Design and implementation of testable fault-tolerant RISC-V system

MATTIAS RODAN
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Design and implementation of testable fault-tolerant RISC-V system

Mattias Rodan
mattiasrodan@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Hemanth Prabhu, Xenergic
Hemanth.Prabhu@Xenergic.com
Supervisor: Steffen Malkowsky, Lund University
steffen.malkowsky@eit.lth.se

Examiner: Pietro Andreani, Lund University

December 17, 2020

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

This thesis aims to investigate and implement a fault-tolerant energy-efficient RISC-V based system on chip (SoC). Key features of the SoC is the testability and reliability of the low power on-chip embedded memories. A built-in self-test (BIST) for the on-chip memories has been designed and implemented to run on-demand diagnostic tests to detect manufacturing errors in the memories. It incorporates three different algorithms to test for common manufacturing memory faults. Runtime soft errors are detected and corrected using an error correction code unit (ECC), which can correct up to two errors. The ECC is integrated with the RISC-V core and memories to increase the fault-tolerance of the SoC at low voltages. The ECC components importance increases as the probability of soft errors increase with lowering of the supply voltage. Power savings up to 46% for the entire system was simulated when the supply voltage was decreased from 1.2V down to 0.8V. The addition of the ECC components resulted in a 3.5% core area increase. The integrated memory built-in self-test contributed to another 24.4% area increase of the core.

Popular Science Summary

Improved semiconductor manufacturing processes and techniques have paved the way for the development of more complex and power-efficient integrated circuits. The higher achievable transistor density enables more functionality to be packed into the same circuit die than before. These advanced computational circuits are embedded in all handheld electronic devices and the internet of things that are present in our day to day life. Higher transistor density combined with high consumer demands on battery lifetime and functionality requirements creates difficult challenges in the development phase of these circuits. As the complexity increases the risk of manufacturing process variations is more likely to affect the system functionality. Fault-tolerant and error detection techniques can be incorporated into the system to counteract the effect of these potential manufacturing errors.

As the systems get more complex over time, the task of conducting tests and diagnostic evaluation of the chips grow. Typically, specific circuits are developed and integrated on-chip that can conduct tests when the chips are manufactured. The data collected can be used to get statistical information about which parts of the design that are most prone to failures.

In this thesis, a system has been implemented where fault-tolerant components have been added to the interface between a 32-bit RISC-V core and its memory subsystem. Because of the high memory usage, a memory built-in self-test has been implemented and integrated to test and provide fault diagnosis. A built-in self-test has also been developed to run diagnostic test of the fault-tolerant components at the memory interfaces. The supply voltage was decreased lower than the technology standard to decrease power usage. Voltage scaling provides a trade-off between operational performance speed of the circuit and power. A JTAG interface is used to control the system with configuration registers and to perform memory operations such as inserting the program instruction that the RISC-V core runs.

Acronyms

ADF Address Decoder-Fault
AF Address Fault
BL Bit-line
BLB Bit-line Inverse
BIST Built-In Self-Test
BEF Byte Enable Faults
CMOS Complementary Metal Oxide Semiconductor
CF Coupling-Fault
ECC Error Correction Code
EX Execution
FSM Finite State Machine
GPIO General Purpose Input-Output
HDL Hardware Description Language
ID Instruction Decode
IF Instruction Fetch
ISA Instruction Set Architecture
IP Intellectual Property
JTAG Joint Tag Action Group
MBIST Memory Built-In Self-Test
MSB Most Significant Bit
PnR Place and route
RISC Reduced Instruction Set Computer
RTL Register Transfer Logic
SRAM Static Random Access Memory
SAF Stuck-At-Fault
SoC System-on-Chip
TF Transition-Fault
WL Word-line
WB Write Back
6-T 6 transistor

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

System-on-chip (SoC) with embedded processing units are commonly integrated into devices used in the everyday life such as phones, cameras, and smartwatches. The SoCs main task in these devices is to enable the data processing required for the product to meet its application requirements. These handheld devices are often powered by a battery with limited energy budget before recharging is needed. The data processing within the processing unit consumes energy which drains the battery of its charge and is thus an important factor of the device battery life.

The design requirements of handheld products are a strict trade-off between battery life, performance, and size. An important approach to achieve a good trade-off between the previously mentioned requirements is to improve the computational energy efficiency. This is especially interesting in applications that require extensive computations and where the processing accounts for a large part of the energy consumption. It is therefore interesting to investigate the possible improvement of the SoC that contains single and multi-core processors to attain higher computation efficiency while still meeting the performance requirements.

Other important problems is how the circuit functionality can be tested, verified to reduce faults during its lifetime. Especially tests of memories are of interest which often accounts for the majority of the total chip area. A common solution to the problem is to develop additional circuit components that are embedded in the chip that target the test of the circuit, called built-in self-test (BIST). A similar approach is taken in order to increase the system fault-tolerance where additional components are used to detect data errors and correct these during run time.

1.1 Background

Development of SoC is a complex and expensive process which most often does not allow the organization to design all circuit components in-house. This is mainly due to the extensive development process and increased time to market which is considered as a very important. The drawback of complete in-house development often forces organizations to use other companies intellectual property (IP) against license payments which in turn also can be very costly. A solution to these expensive fees is then to use a non-license open-source alternative.

There has for a long time existed different open-source alternatives for processing units. However, the open-source community has flourished over the past years and there are many available alternatives. An instruction set architecture (ISA) for embedded processing units that have gained popularity in the last year in both academia and industry is RISC-V. This ISA is based on the well established reduced instruction set computer (RISC) concept. RISC-V originated from research at Berkeley. Several open-source cores based on this ISA have been released to the public. Performance of these cores, in many cases showed to be able to achieve competitive performance compared to available licensed products [3].

The processor cores are usually well-integrated with a memory which it uses to fetch instructions and store data during computations. The memory system can be divided into two different sub-systems called program and data memory, which fulfill these functions. The trend of memory utilization in typical SoC shows the increasing importance of memory [4]. It is common that embedded memory occupies a majority of SoC and the utilization is forecasted to reach levels up to 90% [5]. Due to the high utilization of memory, it becomes further important to ensure its functionality and increase its fault tolerance.

1.2 Project aim

This thesis main goal is to implement a minimal and fault-tolerant and energy-efficient SoC based on a 32-bit open-source processor with compiler support. The developed SoC aims to attain the best possible energy numbers per operation while the test and fault-tolerance functionality still is embedded on the SoC.

The processor within the SoC should utilize the available open-source RISC-V ISA. A testable fault-tolerant memory sub-system should be incorporated into the SoC. A BIST and Error Correction Code (ECC) components will be used to obtain a testable fault-tolerant memory sub-system.

The BIST is required to run several test sequences to increase the fault coverage and the possibility to detect functionality problems that may arise in fabrication and during its lifetime. In particular, the debug functionality error log mechanism is of special importance. The error logger is a key component within the BIST. Its main task is to debug the detected problem and make it accessible. The information stored can be used to improve the understanding of the problem and ease the process of improving the circuit in future iterations.

A common failure apart from fabrication errors are soft errors. Soft error are data corruption errors that may arise randomly during run time and affect the system functionality and must, therefore, be counteracted. ECC components are used to detect and resolve the issue of potential data corruption. ECC consists of a decoder and encoder and are placed at the write and read path of the memory. These are used to implement different algorithm to express the data sequence in

a certain manner. These algorithms embed information so the errors can be detected and corrected. Xenergic will provide a version of an ECC component. The challenge is, therefore, to integrate an existing version of the ECC component into the SoC to maintain robust functionality.

The combination of BIST and ECC will be the major tasks, which are implemented to achieve a testable and fault-tolerant SoC. To control these peripheral circuits, a hardware controller needs to be designed and implemented. One feature of the controller should be to detect if soft errors occur too often. If so, the BIST should trigger again to run test sequences in pursuit to detect potential hard errors in the memory.

1.3 Thesis structure

Chapter 1. Introduction This chapter will provide an introduction to the problem and why a fault-tolerant energy-efficient RISC-V system is of interest.

Chapter 2. Theoretical background In this chapter all required background information regarding the problem and system are presented.

Chapter 3. JTAG interface Describes the JTAG interface implementation and all its configurations registers used to control the system operations.

Chapter 4. Built-in self-test and RISC-V Implementation Describes the implementation and architecture of the system. The system includes the designed test components and RISC-V core.

Chapter 5. Result and verification Contains waveforms to display scenarios of the test data sequences. Results regarding performance, power, and area are also presented in this chapter.

Chapter 6. Conclusion Present final result analysis and final thoughts.

Chapter 2
Theoretical background

This chapter will introduce and explain related background knowledge to provide a solid foundation needed for further discussions in this thesis. It will also contribute to understanding the reason why a fault-tolerant and testable system is desired.

2.1 Static random access memory

Memories are commonly used by the processor cores to store instructions and data during computations. The memory structure often consists of several levels of embedded cache memory to decrease operation latency time and optimize performance. A common factor for these embedded memories is that all of them are volatile memories. Volatile memory forfeits the values stored when the supply voltage is off. A very common version used in these embedded memories is static random access memory (SRAM). The static version does not need to have its value periodically updated as to its dynamic counterpart. It is also considered to be the faster version and is thus placed close to the core for performance optimization. A standard 6 transistor (6-T) bitcell version of an SRAM is shown in Figure 2.1. The single-port bitcell displayed in Figure 2.1 is connected to three different in-output ports called word-line (WL), bit-line(BL), and inverse bit-line(BLB). These are used to perform the read and write data operations to the bitcell.

A write operation is initiated by setting the selected data value on BL and the inverse value on BLB. However, transistor M5 and M6 currently prevent the value from being written to the data storage Q and $\overline{Q}$ nodes. WL is then activated high and the values on both bit-lines are latched into the data storage nodes and a successful write has been performed. A read operation usually starts with a pre-charge of both bit-lines to the supply voltage. The WL is then activated and the values on the data nodes Q and $\overline{Q}$ will be connected to the bit-lines through transistor M5 and M6. This will result in a small voltage shift between BL and BLB which can be detected and the read value can be determined by peripheral circuits.

The SRAM bitcell alone can only contain one-bit data which seldom is useful by itself. Therefore, several bitcells are combined into array configuration as shown in Figure 2.2. This is useful to increase the storage density. In the figure, it can be seen that each added column adds one bitcell to the common shared WL and

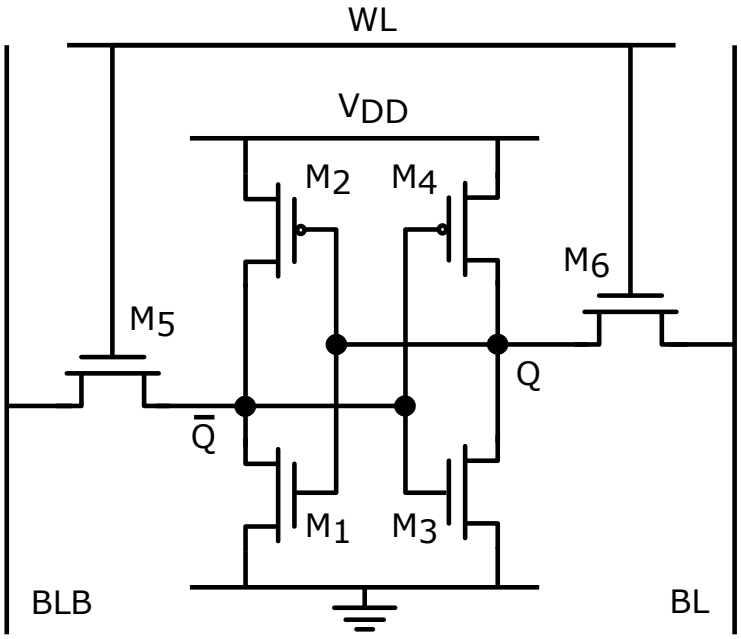

Figure 2.1: Standard 6-T SRAM bitcell

each added row adds a bitcell to the shared BL and BLB. Instead of performing read and write operations on single bitcells, the operations are applied to each row at a time.

In order to target which of the common WL to be activated during operation, an address decoder is used. It can be seen as a large multiplexer that only selects and activates one of the rows at a time to avoid conflict between them. A simplified overview of a memory bank can be seen in Figure 2.3.

2.2 Common memory faults

Two different but common error types that often emerge in these types of memories are soft errors and hard errors. Soft errors are data corruption that mainly arises from random bit-state flip due to charged alpha particles [6]. These alpha particles can induce a voltage at the data nodes Q and $\overline{Q}$ that can result in the harmful bit-flips. The negative impact of alpha particles increases when scaling down the voltage due to the increased risk of change of bit-state in low power designs[7]-[8]. Hard errors are caused by real errors in the circuit which may arise during fabrication due to design or process flaws. Hard errors will occur repeatedly during operation while the soft errors are randomly distributed across the memory.

