Processor Design

Spencer Melnick

December 19, 2018

Contents

1 Introduction

1.1 Objective

The objective of this project is to design a simple, single-cycle processor capable of running a very basic program with a small number of instructions.

1.2 Tools

This processor design was created and tested using Verilog in both Icarus Verilog and ISE Studio. Diagrams were produced using the draw.io online flowchart software. Images were edited in GIMP, and this report was created using LaTeX.

2 Modules

2.1 Adder

The adder used in this processor is a simple 32 bit ripple-carry adder. The 32 bit adder is created using a series of smaller adder modules including:

- Half Adder
- Full Adder
- 2 Bit Adder
- 4 Bit Adder
- 16 Bit Adder
- 32 Bit Adder

2.1.1 Half Adder

2.1.1.1 Inputs

- Operand A - 1 bit
- Operand B - 1 bit

2.1.1.2 Outputs

- Sum - 1 bit
- Carry - 1 bit

2.1.1.3 Functionality

Takes two single bit operands and produces the single bit sum and the carry bit.

2.1.1.4 Diagrams

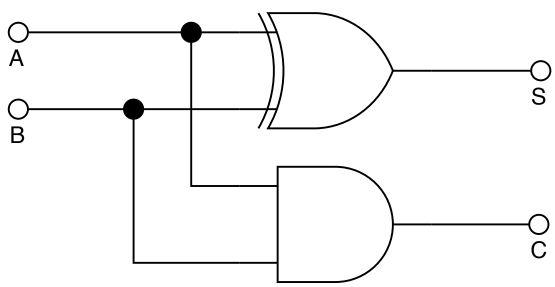

Figure 1: Logic diagram of the half adder

2.1.1.5 Testing

All possible inputs for the half adder are stimulated by the test bench and are compared to the expected outputs according to the following table:

A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

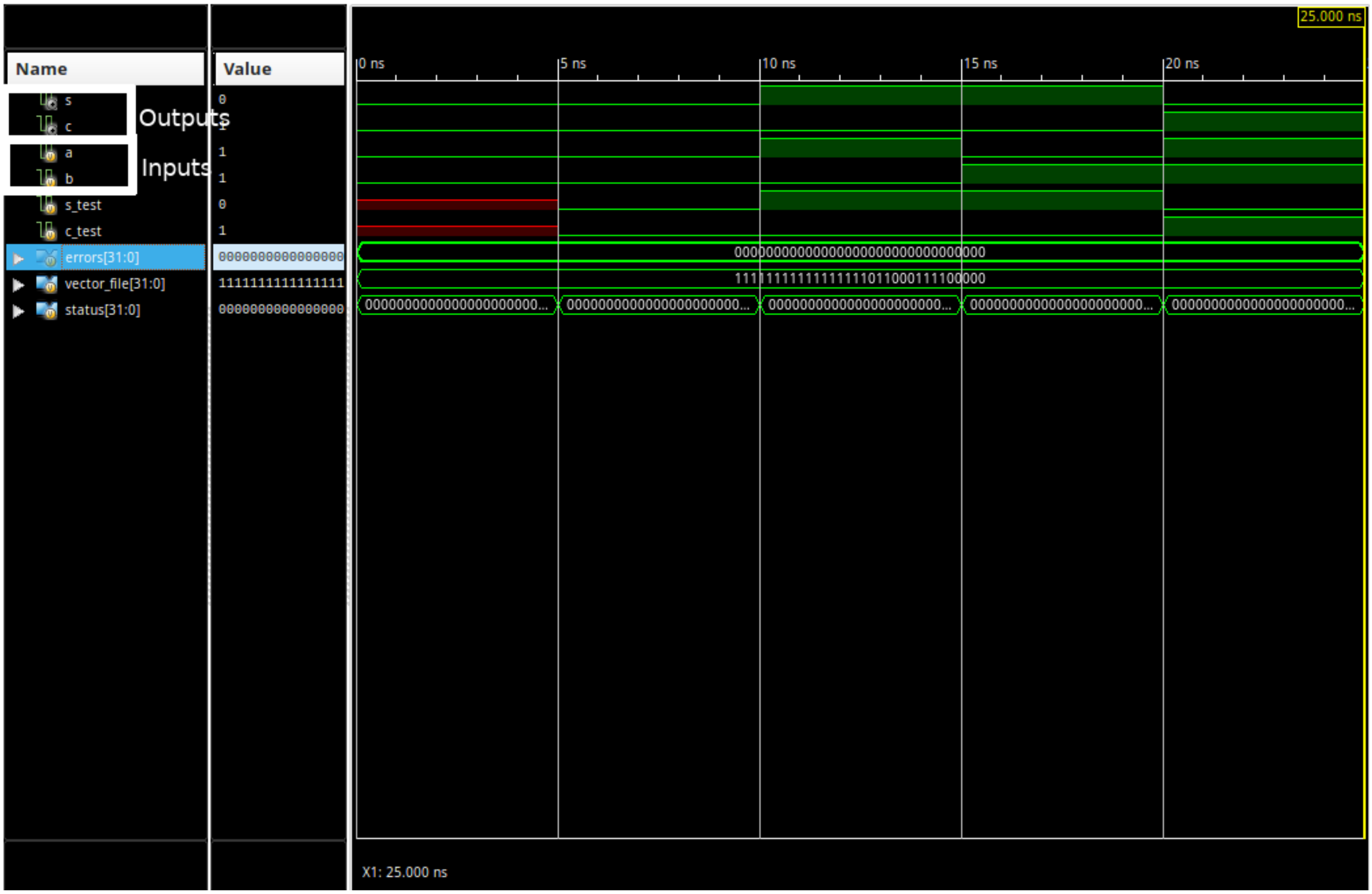

Figure 2: Simulation output of half adder

2.1.2 Full Adder

2.1.2.1 Inputs

- Operand A - 1 bit
- Operand B - 1 bit
- Carry In - 1 bit

2.1.2.2 Outputs

- Sum - 1 bit
- Carry Out - 1 bit

2.1.2.3 Functionality

Takes three single bit operands and produces the single bit sum and the carry bit.

2.1.2.4 Diagrams

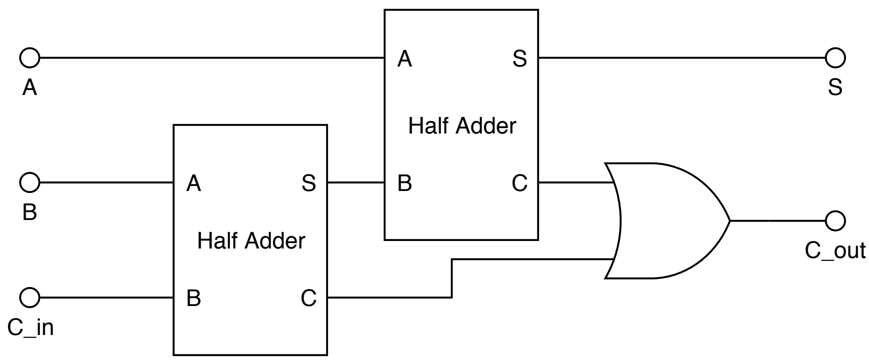

Figure 3: Logic diagram of the full adder

2.1.2.5 Testing

All possible inputs for the full adder are stimulated by the test bench and are compared to the expected outputs according to the following table:

A	B	Carry In	S	Carry Out
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Figure 4: Simulation output of full adder

2.1.3 2 Bit Adder

2.1.3.1 Inputs

- Operand A - 2 bit
- Operand B - 2 bit
- Carry In - 1 bit

2.1.3.2 Outputs

- Sum - 2 bit
- Carry Out - 1 bit

2.1.3.3 Functionality

Takes two 2 bit operands and a third single bit operand and produces the 2 bit sum and the carry bit.

