 8B is defined as follows:

- 0..2: Type of function to perform to calculate a value.
- 3..6: Type of data for the operation.
- 7 : Indicator for pre-computed or cached value.

CoreMark will perform operations on the data16 item during each cycle of the testbench codes, depends on the processes involve.

The sample c codes of list_head structure also provided as shown below:

```
typedef struct list_head_s {
struct list_head_s *next;
struct list_data_s *info;
```

} list_head;

The list head objects value will also changes after each iteration of the benchmark, For example, the *next* pointer will changes its value according to the modification done when the list is reversed or sorted. Upon each iteration of the testbench codes, the algorithm will sort the list based on the details stored in the *data16* member. Then, the algorithm will perform some specific test to the data before restructure the original list. Figure 2.6.1 F1 below shows the basic structure of the linked-list access mechanism. [3]

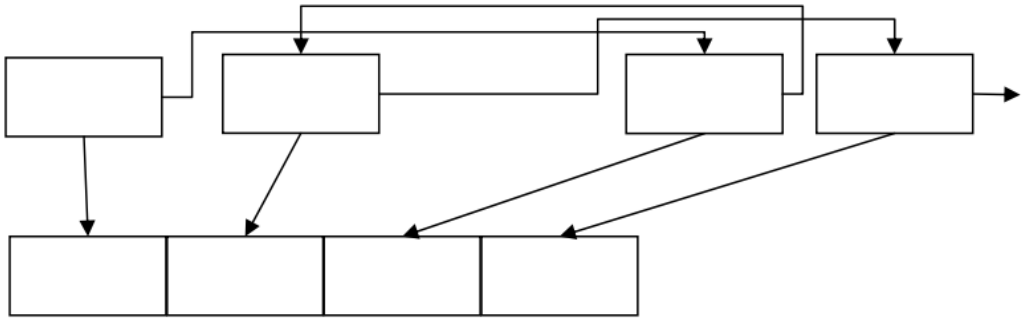


Figure 2.6.1 F1: Basic structure of linked-list access mechanism [3]

2.6.2 CoreMark Matrix Processing

Matrix processing is proposed as one of the most important algorithms in CoreMark as many algorithms use matrixes and arrays to perform calculations. So, matrix processing plays an important role in evaluating the efficiency of a processor. In the view of CoreMark, it performs some simple functions on the input matrixes, such as multiplication with a vector, a constant, or other matrix. Sometimes, CoreMark will tests on only a part of data in the matrix by extracting the bits from each matrix elements before operations perform. As in the list processing, CoreMark will also perform cyclic redundancy check (CRC) at the end of the iterations to make sure that all required tested functions have been performed to the data items.[3]

2.6.3 CoreMark State Machine Processing

Lastly, another main tasks that commonly performed by a processor core is to handle control statements such as if else, which are besides loops. A state machine supported if-else or switch statement thus becoming the ideal agent for testing a CPU core in terms of state machine processing. Generally, state machine handling can be subdivided into two common methods, which are employing a state transition table, or utilizing switch statements. In state machine processing, CoreMark put more focus on testing the switch statement, this is because CoreMark already test the state transition table when running the list processing algorithm when testing the load/store behaviours. Figure 2.6.3 F1 below shows the simplified view of the functionality tested in state machine processing. [3]

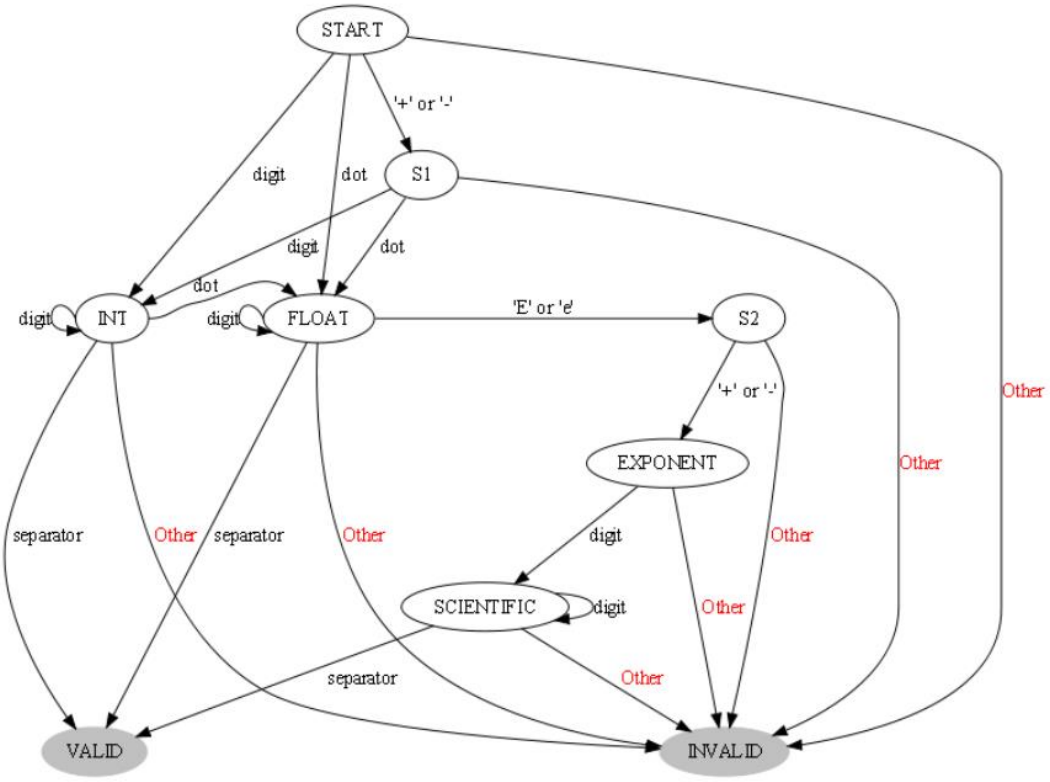


Figure 2.6.3 F1: Simplified view of overall functionality of CoreMark's state machine processing [3]

CHAPTER 3: Proposed Method/Approach

3.1 Proposed solution

To standardize the performance evaluation on the RISC32-E cryptography processors, for better and reliable results for comparison, a standard benchmarking system should be used. In this project, CoreMark, a standard benchmark certified by EEMBC is used as a reference benchmark source instead of developing a functional test without any references. The performance evaluation will perform on both RISC32-E processor without queue system (RISC32-E-NQ) and RISC32-E with queue system (RISC32-E-Q) and compare the results one to another so that users can have a better idea on how much efficient of RISC32-E-Q on cryptography operations compare to conventional implementation, RISC32-E-NQ and RISC32E.

3.2 Research Methodology for the project

3.2.1 Analyze the RISC32, RISC32-E and RISC32-E-Q processors

In order to port the benchmarking codes to the processors, it is important to study the designed architectures and know how to configure the ports and internal hardware like UART and internal clock in CP0. Besides that, memory mapping is also important ensure the instruction loaded into the correct memory spaces. So, some past works that are relevant will be reviewed in this project.

3.2.2 Setup the LLVM compilation toolchain

LLVM compilation toolchain is important in this project to compile the benchmarking codes which is in C language into processors understandable machine language. In this stage, the work by [4] will be reviewed to understand how the compilation toolchain works. Besides that, Linux commands to operate the LLVM compiler will be studied because the LLVM compilation toolchain is only working in Linux OS right now. So, an Ubuntu OS will be setup in a host computer. Besides that, to setup the LLVM compiler, CMake is also needed to be downloaded and install in the host computer. Configuration of LLVM will later be done after CMake installation using the codes “cmake -G "Unix Makefiles" ../llvm -DCMAKE_BUILD_TYPE=Debug -

DLLVM_TARGETS_TO_BUILD= "Mips;X86"" , which meant to configure LLVM compiler to output RISC32 understandable machine codes. One thing to mentioned here will be the source files will be compiled with different optimization level, from the least optimize level, -O0 up to the highest optimize level, -O3 to see the performance of the RISC32 processors under different workloads. -Os optimization level which means the code size optimization will also be one of the independent test set.

3.2.3 Setup the CoreMark benchmarking

The main objective of this project is to utilize the CoreMark benchmarking codes into the processors to get the idea of the performance of each processor. So, CoreMark codes will be studied to understand how the benchmarking system works to obtain accurate result. After that, minimal modification will be done to some c source files provided by EEMBC such as core_portme.c and ee_printf.c, to accommodate the desired functions like sending the results through UART or getting the accurate CPU clock cycles and thus calculate the timing for RISC32 processors.

3.3 RTL Modeling and Verification

The RTL level coding from previous works done by seniors is tested to ensure functionality correctness. Each block (smallest units) will be verified before they piece up together into unit level. In the precondition that all the hardware components can successfully meet the necessary specifications, they will be synthesis and implemented onto the Xilinx Artix-7 XC7A100T FPGA in Digilent Nexys 4 DDR board using Xilinx Vivado HLx 2020.2 IDE. Besides that, the RISC32 compilation toolchain will be installed on a host computer with Ubuntu 16.04 LTS operating system to compile the provided CoreMark codes into compatible codes to be executed on RISC32 processor. The results will be output from the processors through UART port.

3.4 Convert the CoreMark Test Program into RISC32 compatible codes

CoreMark is chosen to be the standard benchmark for the 2 architectures selected. It comes with some test code that needed to passthrough the RISC32 compilation

toolchain developed in the past projects before it can be used to test the functionality of the processors. The detail steps are shown below:

1. Tuned the CoreMark c language files given by the EEMBC to support the RISC32-E platform.
2. Redefine the ee_printf functions to send out the results through UART
3. Setup RISC32 on a FPGA board using Xilinx Vivado simulation tool.
4. Compile the code using the RISC32 compilation toolchain.
5. Run the project by loading the hex codes into RISC32 processors memory.
6. Observe the result through signal bitx_fifo_data_in[7:0] (The data will be loaded into this signal before send out using UART).