There are a few basic common hard errors that will be targeted while performing

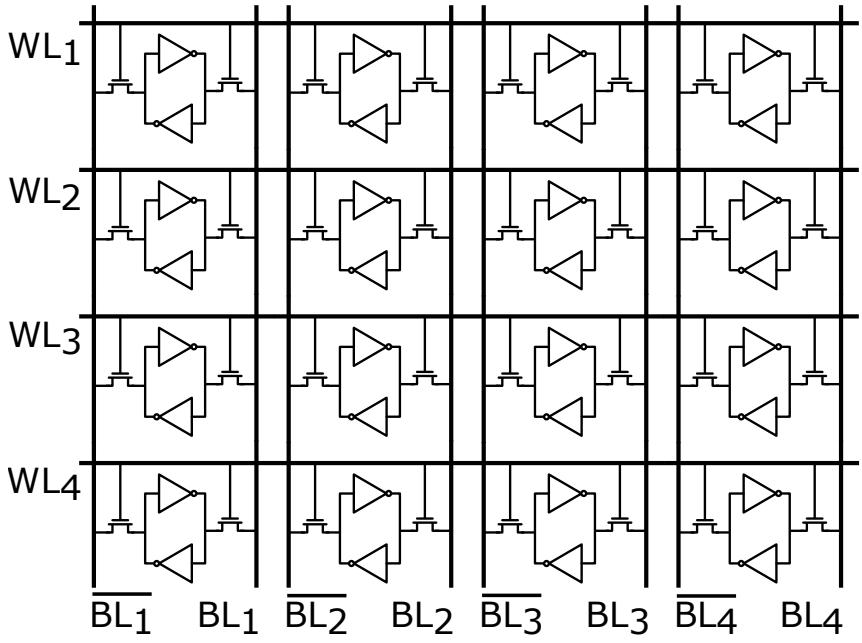

Figure 2.2: 4x4 SRAM bitcells connected to build a larger memory bank

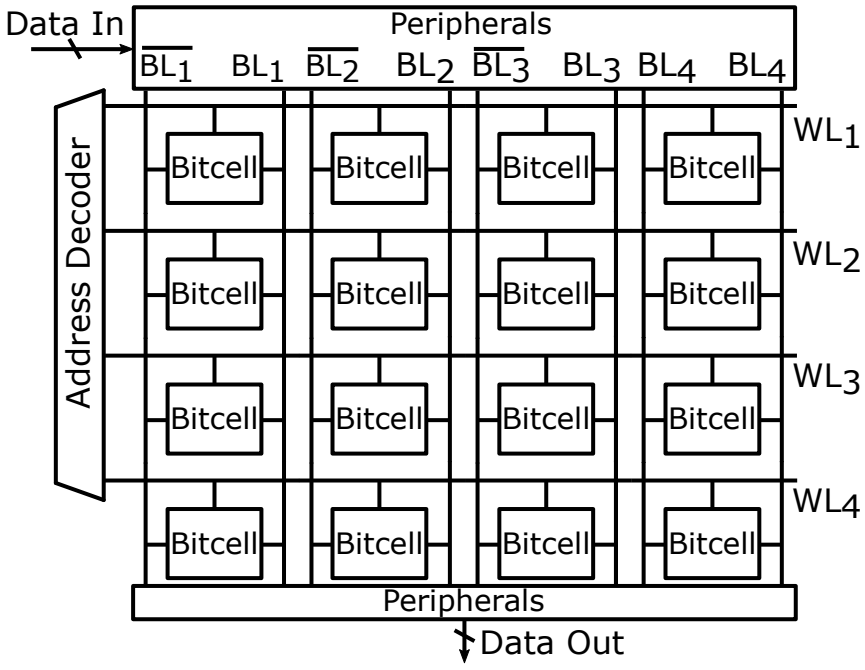

Figure 2.3: SRAM memory array connected with address decoder and other peripherals

the test sequence on the memory. These are the following, stuck at fault (SAF), address decoder fault (ADF), transition fault (TF), and coupling fault (CF). All these errors are mainly located in the memory banks of memory bitcells while ADF is in the peripheral circuits.

SAF errors deals specifically with problems with the data value in SRAM bitcells. It may be that the value can not be modified and is stuck at a certain value despite all other components work properly. This can be the case if either data node Q and $\overline{Q}$ is shorted to a supply rail.

ADF concerns issues with the address decoder which selects and activates WL. An example of potential problems is if different row addresses can not be selected due to a short or any other malfunction in the large address multiplexer. This causes incorrect behavior and affects the SoC functionality.

TF occurs when the SRAM bitcell fails to perform a transition in either direction. This will give a similar effect as in SAF errors and thus considered as a special case of SAF. The problem mainly arises because the data node can not be charged or discharged fast enough to meet the frequency requirement. This might be a consequence of a faulty sized connection than designed in layout due to fabrication errors. This can be fixed by lowering the timing requirements to enable more time to discharge and charge the data nodes.

CF happens when operations carried out on targeted memory cells affects the functionality of non-targeted memory cells. An example of CF could occur when data written to a selected byte in a word might cause a bit-flip in the other non-targeted part of the word due to capacitance coupling.

2.3 Built-in self test and memory test algorithms

There exist a large variety of different types of algorithms used to perform the test and diagnose of embedded memory. Common algorithm types that are used are the MARCH based test algorithms. These types of algorithms are easy to implement and exist in different versions to achieve the requested fault coverage. Another benefit of the algorithms is the linear dependency on memory size. This behavior is especially desirable because of the increased memory trend.

The MARCH based test algorithms can be seen as an iterative process where sequences of write and read operations are performed in different patterns to detect faults in the memory. A commonly used algorithm in the industry is MARCH-C. The algorithms write and read sequences are demonstrated in algorithm 2.1. The MARCH-C algorithm has gained popularity because of its large fault coverage of the previously mentioned common memory faults and due to its low computational complexity. The arrows are the directions in the address space in which the operations should be performed. The characters w0 and w1 correspond to a write operation with values 0 and 1. Similar, r0 and r1 are read operations where the

value represents the expected read value for correct functionality.

$$\{\updownarrow (w0); \uparrow (r0, w1); \uparrow (r1, w0); \downarrow (r0, w1); \downarrow (r1, w0); \updownarrow (r0)\} \tag{2.1}$$

2.3.1 Detection of memory faults with MARCH algorithm

Different sections of the MARCH-C algorithms are added to target different errors. SAF is detected by applying a write action to a target address with the subsequent action being a read operation. To cover all SAF faults this procedure must be performed with both values 1 and 0 for all address locations. A potential write and read can be seen in algorithm 2.2.

$$\{\updownarrow (w0); \updownarrow (r0, w1); \updownarrow (r1)\} \tag{2.2}$$

The other fault requires more advanced algorithm since the detection of these errors require more operations, and set higher requirement on the operations performed in the address space. As an example, ADF is a special case of SAF, the algorithm used to detect these errors involve detection of SAF errors as well. Algorithm 2.3 cover all AF and SAF faults. The main difference between ADF and SAF error technique is that the sequence direction of read and write is specified.

$$\{\updownarrow (w0); \uparrow (r0, w1); \downarrow (r1, w0)\} \tag{2.3}$$

2.3.2 Built-in self-test

BIST is an additional component added into integrated circuits (IC) with the main purpose to ease the test procedure. The basic functionality of the BIST is to provide a test pattern to a device under test (DUT) and then analyze the output from the DUT. The output is compared against the expected result that would be generated from a device with correct functionality. A mismatch in the comparison identifies incorrect functionality and an error event can be triggered. The BIST contains a controller to handle the test generator and comparison units. A basic overview of a simple BIST is displayed in Figure 2.4.

One of the advantages to include a BIST is that tests can be initiated by the IC itself. The test diagnostics can be triggered during any phase of its lifetime to verify its functionality and alert the system when errors have been found so counteraction can be taken. It is therefore common to run periodical test sequences during its lifetime to assure system functionality.

2.4 Error correction code

ECC utilises information encoding and decoding algorithms commonly used in sensitive communication channels where faults due to interference should be counteracted. The interference can result in corrupt information which might be harmful to the critical system functionality. The occurrence of corrupt information increases as the supply voltage decreases [7],[8]. In memories, this interference can

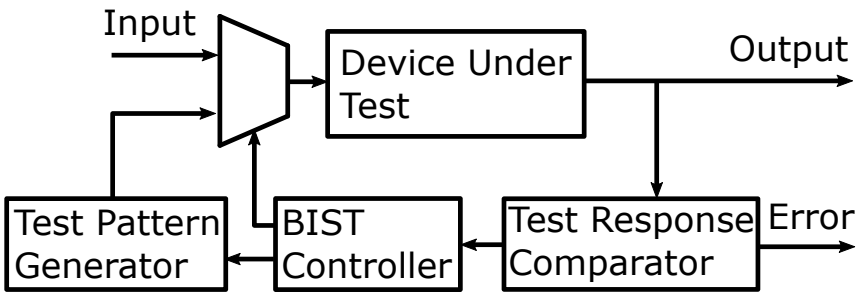

Figure 2.4: BIST architecture overview

be the result of data bit-flip caused by the soft error alpha particles or capacitive coupling. Hard errors can also be interpreted as permanent and constant interference that corrupts the memory data.

The ECC algorithms require extra memory bits to store an ECC code parity bits based on the memory data. This code is generated by an encoder that is placed at the memory input path. The memory output is in a similar fashion connected to a decoder which will analyze the data with corresponding ECC parity code according to the implemented algorithm. The decoder can detect and correct errors caused by interference with the help of the information stored in the additional bits. However, in which degree it can resolve the errors depends heavily on the selected and implemented encoding and decoding algorithm. Two different ECC implementations can be seen in Figure 2.5 and 2.6. The data word is marked W and parity bits (P) in the figures. In Figure 2.5 the extra parity bits have been embedded into the data memory. The consequence of this may result in the use of non-standard memory cuts with data width as an integer power of two. Figure 2.6 shows an ECC implementation where data is stored into one memory while its corresponding ECC parity bits are stored into a different memory. This enables the possibility to use two different standard memory cuts.

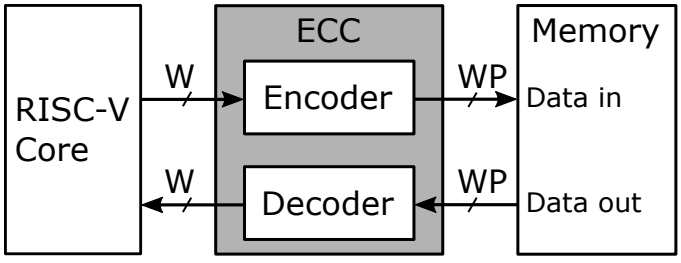

Figure 2.5: ECC parity code embedded with data

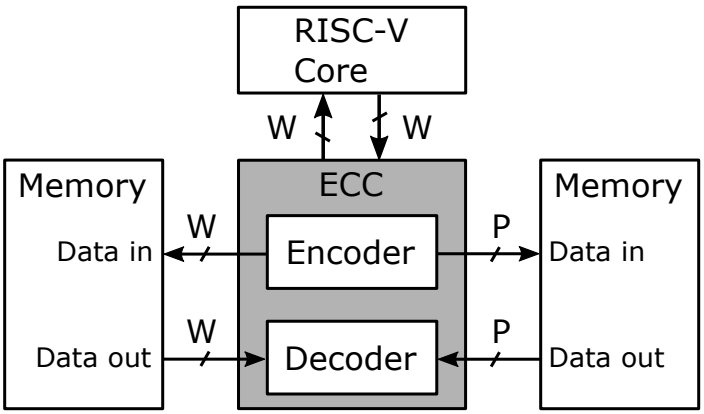

Figure 2.6: Seperate memory used to store ECC parity bits

2.5 Digital design power consumption

The power consumption of an integrated complementary metal-oxide-semiconductor (CMOS) SoC is generally divided into three major categories. These three major parts are the static, dynamic, and short-circuit and the sum of these components adds up to the total power consumption as in equation 2.4.

$$P_{total} = P_{static} + P_{dynamic} + P_{short\ circuit} \tag{2.4}$$

The static power component is used to describe the energy consumed when there is no transistor switching activity. This means that there is energy consumption even when no operations are active. The transistors used in CMOS still have a finite resistance despite being in their off state. This finite resistance between supply and ground results in a leakage current if they do not have the same potential. A simple scenario where the path and direction of the current I_{static} can be seen in Figure 2.7. Static power consumption is calculated as in equation 2.5 when a supply of V_{dd} is applied.

$$P_{static} = I_{static} \cdot V_{dd} \tag{2.5}$$

Dynamic power consumption is the energy needed to charge and discharge the internal circuit capacitance and the load capacitance during transitions. The power is based on the number of transitions per second which is determined from the operation frequency f and the switching activity α. The equation for dynamic power consumption is calculated as in 2.6. The simplified current path for the dynamic part is shown in Figure 2.7.

$$P_{dynamic} = \alpha \cdot (C_{load} + C_{internal}) \cdot f \cdot V_{dd}^2 \tag{2.6}$$

Short circuit power consumption is often combined with the dynamic because it

occurs during switching activity in CMOS. During a portion in the transition of the inverter in Figure 2.7 there is an occasion where both of the transistors are in their non off state. The finite resistance between the supply and ground that is caused by this will conduct a current that results in increased power consumption. Equation 2.7 is used to calculate the short circuit power consumption, where t_r and t_f is the rise and fall time of the transition.

$$P_{short\ circuit} = (\frac{t_r + t_f}{2}) \cdot f \cdot I_{peak} \cdot V_{dd}^2 \tag{2.7}$$

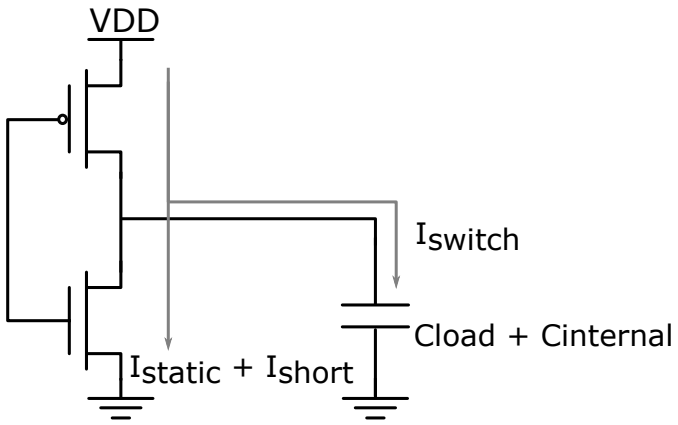

Figure 2.7: Standard inverter to demonstrate different power consumption components of circuits.

