Testing Platform for Memory IPs using PULPissimo

ADITHYA SAIKRISHNA PABBISETTY
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Testing Platform for Memory IPs using PULPissimo

Adithya Saikrishna Pabbisetty
ad5003pa-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Academic Supervisor: Steffen Malkowsky

Supervisor: Allan Andersen

Examiner: Pietro Andreani

October 16, 2022

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Memory IPs are important components in SoC designs. Hence, making sure that the memory IPs are functioning as expected is crucial for any organization. In order to do so, memory IPs must be tested. In addition, the testing capabilities can be enhanced by integrating a processor to the memory test chip. In this project, an open-source PULPissimo platform based on RISC-V ISA(Instruction set architecture) is used as this gives freedom to the system designer and the organization to configure the processor core as per the requirements. Further in this project, various checksum algorithms such as MD5, SHA-1 and SHA-256 are implemented in RTL which can be used to test the organization's ROM IPs. Subsequently, each of the algorithms are integrated to the PULPissimo to provide a platform for testing the ROM IPs. Finally, various comparisons are made using synthesized results. The three implemented algorithms are compared with respect to the number of gates used and latency to identify the suitable algorithm for the organization. Similarly, the cores in the PULPissimo are also compared to identify the preferable core to be used by the organization.

Popular Science Summary

21st century is a new chapter in human development enabled by extraordinary technology advances. The speed, breadth and depth of these advancements demand the organizations to create value. In the domain of semiconductor industries, advances in technology have had a massive impact with numerous electronic devices. According to Moore's law, the number of transistors on a chip doubles every two years. This has made drastic increase in cost per unit area while delivering a lot more functionality within that area. A large portion of the silicon area of many digital designs is dedicated to the storage of data. More than half of the transistors in today's designs are devoted to memories and this ratio is expected to increase further. Hence, memories are very important and crucial components to store data in the chips. To ensure their functionality and reliability, they need to be tested and verified thoroughly.

Electronic memories come in many different formats and sizes. They are most often classified as read-only (ROM) and read-write (RWM) memories based on memory functionality. The RWM structures have the advantage of offering both read and write functionality. Whereas ROMs are encoded and hardwired with data that cannot be modified. Among the various testing techniques, MBIST is the most popular one used by the organizations. However, MBIST algorithms include steps which involve writing and reading. Hence, they are mostly suited for read-write memories. Read-only memories, on the other hand, are tested for bit functionality using various algorithms like checksum.

In this project, three checksum algorithms namely MD5, SHA-1 and SHA-256 were implemented to test the bit functionality of ROMs. In addition, these algorithms were implemented in such a way that they could test any size of ROM. Further, these algorithms were compared with respect to various parameters such as area used and time taken to test the ROM. As a result of this comparison, MD5 was chosen as it uses less area and also less time to verify. To further improve the testing capabilities, a testing platform was considered for integrating the developed algorithm modules. In this project, PULPissimo platform which is an open-source platform was used to test the memories. Finally, the outcome of this project provides a testing platform for the organization to test their memories.

List of Abbreviations

APB: Advanced Peripheral Bus
ASIC: Application-Specific Integrated Circuit
AXI: Advanced eXtensible Interface
FLL: Frequency-Locked Loop
GPIO: General Purpose Input/Output
I2C: Inter-Integrated Circuit
I2S: Integrated Inter-IC Sound
IP: Intellectual Property
ISA: Instruction Set Architecture
JTAG: Joint Test Action Group
mBIST: Memory built-in self-test
MD: Message Digest
MSB: Most-significant bit
NIST: National Institute of Standards and Technology
NSA: National Security Agency
PPA: Power-Performance-Area
PULP: Parallel Ultra Low Power
PnR: Place and Route
QoR: Quality of Result
RISC: Reduced Instruction Set Computer
ROM: Read-only Memory
RTL: Register Transfer Level
SHA: Secure Hash Algorithm
SPI: Serial Peripheral Interface
SRAM: Static Random Access Memory
SoC: System-on-Chip
uDMA: Ultra Direct Memory Access

Table of Contents

1 Introduction **1**
1.1 Goal of the project . . . 2
1.2 Previous work . . . 2
1.3 Thesis Outline . . . 3

2 Theoretical Background **5**
2.1 Checksum Algorithm . . . 5
2.2 PULPissimo . . . 11

3 Design of Checksum algorithms **17**
3.1 Wrapper module . . . 17
3.2 Sub-module . . . 21

4 Customization of PULPissimo **27**
4.1 Integration of algorithm to PULPissimo . . . 27
4.2 Physical Implementation . . . 27

5 Results **29**
5.1 Results of Simulations . . . 29
5.2 Results of Synthesis . . . 31
5.3 Results of PnR . . . 32

6 Conclusion **35**
6.1 Future work . . . 35

References **37**

A Values stored in the ROM IP **39**
A.1 Values stored in the ROM IP . . . 39

List of Figures

2.1 MD5 algorithm overview 6
2.2 MD5 algorithm - Processing of a data block 6
2.3 MD5 algorithm - Update of buffers 7
2.4 SHA-1 algorithm - Update of buffers 8
2.5 SHA-1 algorithm - Processing of a data block 9
2.6 Block Diagram of PULPissimo 11
2.7 Memory-map of PULPissimo 12
2.8 APB State Diagram 14

3.1 Wrapper module overview 17
3.2 Wrapper module parent state machine 19
3.3 Wrapper module child state machine 20
3.4 MD5 sub-module overview 21
3.5 MD5 sub-module state machine 22
3.6 SHA-1 sub-module overview 23
3.7 SHA-1 sub-module state machine 24
3.8 SHA-256 sub-module overview 25
3.9 SHA-256 sub-module state machine 26

4.1 Block Diagram of customized PULPissimo 28

5.1 MD5 sub-module waveform 29
5.2 SHA-1 sub-module waveform 30
5.3 SHA-256 sub-module waveform 30
5.4 Layout of PULPissimo SoC 33

List of Tables

1.1 Checksum algorithms comparison - 1 3
1.2 Checksum algorithms comparison - 2 3

2.1 Description of all the main pins used to implement the industry standard JTAG 15

5.1 Comparison of checksum algorithms synthesis results 31
5.2 Comparison of PULPissimo synthesis results 32

A.1 Values stored in the ROM IP 39

Chapter 1

Introduction

Development of SoC(System-on-Chip) is a complex and time-consuming process for any organization. Having the resources and manpower to design and develop every component in a SoC is not preferable for many organizations to be sustainable. Instead, they rely on the third party components, also referred to as IPs (Intellectual Property). The licensing cost of the IPs can be avoided by using open-source IPs to build the system and to exploit the flexibility of tailoring the existing design to meet the organization's requirements.

In the space of well-structured and matured open-source processor IP, PULPissimo designed by ETH Zurich is the most popular one. PULPissimo[2] is a microcontroller architecture of the more recent PULP (Parallel Ultra Low Power) chips. This architecture uses open-source RISC-V ISA and is available completely without any license cost on Github[6]. Making use of this open-source processor IP against other commercial licensed processor IPs such would be advantageous in terms of license cost and flexibility in integrating new modules to the test chip such as accelerators.

Testing and verifying the memory IPs is one of the rigorous processes carried out by all the organizations. By having a processor core inside a test chip, the testing of the memory IPs becomes flexible as it can be evaluated using various algorithms besides mBIST[3](Memory built-in self-test) which is currently used in the semiconductor industry. Apart from the advantage of flexibility of customization, one can also test the memories with real-time applications by integrating a processor.

With the freedom to integrate new RTL modules to the PULPissimo, it will provide an opportunity to bring in a way to test the memory IPs. To test ROM IPs in particular, various checksum[5] algorithms can be considered. A checksum algorithm performs numerous mathematical operations to create an unique checksum value for a given data. Typical checksum algorithms are MD5[13], SHA-1[14] and SHA-256[15]. These algorithms can be used to measure the quality of the ROM IPs.

1.1 Goal of the project

Memory IPs are important components in the current SoC designs. Making sure that the memory IPs are functioning as expected is crucial for the organization. In order to do so, they must be tested. At present, in Xenergic, the memory IPs are tested using mBIST to ensure the functionality of the memories designed by them. With the possibility of integrating a processor to the memory test chip, the testing capabilities would be enhanced. In this project, this will be done using open-source PULPissimo platform based on RISC-V ISA(Instruction Set Architecture). This gives freedom to the system designer and the organization to configure the processor core as per the organization requirements. The PULPissimo has two different configurable cores namely Ibex(zero-riscy)[12] and RISCY. RISCY[11] is an in-order, single-issue core with 4 pipeline stages whereas Ibex is an in-order, single-issue core with 2 pipeline stages. It would be an interesting process to evaluate these two cores as there could be a difference in the PPA(power-performance-area) numbers and critical path due to the difference in the way the cores are configured. This would help the organization to draw conclusions on which core would be suitable for testing their memory IPs.

Further in this project, various checksum algorithms will be implemented in RTL to test the ROM IPs. Checksum algorithms are used to verify the correctness of the data bit stored in a memory location. To verify that the ROM IPs are functioning as expected without a change in bit location value, testing the ROM IPs with checksum algorithm plays a vital role. In this regard, RTL modules will be designed for different checksum algorithms such as MD5, SHA-1 and SHA-256 for verifying the ROM IPs. The designed RTL modules of the algorithms will further be compared by using various parameters such as area, latency and critical path. These modules will then be integrated to the PULPissimo platform to test the ROM IPs.

Physical implementation of the PULPissimo with the integrated modules (algorithms) will be performed and verified through post-synthesis simulation and post-PnR simulation. The PULPissimo SoC will be programmed using the JTAG[4] ports available on the PULPissimo.

1.2 Previous work

Past studies have been done in the comparison of various checksum algorithms with respect to area, power and latency. In one of the study[9], M. Feldhofer, and J. Wolkerstorfer made a comparison of Low-Power Hardware Implementations of MD5, SHA-1 and SHA-256 algorithms. The summary of the results from this study has been given in Table 1.1.

Algorithm	**Tech (μm/V)**	**Area (gates)**	**Power(μW) @100kHz**	**Latency (cycles)**
MD5	0.35/1.5	8001	4.74	712
SHA-1	0.35/1.5	8120	5.9	1274
SHA-256	0.35/1.5	10868	8.79	1128

Table 1.1: Checksum algorithms comparison - 1

In another study[10], A. Satoh and T. Inoue made an ASIC-Hardware-Focused comparison of checksum algorithms. Table 1.2 shows the comparison of different checksum algorithms with respect to area and latency. These results provide the required numbers to validate the work that will be carried out in this master thesis project.

Algorithm	**Tech(μm)**	**Area(gates)**	**Latency(cycles)**
MD5	0.13	10332	68
SHA-1	0.13	7971	85
SHA-256	0.13	11484	72

Table 1.2: Checksum algorithms comparison - 2

1.3 Thesis Outline

The thesis is organized in the following chapters.