3.5 Calculate the CoreMark/MHz Score

After the simulation done, the CoreMark value will be generated from the RISC32. The CoreMark value has a similar meaning as the MIPS (millions instruction executed per second). To calculate the CoreMark/MHz score (the standard score to show how efficient a processor is relatively in the context of CoreMark benchmarking), CoreMark value must be divided by the clock speed used when the benchmark is performed, noted that the clock frequency needs to be calculated using clock cycle counts that can be obtained from coprocessor 0 (CP0) in RISC32, using the formula as below:

$$Clock\ Frequency(MHz) = \frac{programme\ execution\ time\ (c.c.)\ (obtain\ from\ CP0)}{programme\ execution\ time\ (ns)}$$

and the CoreMark score is calculated using the formula as follow:

$$CoreMark\ Score/MHz = \frac{CoreMark\ value\ (obtained\ from\ RISC32)}{Clock\ Frequency}$$

3.6 Technologies Involved

3.6.1 LLVM

LLVM is a collection of modular and reusable compiler and toolchain technologies. LLVM is famous for its usage that can be used to build up front end for nearly all programming languages and backends for any ISA. LLVM is designed around a language-independent intermediate representation (IR) that serves as a highly flexible, high-level assembly language that can be optimized using different modern source and target-independent optimizers, over multiple entries. The LLVM intermediate codes can be categorized into 3 categories: in-memory compiler IR which are going to be used to generate object files, on-disk bit code representation, and in the form of human understandable assembly language. The criteria mentioned above allows LLVM to be a capable compiler to deliver powerful intermediate representation which are used for compiler transformation and analysis. In shorts, LLVM provide a natural way to debug and visualize the transformation of the any kinds of programming language, no matter frontend or backend. LLVM is used in this project to compile the CoreMark benchmarking codes which are in c language, into RISC32 understandable machine codes. (Lim, 2019)

3.6.2 Xilinx Vivado

Xilinx Vivado is a software produced by Xilinx Incorporation for simulation, synthesis, and analysis of hardware description language designs. Similar to Xilinx Integrated Synthesis Environment (ISE), Vivado includes the in-build logic simulator. The components of Vivado are summarized as below:

- The Vivado high-level synthesis - A compiler that enables high level languages such as C, C++, and SystemC programs to be directly targeted to Xilinx devices without the requirements of building RTL manually.
- The Vivado simulator – A compiled-language simulator that supports mixed-language, encrypted IP, enhanced verification and Tcl scripts.

3.7 Timeline

3.7.1 Gantt chart for Project II

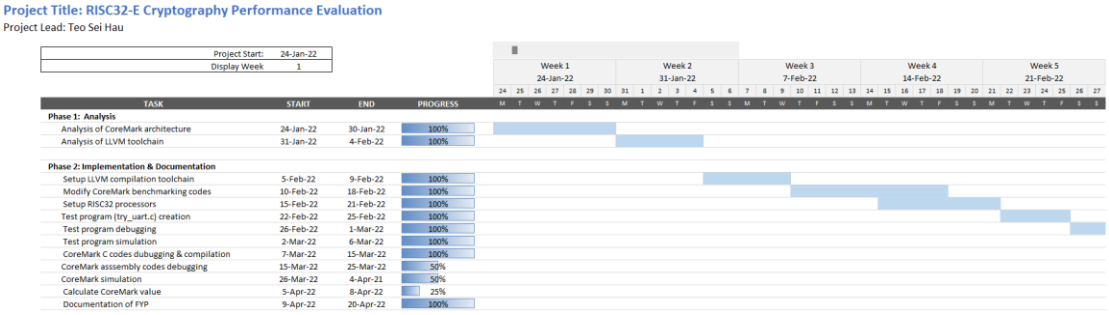


Figure 3.7.1 F1: Gantt chart for week 1 until week 5

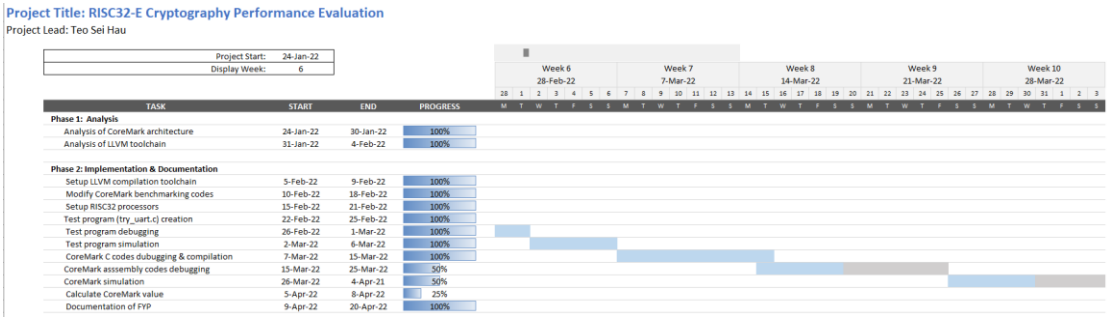


Figure 3.7.1 F2: Gantt chart for week 6 until week 10

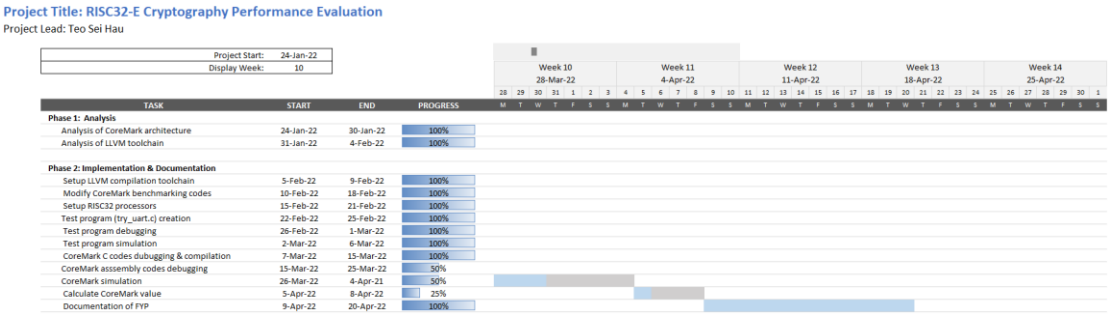


Figure 3.7.1 F3: Gantt chart for week 10 until week 14

CHAPTER 4: System Design

4.1 CoreMark Architecture Analysis

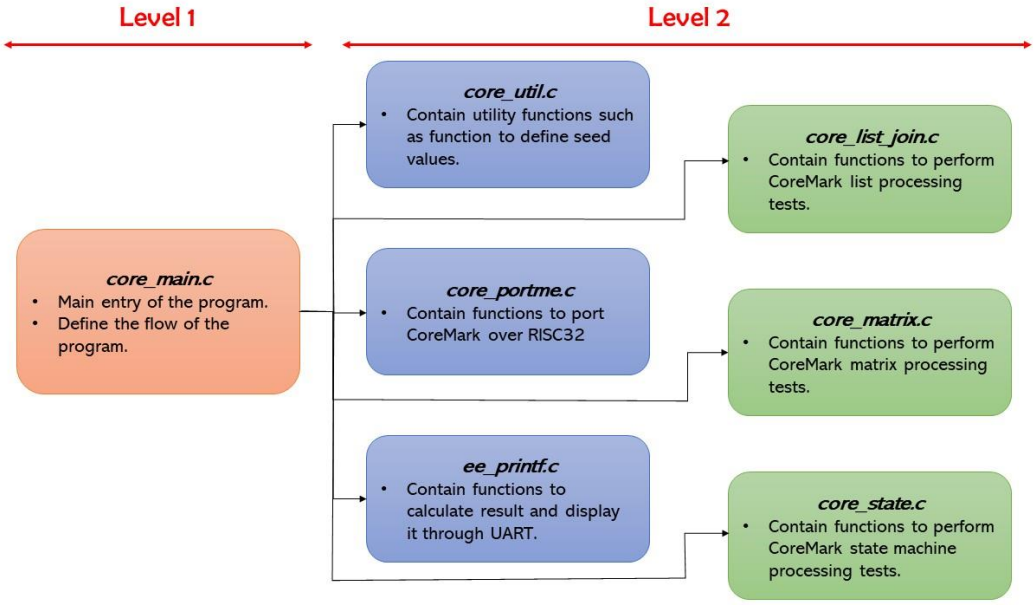


Figure 4.1 F1: CoreMark architecture

Figure 4.1 F1 above show the CoreMark architecture. Generally, the hierarchy of CoreMark architecture can be categories into 2 levels. The first level only contains core_main.c which is the main entry of the CoreMark program and functions to control the entire program flow. Level 2 consists of 6 files in total, which are core_util.c that stores utility functions such as to define the seed values, core_portme.c that stores functions like get time functions that need to be ported over RISC32, ee_printf.c that defines functions to print out the results in UART, core_list_join.c that contains functions to perform CoreMark list processing tests, core_matrix.c that stores functions to test on CoreMark matrix processing and also core_staete.c that contains functinons to perform CoreMark state machine processing tests.

Noted that Figure 4.1 above only shows the selected files, which will be used in this project. There are another file named cvt.c that originally contains in the CoreMark architecture, but it is not included here as cvt.c only contains functions to handle floating points, which are not supported by RISC32.

4.2 CoreMark Codes Analysis and Modification

To port the CoreMark testbench codes over to RISC32 processors, some minor modifications have been done to suit the hardware architecture. Section 4.2 discusses the modifications that have been carried out on the benchmark codes without changing its functions.

4.2.1 core_portme.h file

core_portme.h file includes configuration settings of all the necessary parameters for the testbench to run. For example, Figure 4.2.1 F1 shows the define of compiler version, compiler flag and the memory location to be allocate by the data block.

```
/* Definitions : COMPILER_VERSION, COMPILER_FLAGS, MEM_LOCATION
        Initialize these strings per platform
*/
#ifndef COMPILER_VERSION
    #define COMPILER_VERSION "GCC 5.4.0"
#endif
#ifndef COMPILER_FLAGS
    #define COMPILER_FLAGS "-o0"
#endif
#ifndef MEM_LOCATION
    #define MEM_LOCATION "STACK"
#endif
```

Figure 4.2.1 F1: Compiler version, compiler flags and memory location definitions

The configurations were done based on the capability of RISC32 processors, for example, RISC32 cannot support floating point, time.h functions, stdio.h functions, multithreads, and others. All this configuration were set in this file as shown in Figure 4.2.1 F2, 4.2.1 F3 and 4.2.1 F4 below.

```
/************************/
/* Data types and settings */
/************************/
/* Configuration : HAS_FLOAT
        Define to 1 if the platform supports floating point.
*/
#ifndef HAS_FLOAT
#define HAS_FLOAT 0
#endif

/* Configuration : HAS_TIME_H
        Define to 1 if platform has the time.h header file,
        and implementation of functions thereof.
*/
#ifndef HAS_TIME_H
#define HAS_TIME_H 0
#endif

/* Configuration : USE_CLOCK
        Define to 1 if platform has the time.h header file,
        and implementation of functions thereof.
*/
#ifndef USE_CLOCK
#define USE_CLOCK 0
#endif

/* Configuration : HAS_STDIO
        Define to 1 if the platform has stdio.h.
*/
#ifndef HAS_STDIO
#define HAS_STDIO 0
#endif
```