2.1.3.4 Diagrams

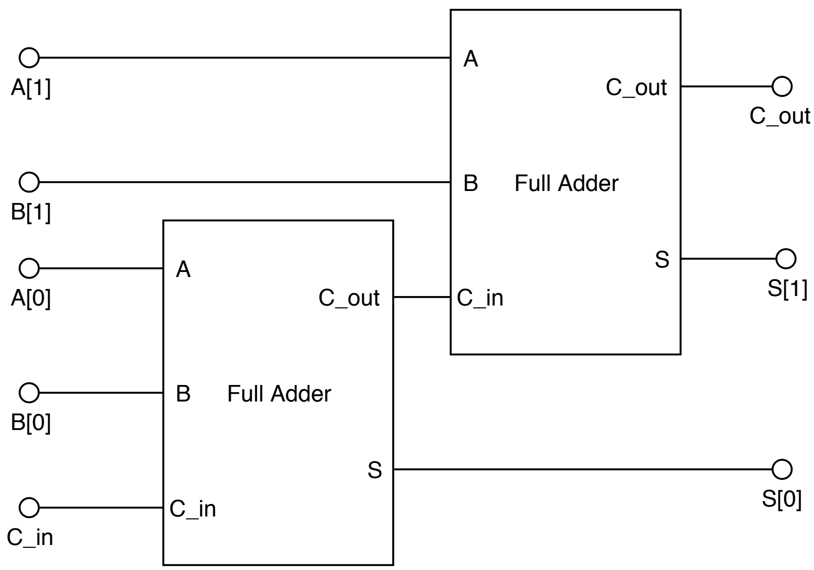

Figure 5: Logic diagram of the 2 bit adder

2.1.3.5 Testing

Due to the larger range of possible inputs, inputs are selected to include several base cases, as well as the maximum input value case and minimum input value case. The inputs and expected outputs are as follows:

A	B	Carry In	S	Carry Out
3	3	0	2	1
3	1	0	0	1
3	3	1	3	1
0	0	0	0	0

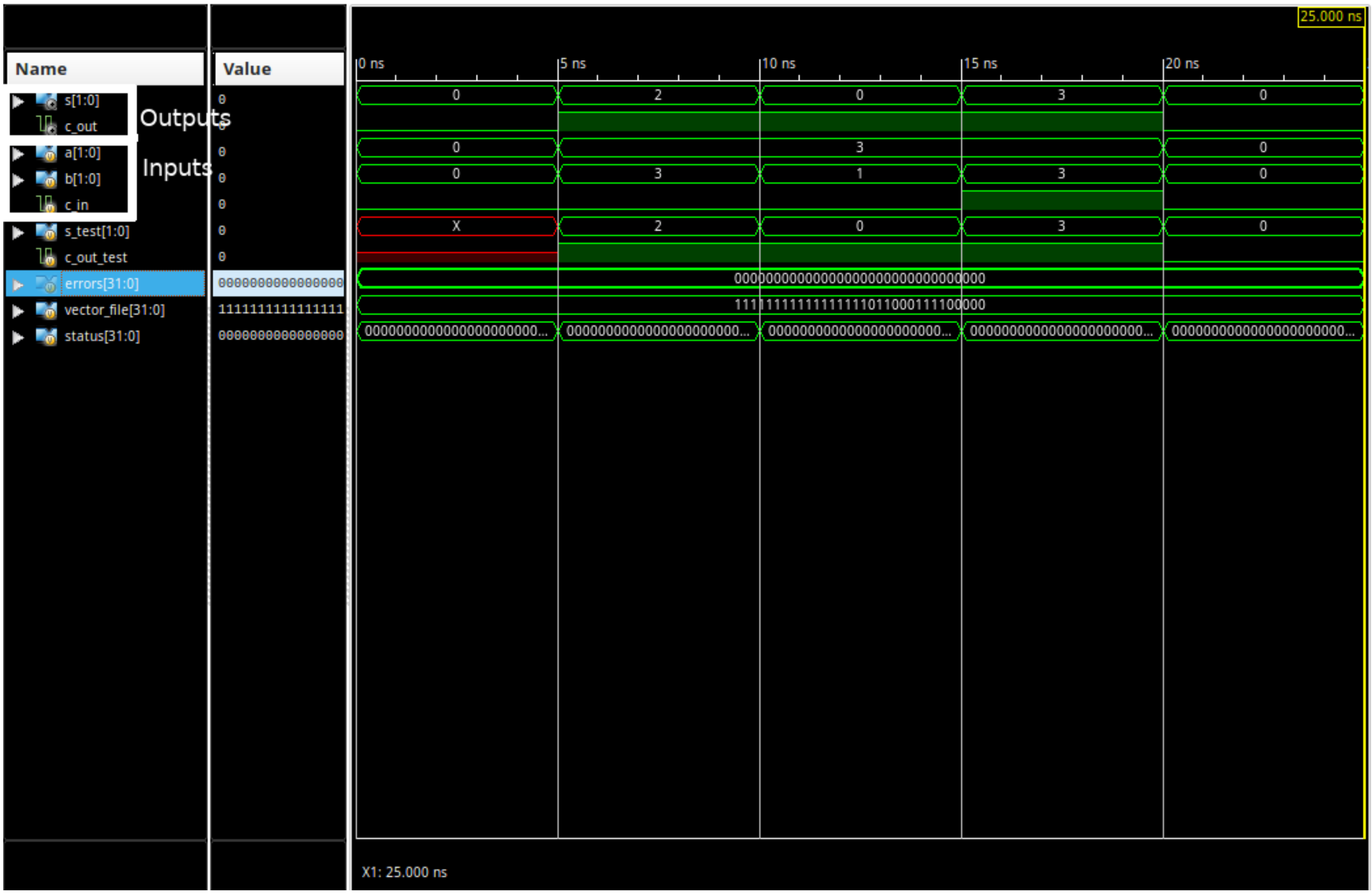

Figure 6: Simulation output of 2 bit adder

2.1.4 4 Bit Adder

2.1.4.1 Inputs

- Operand A - 4 bit
- Operand B - 4 bit
- Carry In - 1 bit

2.1.4.2 Outputs

- Sum - 4 bit
- Carry Out - 4 bit

2.1.4.3 Functionality

Takes two 4 bit operands and a third single bit operand and produces the 4 bit sum and the carry bit.

2.1.4.4 Diagrams

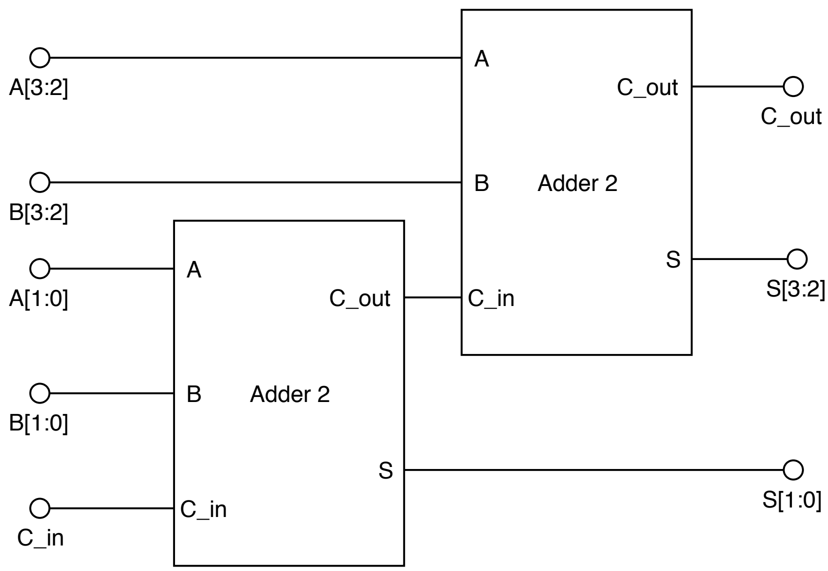

Figure 7: Logic diagram of the 4 bit adder

2.1.4.5 Testing

As this module follows a nearly identical functionality as the 2 bit adder, and nearly identical Verilog code, testing was ommitted. This module's test is included within the 32 bit adder test, as the 32 bit adder will only function correctly if this module functions correctly.

2.1.5 8 Bit Adder

2.1.5.1 Inputs

- Operand A - 8 bit
- Operand B - 8 bit
- Carry In - 1 bit

2.1.5.2 Outputs

- Sum - 8 bit
- Carry Out - 8 bit

2.1.5.3 Functionality

Takes two 8 bit operands and a third single bit operand and produces the 8 bit sum and the carry bit.

