Extension of RISC V Bit Manipulation Instructions with Reliable Memory

Project ID: 833

8 Credits

CONTENTS

INTRODUCTION

Reasons for using a RISC V processor:

The RISC-V architecture based processor has an open source instruction set. It enables developers to develop a product that is tailored specifically to their workload, so they start with the RISC-V core and can add whatever it is they specifically need, saving both time and money.

Need for fault-tolerance on hardware:

A system is prone to errors due to unexpected noise or soft errors/flipping of bits as a result of being stored for a long duration in the memory. Adding Error Correction Coding corrects the errors that occur during transmission of signals or storage.

PROBLEM STATEMENT

To **design** and **verify** a 5 stage pipelined RISC V synthesizable processor with bit manipulation instructions extensions and a fault-tolerant memory using Verilog and SystemVerilog.

LITERATURE SURVEY

[1] B. Koppelmann, P. Adelt, W. Mueller and C. Scheytt, "RISC-V Extensions for Bit Manipulation Instructions" *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), Rhodes, Greece, 2019, pp. 41-48*.

- Reference for the bit manipulation instructions to be implemented

[2] Sahan Bandara, Alan Ehret, Donato Kava and Michel A. Kinsy, "BRISC-V: An Open-Source Architecture Design Space Exploration Toolbox"

- Toolbox for skeleton for RISC V Processor

[3] Tshagharyan, G., Gurgen Harutyunyan, S. Shoukourian, and Yervant Zorian. "Experimental study on Hamming and Hsiao codes in the context of embedded applications." In *2017 IEEE East-West Design & Test Symposium (EWDTS)*, pp. 1-4. IEEE, 2017.

[4] Tam, Simon. "Single error correction and double error detection." *Xilinx Application Note* 645 (2006): 1-12.

MOTIVATION

In hardware applications,performance and system efficiency are of the utmost importance. However, current RISC V processors lack a reliable memory as well as a robust Instruction Set to become efficient in industrial applications.

- Embedded systems require a high energy efficiency in combination with an optimized performance.
- Bit Manipulation Instructions (BMIs) were introduced for x86 and ARMv8 to improve the runtime efficiency and power dissipation of the compiled software for various applications.
- The current RISC V platform only supports two Bit Manipulation Instructions, which are insufficient for hardware applications.

PROCESSOR DESIGN

Core Design:

No of Cores	1
Address Bit Width	20
Data Bit Width	32
Main Memory Type	Synchronous
Pipeline Stages	5

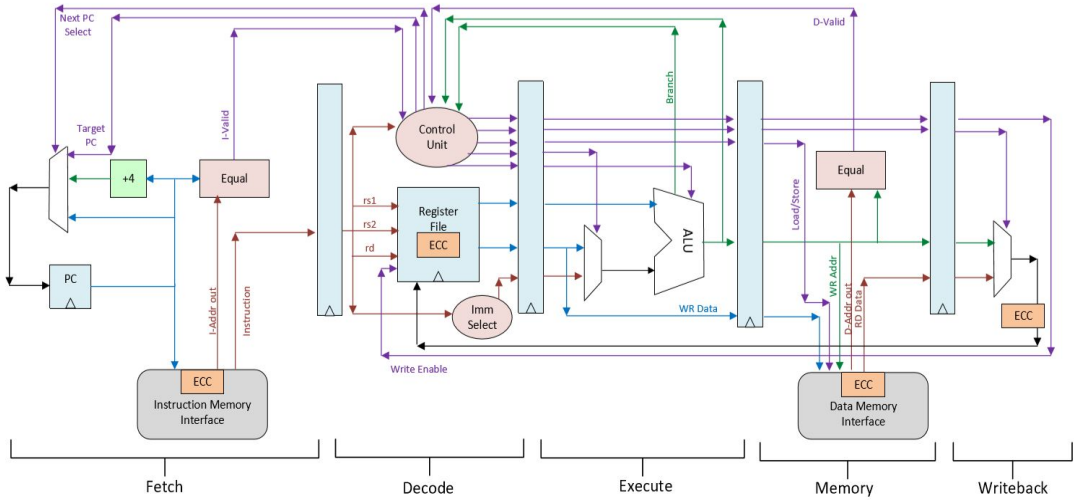


Figure 1: 5 stage pipelined core [2]

BLOCK DIAGRAM

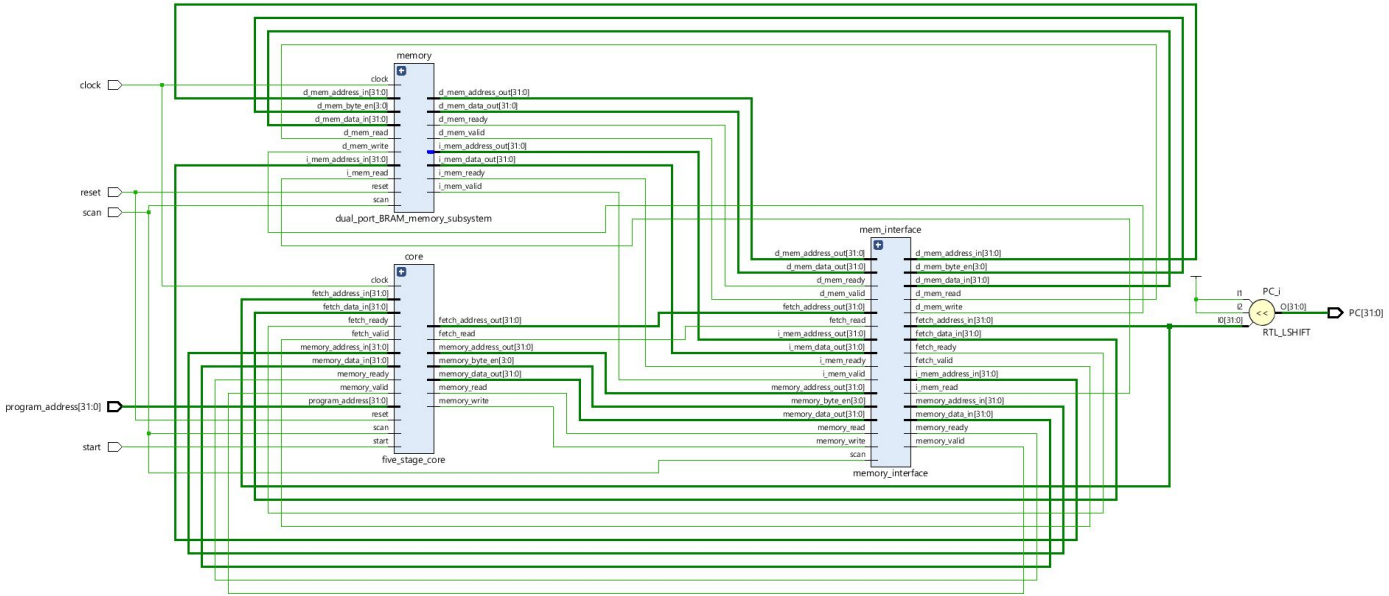


Figure 2: Implemented processor architecture

PROCEDURE

1. Building a RV32I based synthesizable processor with 5 stage pipeline.
2. Addition of Bit Manipulation Instructions extensions.
3. Addition of Error Control Coding module.
4. Verification of the processor using SystemVerilog based on UVM.
5. Measure the performance of the processor with and without extensions

WORKING PRINCIPLE

RISC V ISA:

- Available in 32-bit, 64-bit and 128-bit variants.
- Consists of a small base integer ISA which is usable by itself along with optional standard extensions.
- There are two primary base integer variants, RV32I and RV64I, which provide 32-bit or 64-bit user-level address spaces respectively.

WORKING PRINCIPLE

RISC V ISA: Base Instruction Formats

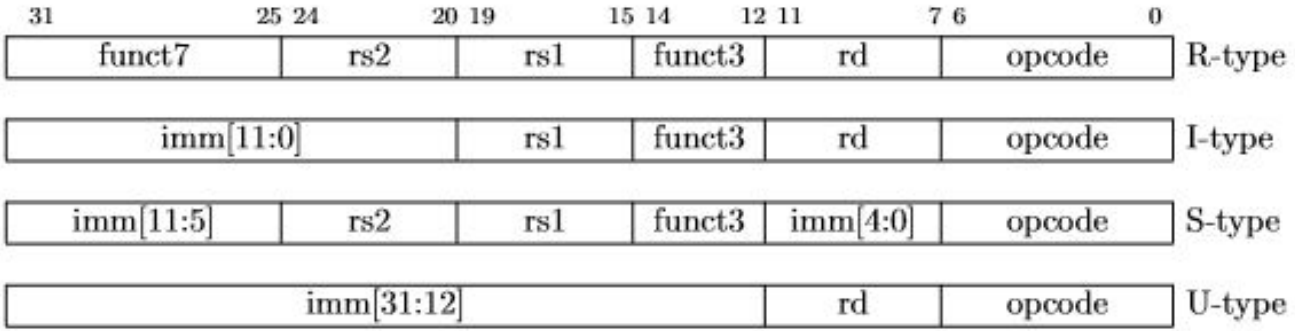


Figure 3: RISC V Base Instruction Formats[6]

- **Type R** *(Register)*: **ADD r4,r5,r6**
- **Type I** *(Immediate)*: **ADDI r4,r5,#30**
- **Type S** *(Store/Load)*: **STR r2,0x4500**
- **Type U** *(Upper Immediate)*: **LUI r3,#67**.

WORKING PRINCIPLE

RISC V ISA: Base Instruction Formats

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Figure 4: Example of S Type Instructions in the RISC V ISA[7]

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Figure 5: Example of R Type Instructions in the RISC V ISA[7]

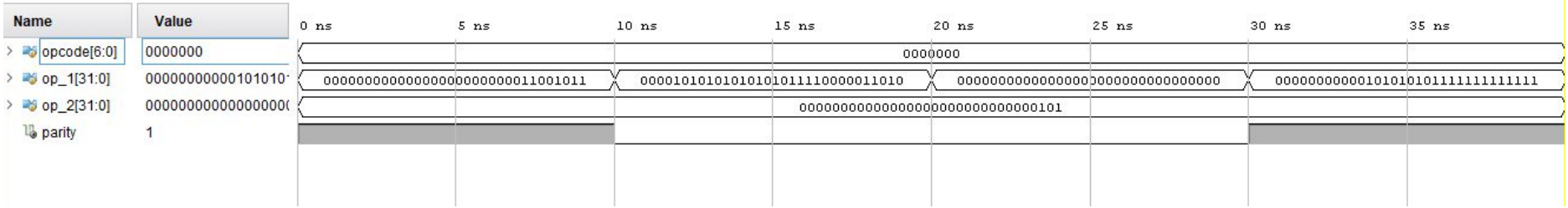
WORKING PRINCIPLE

Bit Manipulation Instructions

1. Parity
2. ByteSwap
3. Rotate Right and Rotate Left
4. Population Count
5. Leading Zeroes
6. Trailing Zeroes
7. Bit Reverse
8. Parallel Gather and Scatter

Implementation of these instructions is R-type.