Figure 4.2.1 F2: Has_Float, Has_Time_H, Use_Clock and Has_STDIO configuration

```
/* Configuration : HAS_PRINTF
        Define to 1 if the platform has stdio.h and implements the printf
    function.
*/
#ifndef HAS_PRINTF
#define HAS_PRINTF 0
#endif

/* Configuration : SEED_METHOD
        Defines method to get seed values that cannot be computed at compile
    time.

        Valid values :
        SEED_ARG - from command line.
        SEED_FUNC - from a system function.
        SEED_VOLATILE - from volatile variables.
*/
#ifndef SEED_METHOD
    #define SEED_METHOD SEED_VOLATILE
#endif

/* Configuration : MEM_METHOD
        Defines method to get a block of memry.

        Valid values :
        MEM_MALLOC - for platforms that implement malloc and have malloc.h.
        MEM_STATIC - to use a static memory array.
        MEM_STACK - to allocate the data block on the stack (NYI).
*/
#ifndef MEM_METHOD
    #define MEM_METHOD MEM_STACK
#endif
```

Figure 4.2.1 F3: Has_printf, seed method and memory method configurations

```
/* Configuration : MULTITHREAD
        Define for parallel execution

        Valid values :
        1 - only one context (default).
        N>1 - will execute N copies in parallel.

        Note :
        If this flag is defined to more then 1, an implementation for launching
   parallel contexts must be defined.

        Two sample implementations are provided. Use <USE_PTHREAD> or <USE_FORK>
   to enable them.

        It is valid to have a different implementation of <core_start_parallel>
   and <core_end_parallel> in <core_portme.c>, to fit a particular architecture.
*/
#ifndef MULTITHREAD
    #define MULTITHREAD 1
    #define USE_PTHREAD 0
    #define USE_FORK    0
    #define USE_SOCKET  0
#endif
```

Figure 4.2.1 F4: Multithread configurations

Noted that the configurations done over here will affect how the whole program treats the data. For example, if HAS_FLOAT set to 1, the time_in_sec variable will be set to float type. Figure 4.2.1 F5 below show how HAS_FLOAT affect the flow of the program.

```
#if HAS_FLOAT
    ee_printf("Total time (secs): %f\n", time_in_secs(total_time));
    if (time_in_secs(total_time) > 0)
        ee_printf("Iterations/Sec   : %f\n",
                  default_num_contexts * results[0].iterations
                    / time_in_secs(total_time));
#else
    ee_printf("Total time (secs): %d\n", time_in_secs(total_time));
    if (time_in_secs(total_time) > 0)
        ee_printf("Iterations/Sec   : %d\n",
                  default_num_contexts * results[0].iterations
                    / time_in_secs(total_time));
#endif
```

Figure 4.2.1 F5: Example of how HAS_FLOAT configuration affect the program.

4.2.2 core_portme.c file

core_portme.c file generally contain the timing functions which define how to capture time and convert to seconds before the codes are ported to the supported platform. The original source codes implement the function using clock() function that required C standard library which is time.h. However, the function is not recognized by LLVM. So, emendations are done to the c language file by substituting the clock() function with the clock cycles counts by the timer inside hardware cp0 to achieve the same purpose. Figure 4.2.1 F1, F2 and F3 below shows the codes to perform timing calculation

```
#define NSECS_PER_SEC              ((long)1000)
#define CORETIMETYPE               uint32_t
#define MYTIMEDIFF(fin, ini)       ((fin) - (ini))
#define TIMER_RES_DIVIDER          1
#define SAMPLE_TIME_IMPLEMENTATION 1
#define EE_TICKS_PER_SEC           (NSECS_PER_SEC / TIMER_RES_DIVIDER)
```

Figure 4.2.2 F1: Emendations done on core_portme.c to calculate timing.

```
/* Function : start_time
        This function will be called right before starting the timed portion of
   the benchmark.

        Implementation may be capturing a system timer (as implemented in the
   example code) or zeroing some system parameters - e.g. setting the cpu clocks
   cycles to 0.
*/
void
start_time(void)
{
    // start timer, in cp0 register
    asm ("mtc0 $0, $9 \n\t");
}

/* Function : stop_time
        This function will be called right after ending the timed portion of the
   benchmark.

        Implementation may be capturing a system timer (as implemented in the
   example code) or other system parameters - e.g. reading the current value of
   cpu cycles counter.
*/
void
stop_time(void)
{
    // Move the result to register $v0, and this register will mapped to stop_time_val
    asm ("mfc0 $v0, $9 \n\t" : "=r"(stop_time_val));
}
```

Figure 4.2.2 F2: start_time and stop_time function.

```
CORE_TICKS
get_time(void)
{
    CORE_TICKS elapsed = (CORE_TICKS)(MYTIMEDIFF(stop_time_val, start_time_val));
    return elapsed;
}

/* Function : time_in_secs
        Convert the value returned by get_time to seconds.

        The <secs_ret> type is used to accomodate systems with no support for
   floating point. Default implementation implemented by the EE_TICKS_PER_SEC
   macro above.
*/
secs_ret
time_in_secs(CORE_TICKS ticks)
{
    secs_ret retval = ((secs_ret)ticks) / (secs_ret)EE_TICKS_PER_SEC;
    return retval;
}
```

Figure 4.2.2 F3: get_time and function to calculate time in terms of seconds.

Besides that, the iterations are set to 3 as shown in Figure 4.2.2 F4 which correspond to roughly 10 seconds. CoreMark specified the CPU must run for at least 10 seconds. To make the benchmark reliable, the program will look for five seed values which are not deterministic during run time. The seed values mean to direct the initialization of values in the data structures in a way which is opaque by user. In this project, the seed values are set based on the recommendations from the CoreMark testbench coding distributors. Figure 4.2.1 F4 below shows the seed values summary for all the possible run modes. In this project, validation run mode is chosen as this is the most basic run mode to test on the program. Validation run mode only required roughly 10 seconds as recommended from CoreMark coding distributors, which match with the iterations set to 3 as discussed above.

```
#define ITERATIONS 3
extern void cache_init(void);

#if VALIDATION_RUN
    volatile ee_s32 seed1_volatile = 0x3415;
    volatile ee_s32 seed2_volatile = 0x3415;
    volatile ee_s32 seed3_volatile = 0x66;
#endif
#if PERFORMANCE_RUN
    volatile ee_s32 seed1_volatile = 0x0;
    volatile ee_s32 seed2_volatile = 0x0;
    volatile ee_s32 seed3_volatile = 0x66;
#endif
#if PROFILE_RUN
    volatile ee_s32 seed1_volatile = 0x8;
    volatile ee_s32 seed2_volatile = 0x8;
    volatile ee_s32 seed3_volatile = 0x8;
#endif
volatile ee_s32 seed4_volatile = ITERATIONS;
volatile ee_s32 seed5_volatile = 0;
```

Figure 4.2.2 F4: Iteration and seeds value summary

4.2.3 ee_printf.c

ee_printf.c defines the way of outputting the result. In this project, UART is used to output the CoreMark score . So, ee_printf.c is modified to send the results out through UART. Figure 4.2.3 F1 and F2 below show the functions to send a character through uorisc_ua_tx_data port. Please refer to Appendix B for the RISC32 port definitions.

```
void
uart_send_char(char c)
{
    /*  Output of a char to a UART usually follows the following model:
        Wait until UART is ready
        Write char to UART
        Wait until UART is done

        Or in code:
        while (*UART_CONTROL_ADDRESS != UART_READY);
        *UART_DATA_ADDRESS = c;
        while (*UART_CONTROL_ADDRESS != UART_READY);

        Check the UART sample code on your platform or the board
      documentation.
    */
    UDR = c;
}
```

Figure 4.2.3 F1: uart_send_char function definition

```
int
ee_printf(const char *fmt, ...)
{
    char    buf[1024], *p;
    va_list args;
    int     n = 0;

    va_start(args, fmt);
    ee_vsprintf(buf, fmt, args);
    va_end(args);
    p = buf;

    //Setup UART
    UCR = UCR | 0x82; //UARTCR = 10000010, UARTEN=>1, TXEIE->0, 9600(baud)->010

    while (*p)
    {
        uart_send_char(*p);
        n++;
        p++;
    }

    //Disable UART
    UCR = UCR | 0x02; //UARTCR = 10000010, UARTEN=>1, TXEIE->0, 9600(baud)->010

    return n;
}
```

Figure 4.2.3 F2: ee_printf function definition

4.3 LLVM Compilation Toolchain Setup

LLVM in this project played a vital role as a translator between human and RISC32 processor. The LLVM is setup on a host computer with Ubuntu 16.04 LTS operating system. The LLVM is tested with cp2_CTR_256_v1.c, the sample test file provided by Mr Mok Kai Ming to ensure the functionality correctness. In this project, a bash script file is written to enhance readability and bugs identification throughout the compilation process. Figure 4.3 F1 below shows some of the important Linux commands that included in bash script file named compile.sh to compile c files into txt files, which contain the instructions in hex codes. Noted that the files will be compile using different optimization level, from least optimization, -O0 up to highest optimization level, -O3, to test the processor under different workload. The extra optimization level, -Os which means code size optimization will also be used.

```
# codes start
clang  -"$o_level" --target=mips-unknown-linux -mips32 $c_portme $c_list $c_main $c_matrix $c_prif $c_state $c_util $c_cvt -emit-llvm -S

## assembly files creation
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_portme" -o "asm_files/$asm_portme" -debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_list" -o "asm_files/$asm_list" -debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_main" -o "asm_files/$asm_main" -debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_matrix" -o "asm_files/$asm_matrix" -debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_prif" -o "asm_files/$asm_prif" -debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_state" -o "asm_files/$asm_state" -debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_util" -o "asm_files/$asm_util" -debug &> debug00.log
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_cvt" -o "asm_files/$asm_cvt" -debug &> debug00.log

## object files creation
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_portme" -filetype=obj -o "obj_files/$obj_portme"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_list" -filetype=obj -o "obj_files/$obj_list"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_main" -filetype=obj -o "obj_files/$obj_main"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_matrix" -filetype=obj -o "obj_files/$obj_matrix"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_prif" -filetype=obj -o "obj_files/$obj_prif"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_state" -filetype=obj -o "obj_files/$obj_state"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_util" -filetype=obj -o "obj_files/$obj_util"
llc -"$o_level2" -disable-mips-delay-filler -march=mips -mcpu=risc32 -mattr=+risc32 "$ll_cvt" -filetype=obj -o "obj_files/$obj_cvt"

## link files
ld.lld --Bstatic -script script.ld "obj_files/$obj_portme" "obj_files/$obj_list" "obj_files/$obj_main" "obj_files/$obj_matrix" "obj_files/$obj_prif" "obj_files/$obj_state" "obj_files/$obj_util" "obj_files/$obj_cvt" -o '
[ -e "$bin_file" ] && echo "$bin_file exist!Deleting file..." && rm "$bin_file" || echo "No existing $bin_file file found."
[ -e "$txt_file" ] && echo "$txt_file exist!Deleting file..." && rm "$txt_file" || echo "No existing $txt_file file found."
[ -e "$dis_txt_file" ] && echo "$dis_txt_file exist!Deleting file..." && rm  "$dis_txt_file" || echo "No existing $dis_txt_file file found."
mips-linux-gnu-objcopy -j .text "$elf_file" -O binary "$bin_file"
mips-linux-gnu-objdump -zS "$elf_file" > "$dis_txt_file"
hexdump -v -e '1/1 "%02X\n"' "$bin_file" > "$txt_file"
```