- **Chapter 1: Introduction** - This chapter describes the main objective of the thesis and why it is important to test the memory IPs.
- **Chapter 2: Theoretical Background** - In this chapter, all the background information relevant to the present study is explained and the algorithms used for testing ROM IPs are also discussed.
- **Chapter 3: Design of Checksum algorithms** - Implementation of the algorithms is explained in this chapter.
- **Chapter 4: Customization of PULPissimo** - Here, the process of integrating the algorithms to the PULPissimo is explained. Also, the changes required for the PULPissimo to be synthesizable and the process of the physical implementation are discussed in this chapter
- **Chapter 5: Results** - In this chapter, the simulation results of the algorithms are presented, and various results are compared.
- **Chapter 6: Conclusion** - In this chapter, conclusions regarding this master thesis and future work of the same are discussed.

Chapter 2

Theoretical Background

2.1 Checksum Algorithm

Checksum algorithm is an algorithm that performs numerous mathematical operations to create an unique checksum value for a given data. Checksum[5] is a small-sized block of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its storage. By themselves, checksums are often used to verify data integrity but are not relied upon to verify data authenticity. However, checksum does not indicate exactly where in the data there was a problem, nor does it provide any error correction. There are various checksum algorithms which include MD5, SHA-1 and SHA-256 algorithms. Each algorithm is explained in detail in the following sections 2.1.1, 2.1.2 and 2.1.3.

2.1.1 MD5

MD5[13] is a checksum algorithm that takes an input data of any length and changes it into a fixed-length message of 128-bits(16 bytes). MD5 algorithm stands for the message-digest algorithm. It is widely used for file authentication and to verify data integrity of any system that stores data.

In this algorithm, all values are considered to be in little endian. Primarily, the input data is broken up into blocks of 512-bits. Then, the data is padded so that its length is divisible by 512. The padding is done in two steps. First, a single bit, "1", is appended to the end of the data(LSB) which is followed by as many zeros as are required to bring the length of the data up to 64 bits fewer than a multiple of 512. Second padding step includes the padding of the size of the original data. The length of this size is 64 bits long. After the padding stage, the input to the next stage will be exact multiple of 512-bits. Each of these 512-bit blocks will go through a process and finally give an output of 128 bit checksum as shown in Figure 2.1.

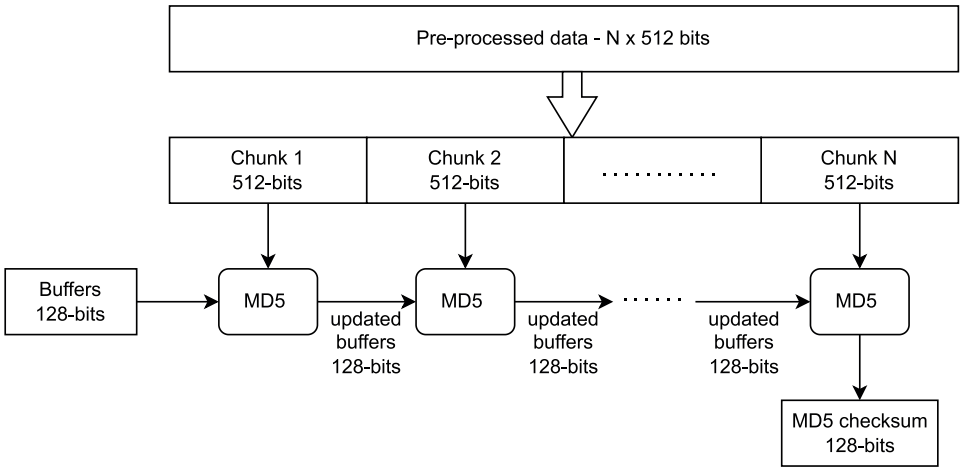

Figure 2.1: MD5 algorithm overview

The main MD5 algorithm operates on a 128-bit state which is divided into four 32-bit buffers, denoted by A, B, C, and D. These are initialized to specific fixed constants. The main algorithm then uses each 512-bit data block in turn to modify the 128-bit state. The processing of a data block consists of four rounds where each round is composed of 16 similar operations based on a non-linear function, a modular addition, and a left rotation as illustrated in Figure 2.2.

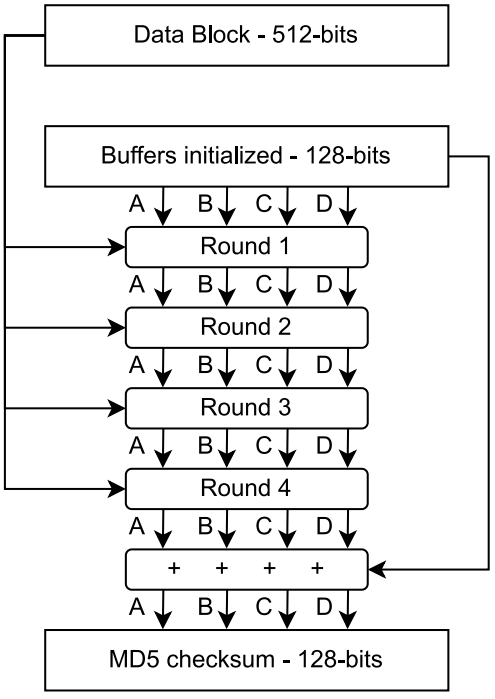

Figure 2.2: MD5 algorithm - Processing of a data block

In each round, a different non-linear function is applied namely F, G, H and I. For calculating these functions, OR, AND, XOR, and NOT operations are performed on the three buffers B, C and D. These functions are given below as equations[14].

$$F = (B \text{ and } C) \text{ or } ((\text{not } B) \text{ and } D) \tag{2.1}$$
$$G = (D \text{ and } B) \text{ or } ((\text{not } D) \text{ and } C) \tag{2.2}$$
$$H = B \text{ xor } C \text{ xor } D \tag{2.3}$$
$$I = C \text{ xor } (B \text{ or } (\text{not } D)) \tag{2.4}$$

The output of these functions is then added with sum of buffer A, a 32-bit word of the 512-bit data W[i] and a predefined 32-bit constant K[i]. Lastly, a left shift operation by s-bits, which is predefined, is performed. This result is then fed to the buffer B. The buffers A, C and D are fed with the values of buffers D, B and C respectively as shown in Figure 2.3.

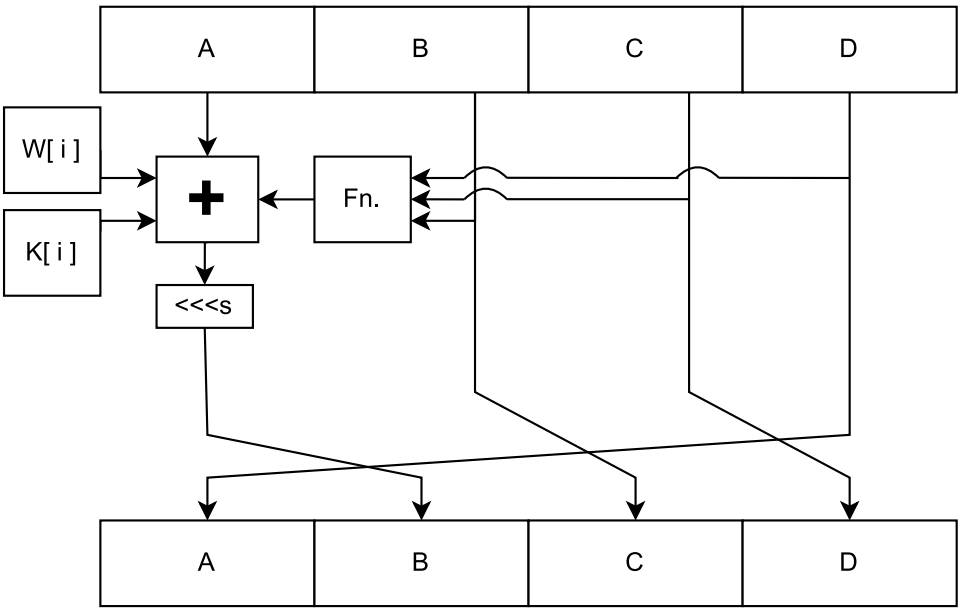

Figure 2.3: MD5 algorithm - Update of buffers

After all the four rounds, i.e. 64 operations have been performed, the resulting buffers A, B, C and D are added with the initial values of the same as shown in Figure 2.2. This process iterates until the pre-processed data is exhausted. The resultant buffers appended together is the MD5 checksum of the given data.

2.1.2 SHA-1

SHA-1[14] or Secure Hash Algorithm 1 is an algorithm which takes an input of any size and produces a 160-bit(20 bytes) checksum value. It is a U.S. Federal

Information Processing Standard and was designed by the United States National Security Agency. This algorithm is used for cryptography and data integrity of a system.

All values in this algorithm are in big endian. Similar to the MD5 algorithm explained in section 2.1.1, the data is initially broken into blocks of 512-bits. Then, this is pre-processed by appending a bit "1" to the end of data(LSB) and followed by zeroes such that the resulting data length in bits is 64 bits short of a multiple of 512. The size of the original data is then padded making the total length of the pre-processed data a multiple of 512 bits.

The next step is to process the data in successive 512-bit blocks. For each block, break the block into sixteen 32-bit big endian words, W[0] to W[15]. Following this, the sixteen 32-bit words are extended into eighty 32-bit words using the below equation[14], where 'i' ranges from 16 to 79.

$$W[i] = (W[i-3] \text{ xor } W[i-8] \text{ xor } W[i-14] \text{xor } W[i-16]) \text{ leftrotate } 1 \quad (2.5)$$

Next, five 32-bits buffers are initialized and stored as A, B, C, D and E. These buffers are updated until there's no more 512-bits blocks of data left. For each 512-bit block, the buffers undergo 80 iterations. Figure 2.4 illustrates the way the buffers are being updated.