2.1.5.4 Diagrams

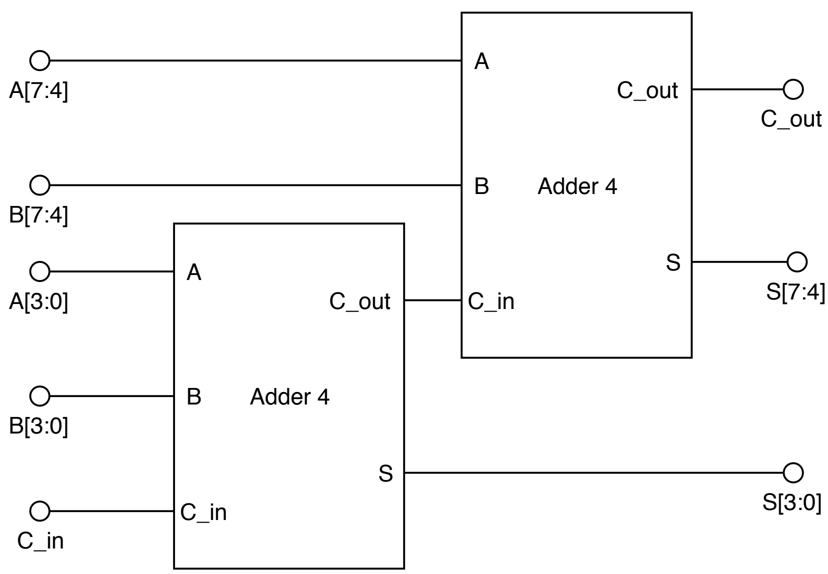

Figure 8: Logic diagram of the 8 bit adder

2.1.5.5 Testing

As this module follows a nearly identical functionality as the 2 bit adder, and nearly identical Verilog code, testing was ommitted. This module's test is included within the 32 bit adder test, as the 32 bit adder will only function correctly if this module functions correctly.

2.1.6 16 Bit Adder

2.1.6.1 Inputs

- Operand A - 16 bit
- Operand B - 16 bit
- Carry In - 1 bit

2.1.6.2 Outputs

- Sum - 16 bit
- Carry Out - 16 bit

2.1.6.3 Functionality

Takes two 16 bit operands and a third single bit operand and produces the 16 bit sum and the carry bit.

2.1.6.4 Diagrams

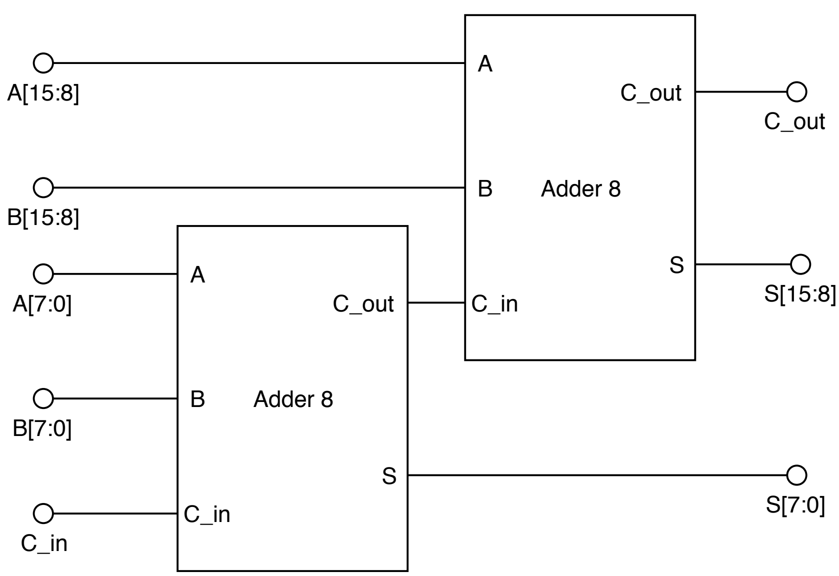

Figure 9: Logic diagram of the 16 bit adder

2.1.6.5 Testing

As this module follows a nearly identical functionality as the 2 bit adder, and nearly identical Verilog code, testing was ommitted. This module's test is included within the 32 bit adder test, as the 32 bit adder will only function correctly if this module functions correctly.

2.1.7 32 Bit Adder

2.1.7.1 Inputs

- Operand A - 32 bit
- Operand B - 32 bit
- Carry In - 1 bit

2.1.7.2 Outputs

- Sum - 32 bit
- Carry Out - 32 bit

2.1.7.3 Functionality

Takes two 32 bit operands and a third single bit operand and produces the 32 bit sum and the carry bit.