Figure 4.3 F1: Extraction of bash script files that include important Linux commands to compile c files into hex codes files

CHAPTER 5: CoreMark Implementation

5.1 LLVM Installation and Compilation

5.1.1 Testing the LLVM Compilation via UART Communication

try_uart.c is a test file to test the functionalities of LLVM and UART of RISC32. Figure 5.1.1 F1 shows codes of try_uart.c in c languages.

```c
#include <stdbool.h>
#include <stdint.h>
#include "io.h"

int
main(void)
{
    // try to send "Hi PQ"
    //Setup UART
    UCR = UCR | 0x82; //UARTCR = 10000010, UARTEN=>1, TXEIE->0, 9600(baud)->010

    //Send data
    UDR = (uint8_t)0x48; //H
    UDR = (uint8_t)0x69; //i
    UDR = (uint8_t)0x20; //space
    UDR = (uint8_t)0x50; //P
    UDR = (uint8_t)0x51; //Q

    UCR = UCR | 0x82; //UARTCR = 10000010, UARTEN=>1, TXEIE->0, 9600(baud)->010
    while(1);
    return 0;
}
```

Figure 5.1.1 F1: try_uart.c implementation

After conversion using LLVM, several files were created. Figure 5.1.1 F2 below show one of the most significant files generated using LLVM, which is the assembly codes after translation.

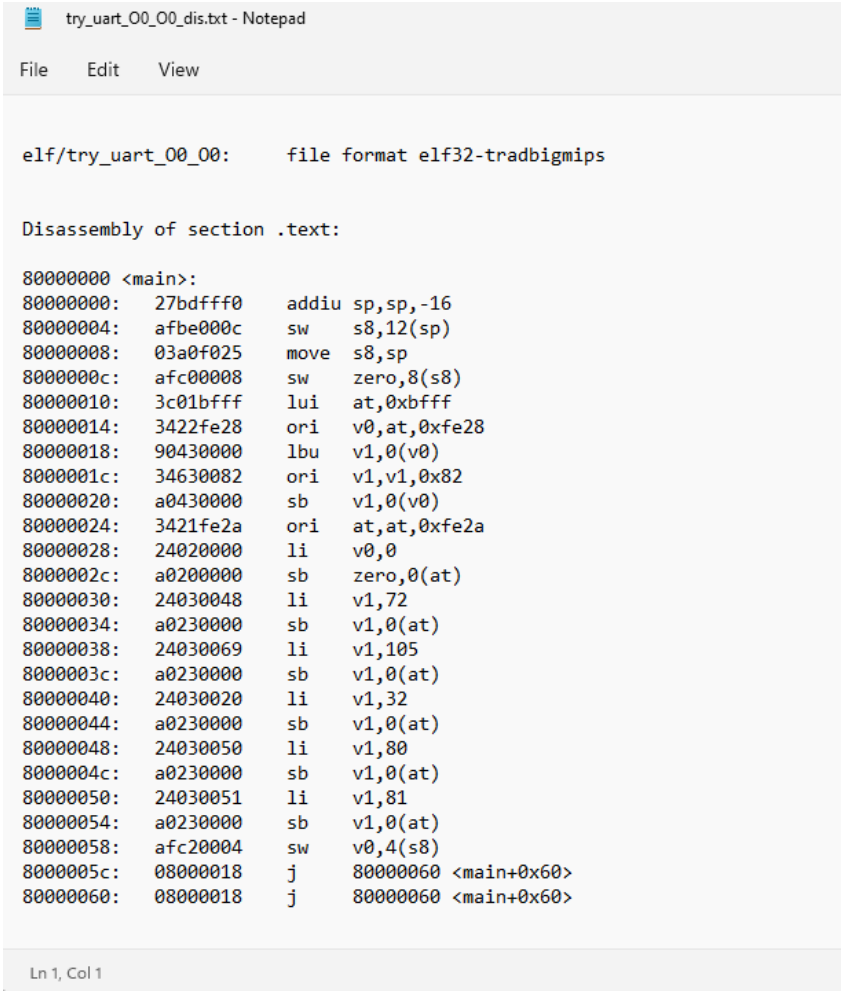


Figure 5.1.1 F2: Assembly code of try_uart after conversion using LLVM with optimization level of -o0

5.1.2 CoreMark Assembly Codes

CoreMark contains several c language files, which needed to be link together using lld linker. All the c files have to be compile using clang independently and later lld linker will link them together using labels, where labels will be the function names contain over the files. Figure 5.1.2 F1 below shows part of the assembly file of CoreMark codes generated using LLVM.

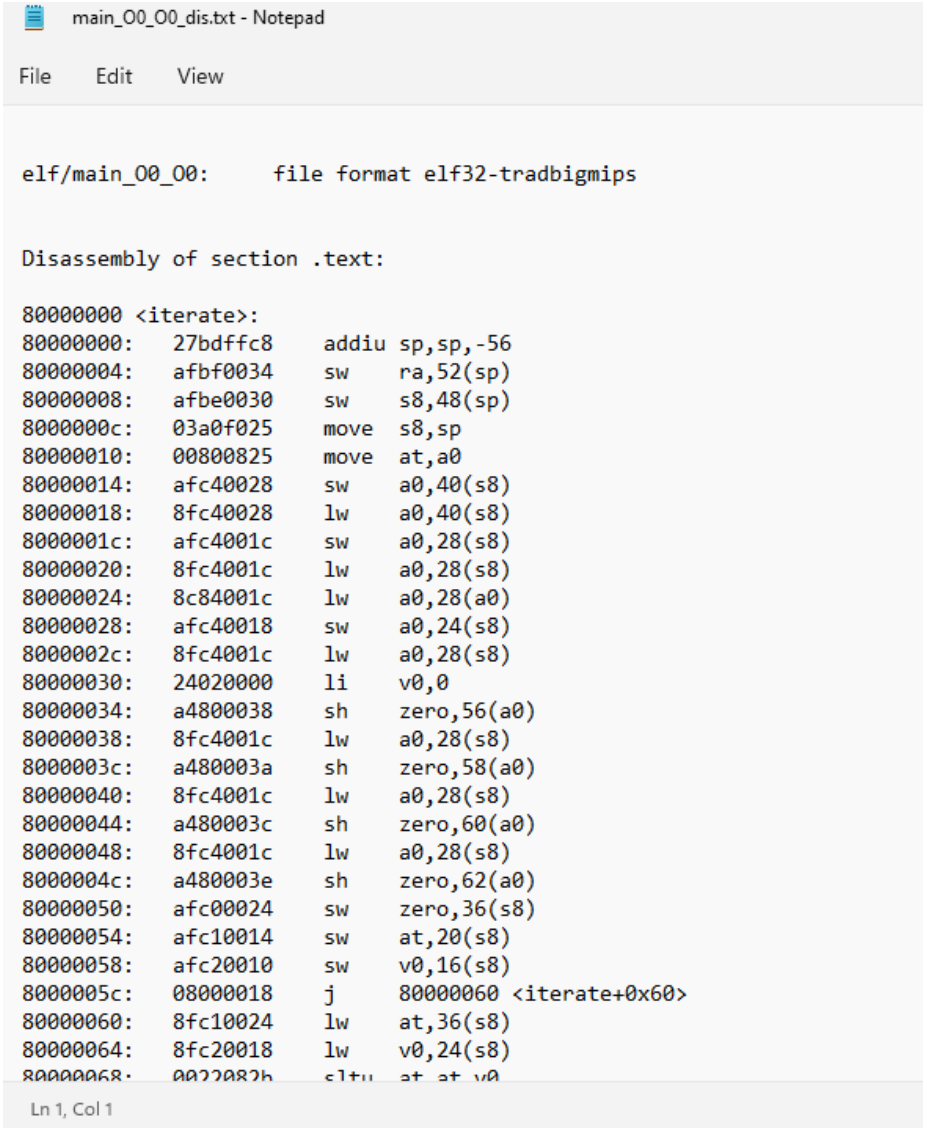


Figure 5.1.2 F1: Assembly codes generated after all the independent files linked together.