Parity

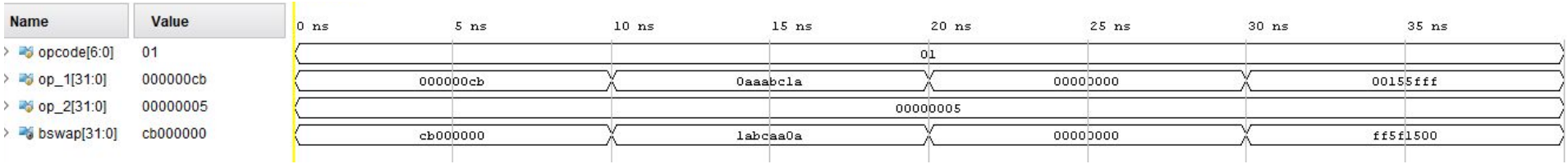


BMI Waveform 1: **Parity** calculation of op_1

- **Parity** is used to detect errors in transmitted data caused by noise or other disturbances.
- Calculated by counting the number on 1's in the data signal to implement **odd** parity.

funct7	rs2	rs1	funct3	rd	opcode
0000000	zero	op_1	000	rd	0111111

Byte Swap

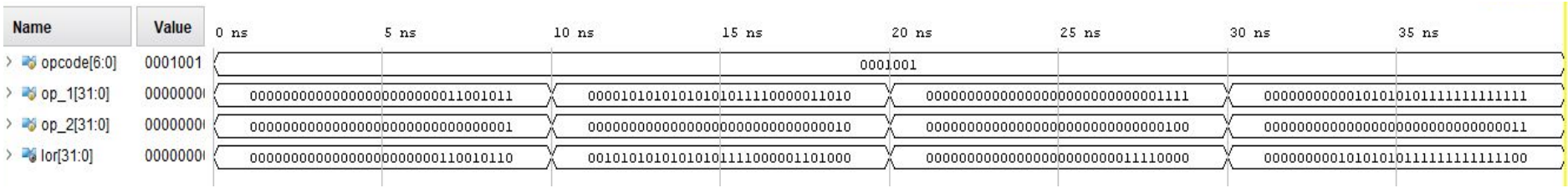


BMI Waveform 2: **Byte Swap** performed on op_1

- **Byte Swap** is calculated by changing the endianness of bytes.
- Useful for applications where the system is communicating with a system with a different endianness.

funct7	rs2	rs1	funct3	rd	opcode
0000000	zero	op_1	001	rd	0111111

Rotate Left

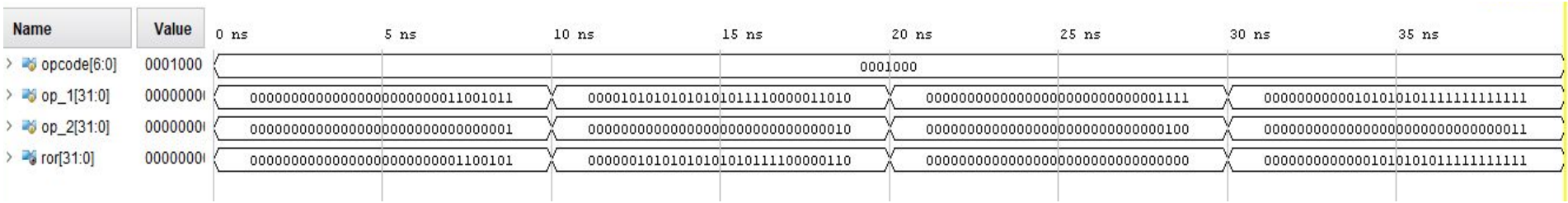


BMI Waveform 3: op_1 **Rotated Left** using op_2 data

- **Rotation instructions** are useful in Cryptography and in encoding/decoding data
- A Rotate Data Signal (op_2) is used to determine the rotation of the data bit.

funct7	rs2	rs1	funct3	rd	opcode
0000000	op_2	op_1	010	rd	0111111

Rotate Right

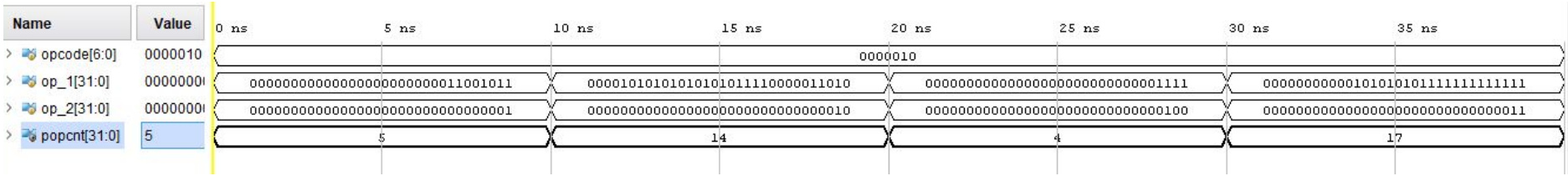


BMI Waveform 4: op_1 **Rotated Right** using op_2 data

- **Rotation instructions** are useful in Cryptography and in encoding/decoding data
- A Rotate Data Signal (op_2) is used to determine the rotation of the data bit.

funct7	rs2	rs1	funct3	rd	opcode
0000000	op_2	op_1	011	rd	0111111

Population Count

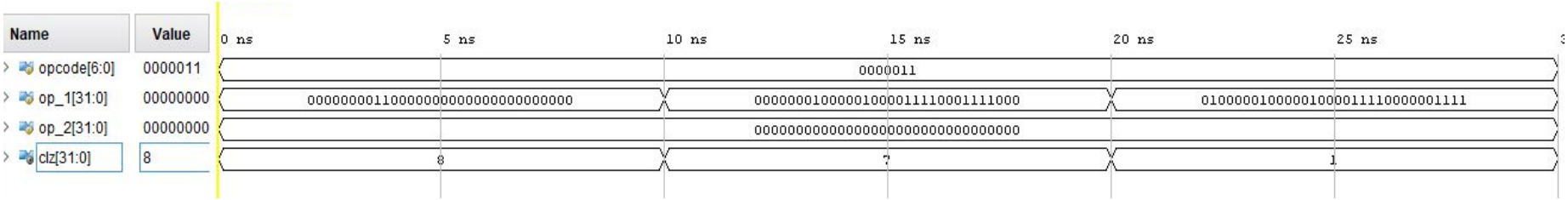


BMI Waveform 5: **Population Count** calculated on op_1

- **Population Count** tells the number of 1's in the data signal.
- This information is useful in Cryptography and related fields.

funct7	rs2	rs1	funct3	rd	opcode
0000000	zero	op_1	100	rd	0111111

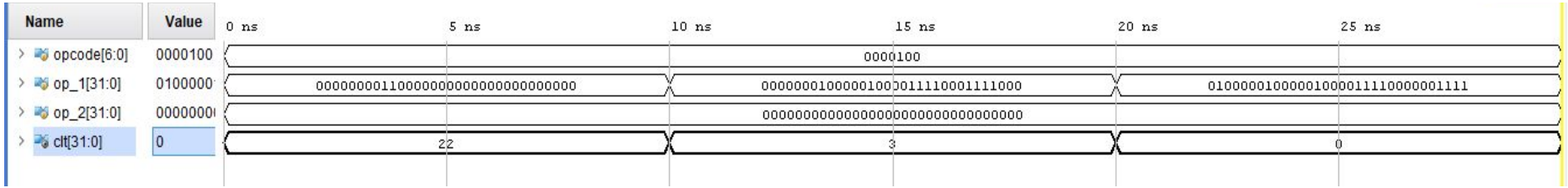
Count Leading Zeros



BMI Waveform 6: **Count of Leading Zeroes** calculated on op_1

funct7	rs2	rs1	funct3	rd	opcode
0000000	zero	op_1	101	rd	0111111

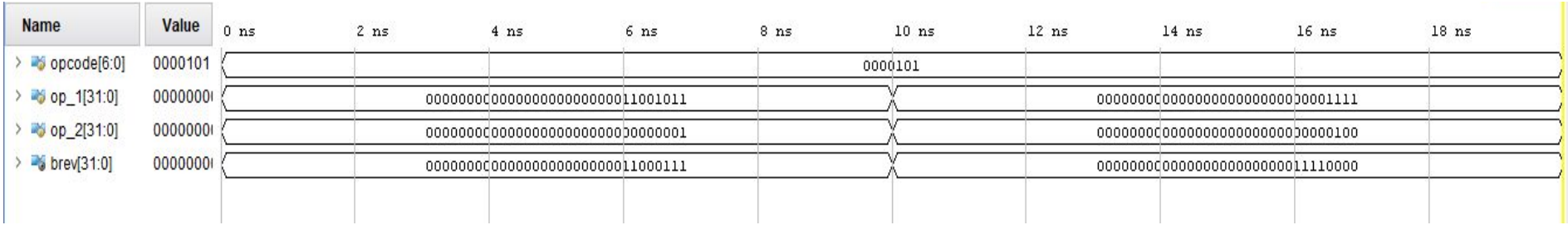
Count Trailing Zeros



BMI Waveform 7: **Count of Trailing Zeroes** calculated on op_1

funct7	rs2	rs1	funct3	rd	opcode
0000000	zero	op_1	110	rd	0111111

Bit Reverse



BMI Waveform 8: **Bit Reverse** calculated on op_1

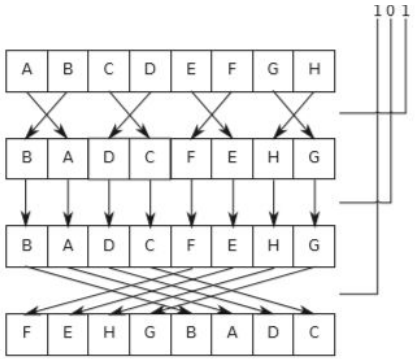


Figure 6: Bit Reverse[1]