2.5.1 Voltage scaling

The total power consumption of the SoC is heavily dependent on the supply voltage which is a factor in each of the power components displayed above. Especially the dynamic power is more dependent on the supply voltage because of its quadratic supply voltage factor. The supply voltage is therefore an important factor to consider to achieve the low power requirements set on a system. Voltage scaling is an effective tool to achieve low power consumption, this can be implemented either in a dynamic or static manner. In the dynamic approach, the supply voltage can be regulated by voltage control circuitry whereas in the static a fixed voltage is selected. The selected supply voltage most often is a trade-off between the power, speed, reliability, and functionality requirement.

2.5.2 Clock gating

Dynamic power consumption is dependent on the switching activity and frequency as seen in equation 2.6. A common strategy to reduce dynamic power in a circuit is to control unnecessary switching transitions of the large capacitance load that toggles in the clock network. This low power method is called clock gating,

and it mainly targets sequential elements such as the flip-flops. Clock gating can be implemented at different levels in the circuitry. A simple and effective implementation is on top system level where logic is added to either enable or disable the clock input to a larger system component depending on control signals. The method can be implemented on all levels down to the very fine grain where clock gating is applied to each flip-flop. The different clock gating implementations can be seen in Figure 2.8.

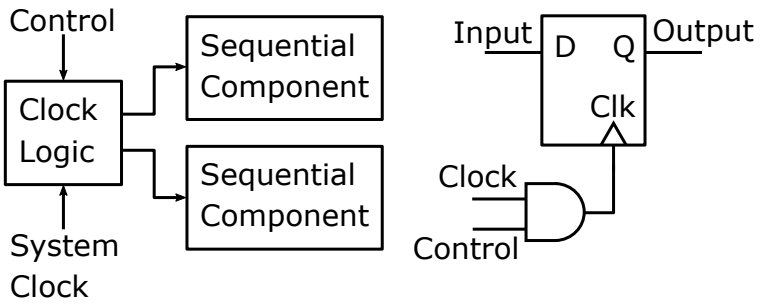

Figure 2.8: System level and fine grain level clock gating

2.6 RISC-V instruction set architecture

RISC is based on the strategy to use a small set of simple but optimized instructions that can be executed within one clock cycle to perform its tasks. More complex instructions are instead broken down into a sequence of several simple instructions. This simplification tends to result in cheaper design and development process. RISC based architectures have increased in popularity and are commonly used in portable low power devices. The RISC is not fundamentally more power-efficient than its counterpart complex instruction set computers [9] but rather optimized for different levels of performance.

RISC-V is a open-source instruction set architecture based on the RISC principles with a licence that require no fee payments. The ISA consists of a base instruction set with several different standard extensions available. The base and common extensions are displayed in Table 2.1. In addition to the available extensions, the ISA enables the possibility to create custom extensions to support application-specific requirements. The flexibility of the RISC-V ISA enables anyone to create their in-house processor implementations. This has resulted in several open-source processor implementations that can be used without license costs.

Table 2.1: Base Instruction set for 32-bit RISC-V ISA and common extensions

Name	Description
Base	
RVI32I	Base Integer Instruction Set for 32-bit
Extensions	
M	Standard Extension for Integer Multiplication and Division Instructions
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point Instructions
D	Standard Extension for Double-Precision Floating-Point Instructions
C	Standard Extension for Compressed Instructions

2.7 JTAG

Joint Test Action Group (JTAG) 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 2.2.

The TMS and TCK pin is used to control the state translations in the 16 state JTAG test access port state-machine as shown in Figure 2.9. The state machine main task is to control JTAG access to the instruction and data shift registers. This is implemented through two similar control paths as shown in the Figure 2.9, which is marked (IR) for instruction registers and (DR) for data registers. Access of the registers involves scanning in data through TDI pin and to stream out data via TDO pin. A general and simplified JTAG system overview of how the state machine controller is interconnected with the registers are shown in Figure 2.10.

Table 2.2: Description of all the main pins used to implement the industry standard JTAG IEEE Std 1149.1 [2]

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.

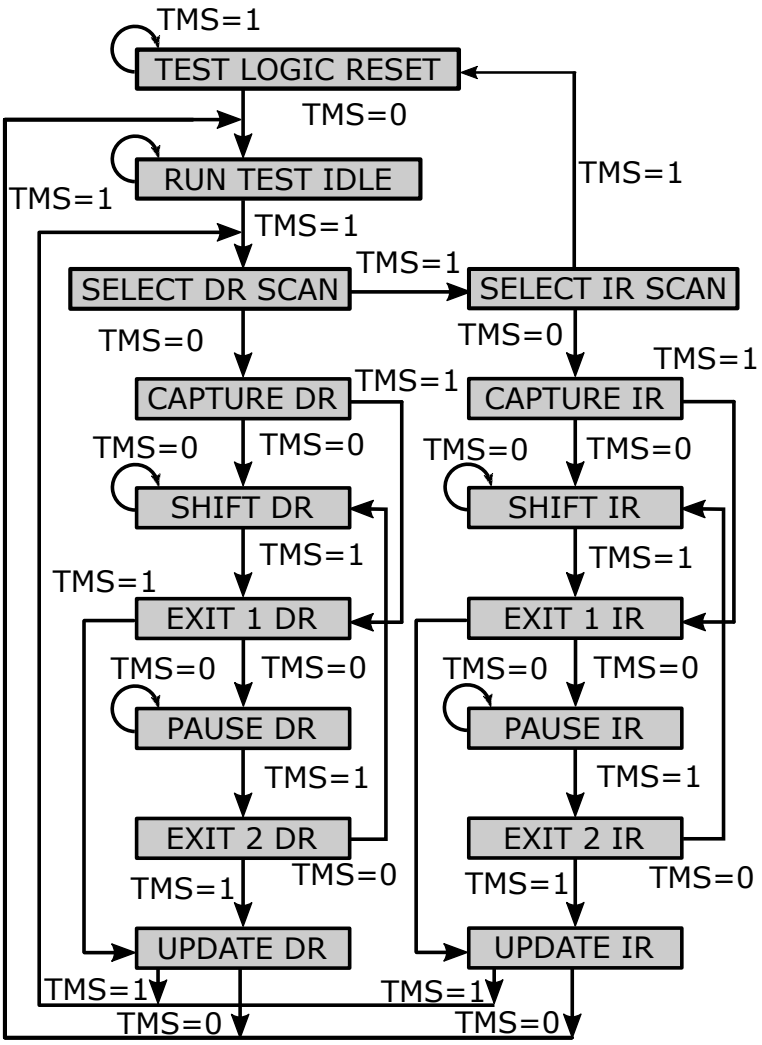

Figure 2.9: JTAG tap state machine used to control JTAG operations

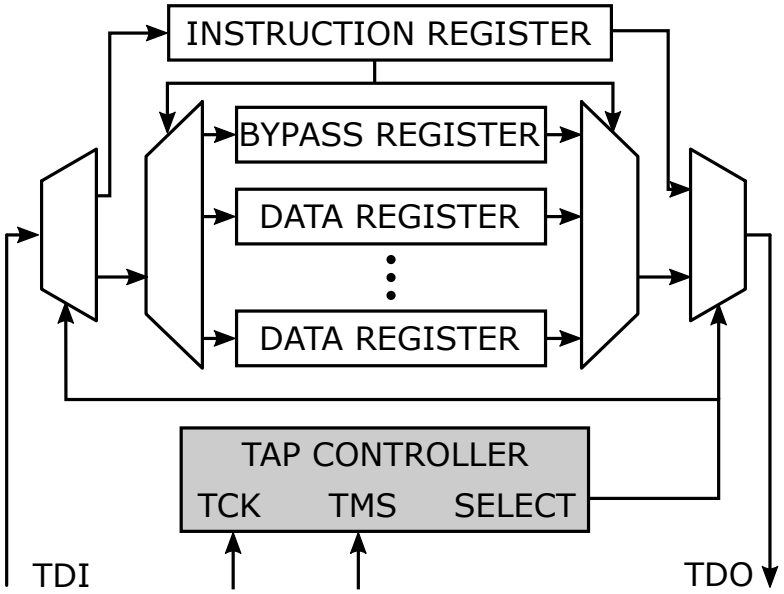

Figure 2.10: General simplified JTAG system overview

Chapter 3
JTAG interface

Basics of the JTAG tap controller and register configuration was recently presented in the background chapter. This chapter will go through the JTAG implementation where these basic structures have been used to create an interface to configure the system and perform memory operations.

3.1 JTAG implementation

The JTAG interface is used to externally configure the BIST system and perform memory operations. The different configurations and operation options are shown in the Table 3.1 and 3.2. The data register is shown in Table 3.3. These registers are implemented similarly as in Figure 2.10 with a bypass register to circumvent the JTAG overhead.

The JTAG registers and operations are controlled by the JTAG tap state machine implemented as in Figure 2.9. Registers include shift register where data are inserted serially through the TDI pin, a trigger flag indicates when all data have been inserted. This valid flag is high until either the registers have been reset or altered. Data from the registers can be parallel loaded to the components that use the data internally. However, the data is still loaded serially through the TDO due to the pin restriction.

3.2 JTAG burst implementation

A JTAG burst component was developed to increase the speed of memory operation via JTAG. This is especially used when writing the compiled program data to the instruction memory. The normal JTAG memory operations involve several steps to perform a write operation to a selected address. Initially, the JTAG tap state machine needs to get into the state where the instruction register is loaded to perform with the correct values to target registers. It should thereafter return to the initial idle state before the next step. The next step is once again to get the state machine into the state where the selected data register can be loaded with values. Tables 3.2 and 3.3 shows that these two data registers need to load values before a valid JTAG memory operation can be done. Register in 3.3 contains either

Table 3.1: JTAG configuration register 1 sub-field information

Configuration Register 1		
Register Section	**Section Length**	**Section Description**
ECC test	1 bit	Bit used to start the ECC bist test instead of the MBIST
Memory margin	Custom for memory size	Memory margin contains the information used to configure internal timing for memory components
End address	Memory address width	End address is used to specify the end of a memory section which is targeted by the test algorithm.
Start address	Memory address width	Start address is used to specify the start of a memory section which is targeted by the test algorithm.
Algorithm select	$\log_2$(Algorithms)	Algorithm select is used to choose one of the available test algorithms.
Overflow enable	1 bit	Overflow enable toggles whether the test should allow overflow of error logger or not. The test is stopped when overflow is detected and not allowed
Pause	1 bit	Used to pause running test.
Stop	1 bit	Used to stop running test.
Start	1 bit	Used to start selected test

Table 3.2: JTAG configuration register 2 sub-field information

Configuration Register 2		
Register Section	**Section Length**	**Section Description**
Burst enable	1 bit	Used to enable the memory burst read and write component
JTAG byte enable	$\frac{\text{data width}}{8}$	JTAG byte enable is used when a memory write operation is performed through JTAG to write to selected bytes.
JTAG write enable	1 bit	JTAG write enable is used to determine memory write or read operation.
JTAG address	Memory address width	JTAG address selects a memory address where read or write operations is performed.
Error log address	Error log address width	Selects one address in error log where read or write operation can be performed.
Error log override	2 bit	Used to select error log operation between three operations. Read error count, read error information, and write operation.
JTAG override	1 bit	Used by to override other components and enable memory read and write operations through JTAG.

Table 3.3: JTAG data section information

Data Register		
Register Section	**Section Length**	**Section Description**
JTAG data	Max(Memory data, error log data)	Register used to load data from read and write operations through JTAG. This is used both by memory and error logger.

data to be read or written to memory, while register in Table 3.2 selects operation and address. This register and state machine operation procedure contains a large amount of overhead.