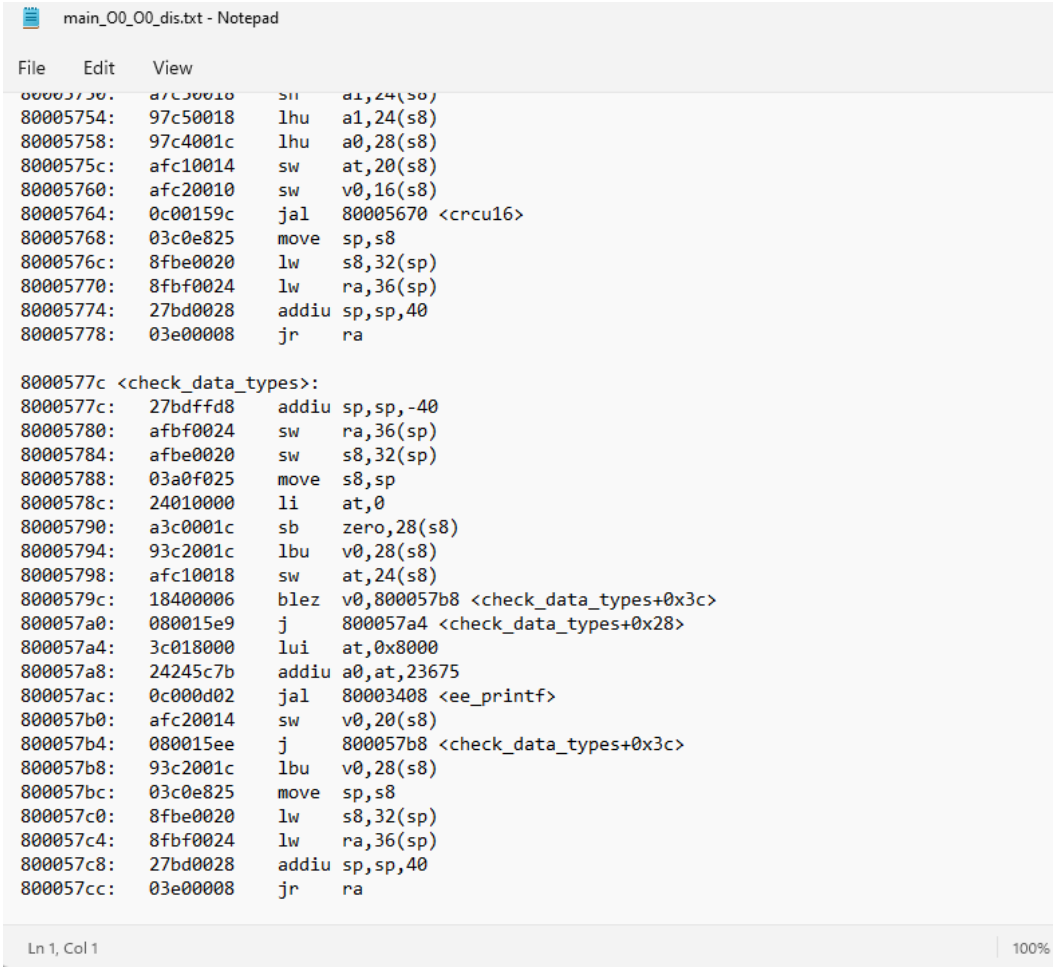


Figure 5.1.2 F2: Assembly codes of the last function, "check data types" in the CoreMark programme

As shown in Figure 5.1.2 F1 and 5.1.2 F2 above, CoreMark programme start from address 0x8000_0000 and end at address 0x8000_57cc. This also implies that there are 22477 lines of assembly codes that will need to use up 22477 words of memory in the i-cache as shown in calculation below:

$$(0x8000\_57CC - 0x8000\_0000) + 1 = 22477$$

According to Figure 2.5 F1 allocated in chapter 2.5, RISC32's i-cache start from address 0x8000_0000 and end at address 0x8001_b400. The total memory available for instructions are 111617 words, as shown in the calculation below:

$$(0x8001\_b400 - 0x8000\_0000) + 1 = 111617$$

Hence, from the instruction memory capacity point of view, the compiled CoreMark codes, that is the assembly instruction programme, can be successfully fit into the i-cache. This also implies there will be no caching for the instruction cache memory during programme execution.

5.2 Simulation on RISC32 Processor

5.2.1 RISC32 Testbench

A testbench is an environment used to verify the correctness of a design or model. In this project, a testbench named tb_r32_pipeline_old_1.v which is written in Verilog code is used to test on functional behaviors of RISC32 processors. The codes of tb_r32_pipeline_old_1.v is shown below:

```
`timescale 1ns / 10ps
`default_nettype none
module tb_r32_pipeline();
//declaration
//======= INPUT =======
// System signal

reg             tb_u_clk;
reg             tb_u_rst;

wire    tb_u_spi_mosi;
wire    tb_u_spi_miso;
wire    tb_u_spi_sclk;
wire    tb_u_spi_ss_n;

wire    tb_u_fc_sclk;
wire    tb_u_fc_ss;
wire    tb_u_fc_MOSI;
wire    tb_u_fc_MISO1;
wire    tb_u_fc_MISO2;
wire    tb_u_fc_MISO3;

wire    tb_ua_tx_rx;
wire [31:0]             tb_GPIO;


crisc dut_c_risc
(//********** INSTANTIATION *************
//======= INPUT =======
//GPIO
.urisc_GPIO(tb_GPIO),

//SPI controller
```

```
.uiorisc_spi_mosi(tb_u_spi_mosi),
.uiorisc_spi_miso(tb_u_spi_miso),
.uiorisc_spi_sclk(tb_u_spi_sclk),
.uiorisc_spi_ss_n(tb_u_spi_ss_n),

//UART controller
.uorisc_ua_tx_data(tb_ua_tx_rx),
//.uorisc_ua_rts(),
.uirisc_ua_rx_data(tb_ua_tx_rx),
//.uirisc_ua_cts(1'b0),

//FLASH controller
.uorisc_fc_sclk(tb_u_fc_sclk),
.uiorisc_fc_MOSI(tb_u_fc_MOSI),
.uirisc_fc_MISO1(tb_u_fc_MISO1),
.uirisc_fc_MISO2(tb_u_fc_MISO2),
.uirisc_fc_MISO3(tb_u_fc_MISO3),
.uorisc_fc_ss(tb_u_fc_ss),

.uirisc_adc_VP(),
.uirisc_adc_VN(),
.uirisc_adc_VAUXP3(),
.uirisc_adc_VAUXN3(),

// System signal
.uirisc_clk_100mhz(tb_u_clk),
.uirisc_rst(tb_u_rst)
);

M25LC1024 SPI_EEPROM
(.SO(tb_u_spi_miso),
.SI(tb_u_spi_mosi),
.SCK(tb_u_spi_sclk),
.CS_N(tb_u_spi_ss_n),
.WP_N(1'b1),
.HOLD_N(1'b1),
.RESET(~tb_u_rst));

s25fl128s SPI_flash
(.SI(tb_u_fc_MOSI), //IO0
.SO(tb_u_fc_MISO1), //IO1
.SCK(tb_u_fc_sclk),
.CSNeg(tb_u_fc_ss),
.RSTNeg(tb_u_rst),
.WPNeg(tb_u_fc_MISO2), //IO2
.HOLDNeg(tb_u_fc_MISO3));  //IO3

//*********************************
//Clock waveform generation
`ifdef MODEL_TECH
initial tb_u_clk <= 1'b1;
always #25 tb_u_clk = ~tb_u_clk; //assume 20MHz
`else
initial tb_u_clk <= 1'b1;
always #5 tb_u_clk = ~tb_u_clk; //assume 100MHz
`endif

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Signals initialization.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

//========================
```

```
always@(posedge tb_r32_pipeline.dut_c_risc.urisc_clk)begin
 if(tb_r32_pipeline.dut_c_risc.u_datapath.uodp_if_pseudo_pc==`END_PC)
         $stop;
end

//read memory to get instruction
initial begin
$readmemh("program.txt",tb_r32_pipeline.SPI_flash.Mem);
$readmemh("exc_handler.txt",tb_r32_pipeline.SPI_flash.Mem);

tb_u_rst <= 1'b1;
repeat(1)@(posedge tb_u_clk);
tb_u_rst <= 1'b0;
repeat(30000)@(posedge tb_u_clk);
#11;
tb_u_rst <= 1'b1;
end
endmodule
```

Noted that after the c language programs were converted into hex code, the hex code is stored in program.txt which will be loaded into the instruction memory of RISC32 processor before the simulation start. The exception handle hex code is store inside exc_handler.txt. It will be loaded into exception handler memory space which starts from 0x8001_b400.

5.2.2 try_uart.c

The machine codes of the try_uart.c generated by LLVM is loaded into the RISC32 instruction memory as described in section 5.2.1. Figure 5.2.2 F1 below show the results of the try_uart program. “Hi PQ” is successfully sent out using UART in RISC32.

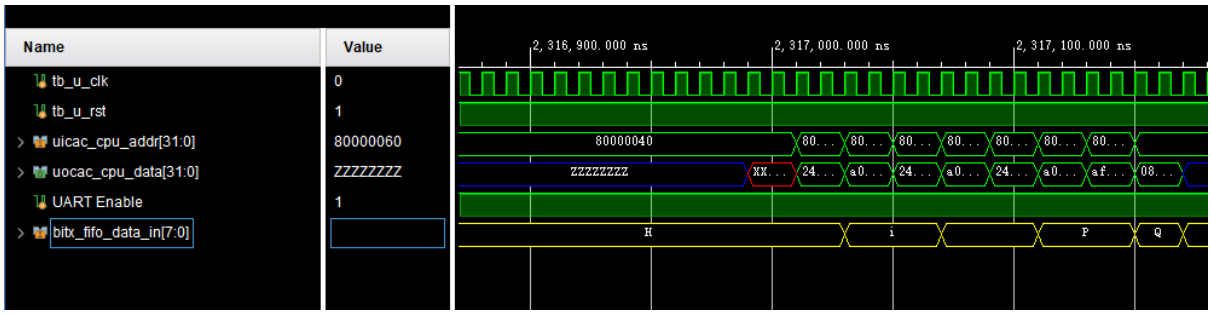


Figure 5.2.2 F1: Final result of try_uart program

Figure 5.2.2 F1 above shown that RISC32 can send out character one at a time through UART. So, modification is done based on this function to send the CoreMark score via UART of the RISC32. So, when the CoreMark codes are ported to the FPGA board, the CoreMark score and

other important messages such as error messages, total CPU run time, and others can be viewed through bitx_fifo_data_in signal. After that, CoreMark/MHz value can be calculated.

5.2.3 CoreMark Code Simulation and Debugging

The CoreMark code in the RISC32 Verilog model can be simulated but due to some logical bugs, the output is not reliable. The CoreMark program wishes to send out result through UART using a function named “ee_printf”. ee_printf is a function similar to printf defined in stdio.h, C standard library, just that the characters are send out using UART. The problem started when the programme runs in a forever loop inside the ee_vsprintf() function. The ee_vsprintf() is used by ee_printf() to perform format conversion based on the format specifier. For example, ee_printf(“Hello %s”, “Jason”). Due to the forever loop problem, the CoreMark programme is unable to proceed to calculate the CoreMark value and send out via UART. Figure 5.2.3 F1 and 5.2.3 F2 below show the infinite looping behaviors of CoreMark Code in the simulation.