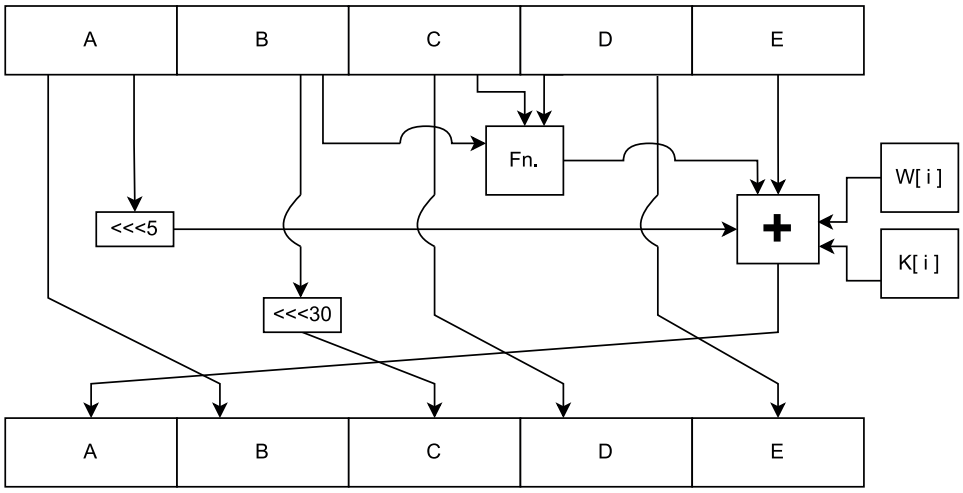

Figure 2.4: SHA-1 algorithm - Update of buffers

In each iteration, the buffers E, D and B are updated with the values of buffer D, C and A. Whereas, the buffer C is updated with value of buffer B left rotated by 30-bits. However, the buffer A is updated using the below equation[14].

$$A = (A \text{ leftrotate } 5) \ + \ F \ + \ E \ + \ k \ + \ W[i] \quad (2.6)$$

where, k is a 32-bits constant and F is the output of a function which vary after every 20 iterations. The definitions of F and k in each set of iterations are given

below

1st 20 iterations - $F = (B \text{ and } C) \text{ or } ((\text{not } B), \text{ and } D)$ and $k =$ 0x5A827999

2nd 20 iterations - $F = B$ xor C xor D, and $k =$ 0x6ED9EBA1

3rd 20 iterations - $F = (B \text{ and } C) \text{ or } (B \text{ and } D) \text{ or } (C \text{ and } D)$, and $k =$ 0x8F1BBCDC

4th 20 iterations - $F = B$ xor C xor D, and $k =$ 0xCA62C1D6

After performing 80 iterations, the resulting buffers are added with the buffers stored in the initial step as shown in Figure 2.5. The sum of these buffers are then read together as a 160-bit checksum value.

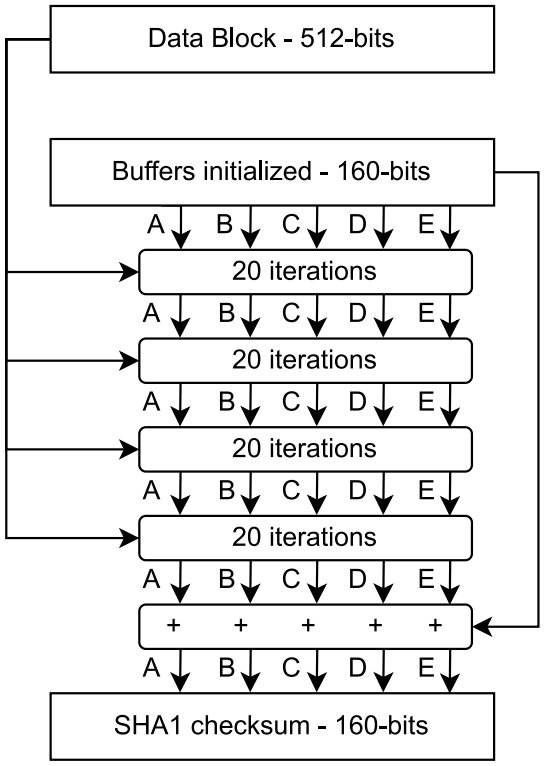

Figure 2.5: SHA-1 algorithm - Processing of a data block

2.1.3 SHA-256

SHA-256[15] is a part of the SHA-2 family of algorithms, where SHA stands for Secure Hash Algorithm. Published in 2001, it was a joint effort between the United States National Security Agency(NSA) and National Institute of Standards and Technology(NIST) to introduce a successor to the SHA-1 family. The significance of the 256 in the name stands for the final checksum value, i.e. irrespective of the size of input data, the checksum value will always be 256-bits. SHA-256 is used for digital signature verification, password hashing and data integrity.

The pre-processing of the data which includes padding of bit "1", zeroes and the size of data is equivalent to the pre-processing of data in SHA-1 explained in section 2.1.2. At the end of the pre-processing stage, the data length is a multiple of 512-bits. The values used in SHA-256 are also in big endian.

Similar to SHA-1, SHA-256 also processes the data in successive 512-bit blocks which are further broken into sixteen 32-bit big endian words, W[0] to W[15]. However, unlike SHA-1, SHA-256 extends the sixteen 32-bit words into sixty four 32-bit words using the below equations[15], where 'i' ranges from 16 to 63.

$$S0 = (W[i-15] \text{ rightrotate } 7) \text{ xor } (W[i-15] \text{ rightrotate } 18) \text{ xor } (W[i-15] \text{ rightshift } 3) \quad (2.7)$$

$$S1 = (W[i-2] \text{ rightrotate } 17) \text{ xor } (W[i-2] \text{ rightrotate } 19) \text{ xor } (W[i-2] \text{ rightshift } 10) \quad (2.8)$$

$$W[i] = W[i-16] \ + \ S0 \ + \ W[i-7] \ + \ S1 \quad (2.9)$$

In SHA-256, eight 32-bits buffers A, B, C, D, E, F, G and H are used. These are then updated until there's no more 512-bits blocks of data left. For each 512-bit block, the buffers undergo 64 iterations. In each iteration, the buffers are updated using the below equations[15].

$$S1 = (E \text{ rightrotate } 6) \text{ xor } (E \text{ rightrotate } 11) \text{ xor } (E \text{ rightrotate } 25) \quad (2.10)$$
$$ch = (E \text{ and } F) \text{ xor } ((\text{not } E) \text{ and } G) \quad (2.11)$$
$$temp1 = H \ + \ S1 \ + \ ch \ + \ k[i] \ + \ W[i] \quad (2.12)$$
$$S0 = (A \text{ rightrotate } 2) \text{ xor } (A \text{ rightrotate } 13) \text{ xor } (A \text{ rightrotate } 22) \quad (2.13)$$
$$maj = (A \text{ and } B) \text{ xor } (A \text{ and } C) \text{ xor } (B \text{ and } C) \quad (2.14)$$
$$temp2 = S0 \ + \ maj \quad (2.15)$$
$$H = G \quad (2.16)$$
$$G = F \quad (2.17)$$
$$F = E \quad (2.18)$$
$$E = D + temp1 \quad (2.19)$$
$$D = C \quad (2.20)$$
$$C = B \quad (2.21)$$
$$B = A \quad (2.22)$$
$$A = temp1 + temp2 \quad (2.23)$$
$$(2.24)$$

At the end of 64 iterations, the updated buffers are added to the initial values of buffers and read together as 256-bits checksum value.

2.2 PULPissimo

PULPissimo[2] is a 32 bit RI5CY single-core System-on-a-Chip. PULPissimo is the second version of the PULPino system and it can be extended with the multi-core cluster of the PULP project.

Unlike the simpler PULPino system, PULPissimo uses a more complex memory subsystem, an autonomous I/O subsystem which uses the uDMA, new peripherals (eg. the camera interface) and a new SDK.

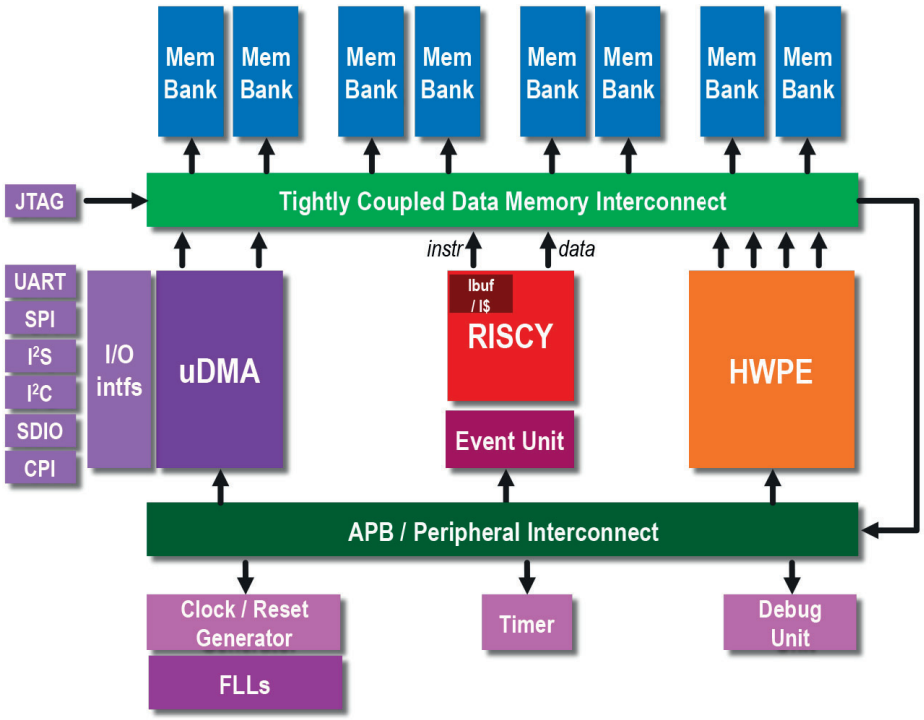

Figure 2.6: Block Diagram of PULPissimo

Figure 2.6 shows a simplified block diagram of the SoC. Similar to PULPino, PULPissimo can be configured at design stage to use either the RISCY or ibex. The peripherals are connected to the uDMA which transfers the data to the memory subsystem efficiently. JTAG and AXI plug have also access to the SoC. AXI plug can be used to extend the microcontroller with a multi-core cluster or an accelerator. GPIO, timers, event unit and event generator, debug and FLLs are not connected to the uDMA instead to the APB bus. In PULPissimo, the advanced debug unit is used to access to system and core registers, memories and memory-mapped IO via JTAG. A logarithmic interconnect allows to link the core and the uDMA to the memory banks simultaneously.

2.2.1 Memory Map

Figure 2.9 shows the default memory-map of PULPissimo. PULPissimo out of the box has 512kB of RAM in L2 memory bank and 8kB boot ROM.