- **Bit Reverse** is used in Cryptography.
- According to Bit Reverse Data Signal, the bit reversal of the data signal takes place.

funct7	rs2	rs1	funct3	rd	opcode
0000000	op_2	op_1	111	rd	0111111

Bit Reverse

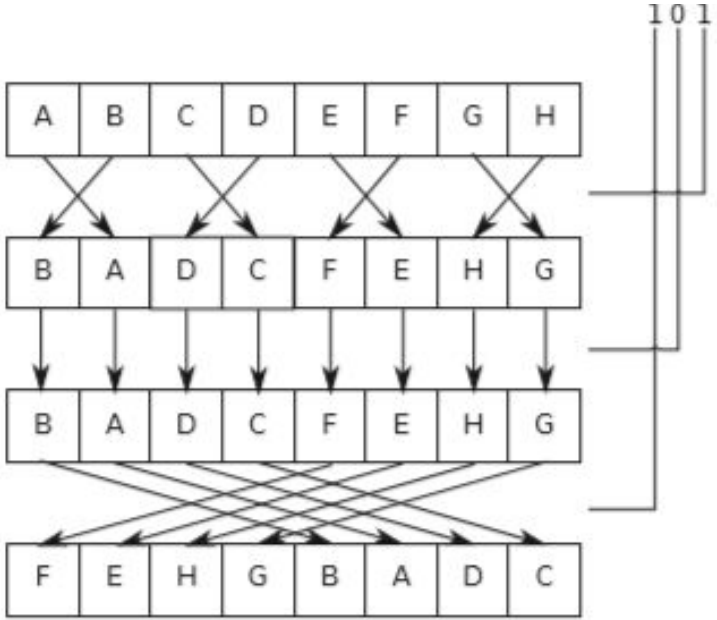
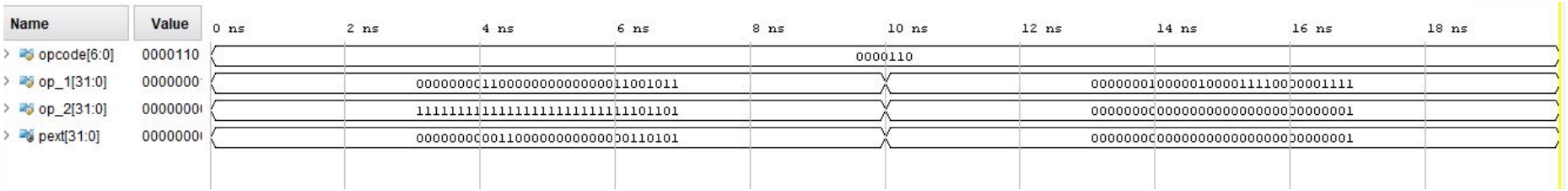


Figure 6: Bit Reverse[1]

Parallel Gather



BMI Waveform 9: **Parallel Gather** calculated on op_1 using mask signal op_2

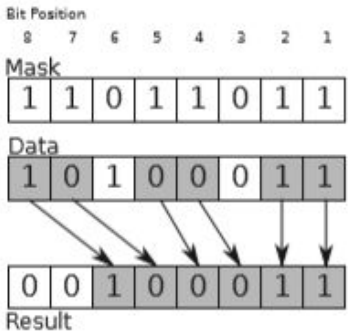
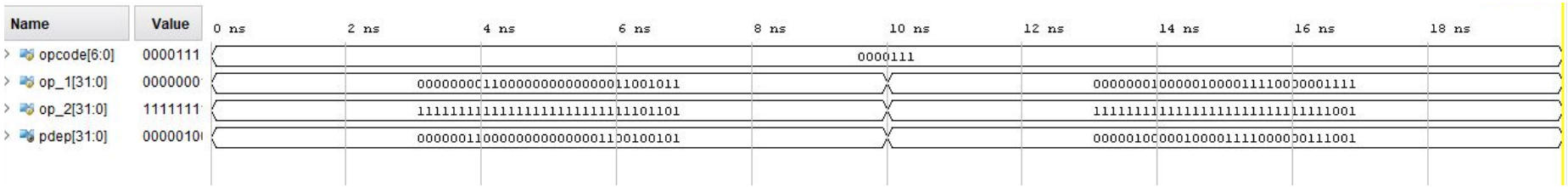


Figure 7: Parallel Gather[1]

- Mask signal (op_2) defines which bits to extract from the data word (op_1).
- **Parallel Gather** (along with Parallel Scatter) is used in Signal Processing and Cryptography domains.

funct7	rs2	rs1	funct3	rd	opcode
0000000	op_2	op_1	000	pext	0111110

Parallel Scatter



BMI Waveform 9: **Parallel Scatter** calculated on op_1 using mask signal op_2

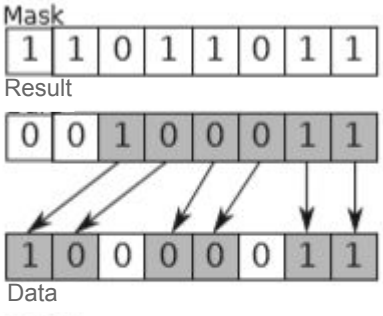


Figure 8: Parallel Scatter[1]

- Mask signal (op_2) defines which bits of result to deposit to the data word (op_1).
- **Parallel Scatter** (along with Parallel Gather) is used in Signal Processing and Cryptography domains.

funct7	rs2	rs1	funct3	rd	opcode
0000000	op_2	op_1	000	rd	0111110

WORKING PRINCIPLE

Error Correction Coding

- Data stored in memories is prone to soft errors.
- Hamming code has been used for Single Error Correction
- Syndrome is computed - which determines the error bit and the error bit is flipped

Error Correction Coding:

Hamming code (n,k):

- Identify all the positions that are powers of 2 in the codeword as positions for the parity bits
- The remaining positions are for the message bits. Fill the message bits into the positions reserved for them.
- Starting from the parity bit <parity bit position> number of bits are taken and the same number of bits are skipped - this is repeated till the end of the codeword.
- Example for a (39,32) code:

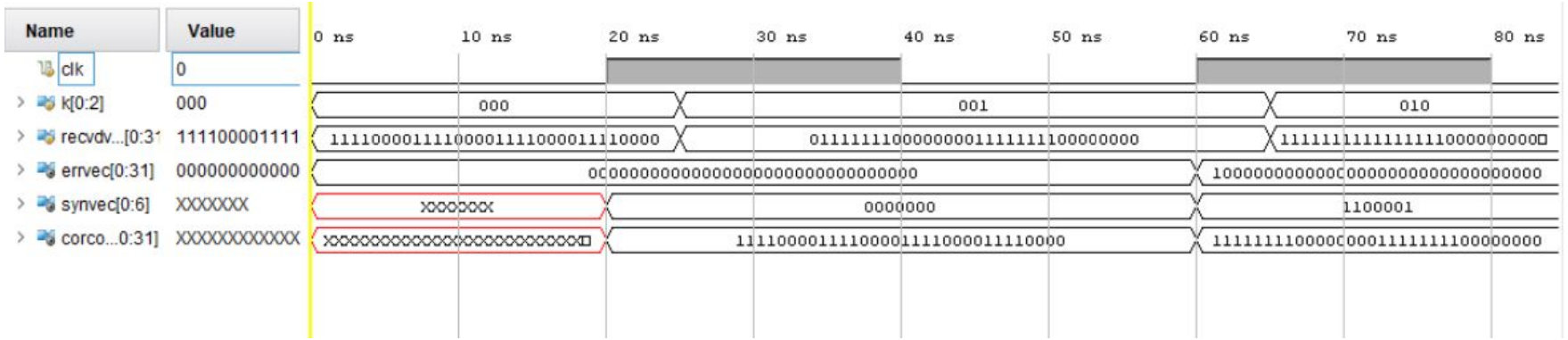
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39(codeword)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 (message)

Bits from the codeword to be taken for each parity bit

Message bits	Generated Parity Bits						
	P1	P2	P3	P4	P5	P6	P7
0	✓	✓					✓
1	✓		✓				✓
2		✓	✓				✓
3	✓	✓	✓				✓
4	✓			✓			✓
5		✓		✓			✓
6	✓	✓		✓			✓
7			✓	✓			✓
8	✓		✓	✓			✓
9		✓	✓	✓			✓
10	✓	✓	✓	✓			✓
11	✓				✓		✓
12		✓			✓		✓
13	✓	✓			✓		✓
14			✓		✓		✓
15	✓		✓		✓		✓
16		✓	✓		✓		✓
17	✓	✓	✓		✓		✓
18				✓	✓		✓
19	✓			✓	✓		✓
20		✓		✓	✓		✓
21	✓	✓		✓	✓		✓
22			✓	✓	✓		✓
23	✓		✓	✓	✓		✓
24		✓	✓	✓	✓		✓
25	✓	✓	✓	✓	✓		✓
26	✓					✓	✓
27		✓				✓	✓
28	✓	✓				✓	✓
29			✓			✓	✓
30	✓		✓			✓	✓
31		✓	✓			✓	✓