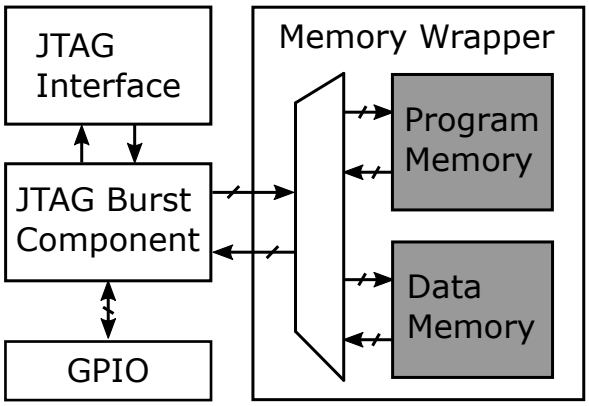

Figure 3.1: JTAG burst component integration with memory and JTAG controller.

The JTAG burst component targets the JTAG memory operation overhead, the proposed architecture is displayed in Figure 3.1. The main idea is to exploit that memory operations often are done in block sections. This enables that only a start and an end address of this block and the operation to be carried out needs to be specified once. This approach will eliminate the constant overhead work to update register in Table 3.2 where the target address and operations information are located. However, it still exists overhead operations to constantly update the data register.

During writing operations, this overhead can be reduced by the assumption that all data streamed in after the JTAG burst component has been configured to be considered as valid data in a write operation. All valid data can then be shifted into a register and then when it is full it can be written to memory. Read operations overhead can be reduced in a similar reversed fashion. A memory read operations can be performed to load the register directly. The data can thereafter be shifted out each cycle without any JTAG register configurations. These processes will be repeated until the selected operation has been applied to all addresses in the specified block.

The expected performance gain is dependent on the burst length and number of pins added to the design to input and output data. Other important factors are the cycles needed to configure the JTAG registers in Table 3.1, Table 3.2 and Table 3.3. The time required for normal JTAG mode to write data for different configurations is shown in equation 3.1. Length variable accounts the number of addresses in the burst length and the T variables represents the time needed to configure each register in the Table 3.1 (T_1), Table 3.2 (T_2) and Table 3.3 (T_{data}).

$$T_{normal} = length \cdot (T_2 + T_{data}) \tag{3.1}$$

$$T_{burst} = (T_1 + T_2) + length \cdot ceil(\frac{32}{pins}) \tag{3.2}$$

In equation 3.1, the total time cost for normal JTAG memory operations increases linear dependent on the length. Equation 3.2 shows a similar linear dependency on the length and an initial cost of the register configurations. The main performance difference when the length increases will be the factor that is repeated for every address as the initial one time cost impact decreases. By default, T_{data} is larger than the length factor due to the cost to control JTAG registers and insert the data. As the *pins* divider increases the performance differences increases.

Chapter 4

Built-in self-test and RISC-V implementation

The required general background information was previously presented to understand the memory problem components. This chapter will explain how the current memory BIST architecture was implemented and what problems it mainly targets.

4.1 BIST architecture

The memory BIST was implemented to be configurable and flexible for extensions. It should also be easy to use the BIST and tailor it to meet the requirements of a system. This specifically includes the ability to update different memory test algorithm to target certain memory faults. Important factors to consider are the memory specifications. The BIST should be able to easily be connected to memory with an arbitrary data width and address width. Other memory proprieties to consider is the operation delay of read and write. All these standard requirements are provided as a parameter to be configured according to the requirements.

Aspect to consider during architecture implementation is the critical paths. The BIST should preferably not be the limiting factor of the system clock speed. This is achieved by making sure that the combinational paths are kept as short as possible. Common techniques to achieve this is to use pipeline or to take advantage of the fact that some calculations can be done in an later stage. The BIST system overview can be seen in Figure 4.1.

4.1.1 Test controller

The test controller's main purpose is to control the test runs according to the input configuration provided through JTAG. This includes control signals such as start, stop, and pause of the selected algorithm test run. An overview of the different control signals can be seen in the Table 3.1. The start and end address of the test can also be specified via JTAG. This enables the possibilities to run a full memory test down to fine-grain specific memory address test. An advantage of this approach is that every error that can be covered by the test algorithms can be found and correctly stored in the error logger. If there exist more errors than

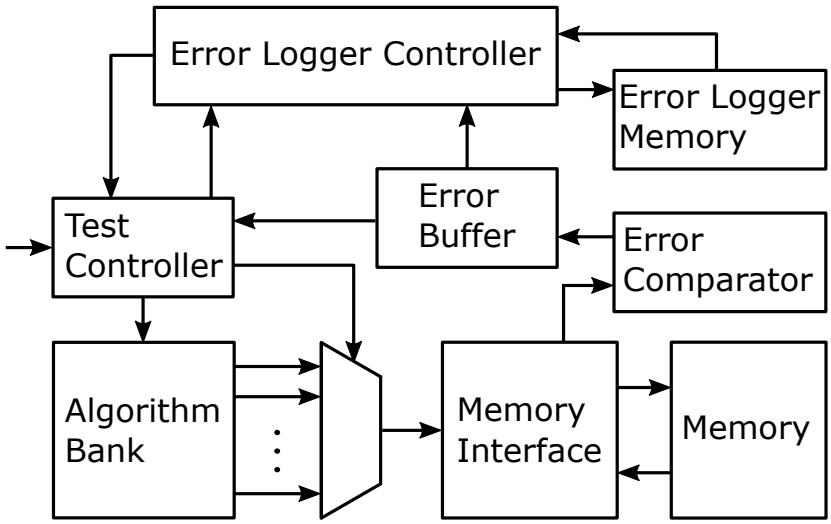

Figure 4.1: BIST implementation architecture overview

error logger memory could store, the detailed information regarding these excess errors would be lost due to memory overflow in the error logger memory. This is solved by running the algorithms on targeted sections of the memory.

The controller also incorporates status signals from other BIST components such as the error buffer and error logger controller. The Error buffer status signal includes the information regarding if the buffer is about to overflow or not. The test controller then uses this information to toggle the pause and start of the test run to avoid overflow and information loss. Signals between the test and error logger controller are mostly used to execute operations on the error logger memory and to configure the error logger.

4.1.2 Algorithm bank

An algorithm bank BIST architecture was presented and used in [10] with a demultiplexer to select between the generated memory sequences. This approach enables an easy extension of BIST with additional algorithms as new algorithms only have to be added to the bank and connected to the output demultiplexer and input multiplexer which selects between algorithms. The algorithm bank contains all the hardware of the implemented test algorithms. All implemented test algorithms are displayed in the Table 4.1. The fault coverage and the operation count of the algorithms are shown in Table 4.2. These algorithms were implemented due to the fault coverage of the basic and common errors in memories discussed in the theoretical background chapter. The algorithm bank can easily be extended with parameter configuration and implementation of the algorithm. Additional hardware costs beyond the implementation itself are the larger multiplexers used to select between the algorithm signals passed onto the memory interface. This extra deep multiplexers also add extra latency to a potential critical path to the memory

input. Another potential but small hardware cost is the additional flip-flop in the algorithm select register needed to select each algorithm.

In addition to the standard MARCH memory test algorithm, an enable test algorithms were developed to mainly target the byte enable faults (BEF) in certain memories. The byte enable test is based on the basic MATS algorithm with an extended write and read sequence to test each byte. It will only write a particular data set to an address where one byte is targeted at a time. The comparison stage will then check if the other bits in the word have been altered even though they were not targeted at a byte write. The byte read operation in Table 4.1 indicates the expected value of the targeted byte section, while the other bits are expected to have the opposite value.

Table 4.1: Implemented test algorithms stored in the algorithm bank and their operation sequence

Algorithm	**Write and Read Sequence**
MARCH MATS	$\{\updownarrow (w0); \updownarrow (r0, w1); \updownarrow (r1)\}$
MARCH C-	$\{\updownarrow (w0); \uparrow (r0, w1); \uparrow (r1, w0); \downarrow (r0, w1); \downarrow (r1, w0); \updownarrow (r0)\}$
BYTE ENABLE TEST	$\{\updownarrow (w0); \uparrow (r0, w1_{B0}); \uparrow (r1_{B0}, w0_{B0}, w1_{B1}); \uparrow (r1_{B1}, w0_{B1}, w1_{B2}); \uparrow (r1_{B2}, w0_{B2}, w1_{B3}); \uparrow (r1_{B3}); \updownarrow (w1); \downarrow (r1, w0_{B0}); \downarrow (r0_{B0}, w1_{B0}, w0_{B1}); \downarrow (r0_{B1}, w1_{B1}, w0_{B2}); \downarrow (r0_{B2}, w1_{B2}, w0_{B3}); \downarrow (r0_{B3})\}$

Table 4.2: Implemented test algorithms fault coverage and operation count. N is the number of memory addresses

Algorithm	**Fault Coverage**					**Operation Count**
	SAF	**AF**	**TF**	**CP**	**BEF**	
MARCH MATS	**X**	**X**				4n
MARCH C-	**X**	**X**	**X**	**X**		10n
BYTE ENABLE TEST	**X**	**X**			**X**	26n

4.1.3 Memory interface

Memory interface components contain a combinational connection from the algorithm block multiplexer directly towards the memory wrapper. Logic in this stage is kept as small and fast as possible to have low delay to ease memory timing closure. The only combinational logic in the data path is a small multiplexer to enable memory operations through JTAG.

The component also includes a configurable delay chain that is used to pass on values to the error comparator from the algorithm bank. Information passed on to the error comparator are the data expected from the memory read operation, address, and byte enable. If the memory has a read latency of 1 clock cycle, all values need to be delayed by the same time.

4.1.4 Memory wrapper

Memory wrapper is implemented in a generic manner to ease the integration process for different configurations. A single IP memory of arbitrary size can be integrated into the wrapper and used as a standard block to create larger memories if required.

The wrapper structure consists of memory instances and the internal control logic. Control logic component is bidirectionally connected to each of the memory instances. This connection involves all the required signals to perform memory operations. All memory instances added together create a larger memory to cover the configured address space. The control logic will use a part of that address space to determine which one of the memory instances to be selected and the other part is used internally by the memory instance to select a word. Figure 4.2 shows an overview of the memory wrapper.

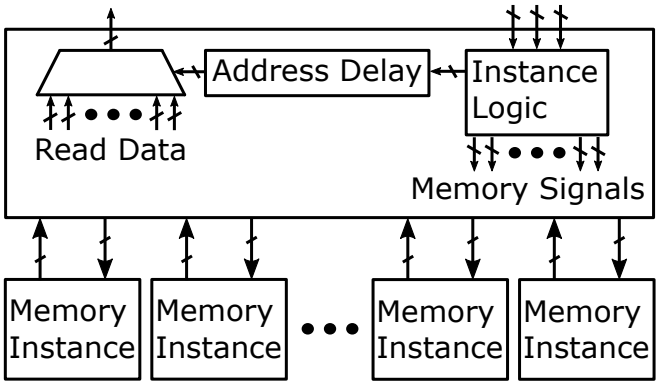

Figure 4.2: Memory wrapper structure with memory instances connected to the control logic

A sequential BIST architecture proposed in [11] and a parallel BIST architecture

proposed was proposed in [12]. The parallel architecture has the main benefit of testing multiple memory instances at once which improves testing time significantly. This requires a more complex system with more hardware which results in more area and power. The sequential architecture was proposed in [11] as the more area-efficient architecture as the hardware could be shared between the instances. As area and power are more of a concern the sequential approach was selected instead of the high test time performance parallel architecture.

4.1.5 Error comparator

The error comparator is a pure combinational component with the main task to identify if there exists an error in the memory read data. Its input used to perform comparison are the memory read data and the algorithm generated expected data at the targeted address. The algorithm generated data is reduced down to only 1 reference bit instead of its data width. This compression has some beneficial consequences such as the reduced hardware and power usage in its data path. Hardware savings such as fewer data wires, the smaller multiplexer in the algorithm block, and fewer flip-flops in the memory interface delay chain. The cost of this increases complexity in the error comparator where the complete data width has to be generated. Generation of the data array is based on the byte enable and reference data. This can be constructed in a basic structure as in Figure 4.3. When byte enable is low it will set each bit in the byte to the reference data. When byte enable is high it will set each bit in the byte to the inverse of reference data. This approach adds a small stages of logic levels keeping the combinational propagation delay as low as possible after the memory. The propagation delay of this stage was of concern during implementation because of all the combinational logic depth from the memory read path.