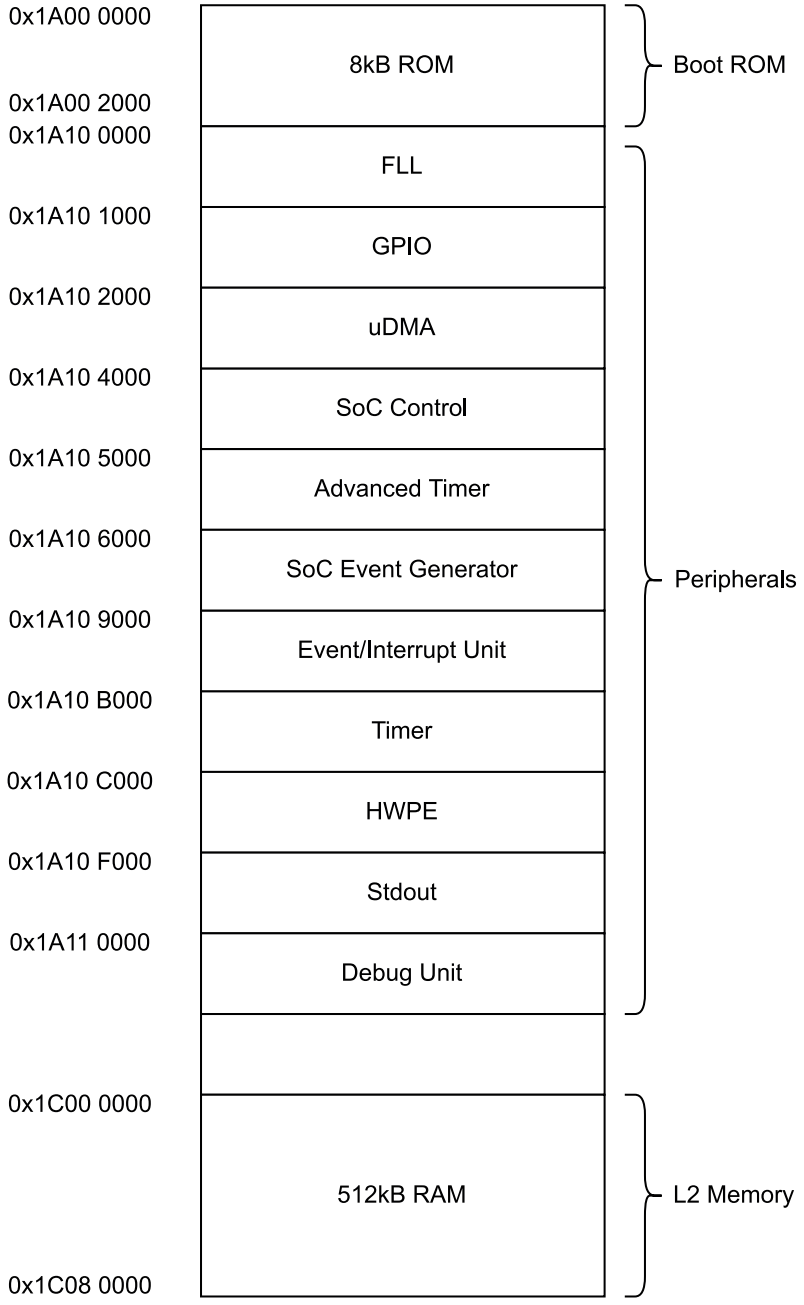

Figure 2.7: Memory-map of PULPissimo

2.2.2 CPU Cores

PULPissimo supports both the RISCY and the Ibex core. These two cores have the same external interfaces and are thus plug-compatible. For debugging purposes, all core registers have been memory mapped which allows them to be accessed over the logarithmic-interconnect subsystem. The debug unit inside the core handles the request over this bus and reads/sets the core registers and/or halts the core.

RISCY is an in-order, single-issue core with 4 pipeline stages which are Instruction Fetch(IF), Instruction Decode(ID), Execution and Write to memory stages. Whereas, Ibex, formerly Zero-riscy, is an in-order, single-issue core with 2 pipeline stages which are Instruction Fetch(IF) and Execution stages.

2.2.3 FLL

A frequency-lock, or frequency-locked loop (FLL), is an electronic control system that generates a signal that is locked to the frequency of an input or "reference" signal. This circuit compares the frequency of a controlled oscillator to the reference and automatically raises or lowers the frequency of the oscillator until its frequency is matched to that of the reference.

PULPissimo contains 3 FLLs. These FLLs are meant for generating the clock for

(i) the peripheral domain

(ii) the core domain which includes core, memories, event unit, etc.

(iii) the cluster

The latter is not used. All the three FLLs can be bypassed and an external clock can be used.

2.2.4 APB Protocol

Figure 2.8 shows the state diagram of the APB protocol[16]. There are three states on which the APB protocol works: IDLE, SETUP and ACCESS. These 3 states are explained below.

IDLE : This is the default state of the APB.

SETUP: When a data transfer has to be made the bus moves into SETUP state after assertion of the appropriate select signal PSLEx. The bus stays in this state for one clock cycle and moves to ACCESS state on the next rising edge of the clock.

ACCESS : **PENABLE** is an enable signal that is asserted in the ACCESS state. The address, write,select, and write data signals must remain stable during the transition from the SETUP to ACCESS state.

Exit from this state is controlled by the **PREADY** state from the slave:

- If the **PREADY** signal is held LOW by the slave, the peripheral bus remains in ACCESS state.

- If the **PREADY** signal is driven HIGH by the slave, the ACCESS state is exited and the bus returns to IDLE state if no more transfers are required. The bus moves directly to SETUP state if another transfer of data has to be made.

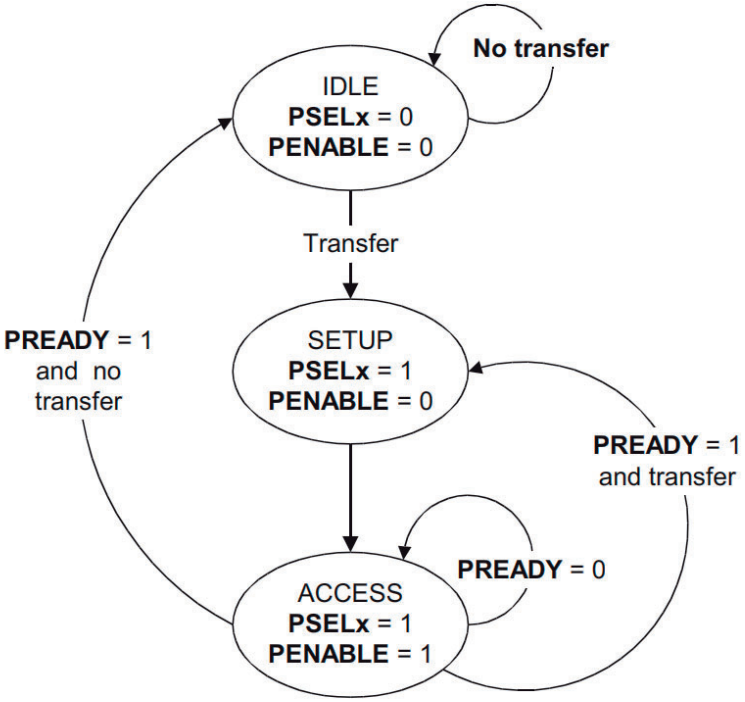

Figure 2.8: APB State Diagram

2.2.5 JTAG

Joint Test Action Group (JTAG)[4] which is also known as boundary scan is a commonly used industry-standard on-chip hardware interface. JTAG provides a solution to serially communicate between the chip and external devices. It is commonly used to program and debug on-chip components through a small number of test pins. Basic JTAG implementation requires at least four different ports with one optional pin. These pins and their functionality are explained in Table 2.1.

Pin	Description
TCK (Test Clock Input)	TCK is an input pin used by an external device to synchronize the serial data stream at input and output pins with the JTAG test access port state-machine
TDI (Test Data Input)	TDI pin is used as an input for the external devices to transfer a serial stream of data. The test data is loaded at the rising edge of TCK.
TMS (Test Mode Select)	TMS input is used to control the movement in JTAG test access port state-machine. TMS signal is loaded at the rising edge of TCK.
TDO (Test Data Output)	TDO is an output pin to a serial stream of data to external test devices. Output data is returned at the falling edge of TCK.
TRST (Test Reset)	TRST is an optional input pin used to asynchronously reset the JTAG regardless of the state of other signals.

Table 2.1: Description of all the main pins used to implement the industry standard JTAG

Chapter 3
Design of Checksum algorithms

Each of the checksum algorithms, viz. MD5, SHA-1 and SHA-256, are implemented in RTL. A wrapper module reads data from the ROM IP and pre-processes the data into 512-bits block of data. This block of data is fed to a sub-module which implements the algorithm. The wrapper module is developed for all the three algorithms. It is to be noted that the pre-processing of the data for all the three algorithms is same. However, the sub-module is unique for each of the three algorithms. The developed modules are made to be flexible to work with ROM IPs of any address width and data width.

3.1 Wrapper module

The wrapper module reads data from the ROM and pre-processes the data into 512-bits block which includes the padding of bit "1", zeroes and size to the end of data which was explained in section 2.1. This module has five input ports which include *CLK*, *RESET*, *start_addr*, *stop_addr* and *config_mem*. The input ports *start_addr* and *stop_addr* will receive the range of addresses for which the algorithm needs to calculate the checksum. In case of error in the data of the ROM, this particular implementation with *start_addr* and *stop_addr* as input ports helps in identifying the location of the error by adopting brute force approach. Among the input ports, the *config_mem* port inputs the configuration of the ROM IP. In case of the output ports, viz. *data_out* and *data_valid*, the calculated checksum is fed to the output port *data_out*.

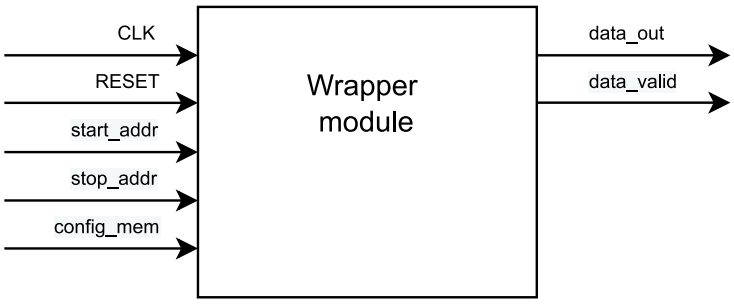

Figure 3.1: Wrapper module overview

A few parameters are also defined namely *address_ width* and *data_ width*. Figure 3.1 illustrates the overview of the module. The wrapper module works on a history transition hierarchical type state machine in which when it enters the parent state, it resumes from the last known child state. In the implemented state machine, the parent state machine works with four states, viz. *wait_ state*, *process_ data*, *data_ read* and *finish_ state*, where the *process_ data* is refined further into another child state machine. These state machines are depicted in Figure 3.2 and Figure 3.3.

In this module, the data is an array of words of length *data_ width*. The length of the array is calculated as 512/*data_ width*. The initial state of the parent state machine(refer Figure 3.2) is the *wait_ state* where it waits for the input trigger from the *config_ mem[0]* port. Upon receiving the trigger from *config_ mem[0]* port, it moves to the *process_ data*. In the *process_ data* state, the data is read from the ROM and padded to form a 512-bit block. This process further works as another state machine which will be explained later in this section. When a 512-bit block is formed it moves to *data_ read* state. In this state, the sub-module is triggered. Then, it will be checked for the last block. If the last block is processed, it moves to the *finish_ state* else, it moves back to the *process_ data* state. In the finish state, the final checksum is received from the sub-module and fed as an output.