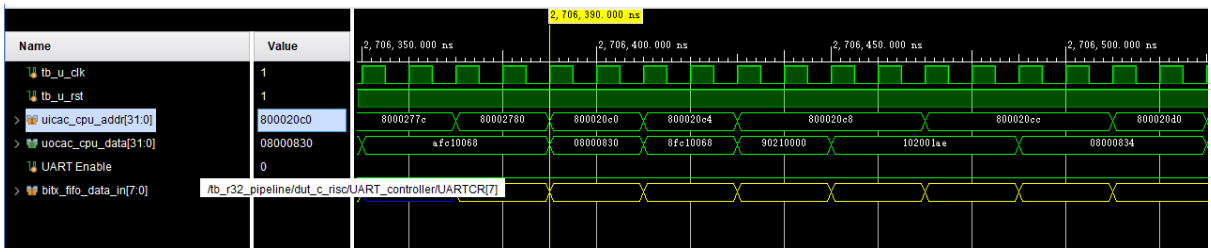


Figure 5.2.3 F1: First iteration observed in simulation

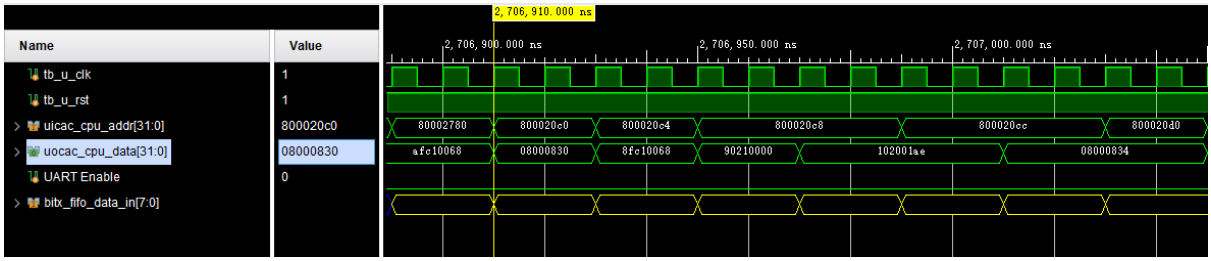


Figure 5.2.3 F2: Second iteration observed in simulation

The infinite looping behaviors continue, and the corresponding instructions are shown in Figure 5.2.3 F3 and Figure 5.2.3 F4 below:

```
800020c0:       8fc10068        lw      at,104(s8)
800020c4:       90210000        lbu     at,0(at)
800020c8:       102001ae        beqz    at,80002784 <ee_vsprintf+0x704>
800020cc:       08000834        j       800020d0 <ee_vsprintf+0x50>
800020d0:       8fc10068        lw      at,104(s8)
800020d4:       80210000        lb      at,0(at)
800020d8:       24020025        li      v0,37
800020dc:       10220008        beq     at,v0,80002100 <ee_vsprintf+0x80>
800020e0:       08000839        j       800020e4 <ee_vsprintf+0x64>
800020e4:       8fc10068        lw      at,104(s8)
800020e8:       90210000        lbu     at,0(at)
800020ec:       8fc20050        lw      v0,80(s8)
800020f0:       24430001        addiu   v1,v0,1
800020f4:       afc30050        sw      v1,80(s8)
800020f8:       a0410000        sb      at,0(v0)
800020fc:       080009dd        j       80002774 <ee_vsprintf+0x6f4>
80002100:       afc00048        sw      zero,72(s8)
80002104:       08000842        j       80002108 <ee_vsprintf+0x88>
80002108:       8fc10068        lw      at,104(s8)
8000210c:       24210001        addiu   at,at,1
80002110:       afc10068        sw      at,104(s8)
80002114:       8fc10068        lw      at,104(s8)
80002118:       80210000        lb      at,0(at)
8000211c:       2421ffe0        addiu   at,at,-32
80002120:       2c220011        sltiu   v0,at,17
80002124:       afc1002c        sw      at,44(s8)
80002128:       1040001a        beqz    v0,80002194 <ee_vsprintf+0x114>
8000212c:       8fc1002c        lw      at,44(s8)
80002130:       00011080        sll     v0,at,0x2
80002134:       3c038000        lui     v1,0x8000
80002138:       00431021        addu    v0,v0,v1
8000213c:       8c424490        lw      v0,17552(v0)
80002140:       00400008        jr      v0
80002144:       8fc10048        lw      at,72(s8)
80002148:       34210010        ori     at,at,0x10
8000214c:       afc10048        sw      at,72(s8)
80002150:       08000842        j       80002108 <ee_vsprintf+0x88>
80002154:       8fc10048        lw      at,72(s8)
80002158:       34210004        ori     at,at,0x4
8000215c:       afc10048        sw      at,72(s8)
80002160:       08000842        j       80002108 <ee_vsprintf+0x88>
```

Figure 5.2.3 F3: Starting address (0x8000_20c0) of infinite looping behavior

```
800026f0:       afc20064        sw      v0,100(s8)
800026f4:       8c210000        lw      at,0(at)
800026f8:       afc1005c        sw      at,92(s8)
800026fc:       080009d1        j       80002744 <ee_vsprintf+0x6c4>
80002700:       93c1004b        lbu     at,75(s8)
80002704:       30210002        andi    at,at,0x2
80002708:       10200007        beqz    at,80002728 <ee_vsprintf+0x6a8>
8000270c:       080009c4        j       80002710 <ee_vsprintf+0x690>
80002710:       8fc10064        lw      at,100(s8)
80002714:       24220004        addiu   v0,at,4
80002718:       afc20064        sw      v0,100(s8)
8000271c:       8c210000        lw      at,0(at)
80002720:       afc1005c        sw      at,92(s8)
80002724:       080009d0        j       80002740 <ee_vsprintf+0x6c0>
80002728:       8fc10064        lw      at,100(s8)
8000272c:       24220004        addiu   v0,at,4
80002730:       afc20064        sw      v0,100(s8)
80002734:       8c210000        lw      at,0(at)
80002738:       afc1005c        sw      at,92(s8)
8000273c:       080009d0        j       80002740 <ee_vsprintf+0x6c0>
80002740:       080009d1        j       80002744 <ee_vsprintf+0x6c4>
80002744:       8fc40050        lw      a0,80(s8)
80002748:       8fc5005c        lw      a1,92(s8)
8000274c:       8fc60054        lw      a2,84(s8)
80002750:       8fc70044        lw      a3,68(s8)
80002754:       8fc10040        lw      at,64(s8)
80002758:       8fc20048        lw      v0,72(s8)
8000275c:       03a01825        move    v1,sp
80002760:       ac620014        sw      v0,20(v1)
80002764:       ac610010        sw      at,16(v1)
80002768:       0c000a42        jal     80002908 <number>
8000276c:       afc20050        sw      v0,80(s8)
80002770:       080009dd        j       80002774 <ee_vsprintf+0x6f4>
80002774:       8fc10068        lw      at,104(s8)
80002778:       24210001        addiu   at,at,1
8000277c:       afc10068        sw      at,104(s8)
80002780:       08000830        j       800020c0 <ee_vsprintf+0x40>
```

Figure 5.2.3 F4: Ending address (0x8000_2780) of infinite looping behaviors

After analysis, the infinite looping cause by instruction under ee_vsprintf function which stored in address 0x8000_20dc, and correspond to instruction “beq at, v0, 80002100”. However, the processor never reached instruction with address of 0x8000_2100 due to value of register “at” is never equal to value of register “v0”, and this is the start of infinite looping behavior. This part requires further investigation in the LLVM compilation behavior.

CHAPTER 6: Conclusion and Recommendation

6.1 Conclusion

In conclusion, three out of four objectives have been met in this project, and the last objective is done partially. The first objective mentioned for this project have been achieved, where CoreMark architecture and components had been analyzed for functions selection and modification purposes. The second objective for this project, which is to develop a suitable subset of CoreMark programme for the RISC32 processors had been completed. The original CoreMark source codes had been modified and selection had been done so that the CoreMark programme can be port over RISC32 processors. The third objective of this project had been met, where the LLVM was setup successfully and was functioning to convert C language source code into assembly codes and hex codes. Besides that, CoreMark programme had been compiled and debug successfully. The length of the whole programme is also checked so that it can fit into the i-cache of RISC32 processors. So, CoreMark programme is ready to be ported over RISC32 processors. The last objective of this project, to simulate the CoreMark programme in the various RISC32 versions for comparison, is only done partially. The CoreMark programme was simulated using RISC32 processor, however, the programme was stucked in an infinite loop. The problem is suspected to be caused by LLVM compiler where it may generate wrong instructions when compiling C language source codes. Comparison have not done to various RISC32 versions.

6.2 Future Work and Recommendation

In the future, LLVM functionalities need to be checked and make sure that it can generate instructions correctly based on the C language source code given. One way to check LLVM functionalities is to compare the assembly codes generated with the one generated using GCC compiler. Next, after the simulation successfully run on RISC32 processor. The CoreMark score need to be calculated for various RISC32 versions and comparison should be made for each of them.

Bibliography

[1] M. Abd-El-Barr and H. El-Rewini, *Fundamentals of Computer Organization and Architecture*. 2005.

[2] U. Nanda and S. K. Pattnaik, "Universal Asynchronous Receiver and Transmitter (UART)," *ICACCS 2016 - 3rd Int. Conf. Adv. Comput. Commun. Syst. Bringing to Table, Futur. Technol. from Arround Globe*, 2016, doi: 10.1109/ICACCS.2016.7586376.

[3] S. Gal-on and M. Levy, "Exploring CoreMark™ - A Benchmark Maximizing Simplicity and Efficacy," *Embed. Microprocess. Benchmark Consort.*, 2012, [Online]. Available: www.eembc.org.

[4] J. C. See, K. M. Mok, W. K. Lee, and H. G. Goh, "RISC32-E: Field programmable gate array based sensor node with queue system to support fast encryption in Industrial Internet of Things applications," *Int. J. Circuit Theory Appl.*, vol. 48, no. 8, pp. 1209–1226, 2020, doi: 10.1002/cta.2797.

[5] W. P. Kiat, "THE DESIGN OF AN FPGA-BASED PROCESSOR WITH RECONFIGURABLE PROCESSOR EXECUTION STRUCTURE FOR INTERNET OF THINGS (IoT) APPLICATIONS KIAT WEI PAU MASTER OF SCIENCE (COMPUTER SCIENCE) FACULTYOF INFORMATION AND COMMUNICATION TECHNOLOGY UNIVERSITI TUNKU ABD," 2018.

Appendix A: MIPS Green Sheet

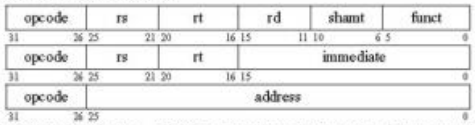
MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FORMAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

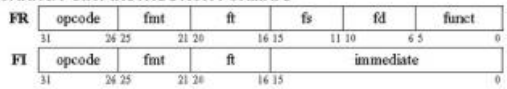
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FORMAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (x is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/ /1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

OPCODES, BASE CONVERSION, ASCII SYMBOLS ③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal	Hexadecimal	ASCII Character	Decimal	Hexadecimal	ASCII Character
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

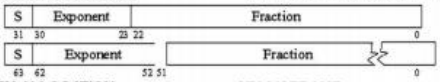
(1) opcode(31:26) == 0
(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)==17_{ten} (11_{hex}) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Single Precision Bias = 127, Double Precision Bias = 1023.