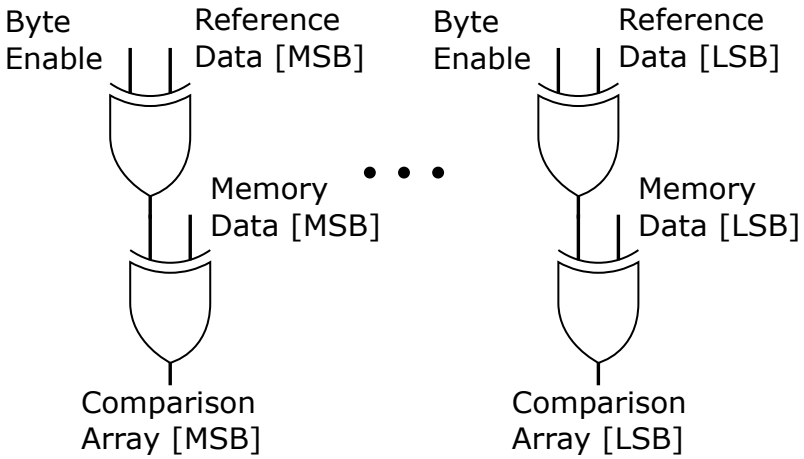

Figure 4.3: Generation of comparison data with two layers of XOR gates when byte enable is active low.

These two data sets are then compared with another array of XOR gates that will output a result array with the same width. The resulting array will only

contain zeros if the two compared data sets are similar. Any non zero result array indicates an error has been detected. As a result of the detected error, it will then raise an error flag. The error flag will be sent to both the error buffer and as a system output from the BIST. The error status flag will be used as a write enable to the error buffer. The data array provided by the error comparator to the error buffer will contain the memory address, an error code, and the comparison result.

4.1.6 Error buffer

All components previously mentioned in the BIST can operate at a speed of a single clock cycle. However, the error logger controller performs multi-cycle evaluation of the detected errors. These two domains with different throughput speed can operate smoothly if errors occur with at least the error logger latency apart. However, if the errors occur faster than this latency, the system needs to pause and wait for the error logger controller to finish its evaluation. The consequence of this is a lower test speed. Another problem with this pause approach is that test operations are already in the pipeline about to execute. The information after the pause, therefore, needs to be stored or to run at a lower speed not to lose any information.

The error buffer added to counteract the speed penalty related to detected errors within the error logger controller latency. The implementation is based on a circular buffer displayed in Figure 4.4, with its corresponding, read and write buffer pointers. Initially both read and write pointer is located at the origin address when the buffer is empty. When the write enable signals from the error comparator have a rising edge the data will be stored at the write pointer location. Write pointer buffer address will then be incremented to point at the next address. The read pointer points at the address which is currently under evaluation by the error logger controller. When the evaluation of the buffer data has been finished, it will be incremented to point at the next address. A system pause will only occur if the write pointer increases to fast and overflow to eventually catch up with the read pointer. The pause signal will be active until the read pointer increments. The buffer is considered to be empty when the read pointer catches up with the write pointer and the error logger controller signals that the data have proceeded.

The different data section width is configurable through parameters to ease integration in an arbitrary system. The size of the buffer itself is also configurable as a parameter. A trade-off has to be made between the hardware cost of the buffer size and the potential speedup possible.

4.1.7 Error logger controller

The error logger controller main tasks are to provide memory error information for further analysis. Information to be extracted is the memory address, the position of the most significant bit (MSB) error in the data array, the number of errors and the error code. However, the controller will initially iterate through the used space in the error memory to survey if the error address already have been stored.

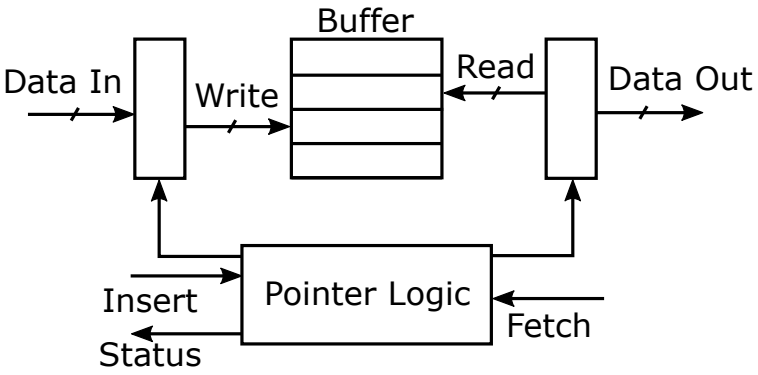

Figure 4.4: Circular buffer with read and write pointers

This is to avoid multiples of similar error information as the algorithm will write and read to each address several times.

Error logger controller uses the data comparison array provided by the error buffer. The initial step taken is the error count from the comparison. The previous XOR comparison generates a value 1 at every bit where a detected miss-match occurred. The number of errors in that memory read is therefore equal to the sum of bits with the value 1. To calculate the sum of these bits, a chain of full adders is used to perform an addition between all bits. The output of this chain is then a $log_2(N)$ bit array with the sum. The next step in the error logger is to calculate the position of the MSB 1 which is obtained with an integrated priority encoder. The generated error code from earlier stages is passed onto the memory.

A priority encoder is a circuity that has an arbitrary N bit input array which converts down to a $log_2(N)$ bit binary array. This array contains the position of the leading 1 bit. When a high priority 1 bit in the input array is detected all other input values are neglected. A truth table of a 4 input priority encoder is displayed in Table 4.3. Only inputs arrays larger than 0 are considered as valid inputs. Figure 4.5 contains the circuit of truth Table 4.3.

Table 4.3: 4 input priority encoder truth table. X is used to indicate neglected do not care bits. Higher number in input indicates higher priority.

I3	I2	I1	I0	O1	O0	Valid
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

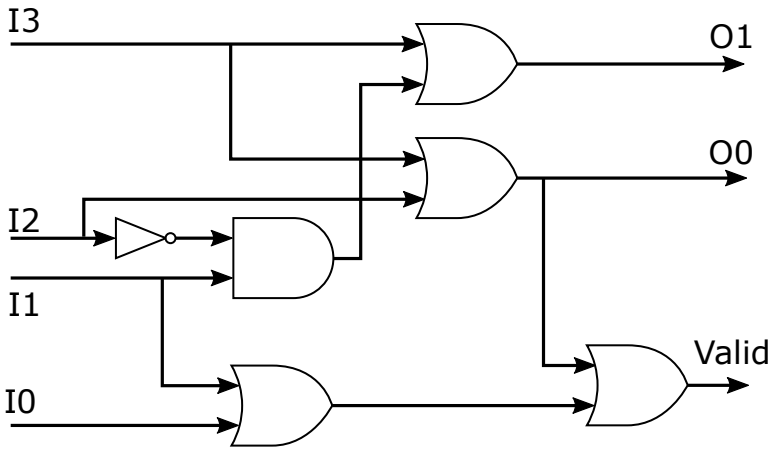

Figure 4.5: 4 input priority encoder circuit

The controller enables read and write operations via JTAG which is passed on from the test controller. Two different read operations can be done, one to read the total amount of errors registered and the other to read an address in the error memory. The write operation is simplified to write either an array with only 0 or 1 bits.

The algorithm test will be stopped when the controller detects an overflow in the error memory is about to occur as a default setting. The overflow of the error memory can also be toggled through the controller to enable the possibility of counting all the errors in the memory without stopping. This is done through the JTAG configuration passed on from the main test controller. When all data in the buffer have been processed it will send a done signal to the main test controller to make it aware of the event. An overview of the error logger is seen in Figure 4.6.

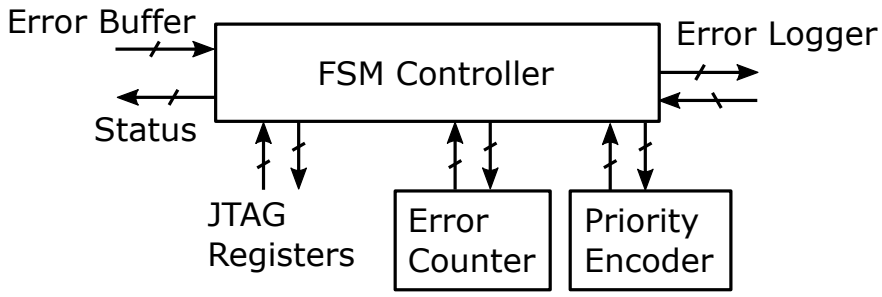

Figure 4.6: Error logger overview

4.1.8 Error logger memory

The error logger memory contains configurable storage of the final error results. It mainly consists of the error log register where each address contains the data sections in Table 4.4. It also includes an error counter. This counter is only used to keep track of the total number of errors found which is especially of interest then error overflow is enabled via the control signals. An overview of the error logger is seen in Figure 4.7

Table 4.4: Error logger data sub-field

Error log register		
Register Section	**Sub-filed Length**	**Sub-field Description**
Memory address	Memory address width	Contains the address information where in the main memory where a fault was detected
Error position	$\log_2$(Data width)	Information where the least significant fault was detected target address
Error count	$\log_2$(Data width)	Sum of all errors detected on target address
Error code	$\log_2$(Algorithms)	Contains information about the algorithm used in the test

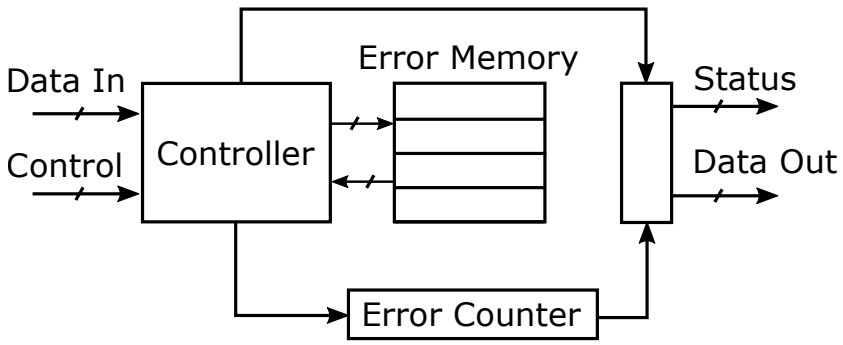

Figure 4.7: Error logger overview

4.1.9 Memory error models

A memory error model was created during the verification of the BIST functionality to detect and process the errors. These error models were implemented in the memory hardware description language (HDL) as a configurable addition. The first errors models implemented was the bit-cell transition failure. The result of a transition failure is that the current value on a bit is stuck at the current value and when a write operation is performed, it will remain the same.

The faulty coupling effect between the bit-cells was also modeled. The error model is implemented in a way that during a write operation to a certain row it would affect the value of another row. The interference between rows is fully configurable and did always occur when one of the addresses were targeted. The implementation would equate as if the address decoder would target both rows during the write operation.

Address decoder fault is also included in the error models. This error model maps the input address to the address decoder to a different address in the memory macro.

4.1.10 Memory test redundancy

The testability of the memory is considered a key feature of the SoC as mentioned before. Therefore, different precautions was taken to counteract the effect of potential manufacturing defects on this functionality.

A critical error would be if the JTAG were to fail. This would results in a non-usable MBIST, because of the required configuration and control signals can not be set. A set of default configurations are therefore hardcoded and a top-level start pin added to enable the test. This would trigger a full memory test with the MARCH-C algorithm. Detected errors would only be visible at an error pin and thus all detailed error information is lost. Only the address of the detected errors can be determined by the time between the start and the error found.

Another critical error would be if the MBIST itself would fail. A memory test could then only be done through the JTAG or the JTAG burst interface. The test time needed would increase and all read data processing has to be done externally.

4.2 ECC and BIST architecture

An ECC interface between the RISC-V core and the memory system is used to increase the systems fault tolerance against potential memory errors. The ECC component is responsible for correct decoding and encoding the data transmitted for all memory operations. Correct functionality is vital for a properly working system. A test component has therefore been developed to verify that basic ECC functionality is fulfilled. The test system needs to override the processor core sig-

nals to the ECC modules when test stream data are applied. A system view of how the ECC tester shares the ECC modules with the RISC-V core is displayed in Figure 4.8. ECC tester controls the muxing between the processor and test data while active. The proposed internal architecture of the ECC bist component can be seen in Figure 4.9.