As mentioned earlier, the *process_ data* state is further refined into a child state machine(refer Figure 3.3) with four states, viz. *read_ data_ from_ mem*, *read_ last_ data_ from_ mem*, *pad_ bit_ one* and *pad_ zeroes_ size*. As it is a history transition type state machine, in the parent state machine, the *process_ data* state moves to the *data_ read* state upon forming a 512-bit block irrespective of the current child state. However, it resumes from the last known child state upon entering the *process_ data* state.

In the child state machine, the initial state is *read_ data_ from_ mem*. In the *read_ data_ from_ mem* state, the data is read from the ROM as words of length of the *data_ width* from the *start_ addr* to the *stop_ addr*. While the address reaches the *stop_ addr*, the *read_ data_ from_ mem* state moves to *read_ last_ data_ from _ mem* state where the last data from the ROM is read. Subsequently, it moves to the *pad_ bit_ one* state. In the *pad_ bit_ one* state, a word of length *data_ width* with MSB as "1" and others as "0" is padded. Further, it moves to the *pad_ zeroes _ size* state where it pads words of zeroes until it reaches 64-bits less than 512-bits. At last, the size of length of 64-bits is padded to the end of the data and triggers the *last_ block_ received* signal. Then, it moves out of the parent state.

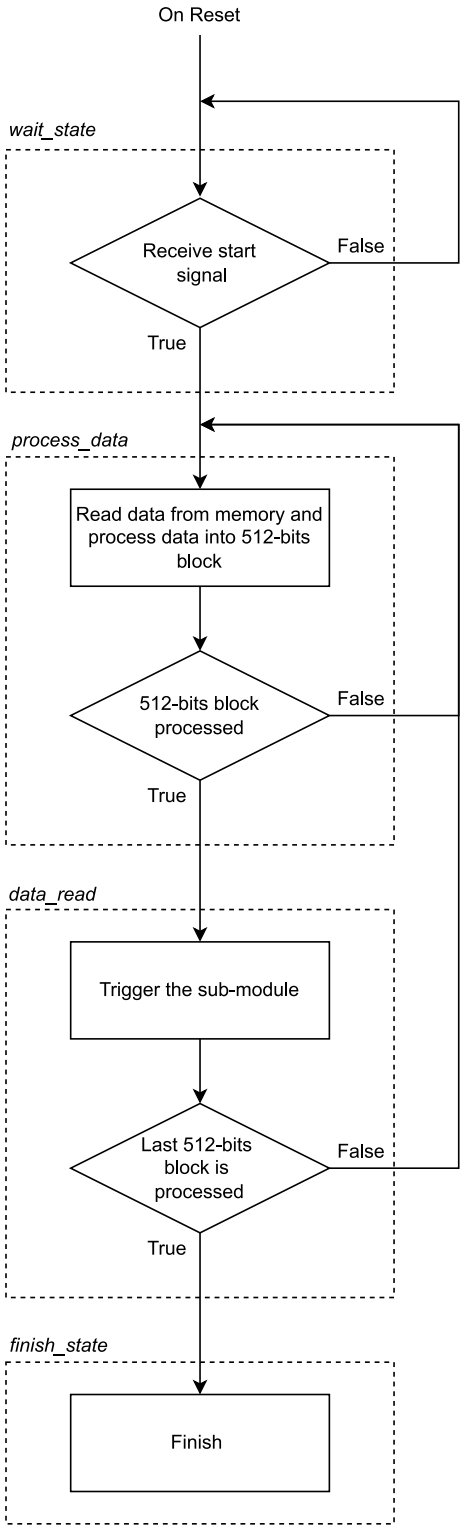

Figure 3.2: Wrapper module parent state machine

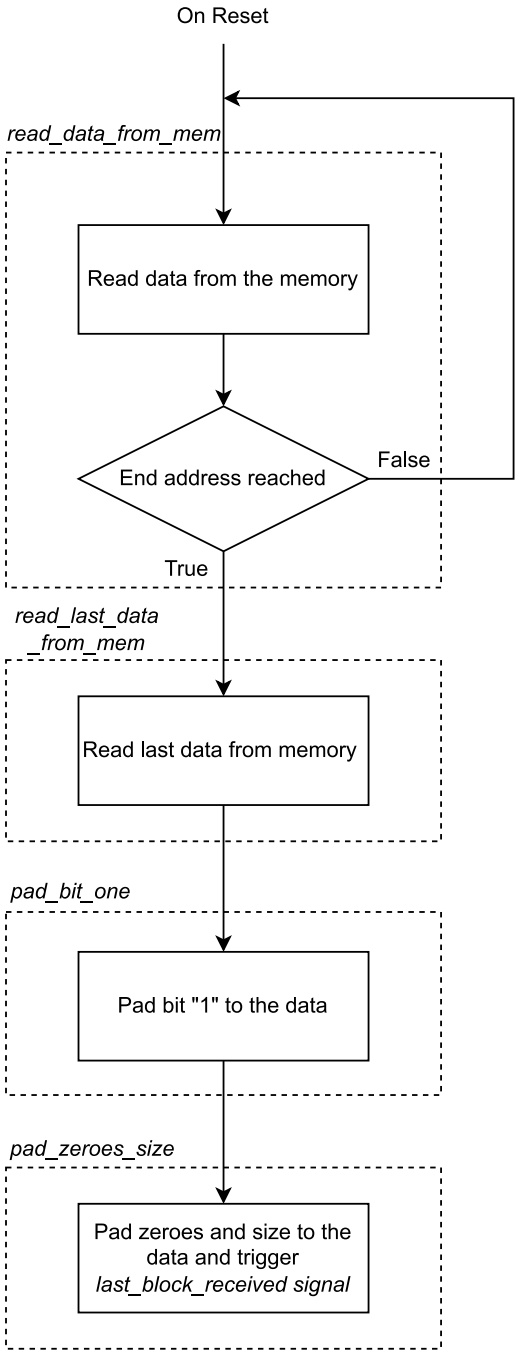

Figure 3.3: Wrapper module child state machine

3.2 Sub-module

3.2.1 MD5

The MD5 sub-module takes an input of 512-bit block and outputs a 128-bit checksum value. The input ports of this sub-module are *CLK*, *RESET*, *start_ in* and *data_ in*. The output ports of the sub-module include *data_ out* and *data_ valid*. The input port *data_ in* receives the 512-bit block of data and outputs a checksum of 128-bits through the output port *data_ out*. Figure 3.4 illustrates the overview of the sub-module.

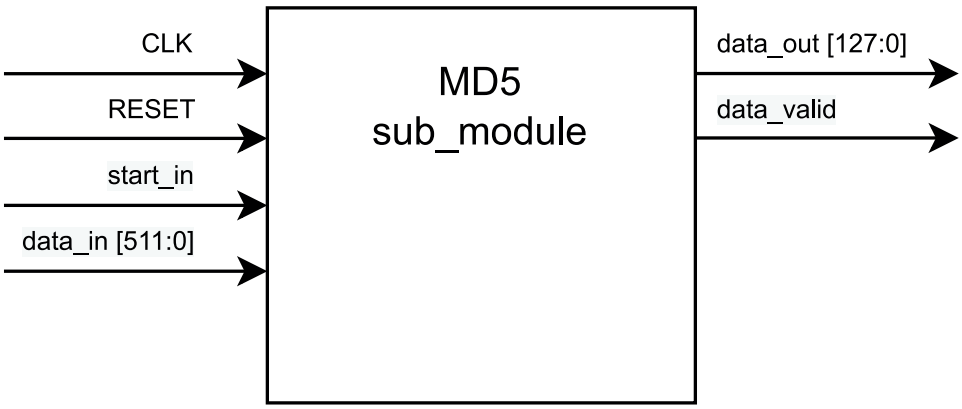

Figure 3.4: MD5 sub-module overview

At the beginning, all the initial buffers, viz. A, B, C and D, and other constants are initialized. This sub-module works on a state machine of three states which are *store_ inp*, *loop_ thro_ rounds* and *checksum_ calc*. Figure 3.5 illustrates the state machine of MD5 sub-module. The initial state of this sub-module is *store_ inp* state where the input block of 512-bits is stored as an array of 32-bit data. When the *start_ in* port receives a bit value "1", it moves to the next state, *loop_ thro_ rounds*. In the *loop_ thro_ rounds* state, 64 rounds of specific operations(refer section 2.1.1) are performed by updating the four buffers. After all the 64 rounds are performed, the state shifts to the *checksum_ calc* state. In this state, the updated buffers are added with the initial buffers and fed to the *data_ out* port and triggers the *finish_ out* port. At last, it returns to the *store_ inp* state.

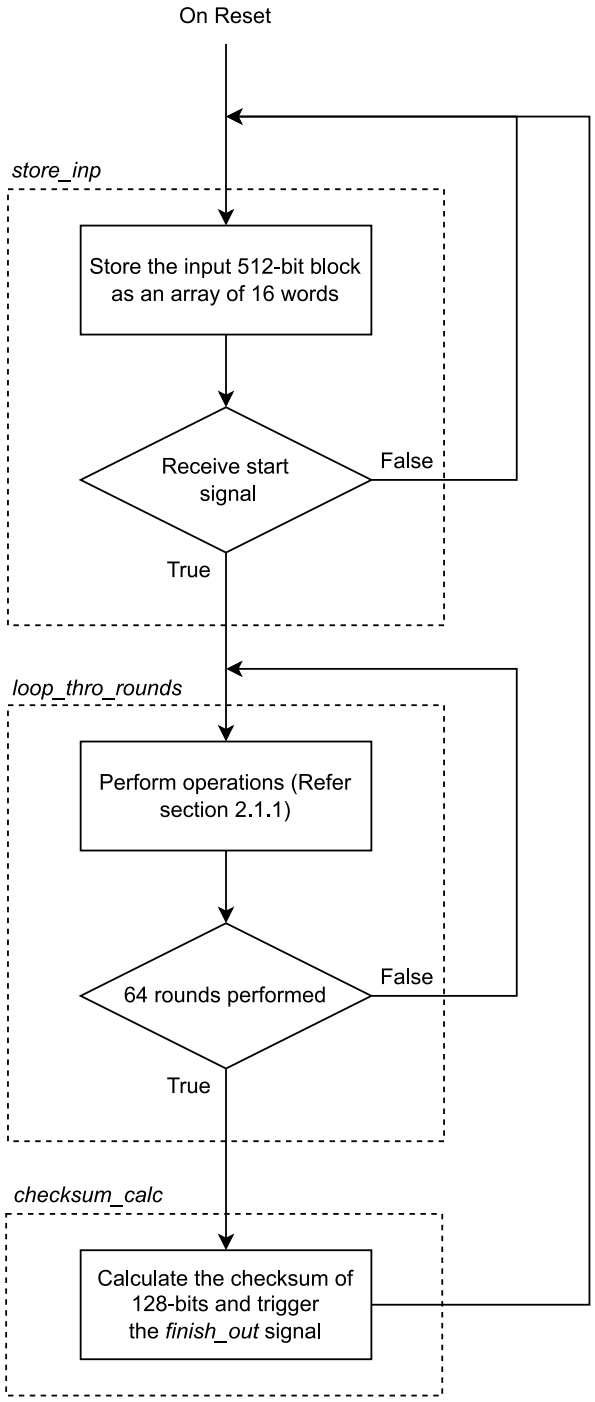

Figure 3.5: MD5 sub-module state machine

3.2.2 SHA-1

The SHA-1 sub-module takes an input of 512-bit block and outputs a 160-bit checksum value. The input ports of this sub-module are *CLK*, *RESET*, *start_ in* and *data_ in*. The output ports of this sub-module include *data_ out* and *data_ valid*. The input port *data_ in* receives the 512-bit block of data and outputs a checksum of 160-bits through the output port *data_ out*. Figure 3.6 illustrates the overview of the sub-module.