2.1.7.4 Diagrams

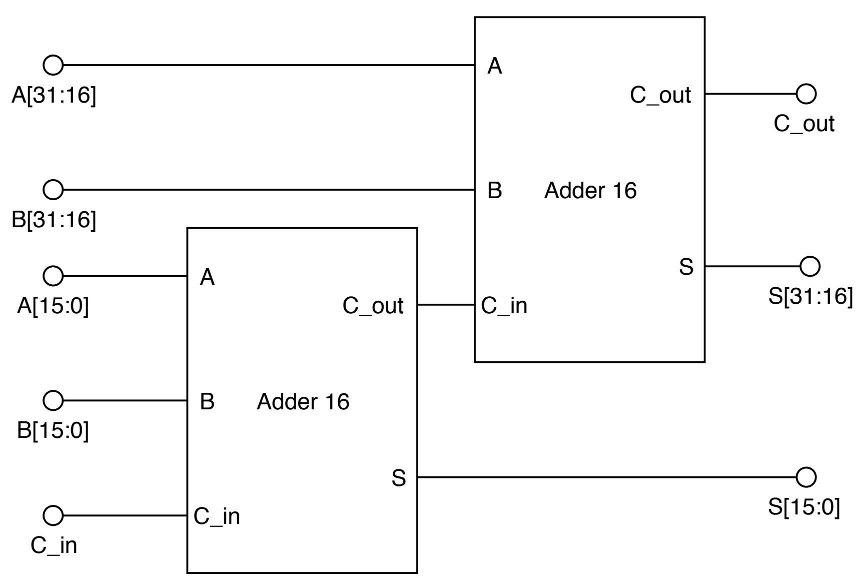

Figure 10: Logic diagram of the 32 bit adder

2.1.7.5 Testing

Due to the larger range of possible inputs, inputs are selected to include several base cases, as well as the maximum input value case. The inputs and expected outputs are as follows:

A	B	Carry In	S	Carry Out
10	10	1	21	0
1	0	1	2	0
4294967295	0	1	0	1
4294967295	4294967295	1	4294967295	1