Table 1: Message bits taken for each parity bit for (39,32) Hamming code

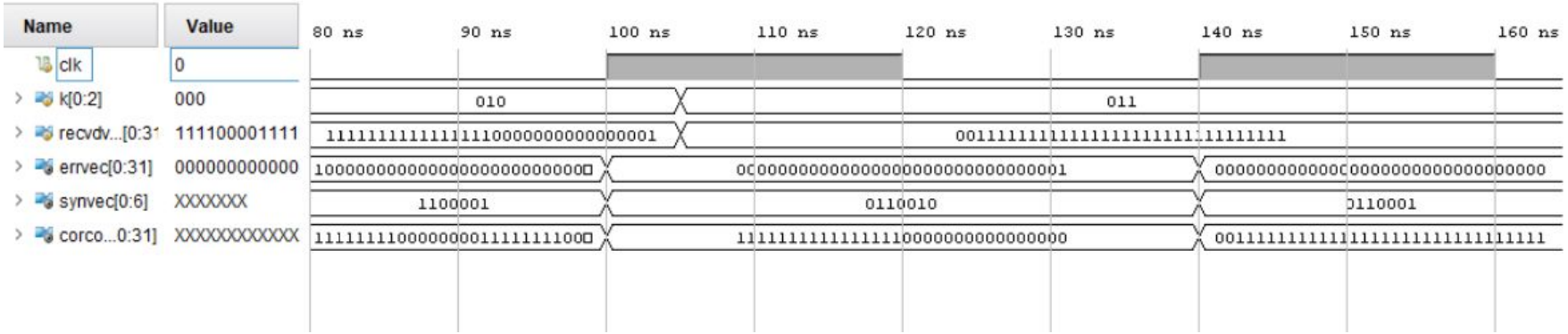
WAVEFORMS



Waveforms of the received data, error vector, syndrome, retransmit and corrected codeword

Data received at first positive edge of clk: No error
Data received at second positive edge of clk: First bit error

WAVEFORMS



Waveforms of the received data, error vector, syndrome, retransmit and corrected codeword

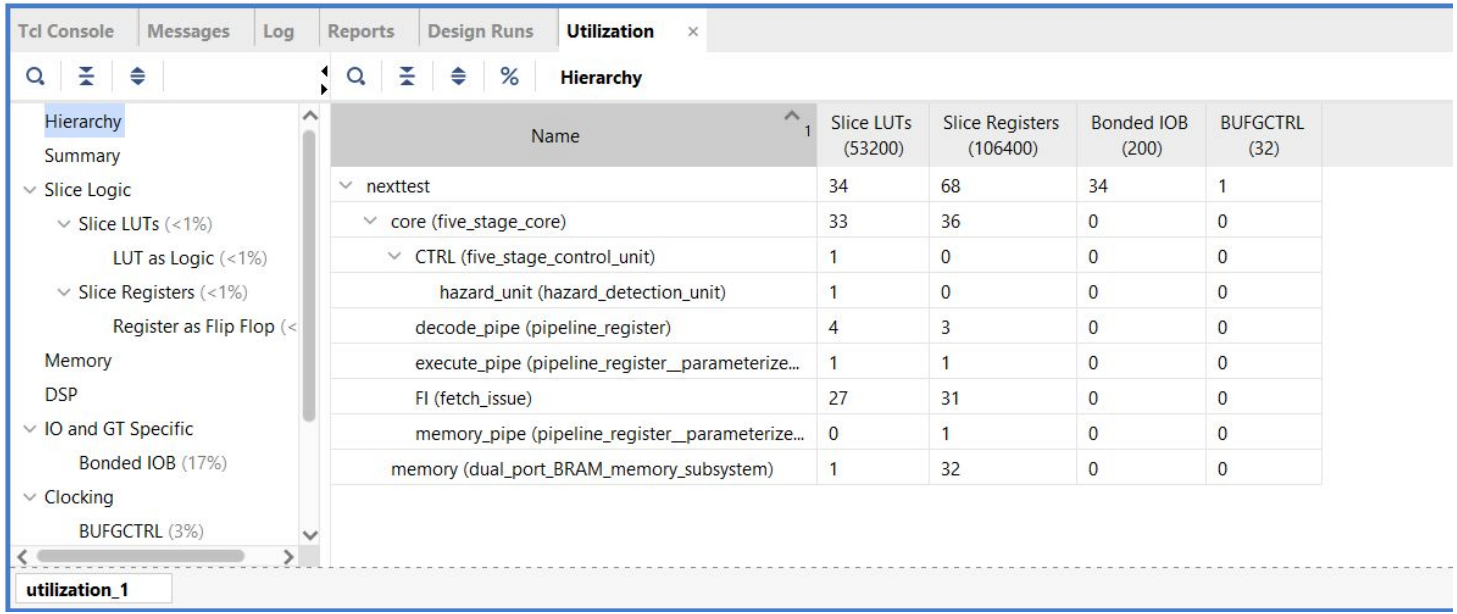
Data received at third positive edge of clk: Last bit error
Data received at fourth positive edge of clk: First and second bit error

IMPLEMENTATION OF ECC IN THE PROCESSOR

- Data has 32 bits → (39,32) Hamming code is used
- The **data memory and the instruction memory** have ECC to correct the data being sent from the memory as it would be stored there for a longer duration of time, having chances of data being corrupted.
- At the **decode stage**, the register file has ECC to detect and correct errors in the data stored in the file.

RESULTS

Synthesis and implementation of the modified processor was done using Vivado 2018.3 for ZedBoard Zynq Evaluation and Development Kit.



Name	Slice LUTs (53200)	Slice Registers (106400)	Bonded IOB (200)	BUFGCTRL (32)
nexttest	34	68	34	1
core (five_stage_core)	33	36	0	0
CTRL (five_stage_control_unit)	1	0	0	0
hazard_unit (hazard_detection_unit)	1	0	0	0
decode_pipe (pipeline_register)	4	3	0	0
execute_pipe (pipeline_register__parameterize...	1	1	0	0
FI (fetch_issue)	27	31	0	0
memory_pipe (pipeline_register__parameterize...	0	1	0	0
memory (dual_port_BRAM_memory_subsystem)	1	32	0	0

Figure 9:Utilization report of the processor with BMI and ECC module

VERIFICATION

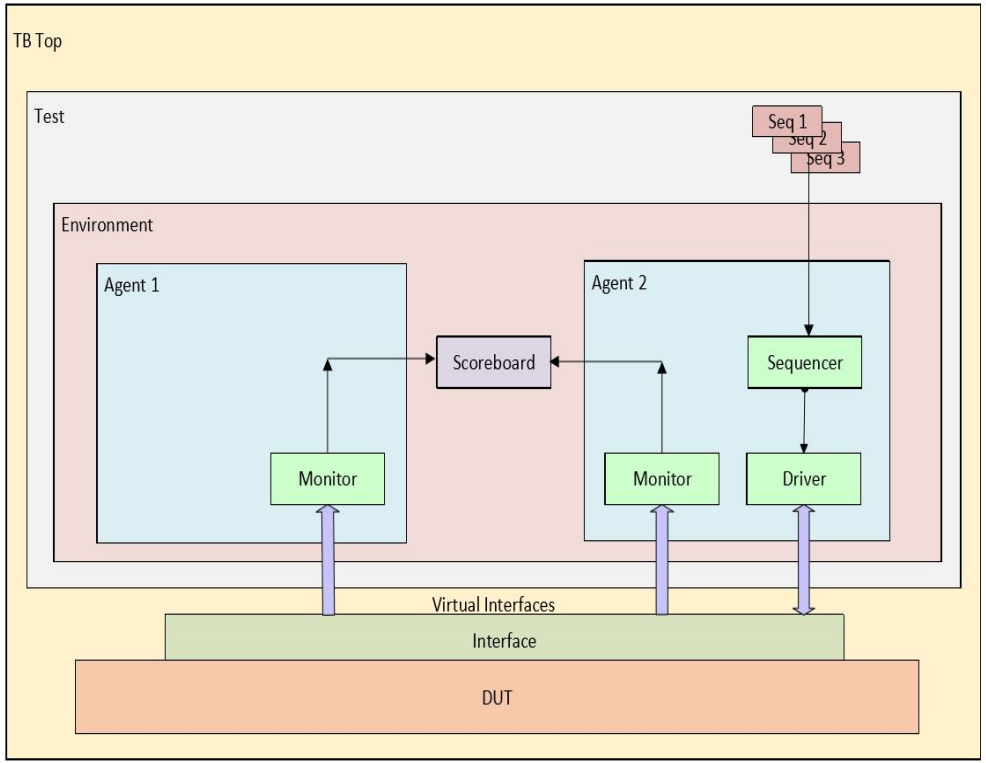


Figure 10: Verification architecture used

UVM Objects:

- Transaction
- Sequence

UVM Components:

- Sequencer
- Driver
- Monitor
- Agent
- Scoreboard
- Environment
- Test

Modules:

- TB top
- DUT

VERIFICATION PLAN

Index	Test name	Description	Type
1	Test Reset	The reset signal goes to one at different times	Positive
2	Test Start	The start values are varied	Positive and negative
3	Test PA	Different program addresses are given at different times.	Positive and negative
4	Test Freq	Different frequencies for clk	Positive

Table 2: Verification Plan used for the test bench

VERIFICATION IMPLEMENTATION

Results obtained after verification:

- The design worked as expected when tested for frequencies from **1Hz to 2GHz**.
- The **instance coverage** and the **code coverage** were obtained on simulating the test bench