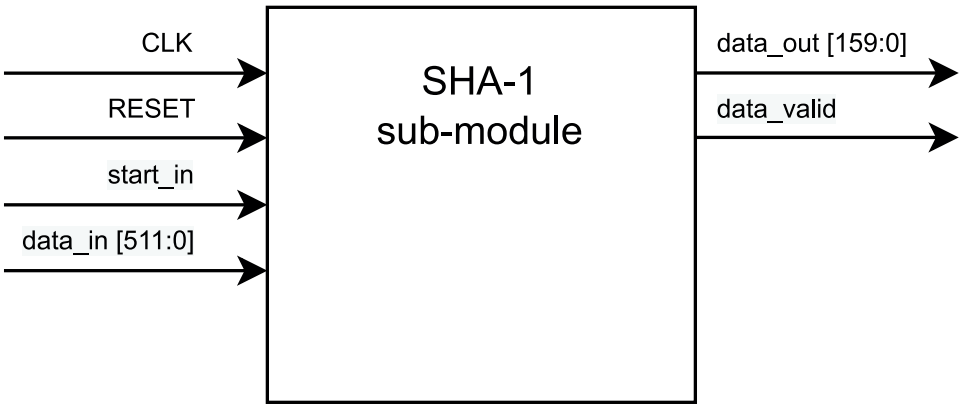

Figure 3.6: SHA-1 sub-module overview

At the beginning, all the initial buffers, viz. A, B, C, D and E, and constants are initialized. This sub-module works on a state machine of four states which are *store_ inp*, *full_ arr_ calc*, *loop_ thro_ rounds* and *checksum_ calc*. Figure 3.7 illustrates the state machine of SHA-1 module. The initial state of this sub-module is *store_ inp* state where the input block of 512-bits is stored as an array of 16 words of 32-bit data. When the *start_ in* port receives a bit value "1", it moves to the next state, *full_ arr_ calc*. In the *full_ arr_ calc* state, the array of 16 words is extended to an array of 80 words by performing a few specific operations(refer section 2.1.2) on the input data. After this array of 80 words is calculated, it enters the next state *loop_ thro_ rounds*. In the *loop_ thro_ rounds* state, 80 rounds of specific operations(refer section 2.1.2) are performed by updating the five buffers. After all the 80 rounds are performed, the state shifts to the *checksum_ calc* state where, the updated buffers are added with the initial buffers and fed to the *data_ out* port and triggers the *finish_ out* port. At last, it returns to the *store_ inp* state.

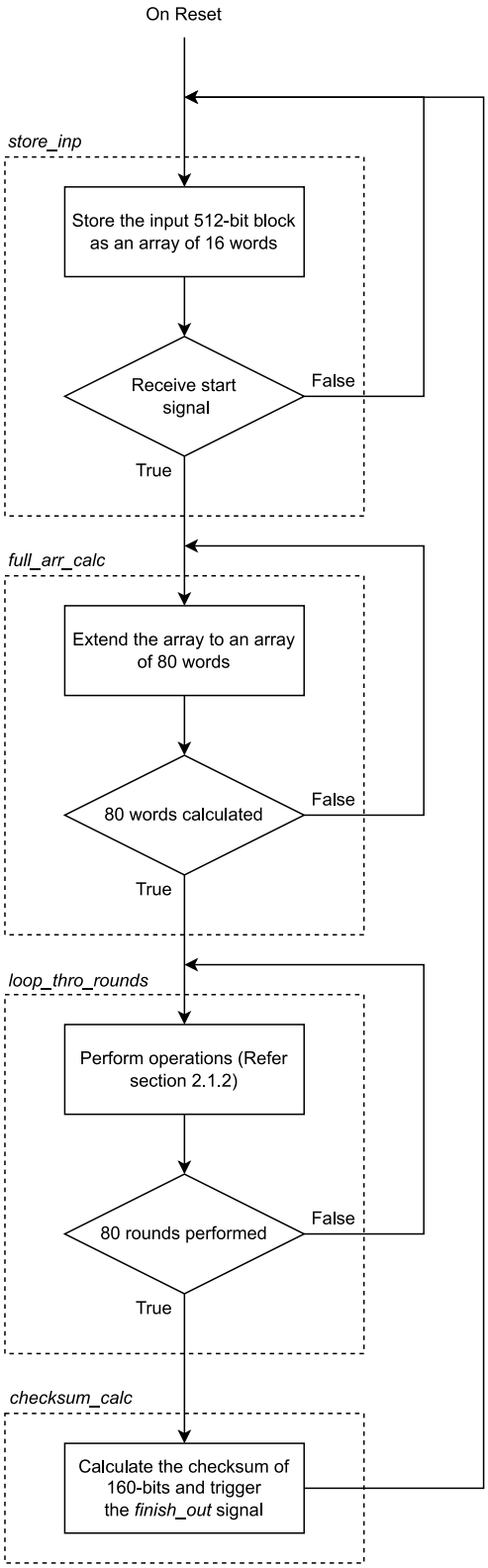

Figure 3.7: SHA-1 sub-module state machine

3.2.3 SHA-256

The SHA-256 sub-module takes an input of 512-bit block and outputs a 256-bit checksum value. The input ports of this sub-module are *CLK*, *RESET*, *start_ in* and *data_ in*. The output ports of this sub-module include *data_ out* and *data_ valid*. The input port *data_ in* receives the 512-bit block of data and outputs a checksum of 256-bits through the output port *data_ out*. Figure 3.8 illustrates the overview of the sub-module.

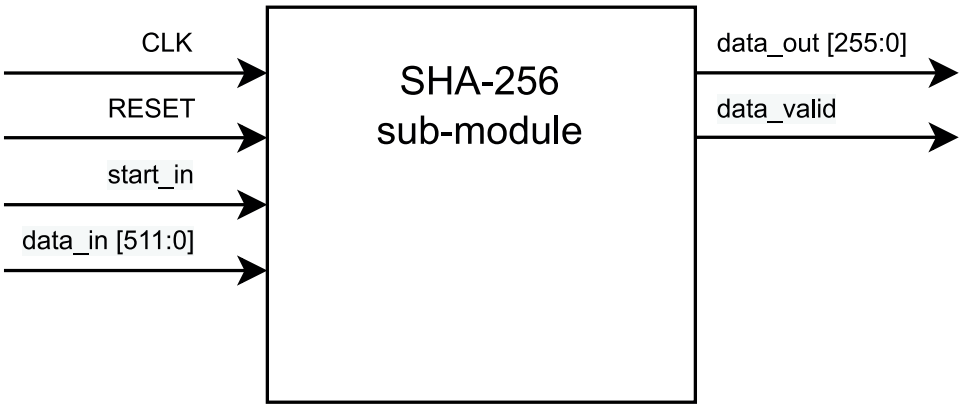

Figure 3.8: SHA-256 sub-module overview

At the beginning, all the initial buffers, viz. A, B, C, D, E, F, G and H, and constants are initialized. This sub-module works on a state machine of four states which are *store_ inp*, *full_ arr_ calc*, *loop_ thro_ rounds* and *checksum_ calc*. Figure 3.9 illustrates the state machine of SHA-256 module. The initial state of this sub-module is *store_ inp* state where the input block of 512-bits is stored as an array of 16 words of 32-bit data. When the *start_ in* port receives a bit value "1", it moves to the next state, *full_ arr_ calc*. In the *full_ arr_ calc* state, the array of 16 words is extended to an array of 64 words by performing a few specific operations(refer section 2.1.3) on the input data. After this array of 64 words is calculated, it enters the next state *loop_ thro_ rounds*. In the *loop_ thro_ rounds* state, 64 rounds of specific operations(refer section 2.1.3) are performed by updating the eight buffers. After all the 64 rounds are performed, the state shifts to the *checksum_ calc* state. In this state, the updated buffers are added with the initial buffers and fed to the *data_ out* port and triggers the *finish_ out* port. At last, it returns to the *store_ inp* state.

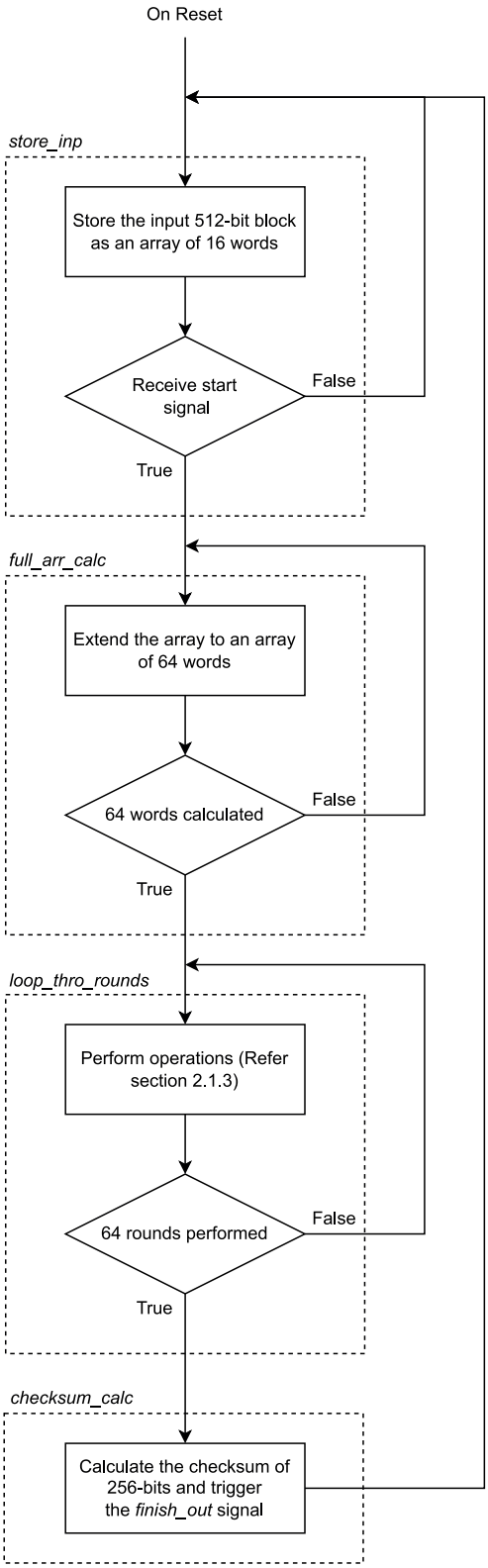

Figure 3.9: SHA-256 sub-module state machine

Chapter 4

Customization of PULPissimo

4.1 Integration of algorithm to PULPissimo

The implemented checksum algorithms are integrated to the PULPissimo. This integration is done using the APB available in the PULPissimo. The first step of the integration includes provision of an address range to the new module. As shown in Figure 2.7, it is clear that addresses upto 0x1A110000 are used. Hence, the algorithm module that was developed is given an address range of 0x1A121000 to 0x1A121FFF. A new bus for this developed module is created and a new peripheral is also instantiated in the PULPissimo. In the peripheral module, the developed module is instantiated. The address 0x1A121000 is used for sending the configuration of the ROM IP and address 0x1A121004 is used for sending the range of addresses the algorithm needs to process. The output ports of the algorithm, viz. *data_ out* and *data_ valid*, are port mapped to the top most module of the PULPissimo for verification purposes.