IEEE Single Precision and Double Precision Formats:



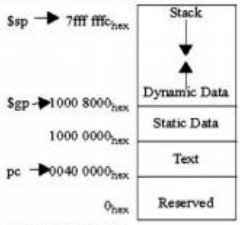
④

IEEE 754 Symbols

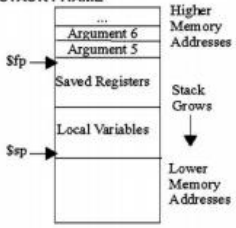
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

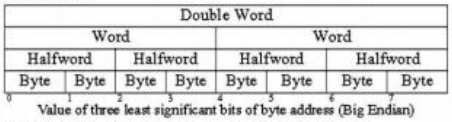
MEMORY ALLOCATION



STACK FRAME

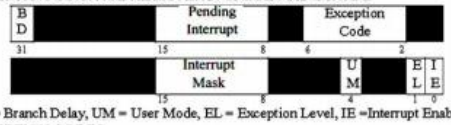


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

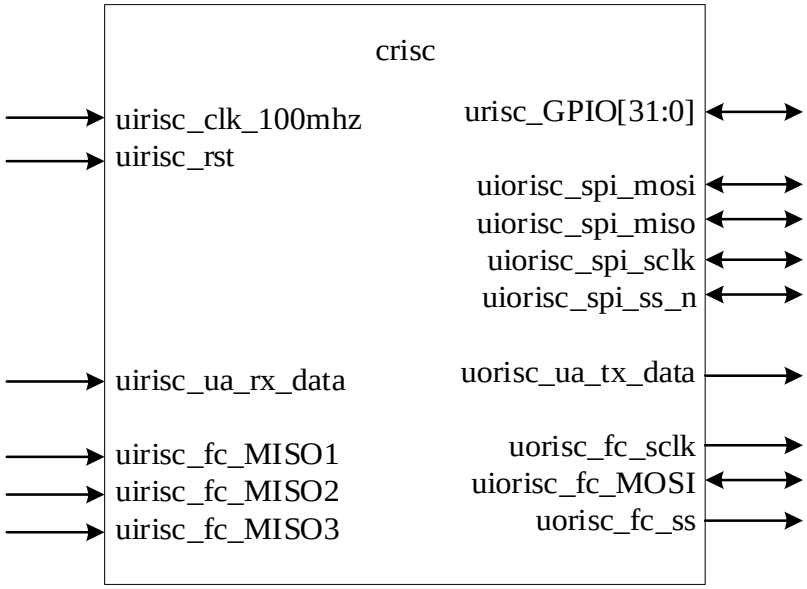
Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	milli-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

Appendix B: Chip Interface of RISC32 processor



Bi-weekly Report

FINAL YEAR PROJECT BI-WEEKLY REPORT

(Project II)

Trimester, Year: Jan 2022	**Study week no.:** 02
Student Name & ID: Teo Sei Hau 18ACB03719	
Supervisor: Mr. Teoh Shen Khang and Mr. Mok Kai Ming	
Project Title: RISC32-E Cryptography Performance Evaluation	

1. WORK DONE

- Analysis of CoreMark architecture for selection and modification purposes.
- Analysis of LLVM compilation toolchain.
- Install Ubuntu version 16.04 LTS on a laptop.
- Setup LLVM compilation toolchain in the laptop with Ubuntu Operating System.

2. WORK TO BE DONE

- Configure LLVM settings so that it output codes for RISC32 processors.
- Select and modify CoreMark codes so that the whole programme can port over RISC32 processors.

3. PROBLEMS ENCOUNTERED

- Clang of LLVM cannot recognizes some functions from the C standard library such as printf from “stdio.h”.
- Need to figure out a way to create functions to calculate the timing for CoreMark programme.

4. SELF EVALUATION OF THE PROGRESS

- The progress is still in the plan as in Figure 3.7.1, Gantt chart for week 1 until week 5.

TEOH

Supervisor’s signature

Student’s signature

FINAL YEAR PROJECT BI-WEEKLY REPORT

(Project II)

Trimester, Year: Jan 2022	**Study week no.:** 04
Student Name & ID: Teo Sei Hau 18ACB03719	
Supervisor: Mr. Teoh Shen Khang and Mr. Mok Kai Ming	
Project Title: RISC32-E Cryptography Performance Evaluation	

1. WORK DONE

- Configure the LLVM compilation toolchain settings to map the hex codes generated to RISC32-E instruction set.
- Analysis the timer of RISC32 processors in coprocessor 0 (cp0).
- Analysis the UART of RISC32 processors.
- Modify the timing functions of the CoreMark programme to calculate timing based on timer of RISC32.
- All the configurations in core_portme.h header files were set accordingly.
- Try to setup RISC32-E processors in Xilinx Vivado for simulation purposes.

2. WORK TO BE DONE

- Create a test program to test on LLVM and UART functionalities.
- Modify the ee_printf function to print out the result through UART
- Find out the signals that should be used to observe the outcome of the CoreMark programme.
- Compile and run simulation of CoreMark programme.

3. PROBLEMS ENCOUNTERED

- The script to compile the CoreMark source codes using LLVM have not created yet. So, compilation will be time consuming.

4. SELF EVALUATION OF THE PROGRESS

- The progress is still in the plan as in Figure 3.7.1, Gantt chart for week 1 until week 5.

TEOH

Supervisor's signature

Student's signature

FINAL YEAR PROJECT BI-WEEKLY REPORT

(Project II)

Trimester, Year: Jan 2022	**Study week no.:** 06
Student Name & ID: Teo Sei Hau 18ACB03719	
Supervisor: Mr. Teoh Shen Khang and Mr. Mok Kai Ming	
Project Title: RISC32-E Cryptography Performance Evaluation	

1. WORK DONE

- Create test program, try_uart.c to test on LLVM and UART functionalities.
- Compile and debug of try_uart.c
- Modify ee_printf function to print out the results through UART.

2. WORK TO BE DONE

- Compile and run simulation of CoreMark programme.
- Debug of CoreMark programme.

3. PROBLEMS ENCOUNTERED

- The script to compile the CoreMark source codes using LLVM have not created yet. So, compilation will be time consuming.

4. SELF EVALUATION OF THE PROGRESS

- The progress is still in the plan as in Figure 3.7.1, Gantt chart for week 1 until week 5 and Figure 3.7.2, Gantt chart for week 6 until week 10.

TEOH

Supervisor's signature

Student's signature

FINAL YEAR PROJECT BI-WEEKLY REPORT

(Project II)

Trimester, Year: Jan 2022	**Study week no.:** 08
Student Name & ID: Teo Sei Hau 18ACB03719	
Supervisor: Mr. Teoh Shen Khang and Mr. Mok Kai Ming	
Project Title: RISC32-E Cryptography Performance Evaluation	

1. WORK DONE

- Create bash script file, compile.sh. compile.sh stored all the commands to compile C source codes until generation of machine language codes.
- Compile CoreMark programme.
- Simulation of CoreMark programme on RISC32 processor.
- Found some problems of CoreMark programme.

2. WORK TO BE DONE

- Continue the debugging process of CoreMark programme.
- Correct CoreMark programme and loader program based on the problems encountered.
- Debug on CoreMark assembly codes generated by LLVM

3. PROBLEMS ENCOUNTERED

- RISC32 processor do not have coprocessor 1 (cp1) to handle floating point. So floating point tests and variable need to be removed from CoreMark programme.
- The loader program always set the starting address to 0x8000_0000, but CoreMark's main entry is at address 0x8000_010c. So, loader program should be changed to set entry point to 0x8000_010c.

4. SELF EVALUATION OF THE PROGRESS

- The progress is still in the plan as in Figure 3.7.2, Gantt chart for week 6 until week 10.

TEOH

Supervisor's signature

Student's signature

FINAL YEAR PROJECT BI-WEEKLY REPORT

(Project II)

Trimester, Year: Jan 2022	**Study week no.: 10**
Student Name & ID: Teo Sei Hau 18ACB03719	
Supervisor: Mr. Teoh Shen Khang and Mr. Mok Kai Ming	
Project Title: RISC32-E Cryptography Performance Evaluation	

1. WORK DONE

- Removed all the floating-point tests and variables from original CoreMark programme.
- Compile latest CoreMark programme again.
- Run simulation and found some bugs due to infinite looping behavior.

2. WORK TO BE DONE

- Solve the bugs that cause infinite looping behavior.
- Calculate the outcome values and compare between different versions of RISC32.
- Documentation of FYP2.

3. PROBLEMS ENCOUNTERED

- Cannot send out the results due to infinite loop in RISC32 processor.

4. SELF EVALUATION OF THE PROGRESS

- The progress is still in the plan as in Figure 3.7.2, Gantt chart for week 6 until week 10.

TEOH

Supervisor's signature

Student's signature

FINAL YEAR PROJECT BI-WEEKLY REPORT

(Project II)

Trimester, Year: Jan 2022	**Study week no.: 12**
Student Name & ID: Teo Sei Hau 18ACB03719	
Supervisor: Mr. Teoh Shen Khang and Mr. Mok Kai Ming	
Project Title: RISC32-E Cryptography Performance Evaluation	

1. WORK DONE

- Try to figure out the bugs cause infinite looping behavior.
- Started to document FYP2.

2. WORK TO BE DONE

- Solve the bugs that cause infinite looping behavior.
- Calculate the outcome values and compare between different versions of RISC32.

3. PROBLEMS ENCOUNTERED

- Cannot send out the results due to infinite loop in RISC32 processor.