VERIFICATION IMPLEMENTATION

CODE COVERAGE

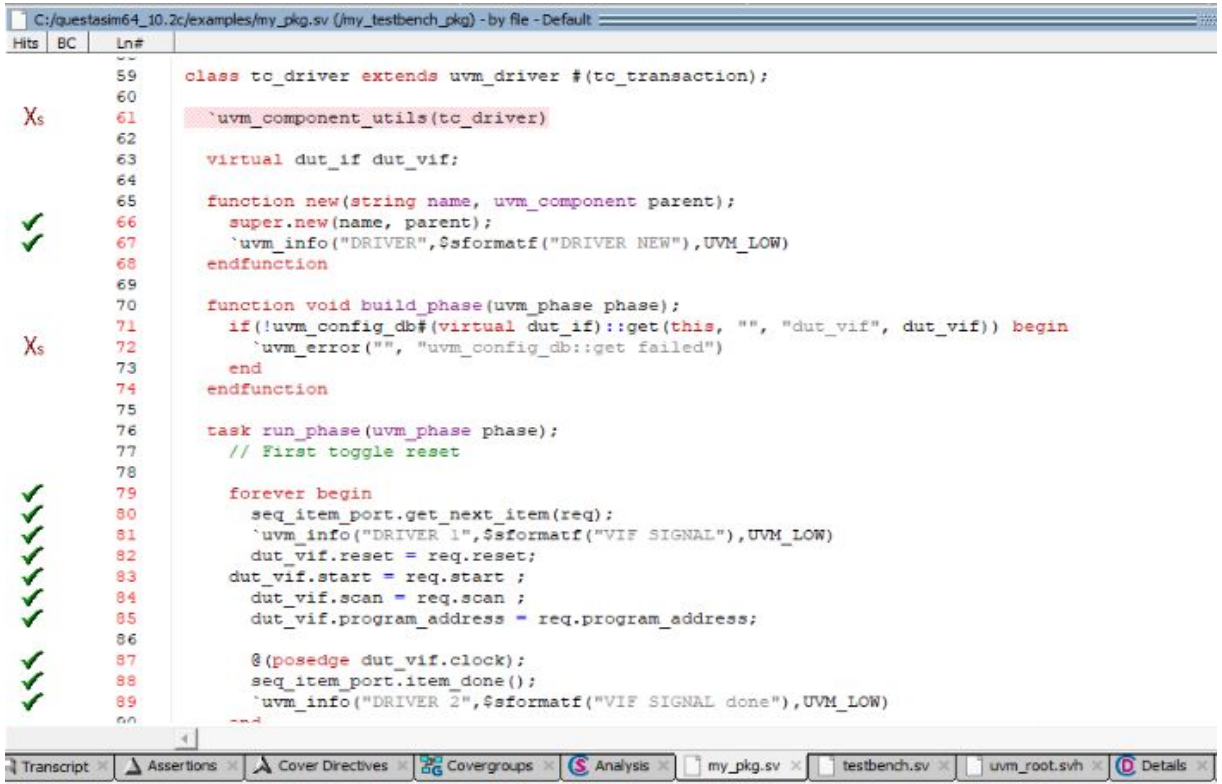


Figure 11: A section of code coverage of the tb

VERIFICATION IMPLEMENTATION

INSTANCE COVERAGE

sim (Recursive Coverage Aggregation) - Default

Instance	Design unit	Design unit type	Top Category	Visibility	Cover Options	Total coverage	Stmt count	Stmts hit	Stmts missed	Stmt %	Stmt graph
uvm_root	uvm_root	SVClassItem	TB Component	+acc=<full>	+cover=<none>						
uvm_test_top	test_rand	SVClassItem	TB Component	+acc=<full>	+cover=bcefst						
top	top(fast)	Module	DU Instance	+acc=<full>	+cover=bcefst	22.3%	664	449	215	67.6%	
BYTE_LOOP[3]	top(fast)	VlGenerateBlock	-	+acc=<full>	+cover=bcefst	100.0%	4	4	0	100%	
BYTE_LOOP[2]	top(fast)	VlGenerateBlock	-	+acc=<full>	+cover=bcefst	100.0%	4	4	0	100%	
BYTE_LOOP[1]	top(fast)	VlGenerateBlock	-	+acc=<full>	+cover=bcefst	100.0%	4	4	0	100%	
BYTE_LOOP[0]	top(fast)	VlGenerateBlock	-	+acc=<full>	+cover=bcefst	100.0%	4	4	0	100%	
dut_if1	dut_if(fast)	Interface	DU Instance	+acc=<full>	+cover=bcefst	51.5%					
dut	nexttest(fa...	Module	DU Instance	+acc=<full>	+cover=bcefst	22.4%	602	413	189	68.6%	
dif	dut_if(fast)	Interface	DU Instance	+acc=<full>	+cover=bcefst						
core	five_stage...	Module	DU Instance	+acc=<full>	+cover=bcefst	20.4%	488	315	173	64.5%	
mem_interf...	memory_in...	Module	DU Instance	+acc=<full>	+cover=bcefst						
memory	dual_port_...	Module	DU Instance	+acc=<full>	+cover=bcefst	35.8%	113	97	16	85.8%	
#ASSIGN#...	nexttest(fa...	Process	-	+acc=<full>							
#INITIAL#71	top(fast)	Process	-	+acc=<full>							
#INITIAL#81	top(fast)	Process	-	+acc=<full>							
#INITIAL#118	top(fast)	Process	-	+acc=<full>							
#ALWAYS#147...	top(fast)	Process	-	+acc=<full>							
uvm_pkg	uvm_pkg	VlPackage	Package	+acc=<full>	+cover=<none>						
my_testbench_pkg	my_testbe...	VlPackage	Package	+acc=<full>	+cover=bcefst	47.7%	124	79	45	63.7%	
std	std	VlPackage	Package	+acc=<full>	+cover=<none>						
questa_uvm_pkg	questa_uv...	VlPackage	Package	+acc=<full>	+cover=<none>						
#vsim_capacity#		Capacity	Statistics	+acc=<none>							

Figure 12: Instance coverage of all modules

VERIFICATION IMPLEMENTATION

INSTANCE COVERAGE FOR UVM TEST BENCH

Instance	Total coverage
my_testbench_pkg::sequence_rand::body	87.5%
my_testbench_pkg::sequence_rand::get_type_name	100%
my_testbench_pkg::sequence_rand::new	100%
my_testbench_pkg::tc_driver::run_phase	100%
my_testbench_pkg::tc_driver::new	100%
my_testbench_pkg::test_rand::run_phase	100%
my_testbench_pkg::test_rand::build_phase	100%
my_testbench_pkg::test_rand::get_type_name	100%
my_testbench_pkg::test_rand::new	100%
my_testbench_pkg::base_test::extract_phase	
my_testbench_pkg::base_test::end_of_elaboartion_phase	100%
my_testbench_pkg::base_test::build_phase	100%
my_testbench_pkg::base_test::new	100%
my_testbench_pkg::my_env::connect_phase	
my_testbench_pkg::my_env::build_phase	100%
my_testbench_pkg::my_env::new	100%
my_testbench_pkg::my_agent1::connect_phase	100%
my_testbench_pkg::my_agent1::build_phase	100%
my_testbench_pkg::my_agent1::new	100%
my_testbench_pkg::my_agent2::connect_phase	100%
my_testbench_pkg::my_agent2::build_phase	100%
my_testbench_pkg::my_agent2::new	100%
my_testbench_pkg::my_passive_monitor::run_phase	100%
my_testbench_pkg::my_passive_monitor::build_phase	66.7%
my_testbench_pkg::my_passive_monitor::new	100%
my_testbench_pkg::my_active_monitor::run_phase	100%
my_testbench_pkg::my_active_monitor::build_phase	66.7%
my_testbench_pkg::my_active_monitor::new	100%

Table 3: Instance Coverage for all instances of the test bench code

PERFORMANCE MEASUREMENT

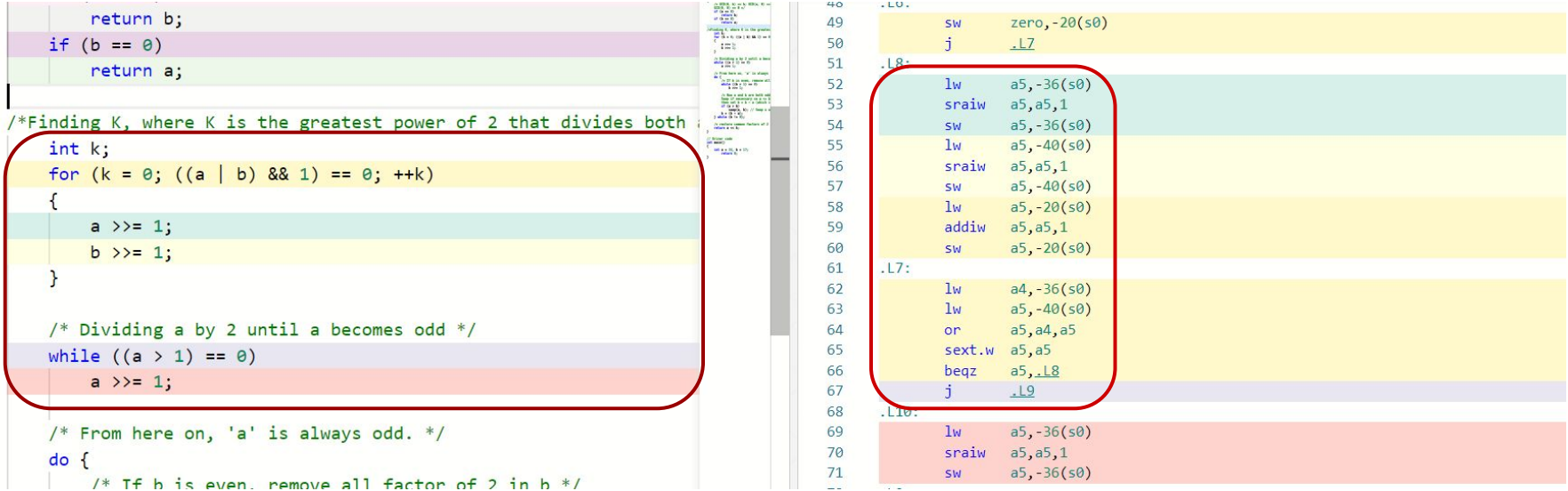
Test Setup

- The same C programs were tested on the design with and without the BMI extension.
- The programs were converted to ELF (Executable Linkable Format) files using the **riscv-gnu-toolchain** for the RV32I instruction set architecture.
- **Compiler Explorer** is used to match the C programs code to the RISC V assembly for analysis. Each instruction in C is colour coded to reflect the corresponding assembly code.
- The RISC V manual was referred to interpret the instructions and the replacement of BMIs was done accordingly.

PERFORMANCE MEASUREMENT

PERFORMANCE MEASUREMENT

Greatest Common Divisor: Stein's Algorithm is used to find the highest power of 2 that both inputs are divisible. Count trailing zeros BMI is implemented to perform the same function.