4.2 Physical Implementation

For further study, the PULPissimo needs to be synthesized by making a few modifications. Firstly, the generic technology cells, viz. clock gating cells and SRAM models, need to be replaced with the technology cells used for synthesis. The generic memory in the PULPissimo is replaced with the organization's SRAM macros. Since both the size of the generic memory used in the PULPissimo and the SRAM macro of organization are equal to 32kB, the generic memory is directly replaced with the SRAM macros. By replacing them, all the interleaved banks(L2 cache) and the private banks(instruction and data memory) use the SRAM macros.

Secondly, the FLLs used in the PULPissimo are bypassed as they are not synthesizable. These FLLs are bypassed by writing the JTAG register before the reset signal is asserted. Hence, the peripheral clock, core clock and the cluster clock use the external clock. Figure 4.1 depicts the PULPissimo architecture after the checksum algorithm module is integrated and FLLs are removed.

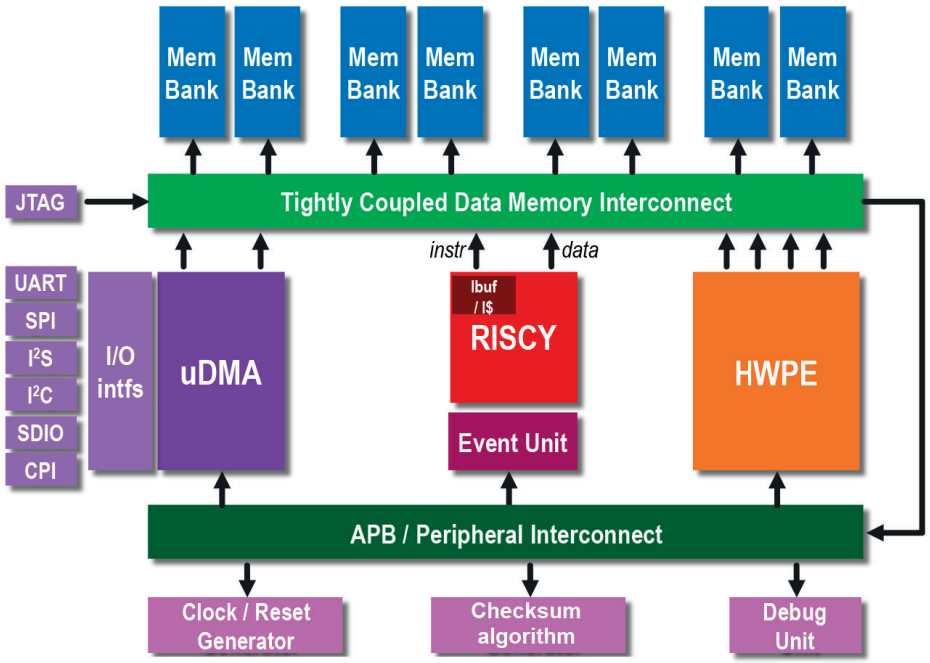

Figure 4.1: Block Diagram of customized PULPissimo

After these modifications are made to the PULPissimo, RTL level simulation was performed and verified. The whole design was then synthesized with effort level high for synthesis generic, mapping, and optimization to achieve best possible optimized hardware at this stage. By performing a few rounds of optimizations a better QoR was achieved. The synthesized netlist was then verified through simulation. Further, PnR was performed on the synthesized netlist with zero timing violations. At last, post-PnR simulation was performed to verify the final netlist. The results of the synthesis and PnR stage are discussed in Chapter 5.

Chapter 5

Results

5.1 Results of Simulations

For the developed checksum algorithm RTL modules, behavioral and post-synthesis simulations have been carried out in Cadence Xcelium with a ROM IP of size 0.5kB. For both the simulations, the inputs are fed through a RTL test bench. The simulation waveforms of the behavioral and post-synthesis were found to be same for each of the algorithms as shown in Figures 5.1, 5.2 and 5.3. These simulations have further been verified with an online tool[8], which calculates various checksums, by feeding the same data stored in the ROM IP. The used ROM IP stored 32-bit words in 16 addresses which are given in Appendix A.1.

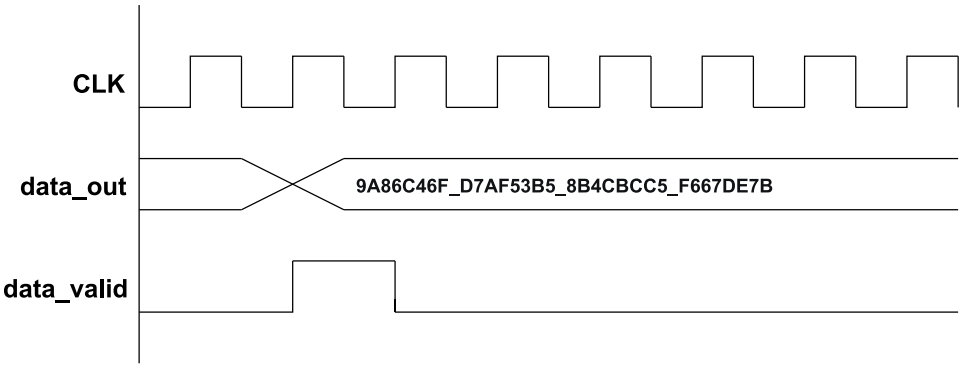

Figure 5.1: MD5 sub-module waveform

Figure 5.1 shows the simulation waveform of the developed MD5 algorithm. The *data_ out* signal value obtained when the *data_ valid* signal goes high is compared with the output of the online tool and are found to be same.

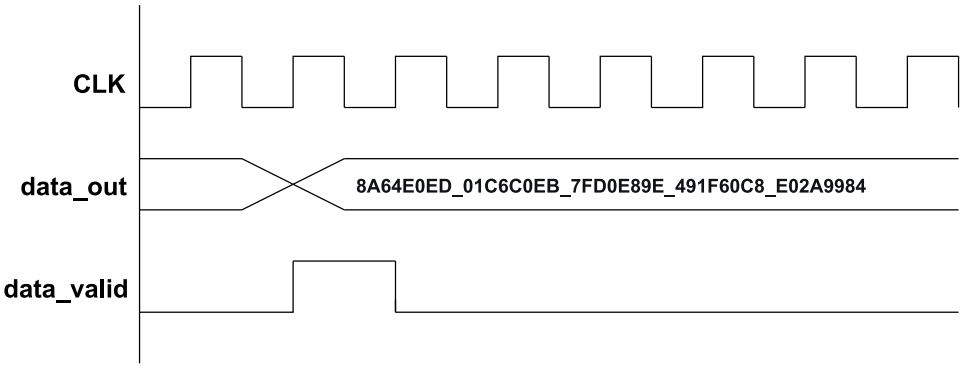

Figure 5.2: SHA-1 sub-module waveform

Figure 5.2 shows the waveform of the developed SHA-1 algorithm. The *data_out* signal value obtained when the *data_valid* signal goes high is compared with the output of the online tool and are found to be same.

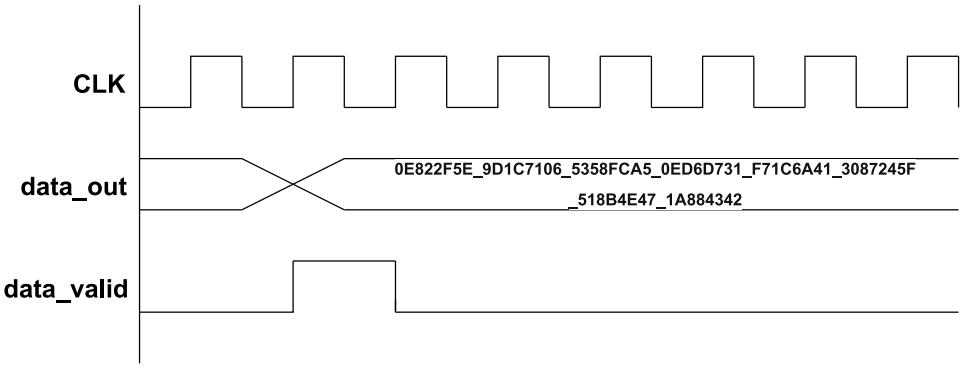

Figure 5.3: SHA-256 sub-module waveform

Figure 5.3 shows the waveform of the developed SHA-256 algorithm. The *data_out* signal value obtained when the *data_valid* signal goes high is compared with the output of the online tool and are found to be same.

Subsequently, each of the developed algorithms were integrated to the PULPissimo as explained in section 4.1. Further, the PULPissimo integrated with the algorithm was verified with the same ROM IP considered in the previous verification process. Instead of feeding the input through a testbench, here, the input is fed to a C-program which is compiled for generating a stimulus file. Then, this stimulus file is fed through the JTAG ports available in the PULPissimo. The output waveforms of each of the integrated algorithms were observed to be same as those given in Figures 5.1, 5.2 and 5.3.

5.2 Results of Synthesis

5.2.1 Comparison of Checksum Algorithms

Each of the developed checksum algorithms was synthesised using Cadence Genus tool with 22nm standard cell technology. They were synthesised at a clock period of 3000ps. The results of various metrics, viz. number of gates used, time slack and clock cycles(latency) to calculate a checksum for an input data of 0.5kB, have been tabulated in Table 5.1. Results of each of the algorithms are compared with the results of Study 1[9] and Study 2[10] reported in section 1.2 which also used an input data of 0.5kB.