4. SELF EVALUATION OF THE PROGRESS

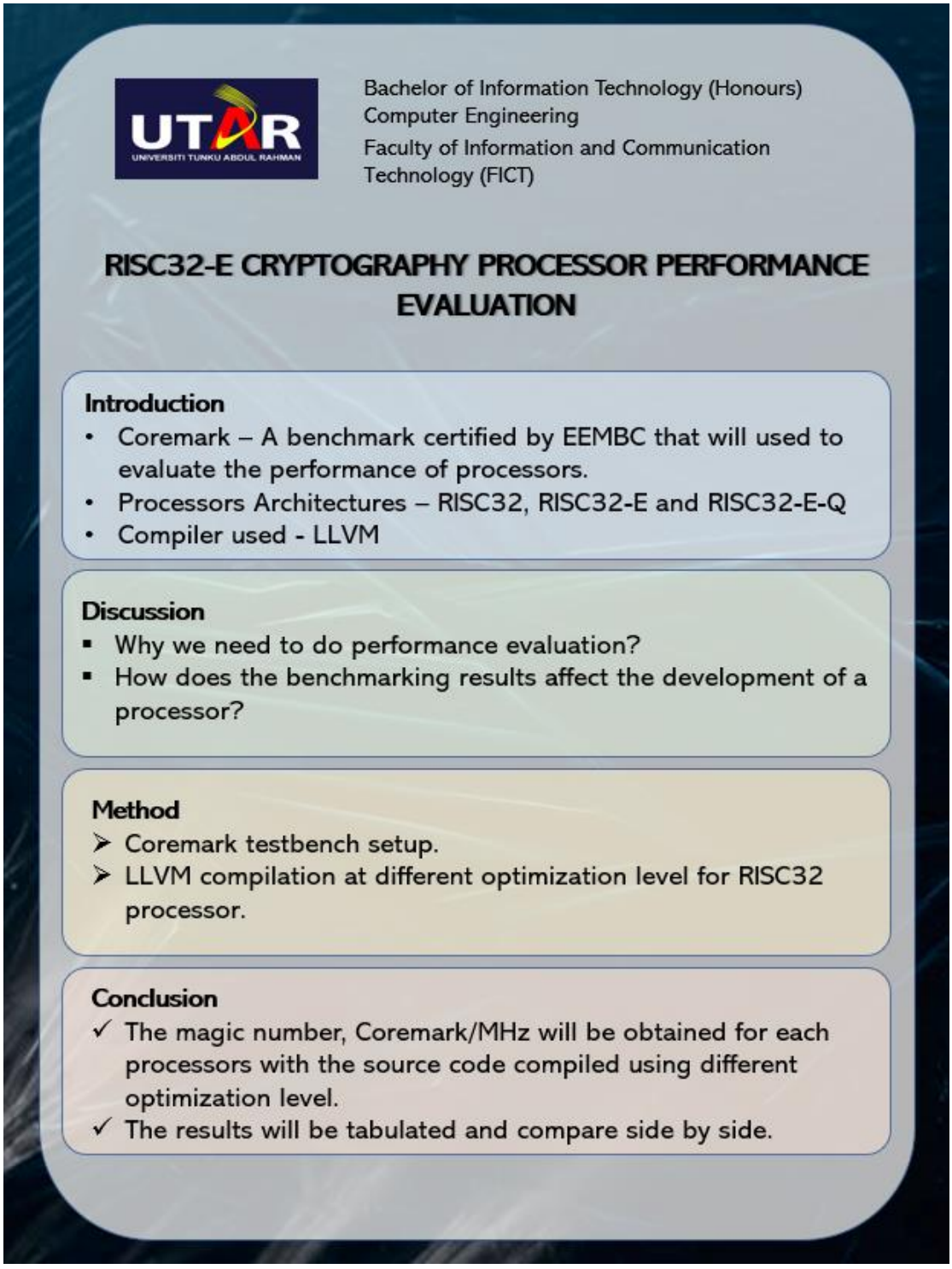
- The progress is behind the plan as the range of scope is bigger than expected.

TEOH

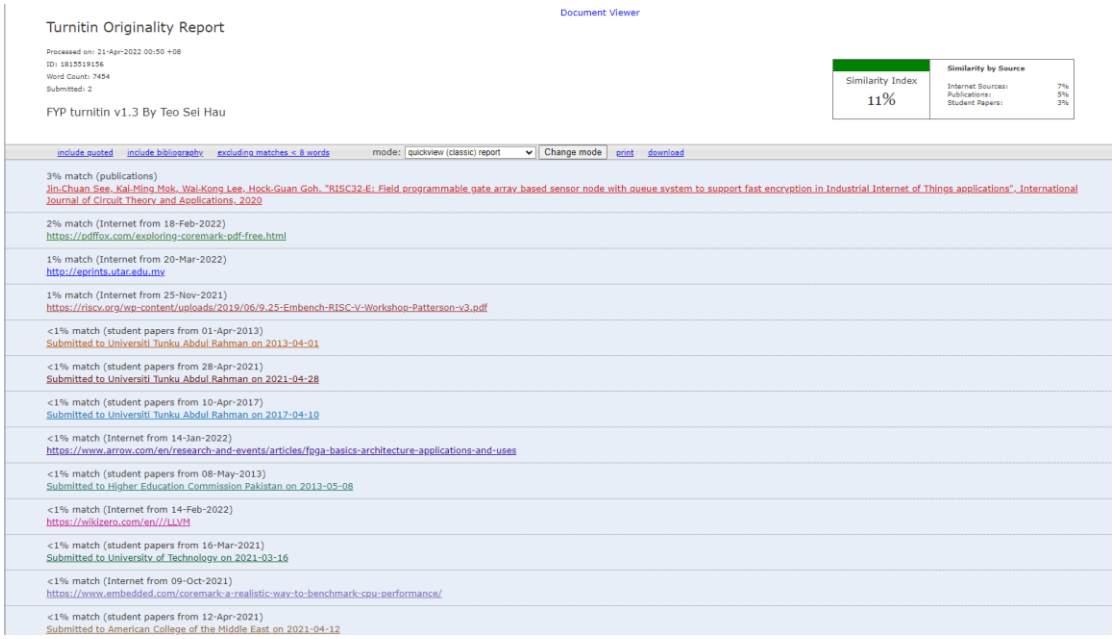
Supervisor's signature

Student's signature

Poster


Bachelor of Information Technology (Honours)
Computer Engineering
Faculty of Information and Communication Technology (FICT)
RISC32-E CRYPTOGRAPHY PROCESSOR PERFORMANCE EVALUATION
Introduction
• Coremark – A benchmark certified by EEMBC that will used to evaluate the performance of processors.
• Processors Architectures – RISC32, RISC32-E and RISC32-E-Q
• Compiler used - LLVM
Discussion
▪ Why we need to do performance evaluation?
▪ How does the benchmarking results affect the development of a processor?
Method
➢ Coremark testbench setup.
➢ LLVM compilation at different optimization level for RISC32 processor.
Conclusion
✓ The magic number, Coremark/MHz will be obtained for each processors with the source code compiled using different optimization level.
✓ The results will be tabulated and compare side by side.

Plagiarism Check Result



Document Viewer

Turnitin Originality Report

Processed on: 21-Apr-2022 00:50 +08
ID: 1815519156
Word Count: 7454
Submitted: 2

FYP turnitin v1.3 By Teo Sei Hau

Similarity Index 11%

Similarity by Source
Internet Sources: 7%
Publications: 5%
Student Papers: 3%

include quoted include bibliography excluding matches < 8 words mode: quickview (classic) report Change mode print download

3% match (publications)
Jin-Chuan See, Kai-Ming Mok, Wai-Kong Lee, Hock-Guan Goh, "RISC32-E: Field programmable gate array based sensor node with queue system to support fast encryption in Industrial Internet of Things applications", International Journal of Circuit Theory and Applications, 2020

2% match (Internet from 18-Feb-2022)
https://pdffox.com/exploring-coremark-pdf-free.html

1% match (Internet from 20-Mar-2022)
http://eprints.utar.edu.my

1% match (Internet from 25-Nov-2021)
https://riscv.org/wp-content/uploads/2019/06/9.25-Embench-RISC-V-Workshop-Patterson-v3.pdf

<1% match (student papers from 01-Apr-2013)
Submitted to Universiti Tunku Abdul Rahman on 2013-04-01

<1% match (student papers from 28-Apr-2021)
Submitted to Universiti Tunku Abdul Rahman on 2021-04-28

<1% match (student papers from 10-Apr-2017)
Submitted to Universiti Tunku Abdul Rahman on 2017-04-10

<1% match (Internet from 14-Jan-2022)
https://www.arrow.com/en/research-and-events/articles/fpga-basics-architecture-applications-and-uses

<1% match (student papers from 08-May-2013)
Submitted to Higher Education Commission Pakistan on 2013-05-08

<1% match (Internet from 14-Feb-2022)
https://wikizero.com/en///LLVM

<1% match (student papers from 16-Mar-2021)
Submitted to University of Technology on 2021-03-16

<1% match (Internet from 09-Oct-2021)
https://www.embedded.com/coremark-a-realistic-way-to-benchmark-cpu-performance/

<1% match (student papers from 12-Apr-2021)
Submitted to American College of the Middle East on 2021-04-12

Form Title: Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 27/8/2021	Page No.: 1of 1

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Teo Sei Hau
ID Number(s)	18ACB03719
Programme / Course	Bachelor of Information Technology (Honours) Computer Engineering
Title of Final Year Project	RISC32-E Cryptography Processor Performance Evaluation

Similarity	**Supervisor's Comments (Compulsory if parameters of originality exceed the limits approved by UTAR)**
Overall similarity index: 11 % **Similarity by source** Internet Sources: 7 % Publications: 5 % Student Papers: 3 %	
Number of individual sources listed of more than 3% similarity: 0	
Parameters of originality required, and limits approved by UTAR are as Follows: **(i) Overall similarity index is 20% and below, and** **(ii) Matching of individual sources listed must be less than 3% each, and** **(iii) Matching texts in continuous block must not exceed 8 words** ***Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.***	

Note: Supervisor/Candidate(s) is/are required to provide softcopy of full set of the originality report to Faculty/Institute

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.

TEOH

Signature of Supervisor	Signature of Co-Supervisor
Name: Mr. Teoh Shen Khang	Name:
Date: 21 April 2022	Date:

FYP 2 Checklist

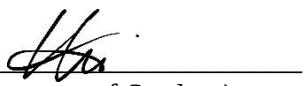
UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
(KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	18ACB03719
Student Name	Teo Sei Hau
Supervisor Name	Mr.Mok Kai Ming and Mr.Teoh Shen Khang
TICK (√)	**DOCUMENT ITEMS** Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
	Front Plastic Cover (for hardcopy)
√	Title Page
√	Signed Report Status Declaration Form
√	Signed FYP Thesis Submission Form
√	Signed form of the Declaration of Originality
√	Acknowledgement
√	Abstract
√	Table of Contents
√	List of Figures (if applicable)
√	List of Tables (if applicable)
√	List of Symbols (if applicable)
√	List of Abbreviations (if applicable)
√	Chapters / Content
√	Bibliography (or References)
√	All references in bibliography are cited in the thesis, especially in the chapter of literature review
√	Appendices (if applicable)
√	Weekly Log
√	Poster
√	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)
√	I agree 5 marks will be deducted due to incorrect format, declare wrongly the ticked of these items, and/or any dispute happening for these items in this report.

I, the author, have checked and confirmed all the items listed in the table are included in my report.



(Signature of Student)
Date:20/04/2022