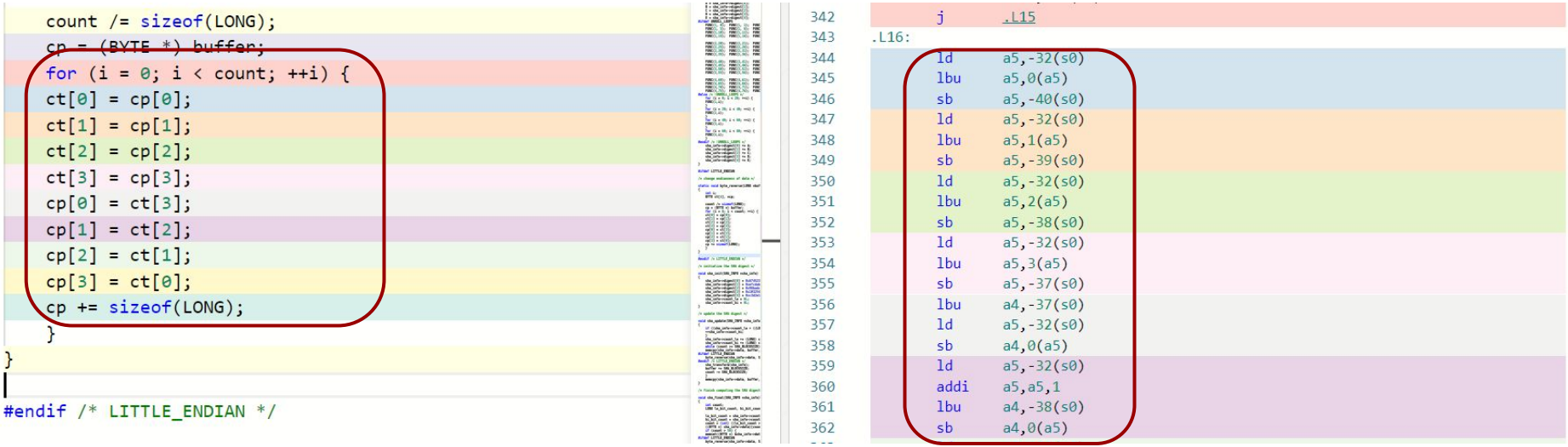
```
        return b;
    if (b == 0)
        return a;

/*Finding K, where K is the greatest power of 2 that divides both
    int k;
    for (k = 0; ((a | b) && 1) == 0; ++k)
    {
        a >>= 1;
        b >>= 1;
    }

    /* Dividing a by 2 until a becomes odd */
    while ((a > 1) == 0)
        a >>= 1;

    /* From here on, 'a' is always odd. */
    do {
        /* If b is even, remove all factor of 2 in b */
```

```
49          sw      zero,-20(s0)
50          j       .L7
51  .L8:
52          lw      a5,-36(s0)
53          sraiw   a5,a5,1
54          sw      a5,-36(s0)
55          lw      a5,-40(s0)
56          sraiw   a5,a5,1
57          sw      a5,-40(s0)
58          lw      a5,-20(s0)
59          addiw   a5,a5,1
60          sw      a5,-20(s0)
61  .L7:
62          lw      a4,-36(s0)
63          lw      a5,-40(s0)
64          or      a5,a4,a5
65          sext.w  a5,a5
66          beqz    a5,.L8
67          j       .L9
68  .L10:
69          lw      a5,-36(s0)
70          sraiw   a5,a5,1
71          sw      a5,-36(s0)
```

Figure 14: Stein's Algorithm is used for GCD

PERFORMANCE MEASUREMENT

Secure Hash Algorithm: Requires big endian input where byte swap BMI can be used



```
    count /= sizeof(LONG);
    cp = (BYTE *) buffer;
    for (i = 0; i < count; ++i) {
    ct[0] = cp[0];
    ct[1] = cp[1];
    ct[2] = cp[2];
    ct[3] = cp[3];
    cp[0] = ct[3];
    cp[1] = ct[2];
    cp[2] = ct[1];
    cp[3] = ct[0];
    cp += sizeof(LONG);
    }
}

#endif /* LITTLE_ENDIAN */
```

```
342         j       .L15
343 .L16:
344         ld      a5,-32(s0)
345         lbu     a5,0(a5)
346         sb      a5,-40(s0)
347         ld      a5,-32(s0)
348         lbu     a5,1(a5)
349         sb      a5,-39(s0)
350         ld      a5,-32(s0)
351         lbu     a5,2(a5)
352         sb      a5,-38(s0)
353         ld      a5,-32(s0)
354         lbu     a5,3(a5)
355         sb      a5,-37(s0)
356         lbu     a4,-37(s0)
357         ld      a5,-32(s0)
358         sb      a4,0(a5)
359         ld      a5,-32(s0)
360         addi    a5,a5,1
361         lbu     a4,-38(s0)
362         sb      a4,0(a5)
```

Figure 15: Byte swap implemented in Secure Hash Algorithm(SHA)

PERFORMANCE MEASUREMENT

Secure Hash Algorithm: Requires rotate left function where the rotate left BMI can be used

```
/* 32-bit rotate */

#define ROT32(x,n)  ((x << n) | (x >> (32 - n)))

#define FUNC(n,i)                       \
    temp = ROT32(A,5) + f##n(B,C,D) + E + W[i] + CONST##n;  \
    E = D; D = C; C = ROT32(B,30); B = A; A = temp

/* do SHA transformation */

    FUNC(4,70);  FUNC(4,71);  FUNC(4,72);  FUNC(4,73);  FUNC(4,74
    FUNC(4,75);  FUNC(4,76);  FUNC(4,77);  FUNC(4,78);  FUNC(4,79
#else /* !UNROLL_LOOPS */
    for (i = 0; i < 20; ++i) {
    FUNC(1,i);
    }
    for (i = 20; i < 40; ++i) {
    FUNC(2,i);
    }
    for (i = 40; i < 60; ++i) {
    FUNC(3,i);
    }
    for (i = 60; i < 80; ++i) {
    FUNC(4,i);
    }
#endif /* !UNROLL_LOOPS */
    sha_info->digest[0] += A;
    sha_info->digest[1] += B;
    sha_info->digest[2] += C;
```

```
108     ld      a5,-64(s0)
109     add     a4,a4,a5
110     lw      a5,-20(s0)
111     slli    a5,a5,3
112     addi    a3,s0,-16
113     add     a5,a3,a5
114     ld      a5,-696(a5)
115     add     a4,a4,a5
116     li      a5,1518501888
117     addi    a5,a5,-1639
118     add     a5,a4,a5
119     sd      a5,-72(s0)
120     ld      a5,-56(s0)
121     sd      a5,-64(s0)
122     ld      a5,-48(s0)
123     sd      a5,-56(s0)
124     ld      a5,-40(s0)
125     slli    a4,a5,30
126     ld      a5,-40(s0)
127     srli    a5,a5,2
128     or      a5,a4,a5
129     sd      a5,-48(s0)
130     ld      a5,-32(s0)
131     sd      a5,-40(s0)
```

Figure 16: Rotate left implemented in the Secure Hash Algorithm (SHA)

PERFORMANCE MEASUREMENT

Integer Square Root: It is a basic math program from MiBench without multiplications or divisions aimed to perform on a variety of processors. Figure 10 shows that the program checks the inconsequential MSBs of the input where count leading zeros BMI is implemented.

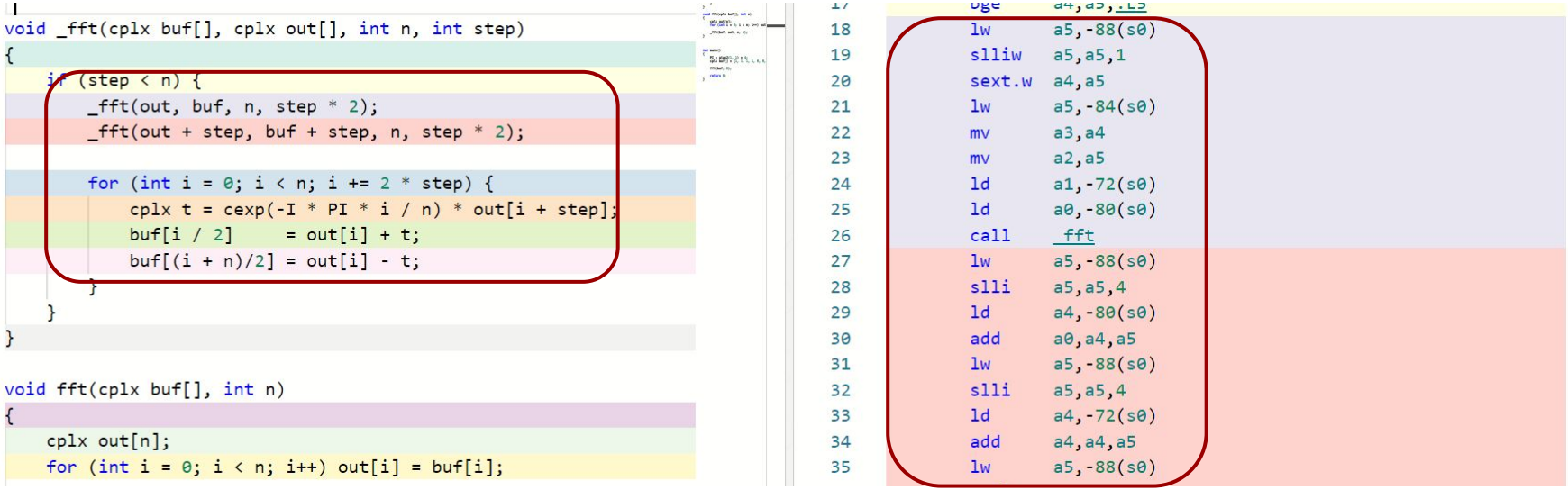
```
#define TOP2BITS(x) ((x & (3L << (BITSPERLONG-2))) >> (BITSPERLONG

void usqrt(unsigned long x,unsigned long q)
{
    unsigned long a = 0L;                   /* accumulator
    unsigned long r = 0L;                   /* remainder
    unsigned long e = 0L;                   /* trial product

    int i;

    for (i = 0; i < BITSPERLONG/2; i++)
    {
        r = (r << 2) + TOP2BITS(x); x <<= 2;
        a <<= 1;
        e = (a << 1) + 1;
        if (r >= e)
        {
            r -= e;
```

```
13      ld      a5,-24(s0)
14      slli    a4,a5,2
15      ld      a5,-56(s0)
16      srli    a5,a5,30
17      andi    a5,a5,3
18      add     a5,a4,a5
19      sd      a5,-24(s0)
20      ld      a5,-56(s0)
21      slli    a5,a5,2
22      sd      a5,-56(s0)
23      ld      a5,-48(s0)
24      slli    a5,a5,1
25      sd      a5,-48(s0)
26      ld      a5,-48(s0)
27      slli    a5,a5,1
28      addi    a5,a5,1
29      sd      a5,-40(s0)
30      ld      a4,-24(s0)
31      ld      a5,-40(s0)
32      bltu    a4,a5,.L3
33      ld      a4,-24(s0)
34      ld      a5,-40(s0)
```

Figure 17 : Finding inconsequential MSB in square root

PERFORMANCE MEASUREMENT

Fast Fourier Transform (FFT): It is a digital signal processing test from MiBench. Figure 14 shows that the algorithm requires bit reversed addressing where bit reverse BMI is replaced.