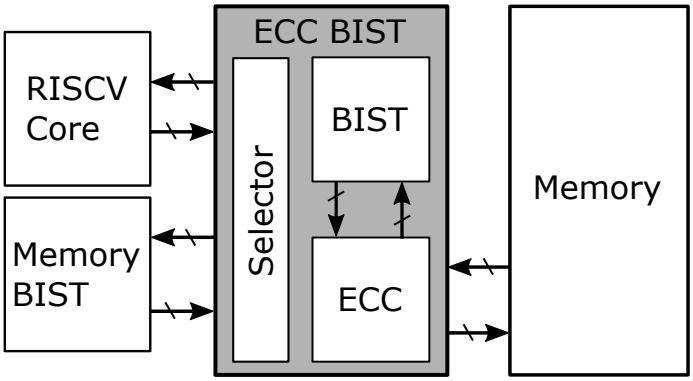

Figure 4.8: ECC top level implementation architecture overview

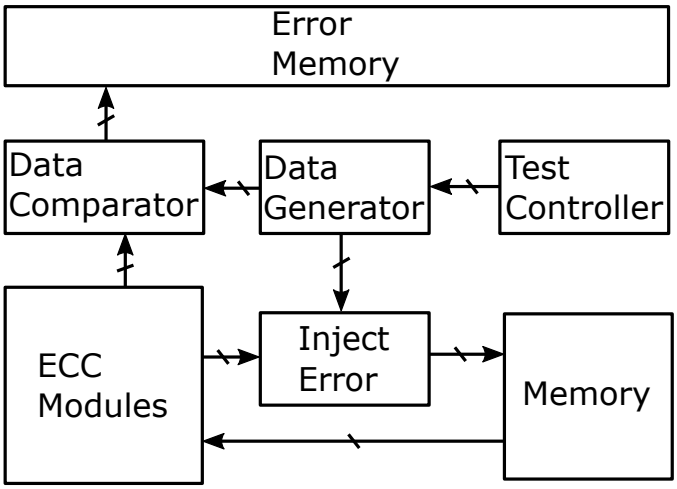

Figure 4.9: ECC implementation architecture overview

4.2.1 Test controller

The test controller is controlled through the JTAG registers similar to the MBIST system. The control signals required are the start address, end address and the

test enable. A control FSM will generate controls signals to the data generator, error injector, data comparator, and finally the memory signals. There are three different tests that will be used to make sure the ECC operates correctly. The initial phase is to mimic normal operations where no faults are inserted. In the second phase, it will insert an error to the initial bit. The last test inserts two errors to the test data. This procedure is repeated on the entire memory section which is selected through JTAG as mentioned before.

4.2.2 Data generator and error injection

Data generator stores a set of hard-coded default values that will be used as test data. Each address increment from the test controller will select a new value to be written. When all values in the data generator have been used it will loop and start to use the first one again. The encoded data is then passed onto the error injection component. The error injector uses a control signal from the test controller to determine which test phase its in and then inject errors accordingly.

4.2.3 Data comparator and error memory

The data comparator main task is to make a comparison of the decoded data from the ECC modules to match the expected data in the test and if not raise an error flag. It has inputs from the test controller, data generator, and the ECC modules. The test controller communicates which test phase is active. This signal includes the expected error pattern injected. The data generator provides the data used as the expected reference data. The ECC modules provide the decoded data to be compared. It also generates a data width signal with a one-hot encoding of the detected error positions. All these signals are then used to perform a comparison between the decoded data and expected data. A comparison of the decoded and expected error position location is also made to verify functionality. An error signal will turn high if any of these comparisons fail. This error signal is passed on to the error memory as a write enable. The information stored in the error memory is the test phase and which comparison failed. This information can then be extracted through JTAG in a similar procedure as in the MBIST error memory.

4.3 RISC-V core integration

This section presents the RISC-V based processing unit of the system which will be implemented in the core wrapper. The wrapper includes the embedded core and its interface to the data memory, instruction memory, and the ECC component multiplexer toggle for redundancy.

4.3.1 RISC-V core

The RISC-V homepage displays available hardware [13] where several processor cores and complete SoC platforms are displayed. Many of them are under company commercial licenses but there also exist several open-source cores. This list was used to select a suitable core to be integrated into the system. Important specifications that the core had to full-fill were a 32-bit core with an open-source license that enables modifications and usage. The requirement for the RISC-V ISA extensions was the use of the RV32I instruction base with the support of integer multiplication and atom instructions. Other important factors for the selection are silicon-proven design with a supported tool-chain as possible.

The selected core was the formerl RI5CY [14] which now goes under the new name cv32e40p. RI5CY originated from the PULP platform where it has been one of the standard base cores in several different architectures. The original core has also been integrated and used in several tape-outs [15]. The supported instruction set is the RV32IM[F]C which covers the previously mentioned RISC-V ISA requirements. The floating-point extension is optional as the hardware is configurable to support it or not. The extensions and their supported functionality description can be seen in Table 2.1. The processor core is made of a four-stage pipeline which consists of the instruction fetch (IF), instruction decode (ID), execution stage (EX), and the write back stage (WB). Pipeline forwarding is also supported to decrease the pipeline stalls negative effect on the performance. The core architecture and the most important component blocks can be seen in Figure 4.11.

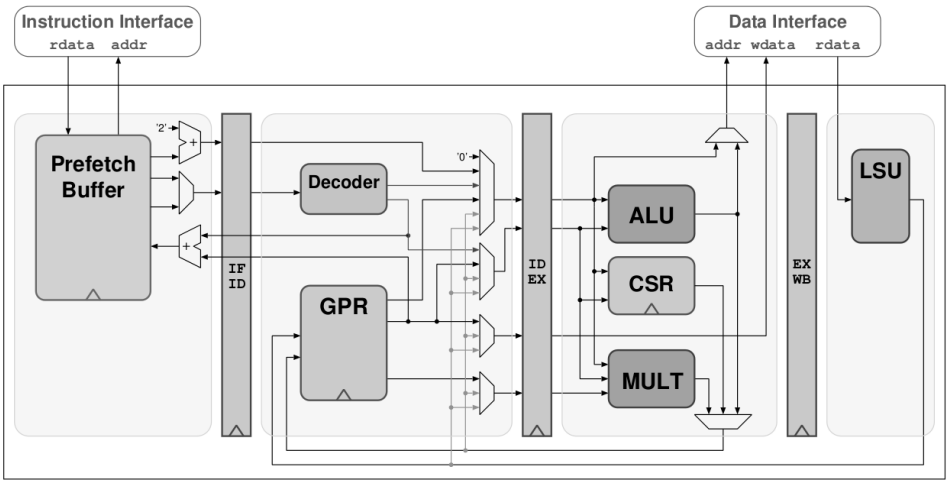

Figure 4.10: Cv32e40p RISC-V core image from [1]

The core has two components which are connected to an external system. The first stage primarily consists of the instruction fetch buffer whose main task is to provide one instruction to the ID stage per clock cycle from the instruction mem-

ory. A potential bottleneck might also be how fast the instruction interface can provide the instructions. The other interface components are the load and store unit which manages the access of the data memory. Access includes executions results writes and data reads. Access can be configured to support 32-bit words, half-words, and single bytes.

Both single- and multi-cycle instructions are supported by the core. The range of allowed cycle instructions is between 1 and 32. Data hazards and load data hazard will result in a single cycle penalty. The data hazard penalty occurs when a jump register instruction depends on the result of an immediately preceding instruction. The load data hazard penalty occurs when the instruction immediately follows a load that used the result of that load.

The core incorporates several control and status registers (CSR). These also have performance counters included and the location of these registers is in the execution stage CSR component. Performance counters will mainly be used to extract executions times of the benchmarks. The utilized performance counter is called mcycles which consists of two 32 bit register where the number of cycles are stored since it was active. Time measurements can be conducted by an initial read of mcycle when the timer starts. The program is finished by a final mcycle read where the execution time is the difference between the values.

4.3.2 RISC-V core interface

Figure 4.11 displays memory interface used by the core. The instruction interface is between the prefetch buffer and the instruction memory while the data interface is between the load and store unit and data memory. The instruction interface is limited to only perform read operations while the data interface can both execute read and write operations. All signals used by the interface are visible in Table 4.5. All signals used to perform write operations are not used by the prefetch buffer because it only performs read operations. All these signals are not standard ports used by the available standard IP and thus the handshake procedure between the core interface and memory is embedded into the core wrapper. When a memory operation from the core is instantiated it will be guaranteed direct access and control of the memory. Therefore, a constant clock cycle delay between the core and memory operations can be used to control the handshake procedure where it is assumed that the correct data will be in place at the correct timing. This handshake procedure involves the signals data_req_o, data_rvalid_i, and data_gnt_i.

The waveforms timing diagram in Figure 4.11 shows how a memory transaction and multiple back to back memory operations are executed via the load-store unit. During operation, the core sets the signal output data_req_o to high when it requests new data to be loaded or written. At the same time, it specifies the targeted address, output data, byte enable, and write enable signals. The memory wrappers will then set data_gnt_i to high when it is ready to serve the operation request. When the request has been approved to be served new values for another memory operations can be set without affecting the previous. When the memory has pro-

Table 4.5: RISC-V core memory interface signals

Signal Name	Port Direction	Signal Description
data_gnt_i	Input	Used by the other component to grant memory operation request.
data_req_o	Output	Single bit used to indicate valid request. Stays high until data_gnt is high for one cycle
data_addr_o	Output	Memory target address
data_we_o	Output	Write enable high for write and low for read operation. Combined with data_req_o.
data_be_o	Output	Byte enable high during write and low during read operations
data_wdata_o	Output	Data to be written to memory, sent together with data_req_o
data_rdata_i	Input	Data retrieved from memory read operation
data_rvalid_i	Input	Used to indicate if read data is valid. High during one cycle per request.

cessed the operation it will provide read data at signal data_rdata_i and flag that the operation finished by setting data_rvalid_i to high. Signal data_rvalid_i is also set to high to indicate a write operation have been executed as well.

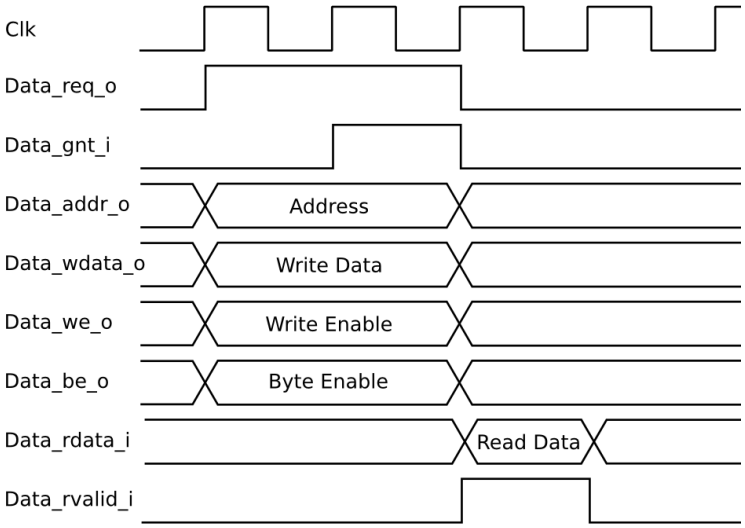

Figure 4.11: Cv32e40p RISC-V memory interface operation

4.3.3 System overview

The system architecture overview with the integrated RISC-V core is seen in Figure 4.12. The thick grey lines represent the configuration signals from the JTAG interface to the sub-modules. All selector components are configurable multiplexers, there exist two modes, test mode and normal operation.

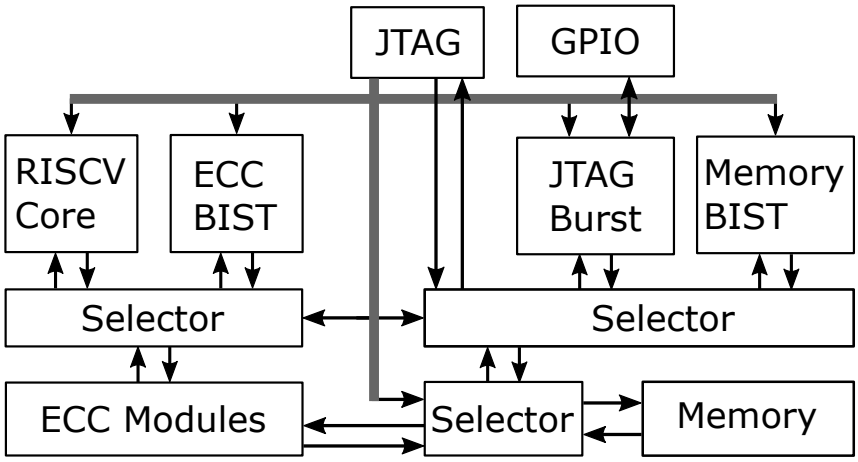

Figure 4.12: System architecture overview

Chapter 5
Result and Verification

This chapter presents post synthesis results and simulations if not stated otherwise used to perform functionality verification and measurements regarding area, power and frequency. Parts of the design has also been sent for fabrication. Memory and digital logic are within the same power and clock domain. The corner used during the results are typical typical under 25 degree temperature for standard threshold voltage transistors.

5.1 JTAG burst

Table 5.1 display the performance comparison between the speed of performing memory read or write operation for 100 memory addresses where the data width was 48 bits. Memory operations performed through normal JTAG procedure required 16400 cycles on 100 different addresses. This number will be used to calculate the performance improvement. The amount of clock cycles used to configure the burst component required 117 cycles. This time cost is fixed for all burst operations. The measurement was taken from the first cycle when the serial JTAG data was shifted onto the input. The end of measurement was when the JTAG tap returned to the idle state and done with the operation.

The addiditon of a single pin

Table 5.1: Comparison of burst memory operation and normal JTAG performance

GPIO PINS	Clock Cycles	Speedup
1	4917	3.34
2	2517	6.52
3	1717	9.55
4	1317	12.45
5	1077	15.23
10	597	27.47

5.2 Memory BIST

The MBIST system configuration used during this verification chapter is a 48 bit memory wrapper constructed from a 32 bit and 16 bit memory. The total memory address width is 16 bits including the instruction and data memory which shares the MBIST.