	Tech (nm)	**Gates**	**Latency (cycles)**	**Time Slack(ps)**
MD5				
Study 1[9]	350	8001	712	-
Study 2[10]	130	10332	68	-
Present study	22	6482	84	363
SHA-1				
Study 1[9]	350	8120	1274	-
Study 2[10]	130	7971	85	-
Present study	22	17416	164	407
SHA-256				
Study 1[9]	350	10868	1128	-
Study 2[10]	130	11484	72	-
Present study	22	17085	132	471

Table 5.1: Comparison of checksum algorithms synthesis results

Generally, a fair comparison with the literature study is difficult to achieve because all published works use different standard cell technologies and have different design goals. In the case of Study 1, the goal was to reduce the power consumption at the cost of extended latency. Whereas, in Study 2, the motivation was to compare various checksum algorithms and achieve a higher throughput, ie. lower latency. The goal of present thesis is to read data from the ROM IP, calculate the checksum and assess the latency.

It is evident from Table 5.1 that the number of gates used by different algorithms in every study varied significantly as the goals are different and the technologies used are also different. It is to be noted that the latency results of Study 1 for each of the checksum algorithms are significantly higher than those of Study 2 and present thesis. This observation is expected as Study 1's goal was achieved at the cost of latency. Hence, the latency of the present thesis can be compared with Study 2 results as their goals are similar. Even though, the latency results are in comparable range, the current thesis results are higher than Study 2 results. The

primary reason for higher latency in all the three algorithms is that the current thesis reads data from ROM IP and pre-processes the data by padding which was not part of Study 2. Further, in case of SHA-1 and SHA-256, it can be observed that the present study's latency results are almost twice that of the Study 2's results. The reason for this observation is, apart from reading from ROM IP and padding, due to the additional processing of extending the array as explained in sections 3.2.2 and 3.2.3 which was not part of Study 2.

While comparing the results of the present thesis among the three checksum algorithms, it is noticed that the number of gates used by MD5 algorithm is almost one-third of that of SHA-1 or SHA-256. Also, the computational cycles used by MD5 is significantly lesser than the other two algorithms. Even though, the time slack in MD5 is observed to be marginally lower than those in SHA-1 and SHA-256, MD5 is a good choice in the present study by considering the results of the other two metrics.

5.2.2 Comparison of cores in PULPissimo

Out of the box PULPissimo was synthesised with each of the two cores (RISCY and Ibex) available using 22nm standard cell technology. Their performance(speed) and area(number of gates) obtained based on the synthesis are tabulated in Table 5.2. Although, PULPissimo with RISCY core is bigger than with Ibex core, it is observed that the former is faster than the latter. Hence, RISCY core is considered for the testing platform. Further, modules and peripherals such as SPI, I2C, I2S, Camera and Advanced Timer in PULPissimo, which are not essential for the organization, were removed to save area. The chosen MD5 algorithm was then integrated and synthesised for final synthesis results. These results are also tabulated in Table 5.2. Due to the removal of various unessential modules and integration of MD5 algorithm, the number of gates are observed to be reduced to 46%. However, the speed is observed to be bound by the speed of the core which is 135MHz.

Design	**Gates**	**Speed(MHz)**
Ibex core	170503	90
RISCY core	208448	135
Present study	96443	135

Table 5.2: Comparison of PULPissimo synthesis results

5.3 Results of PnR

PULPissimo with RISCY core that was synthesized, after the removal of unessential modules and integration of MD5 algorithm, is fed to the PnR flow in order to realize the ASIC layout and discuss the resource utilization. Cadence Innovus tool was used for the PnR flow. After routing of the whole design, the layout is obtained as shown in Figure 5.4. This floorplan resulted in the best QoR (Qual-

ity of Result) for the PULPissimo SoC with 22nm technology node. Further, the design was clean with zero timing violations and congestion free. The core height of the layout was bounded by the height of the memory banks as shown in Figure 5.4. The core area was designed with a width of 1300um and height of 925um. The standard cells were placed in 964 site rows. It was also observed that 82% of the area was occupied by the memory banks and the remaining 18% was occupied by the logic. Such huge area percentage of memories is due to the memory macro size and their count in the design.

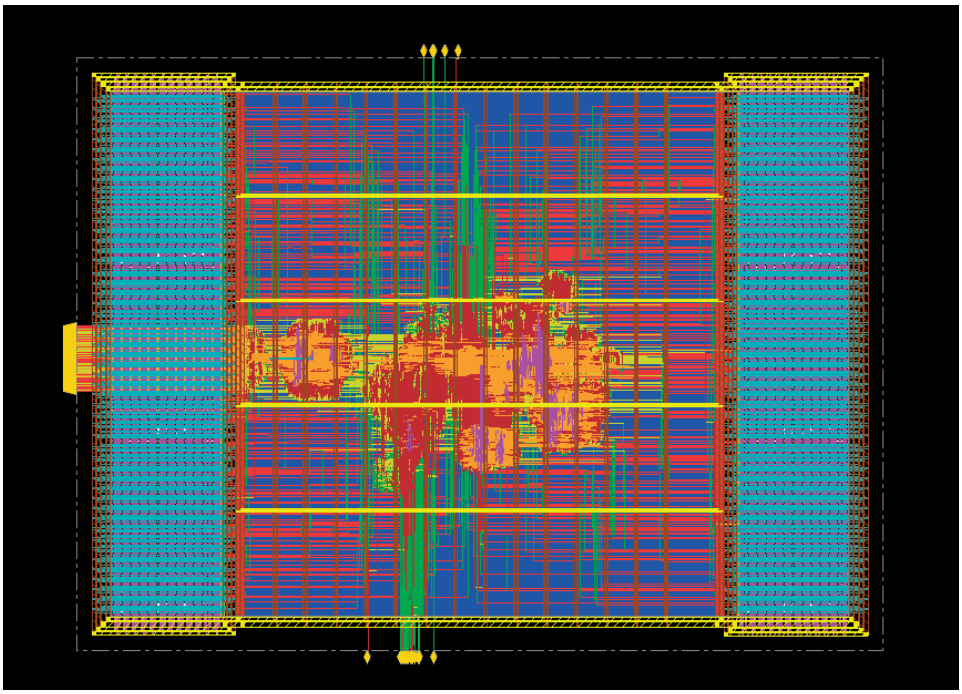

Figure 5.4: Layout of PULPissimo SoC

Chapter 6
Conclusion

Memory IPs are an important component in the current SoC designs. Making sure that the memory IPs are functioning as expected is crucial for any organization. In this thesis, to test the ROM IPs, various checksum algorithms, viz. MD5, SHA-1 and SHA-256, were implemented in RTL. The developed checksum algorithm RTL modules were verified through behavioral and post-synthesis simulations. Further, the synthesis results of each of the algorithms were compared, with respect to metrics such as number of gates used and latency, to analyse and to choose a suitable algorithm for the organization. Based on these comparisons, MD5 was considered as the suitable choice. Further, testing capabilities were enhanced by integrating a processor to the memory test chip. In this project, an open-source PULPissimo platform based on RISC-V ISA(Instruction set architecture) was used as this gives freedom to the system designer and the organization to configure the processor core as per the requirements. Further, PULPissimo was synthesised with each of the available cores, RISCY and Ibex, to make a comparison with respect to area and performance. Based on this comparison of speed results, RISCY was chosen for the final design. In addition, to save area, unessential modules such as SPI, I2C, I2S, Camera and Advanced Timer in PULPissimo were removed. Then, the developed MD5 RTL module was integrated to the PULPissimo through the APB protocol available in PULPissimo. This design was synthesised for further results. The synthesis netlist was fed to the PnR flow in order to realize the ASIC layout and to discuss the resource utilization. Finally, the SoC was verified through post-layout simulation. The verified SoC provides a testing platform for the organization to test their ROM IPs.

6.1 Future work

The following are the future scope of work identified:

- Various other algorithms such as SHA-384, SHA-512, SHA3, etc. can be implemented to identify a more suitable algorithm.
- The developed SHA-1 and SHA-256 RTL modules or any other algorithms existing in the organization can also be integrated to the PULPissimo which can improve the reliability of testing the ROM IPs.

References

[1] PULPino GitHub Project : `https://github.com/pulp-platform/pulpino`

[2] PULPissimo GitHub Project : `https://github.com/pulp-platform/pulpissimo`

[3] mBIST : Marinescu, M., 1982. Simple and Efficient Algorithms for Functional RAM Testing. 1982 IEEE Test Conference, Philadelphia, (Nov.). IEEE Computer Society, pp. 236-239.

[4] IEEE standard for test access port and boundary-scan architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001),* pages 1–444, 2013.

[5] Checksum Algorithm : `https://en.wikipedia.org/wiki/Checksum`

[6] Github Knowledge : `https://github.com/`

[7] `https://github.com/pulp-platform/pulpissimo/blob/master/doc/datasheet/datasheet.pdf`

[8] Checksum online tool : `https://emn178.github.io/online-tools/index.html`

[9] M. Feldhofer, and J. Wolkerstorfer, "Strong Crypto for RFID Tags - A Comparison of Low-Power Hardware Implementations," IEEE Int'l Symp. on Circuits and Systems, 2007, pp. 1839-1842.

[10] A. Satoh, T. Inoue, "ASIC-Hardware-Focused Comparison for Hash Functions MD5, RIPEMD-160 and SHS," ITCC2005, pp.532-537.

[11] RISCY core : Gautschi, Michael, et al. "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 25.10 (2017): 2700-2713.

[12] Ibex core : Schiavone, Pasquale Davide, et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications." 2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). IEEE, 2017.

[13] MD5 : Rivest, Ronald L. (April 1992). "RFC 1321 – The MD5 Message-Digest Algorithm". Internet Engineering Task Force.

[14] SHA-1 : D.Eastlake, P.Jones (September 2001). "RFC 3174 - US Secure Hash Algorithm 1 (SHA1)". Internet Engineering Task Force.

[15] https://www.simplilearn.com/tutorials/cyber-security-tutorial/sha-256-algorithm

[16] https://web.eecs.umich.edu/~prabal/teaching/eecs373-f12/readings/ARM_AMBA3_APB.pdf

Appendix A
Values stored in the ROM IP

A.1 Values stored in the ROM IP

Address	Value
1	0x63FEC795
2	0xD47A476C
3	0xAA5451FE
4	0xBC6109A7
5	0x8A3C006F
6	0xEE3BE66C
7	0xE0C93E0D
8	0xD25C6A92
9	0x6D1AA1C4
10	0xB749A94B
11	0xE8E482D5
12	0xD58AEE44
13	0x6FC0933A
14	0x97CF7526
15	0x18340E33
16	0xB4B0DC97

Table A.1: Values stored in the ROM IP

Printed by Tryckeriet i E-huset, Lund 2022

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2022-895
http://www.eit.lth.se