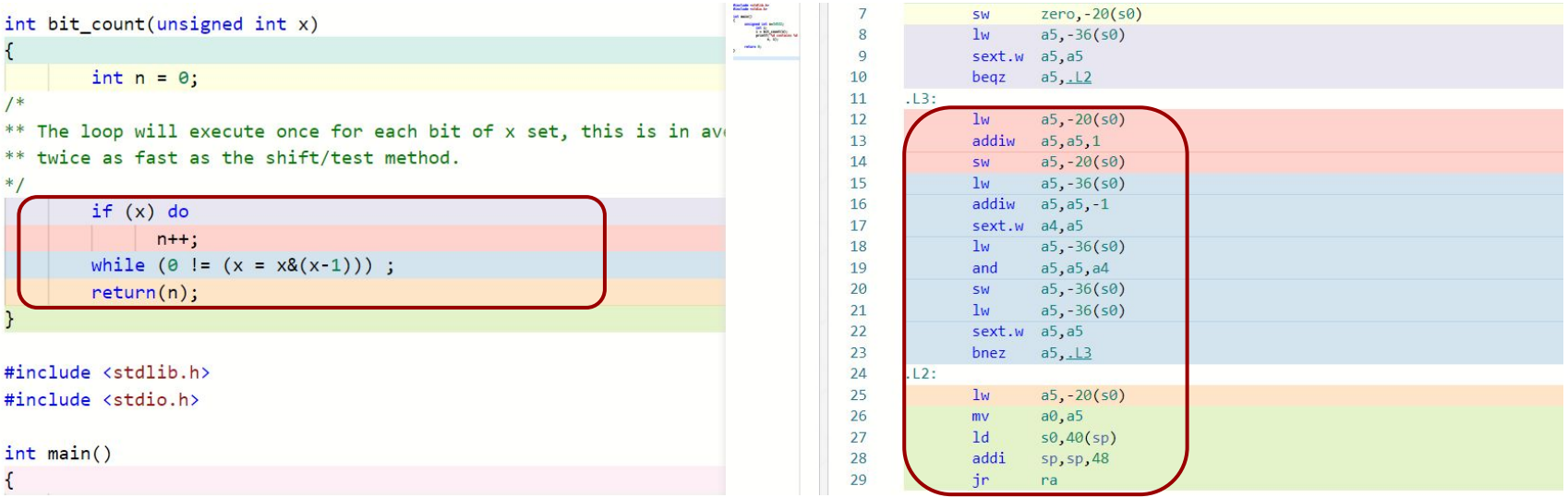
```
void _fft(cplx buf[], cplx out[], int n, int step)
{
    if (step < n) {
        _fft(out, buf, n, step * 2);
        _fft(out + step, buf + step, n, step * 2);

        for (int i = 0; i < n; i += 2 * step) {
            cplx t = cexp(-I * PI * i / n) * out[i + step];
            buf[i / 2]     = out[i] + t;
            buf[(i + n)/2] = out[i] - t;
        }
    }
}

void fft(cplx buf[], int n)
{
    cplx out[n];
    for (int i = 0; i < n; i++) out[i] = buf[i];
```

```
18      lw      a5,-88(s0)
19      slliw   a5,a5,1
20      sext.w  a4,a5
21      lw      a5,-84(s0)
22      mv      a3,a4
23      mv      a2,a5
24      ld      a1,-72(s0)
25      ld      a0,-80(s0)
26      call    fft
27      lw      a5,-88(s0)
28      slli    a5,a5,4
29      ld      a4,-80(s0)
30      add     a0,a4,a5
31      lw      a5,-88(s0)
32      slli    a5,a5,4
33      ld      a4,-72(s0)
34      add     a4,a4,a5
35      lw      a5,-88(s0)
```

Figure 18: Bit reverse implemented in Fast Fourier Transform

PERFORMANCE MEASUREMENT

Miscellaneous Bit Count Programs: A number of bit count algorithms are present in MiBench ranging from simple to sophisticated as shown in Figures 11, 12 and 13.The popcount BMI is used instead of these algorithms.



```
int bit_count(unsigned int x)
{
        int n = 0;
/*
** The loop will execute once for each bit of x set, this is in av
** twice as fast as the shift/test method.
*/
        if (x) do
                n++;
        while (0 != (x = x&(x-1))) ;
        return(n);
}

#include <stdlib.h>
#include <stdio.h>

int main()
{
```

```
        sw      zero,-20(s0)
        lw      a5,-36(s0)
        sext.w  a5,a5
        beqz    a5,.L2
.L3:
        lw      a5,-20(s0)
        addiw   a5,a5,1
        sw      a5,-20(s0)
        lw      a5,-36(s0)
        addiw   a5,a5,-1
        sext.w  a4,a5
        lw      a5,-36(s0)
        and     a5,a5,a4
        sw      a5,-36(s0)
        lw      a5,-36(s0)
        sext.w  a5,a5
        bnez    a5,.L3
.L2:
        lw      a5,-20(s0)
        mv      a0,a5
        ld      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

Figure 19 : Bit Count algorithm 1

```
/*
**  Count bits in each byte
**
**  by Auke Reitsma, works best on Microsoft, Symantec, and other
*/

int AR_btbl_bitcount(unsigned int x)
{
    unsigned char * Ptr = (unsigned char *) &x ;
    int Accu ;

    Accu  = bits[ *Ptr++ ];
    Accu += bits[ *Ptr++ ];
    Accu += bits[ *Ptr++ ];
    Accu += bits[ *Ptr ];
    return Accu;
}
```

```
addi    a5,s0,-36
sd      a5,-24(s0)
ld      a5,-24(s0)
addi    a4,a5,1
sd      a4,-24(s0)
lbu     a5,0(a5)
sext.w  a4,a5
lui     a5,%hi(bits)
addi    a5,a5,%lo(bits)
add     a5,a4,a5
lbu     a5,0(a5)
sw      a5,-28(s0)
ld      a5,-24(s0)
addi    a4,a5,1
sd      a4,-24(s0)
lbu     a5,0(a5)
sext.w  a4,a5
lui     a5,%hi(bits)
addi    a5,a5,%lo(bits)
add     a5,a4,a5
lbu     a5,0(a5)
sext.w  a5,a5
lw      a4,-28(s0)
addw    a5,a4,a5
sw      a5,-28(s0)
ld      a5,-24(s0)
```

Figure 20 : Bit Count algorithm 2

```
int bitcount(unsigned int i)
{
    i = ((i & 0xAAAAAAAAL) >>  1) + (i & 0x55555555L);
    i = ((i & 0xCCCCCCCCL) >>  2) + (i & 0x33333333L);
    i = ((i & 0xF0F0F0F0L) >>  4) + (i & 0x0F0F0F0FL);
    i = ((i & 0xFF00FF00L) >>  8) + (i & 0x00FF00FFL);
    i = ((i & 0xFFFF0000L) >> 16) + (i & 0x0000FFFFL);
    return (int)i;
}