Figure 5.1 show the basic MARCH MATS algorithm generated test data to the memory over the address range 4 to 5. In the figure it can be seen that it initially writes 0 to all targeted addresses. A write is performed when signal Write_Enable is high and the data written is displayed in the Write_Data signal. A write requires one clock cycle to be performed which can be seen at the internal memory signals named Memory that latch the written value one cycle after each performed write. The state machine generates the algorithm data signals when the entire test procedure is done with the Test_End signal at the bottom of the waveform.

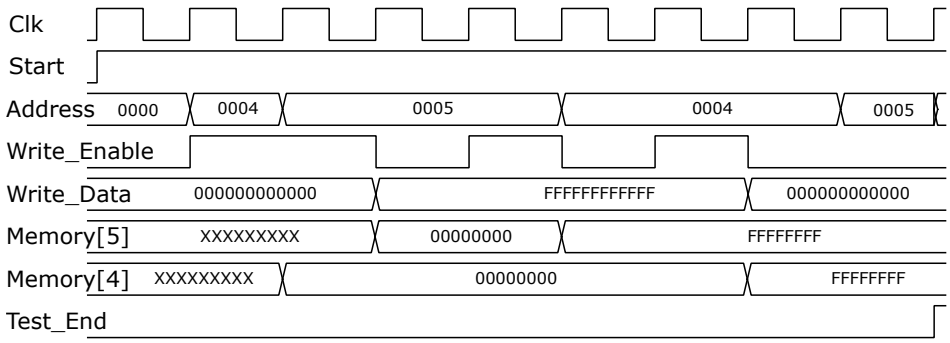

Figure 5.1: BIST default generated memory signals from a run of MARCH MATS algorithm on two addresses

Figure 5.2 display the data comparison stage from the algorithm generated in Figure 5.1. No errors have been inserted during this test and thus it should not give any error indications. The algorithm provides the comparator the expected data which is generated from the signals Reference_Data and Bit_Enable. Low value at signal Bit_Enable will make the corresponding data used in the comparison the same as the Reference_Data. Value 1 will instead use the inverted value during comparison. The comparison is only made when the Compare_Active signals is high. As seen at the Error_Found signals, no errors are detected as expected in this non faulty simulation. The spike at Error_Found is a combinational glitch when Reference_Data changes value.

Figure 5.3 displays test data generated from the MARCH_C algorithm where it targets address 4 to 6. However an address decoder error has been inserted into this model. As seen the address decoder targets address 4 when the MBIST selects both address 4 and 5 during memory operations. The write to address 5 will therefore write to address 4 instead which can be seen in the figure where value 1

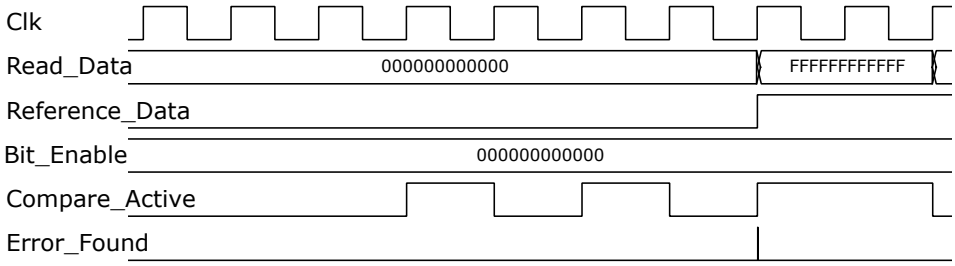

Figure 5.2: BIST default comparison of memory signals and reference data

is written to all bits at Memory[4]. This error results in that the Memory[5] value remains unknown.

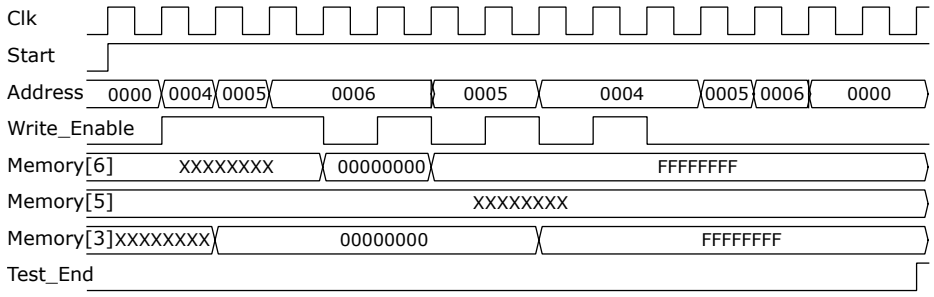

Figure 5.3: BIST default generated memory signals from a run of MARCH C algorithm on three addresses

Figure 5.4 show the data comparison where an error is found due to the inserted error. As seen in the figure, the error is found during the last comparison of address 4 where the algorithm expect value 0. However, when the algorithm wrote values of 1 to address 5 it overwrote value 0 at address 4 due to the address decoder error. When the algorithm then reads values of address 4 it will detect the error.

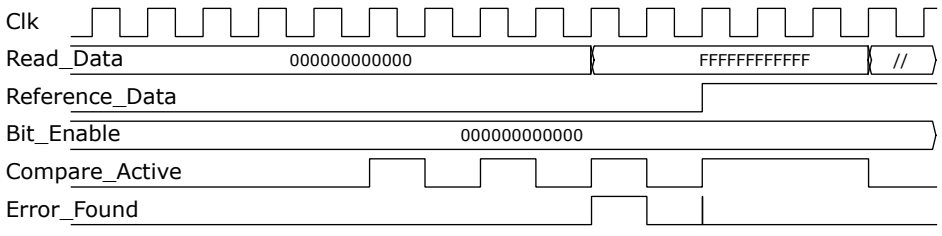

Figure 5.4: BIST comparison of memory signals and reference data with an address decoder fault

5.3 ECC BIST

The ECC BIST system configuration used in this verification is based on a 32 bit system. It can correct up to two errors and require 42 bits to detect and correct these errors. Figure 5.5 shows the test controller and the generated test data. In a test there are three different test phases. These can be seen in the Inject_Error signal under the active high Test_Active signal. The Data_In signal displays the generate data that have been encoded by the ECC that is written to memory. Signal Error_Data is the output from the error injector component. As seen during phase 0 both Data_In and Error_Data both are the same. However during phase 1 the initial bit has been inverted and during the second phase the first two bits has been inverted to inject errors.

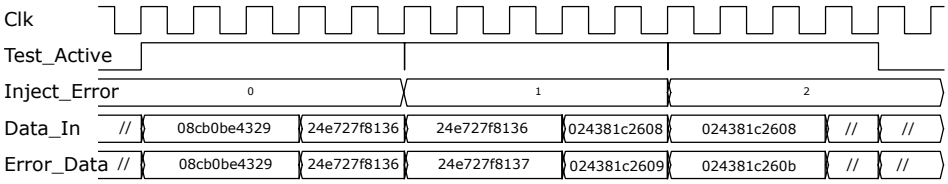

Figure 5.5: ECC BIST test data generation

Figure 5.6 display the comparator unit when the ECC modules work as expected. The reference inputs will be the expected data and error positions while the ECC modules outputs will provided the decoded data and decoded error position. As seen in the figure there are only glitches on the Error_Detected signal during the comparsions and no fault is found as both data and error positions match during the comparsion when Compare_Active is high.

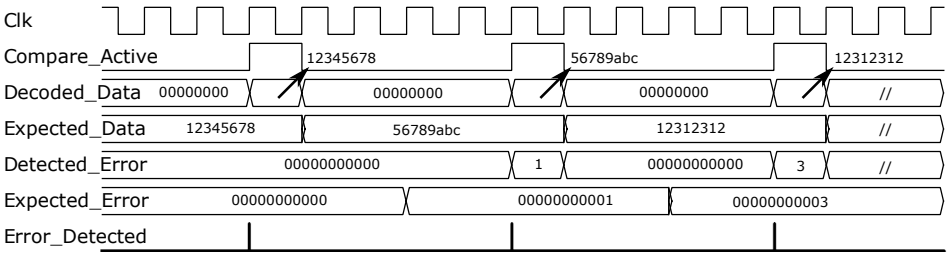

Figure 5.6: ECC BIST test reference and data comparison

In Figure 5.7 faulty ECC modules has been inserted to test the functionality of the ECC tester. In the phase 0 an error was inserted at the first bit. The ECC managed to correct the data but fails to provided correct error position. During the second phase it can be seen that the ECC provides correct decoded error

position while the decoded data does not match the expected data. In the final phase, it fails to correct two errors inserted to the data but still manages to provide the error position. The signal Error_Detected goes active high and all errors are stored into the error logger.

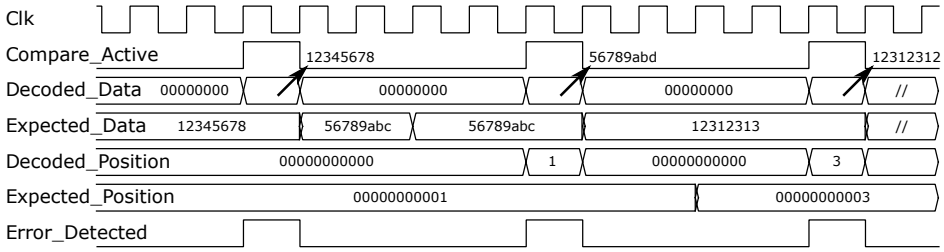

Figure 5.7: ECC BIST test reference and data comparison with an errors inserted

5.4 System area

The synthesis area results was used to calculate gate count of the entire system and important sub-block components. Gate count metric is calculated from the provided cell area of the target circuit divided by the area of the smallest NAND gate in the standard cell library. The RISC-V core was implemented using latch based register instead of flip flop based. This enabled area savings because of the latch standard cell smaller layout than its counterpart flip flop.

Figure 5.8 shows the area for the top level system blocks excluding the memory. As seen the RISC-V core occupies the majority of the area as it stands for over 70% of the total. The second largest block considering area is the MBIST which occupies over 17% of the total area and the ECC bist and ECC modules occupies just over 4%. The JTAG interface utilise just under 5% of the total area.

Figure 5.9 shows the area block distribution for the major blocks within the MBITS component. As seen in the figure the two major area contributes are the circular buffer and error memory which occupies over 60% if the MBIST area. These are mainly small storage memories made out of flip-flops. The implemented circular buffer depth is 8 and the error logger have a depth of 16. The data width of each depth level in the error logger memory is determined as in Table 4.4. In this implementation is has a data width of 28 Algorithm block is the third largest area contributor with just over 16% .

The different processor pipeline stages area distribution are visible in Figure 5.10. The trend here is not that the internal storage elements CS registers occupy the largest area since its not implemented by standard cell flip flops but instead latches.

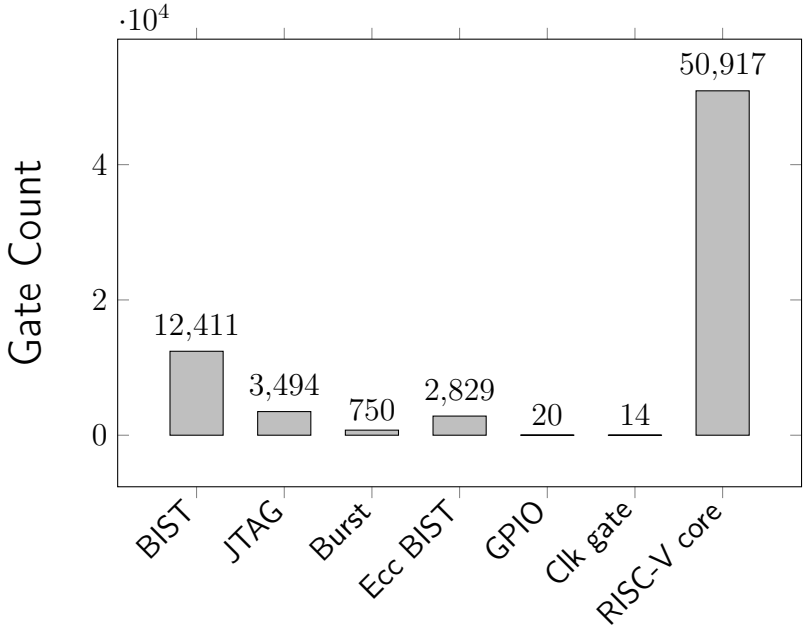

Figure 5.8: Top level gate count for the the complete SoC system.

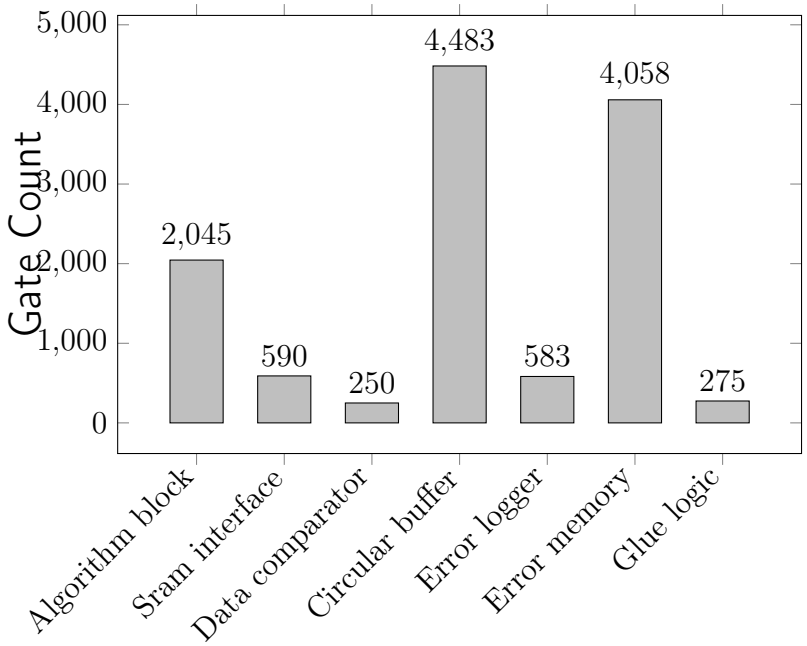

Figure 5.9: Gate count for the important blocks within the BIST system.

The two largest components here are the instruction decoder and the execution block of the processor.

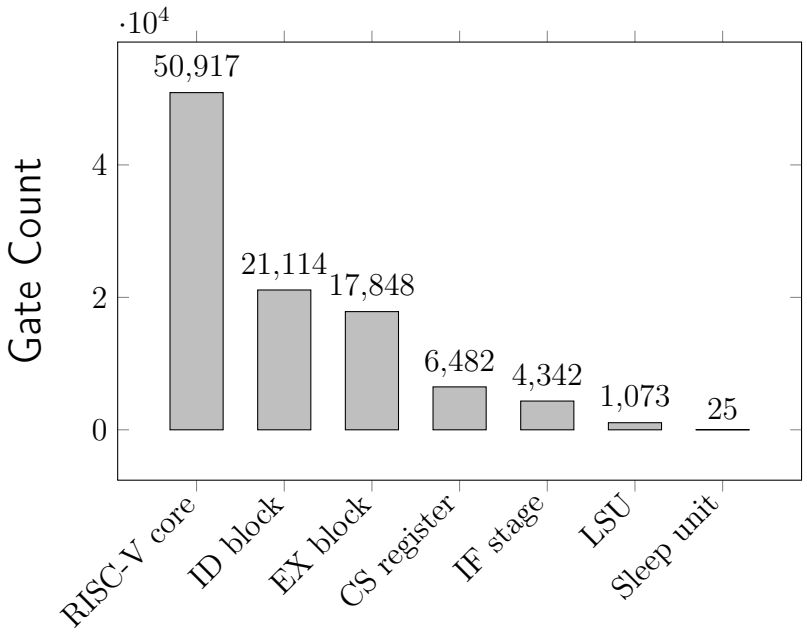

Figure 5.10: Gate count for the important blocks within the SoC system.

5.5 Frequency and power

The maximum operational frequency of the system for different voltages are displayed in the Figure 5.11. The frequency have been normalized with the smallest NAND gate propagation delay as reference at 1.0 V. The initial default supply are 1.2 V where it manages to run at a frequency equivalent of 305 gate depth. When the voltage supply is decreased with 200 mV the frequency drops by one third which compares to a gate depth of 454.3. Another 200 mV decrease in supply voltages decreases the frequency by another 50% to a gate depth of 940.3.

System critical path originate from the memory data outputs. From the output its routed through memory wrapper instance multiplexer and then its routed to the ECC modules. The decoded ECC data is then routed through another mul-

tiplexer which select if the ECC BIST or processor core should receive the data. The data is routed to the core where it first goes through the instruction decoder unit and then is stored in a sequential unit.

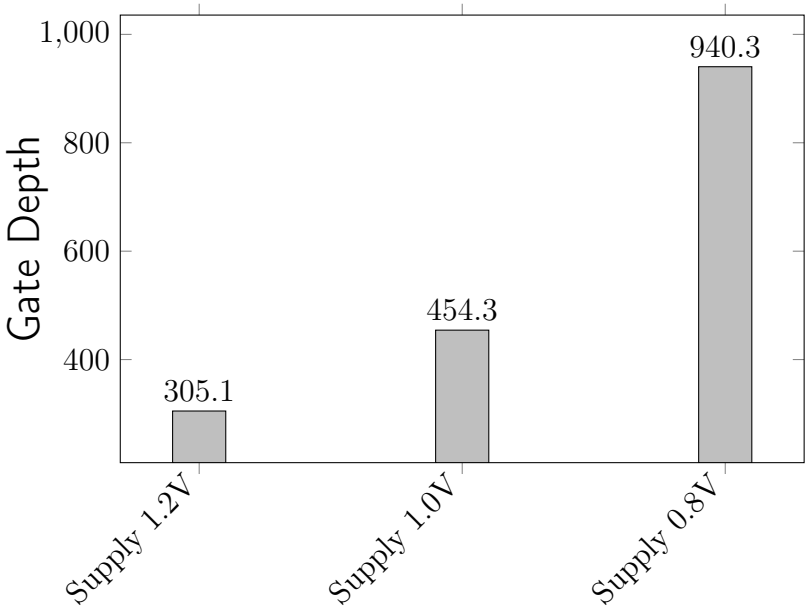

Figure 5.11: System maximum frequency for different supply voltages

In order to give a number on the BIST and ECC including the memory wrapper impact on the critical path of the RISC-V core a gate depth metric is used. The memory instance delay has been excluded from the calculations. Gate depth metric consist of the the number of gates placed in a chain that would result in a similar propagation delay. The base delay used to calculate the result is the propagation of the smallest NAND gate available in the pdk. The fastest time in the cell timing table which is generated during the ideal conditions with the least amount of load capacitance and fastest input slew rate was used. This resulted in a gate depth of 42.3 gates depth. The memory wrapper data output selection contributes with 12.5 gate depth and the ECC decoder with the 28.8 gate depth.

Table 5.2 display the leakage and the dynamic power of the two main block in the system. The power numbers of the smallest NAND gate at 1.0 V have been used to normalize the data. Power numbers are provided in the amount of equivalent NAND gate in the system that would result in the same power consumption. Leakage calculations use the NAND gate cell leakage and the dynamic calculations use the average NAND gate transition power. The numbers are extracted during

a normal MBIST check of 200 first addresses where the BIST does not find any memory errors. During the BIST operation the core is put in to sleep mode where the majority of the core is clock gated to lower dynamic power consumption. The power numbers of the active RISC-V core are generated from the synthesis where no specific program was executed to generate power numbers. The only non clock gated components in the BIST are the combinational path from memories through the ECC to the RISC-V core.

Table 5.2: Leakage and dynamic power numbers for different user cases

Component	**Voltage**	**Leakage**	**Dynamic, BIST active**	**Dynamic, RISC active**
BIST	0.8V	10831	721	Not Active
BIST	1.0V	17628	1097	Not Active
BIST	1.2V	34074	1354	Not Active
RISCV	0.8V	22439	7	12 630
RISCV	1.0V	31554	9	15 369
RISCV	1.2V	79878	16	30 995

The nomralaized power saving for executing the same BIST operations at different supply voltage are calculated by using the simulation time multiplied with the above power numbers for leakage and dynamic added together. Numbers have then been normalised with standard 1.2V supply as the reference 1. Results are displayed in Table 5.3.

Table 5.3: Power comparison between different supply voltage during a standard BIST test

Supply Voltage	**1.2V**	**1.0V**	**0.8V**
	1	0.78	0.54

Chapter 6
Conclusion

A fault-tolerant and energy-efficient RISC-V architecture has been proposed in this project. Key features of the system targets are the test-ability of the system and especially the memory instances. The system includes ECC components on both instruction and data memory connections to increase the fault-tolerant capabilities of the RISC-V core operations. An MBIST has been developed and integrated with the RISC-V core to test the memory. The JTAG interface is used to control the system and to perform memory operations. Recharacterization of the standard cells in the pdk was performed for the voltage scaling to look into the possible power savings that can be achieved.

The selected ECC managed to correct a maximum of 2 errors. It can indicate the rest of the system that corrected errors have been found and corrected. Positions of the error in memory are also available from the ECC modules. An ECC BIST has been developed and integrated to verify correct functionality. The ECC has the consequence of increasing the number of data bits from the system 32 bit to 42 bits. This cost results in increasing memory area and power that comes with more bits. The 42 bit is not a standard power of two numbers that are common for memory sizes. A 42 memory data width might not be available and force the system to utilize two memories to accommodate for the extra bits as used in this thesis where a 16-bit memory was added together with a 32-bit memory to accommodate all bits. This created 6 bits in overhead which create an unnecessary amount of power and area utilization from the memories.

The area impact of including the ECC modules and BIST can be seen in Figure 5.8. Compared to the integrated RISC-V core it increases the gate count by just above 5.5% where the ECC modules stands for 3.5%. The cost of adding the ECC compare was an addition of 28.8 gate depth to the current 12.5. This is an increment of 2.3 times the memory wrapper instance multiplexer in the critical path. The addition of the MBIST to test memories are another 24.4% of the core size. However, the circular buffer and error memory occupies the majority of the entire BIST area. These are mostly pure sequential storage units where the size is based on user constraints and debug requirements. In order to reduce the area impact these can be decreased, especially the circular buffer which can occupy 8 errors in this configuration. If it were to be reduced down to 2 errors it could result in area savings up to 27% of the total MBIST part.

As mentioned previously testing of the memories is considered a key feature. In case of potential manufacturing failure or design faults of different critical digital components the memory should still be able to be tested in some basic manner. Basic test in the meaning that it can detect if any errors are available in the memory. Therefore, different actions have been taken to increase the test-ability. In normal operation the MBIST is configured to target a specified memory section through JTAG with a selected algorithm. However, if the JTAG were to fail it would result in that no memory test can be performed. To counter this an extra pin was added to run default bist settings that target entire memory. Detected errors will be displayed at the error pin. If the BIST were to be faulty memory test can still be done through either normal JTAG operations or with the JTAG burst component.

Voltage scaling resulted in a possible power saving up to 46% when supply was reduced from 1.2 V to 0.8 V when the bist runs its normal test operations in Table 5.3. This comes at the cost of lower operation frequency, where the gate depth in critical path increases with lower supply voltage. The lower supply in the final implementation resulted in 3x slower operation frequency.

At the moment three test cases are included in the algorithm bank. Future work could extend the number of available tests to increase the fault coverage of the memory. A potential large extensions would also be to develop a BIST to test the RISC-V core itself in order to diagnose the key components in the system and to make sure that the system achieves correct functionality. Another topic of interest to be investigated further is the supply voltage selected in the voltage scaling. Mapping and characterization of memories and standard-cells with smaller voltage steps could be used to find the most operation efficient voltage for the system in the used technology.

References

[1] OpenHW group cv32e40p user manual. `https://core-v-docs-verif-strat.readthedocs.io/projects/cv32e40p_um/en/latest/`. Accessed: 2020-08-15.

[2] 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.

[3] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-threshold RISC-V core with DSP extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[4] S. Kaushik andY. Zorian. Embedded memory test and repair optimizes SoC yields. In *Synopsys, Mountain View, CA, USA, Tech. Rep., Jul.*, 2012.

[5] Semiconductor industry association (SIA), "international technology roadmap for semiconductors (ITRS)", 2003 edition.

[6] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, 1979.

[7] V. Chandra and R. Aitken. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. pages 114–122, 2008.

[8] J. Tonfat, J. R. Azambuja, G. Nazar, P. Rech, C. Frost, F. L. Kastensmidt, L. Carro, R. Reis, J. Benfica, F. Vargas, and E. Bezerra. Analyzing the influence of voltage scaling for soft errors in SRAM-based FPGAs. In *2013 14th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pages 1–5, 2013.

[9] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2013.

[10] T. Q. Bui, L. D. Pham, H. M. Nguyen, V. T. Nguyen, T. C. Le, and T. Hoang. An effective architecture of memory built-in self-test for wide range of SRAM. In *2016 International Conference on Advanced Computing and Applications (ACOMP)*, pages 121–124, 2016.

[11] R. Silveira, Q. Qureshi, and R. Zeli. Flexible architecture of memory BISTs. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6, 2018.

[12] Y. Huang, C. Chou, and J. Li. A low-cost built-in self-test scheme for an array of memories. In *2010 15th IEEE European Test Symposium*, pages 75–80, 2010.

[13] RISC-V homepage available hardware. `https://riscv.org/risc-v-cores/`. Accessed: 2020-05-27.

[14] PULP platform homepage. `https://pulp-platform.org/index.html`. Accessed: 2020-05-27.

[15] PULP platform project page. `http://iis-projects.ee.ethz.ch/index.php/PULP`. Accessed: 2020-05-27.

Printed by Tryckeriet i E-huset, Lund 2021

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-803
http://www.eit.lth.se