#include <stdlib.h>
#include <stdio.h>
int main()
{
    unsigned int n;
```

```
mv      a5,a0
sw      a5,-20(s0)
lwu     a5,-20(s0)
srai    a5,a5,1
sext.w  a5,a5
mv      a4,a5
li      a5,1431654400
addi    a5,a5,1365
and     a5,a4,a5
sext.w  a4,a5
lw      a3,-20(s0)
li      a5,1431654400
addi    a5,a5,1365
and     a5,a3,a5
sext.w  a5,a5
addw    a5,a4,a5
sw      a5,-20(s0)
lwu     a5,-20(s0)
srai    a5,a5,2
sext.w  a5,a5
mv      a4,a5
li      a5,858992640
addi    a5,a5,819
and     a5,a4,a5
sext.w  a4,a5
```

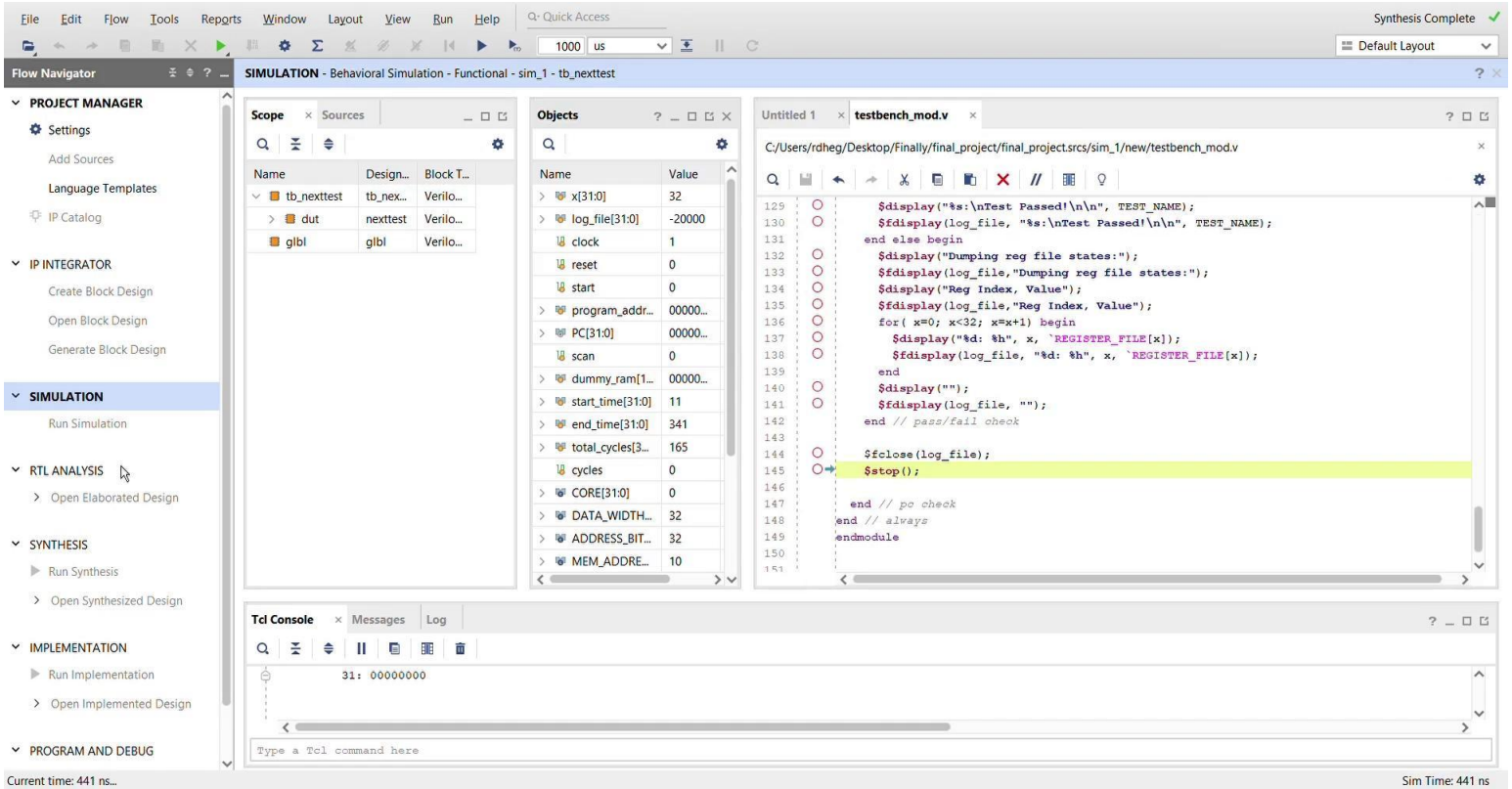
Figure 21 : Bit Count algorithm 3

PERFORMANCE MEASUREMENT

Program	Without BMIs	With BMI	Speedup	BMI used
Binary GCD	165	93	1.79	ctz
Square root	1691	829	2.04	clz
SHA	2027	1748	1.15	rotl and bswap
FFT	28378	24256	1.16	brev
Bitcount 1	208	87	2.39	popcnt
Bitcount 2	845	87	9.71	popcnt
Bitcount 3	4539	87	52.17	popcnt

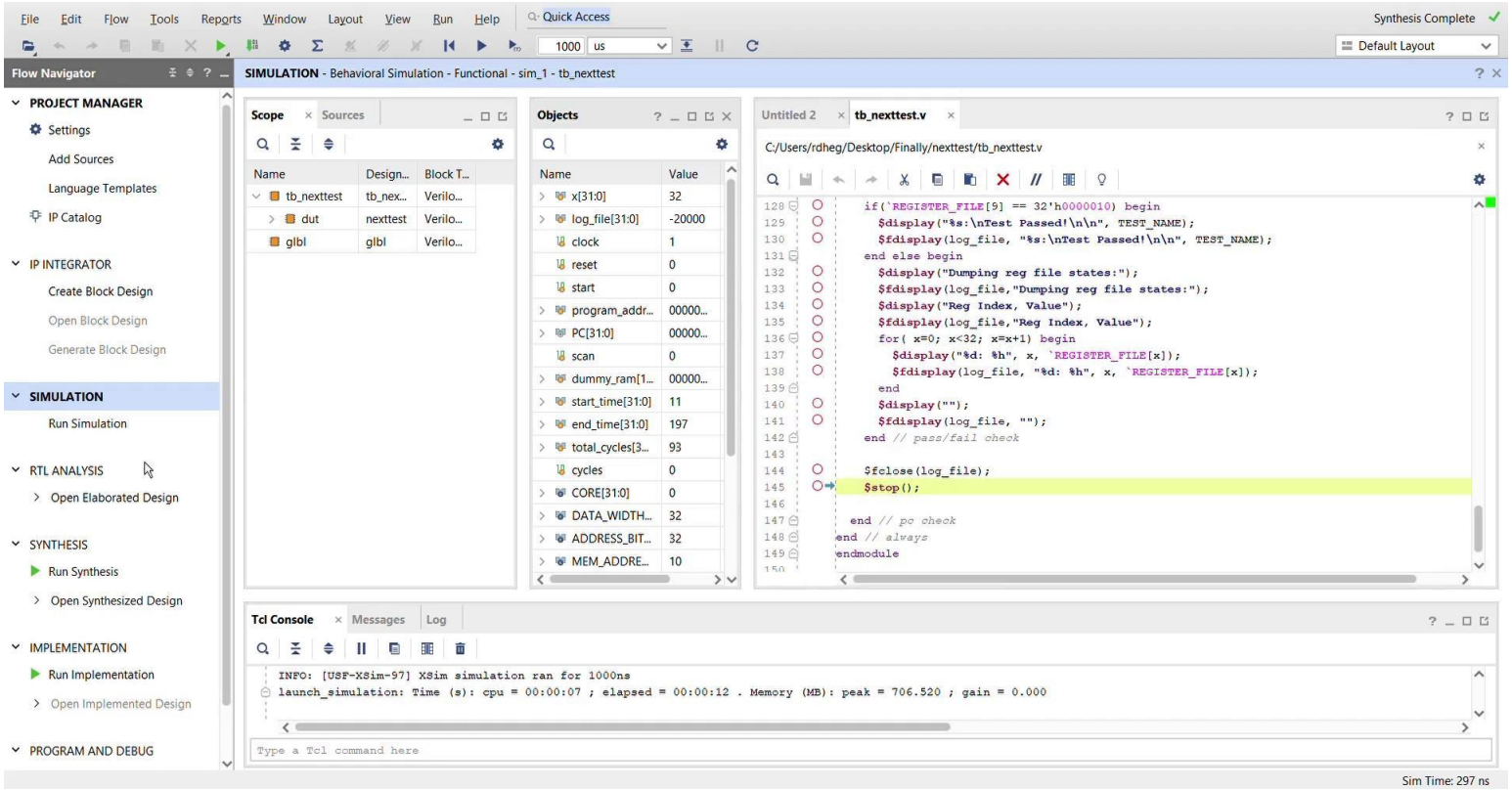
Table 4:Execution time in cycles for the test with and without BMIs

PERFORMANCE MEASUREMENT



Video 1:Demonstration of gcd program without BMI using Vivado.

PERFORMANCE MEASUREMENT



Video 2:Demonstration of gcd program with BMI using Vivado.

ACCOMPLISHMENTS

1. Literature survey of RISC V processors and research on required instructions to enhance efficiency
2. Study and implementation of Bit Manipulation Instructions Extensions
3. Study and implementation of Error Correction Coding
4. Verification plan and implementation
5. Performance analysis

FUTURE SCOPE

1. Adding more bit manipulation instructions.
2. Implementing ECC algorithms that can correct a higher number of bits.

REFERENCES

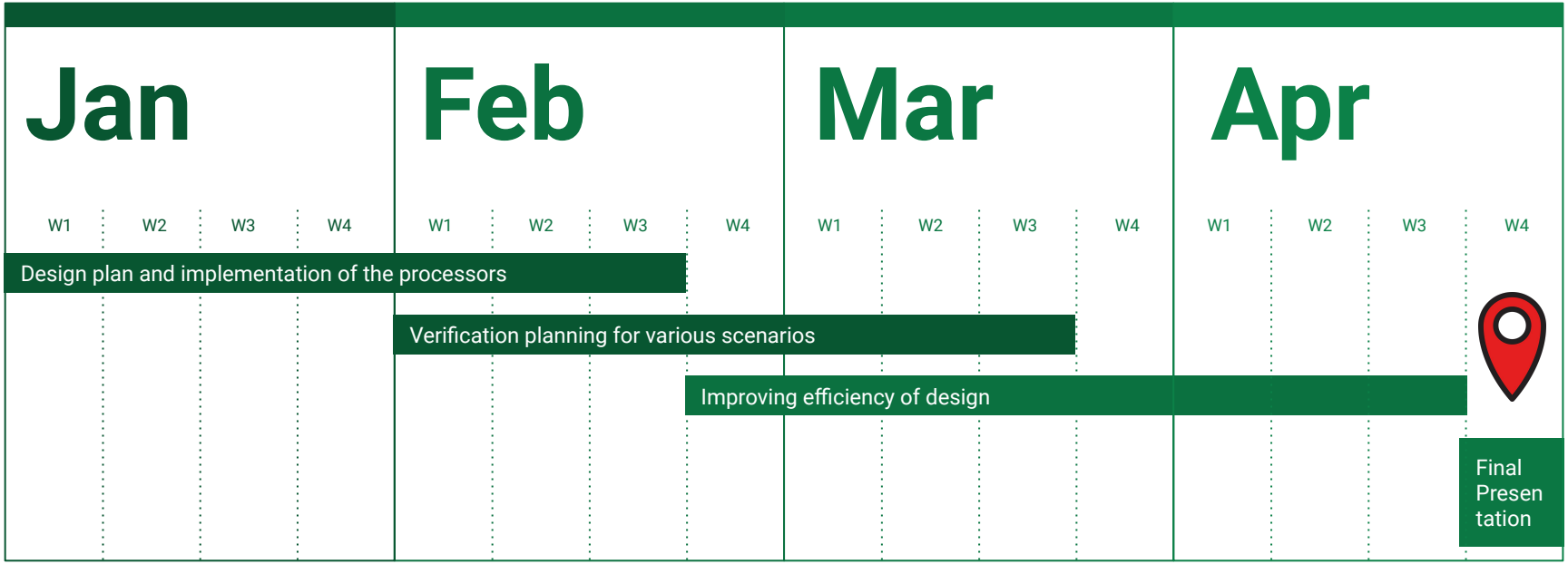
[5] *MiBench: A free, commercially representative embedded benchmark suite*

[6] *https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf*

[7]*https://www.chegg.com/homework-help/questions-and-answers/use-provided-set-risc-v-instructions-seen-answer-following-problem-q32767781*

[8]*https://godbolt.org/*

GANTT CHART


Jan
Feb
Mar
Apr
W1
W2
W3
W4
W1
W2
W3
W4
W1
W2
W3
W4
W1
W2
W3
W4
Design plan and implementation of the processors
Verification planning for various scenarios
Improving efficiency of design
Final Presentation

SUBMISSIONS

Project Report -YES
Project Poster - YES
Project Video - YES
Research Paper - NA
Individual Contribution Statement - YES

PES UNIVERSITY

THANK YOU