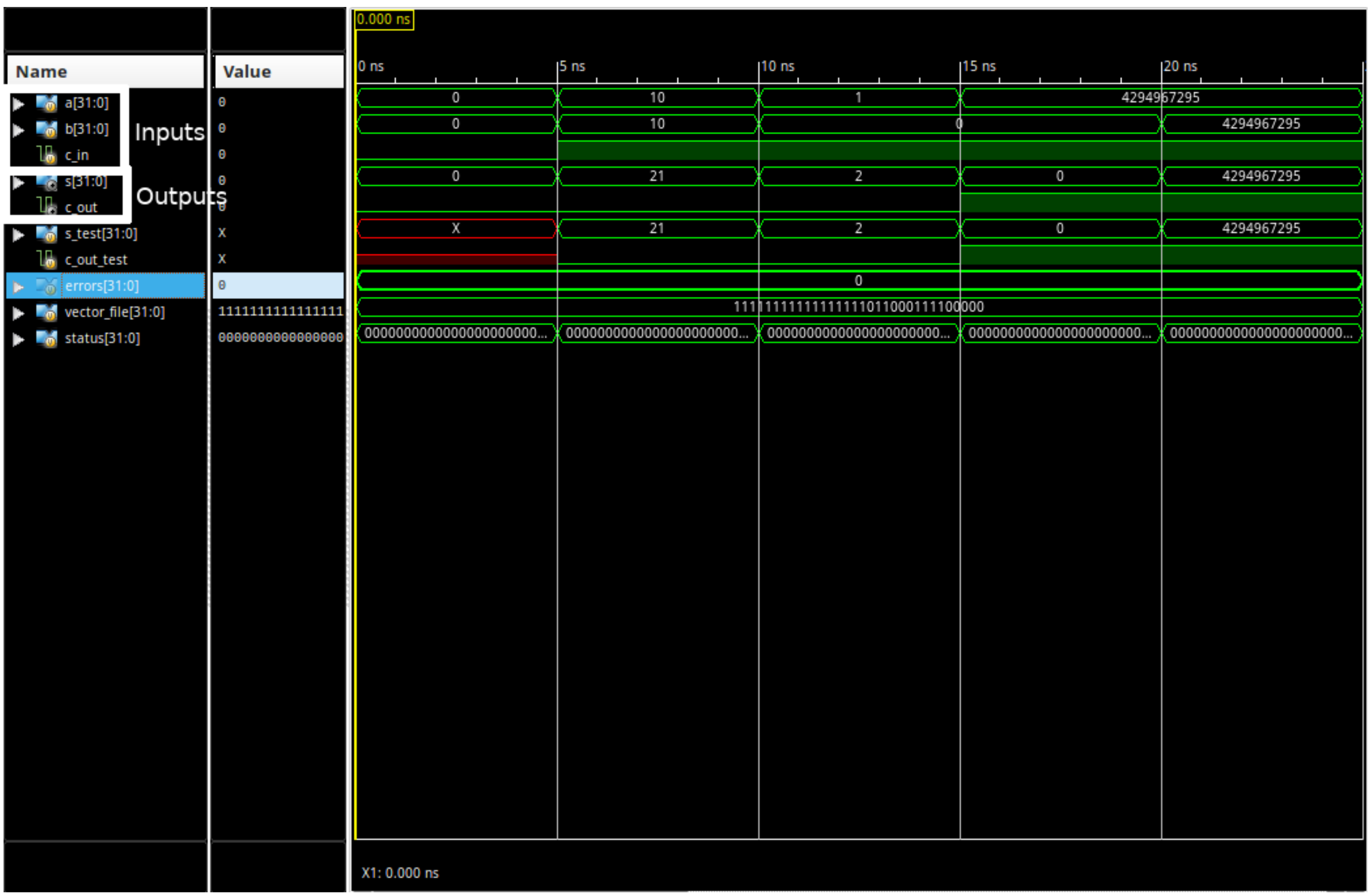

Figure 11: Simulation output of 32 bit adder

2.2 32 Bit Multiplier

2.2.0.1 Inputs

- Operand A - 32 bit
- Operand B - 32 bit
- Clock Signal - 1 bit
- Enable Signal - 1 bit
- Reset Signal - 1 bit

2.2.0.2 Outputs

- Product - 64 bit
- Done Signal - 1 bit

2.2.0.3 Functionality

The multiplier module uses a simple multiplication algorithm, optimized to use the minimum number of gates. As a consequence, the multiplier module takes multiple clock cycles to complete an operation.

On the positive edge of the reset signal, the values of operand A and operand B are stored by the multiplier and the multiplication is reset, but no actual multiplication is performed.

If the enable signal is high, then on the positive edge of the clock signal the module will complete one cycle of the multiplication operation. During this time the product output will change, but it will not be the correct value.

Once the multiplication is complete, the output done signal will be driven high and the product will be accurate.

This module supports signed multiplication.

2.2.0.4 Diagrams

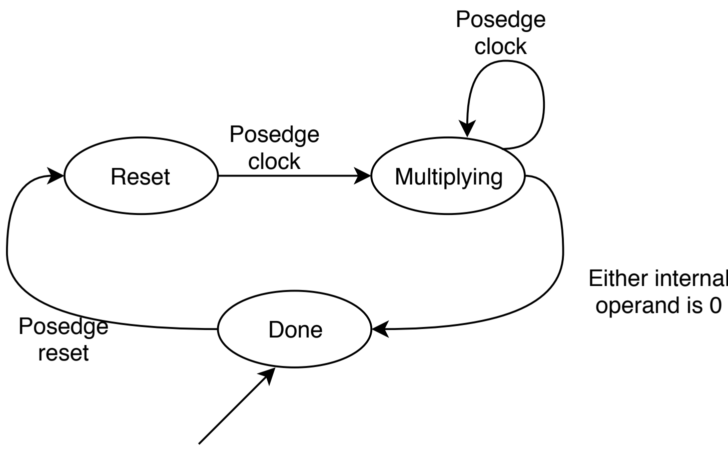

Figure 12: State change diagram of the 32 bit multiplier

2.2.0.5 Testing

A few base cases are tested with positive and negative number multiplication, multiplication by zero, and the largest positive inputs and largest negative inputs are tested. Overflow is not tested, as with a 64 bit product, overflow is impossible. The inputs and expected outputs are as follows:

A	B	Product
5	10	50
7	5	35
0	0	0
1	0	0
1	1	1
1	-1	-1
-1	-1	1
2147483647	2147483647	4611686014132420609
-2147483648	-2147483648	4611686018427387904

Figure 13: Simulation output of 32 bit multiplier

2.3 32 Bit Divider

2.3.0.1 Inputs

- Dividend - 32 bit
- Divisor - 32 bit
- Clock Signal - 1 bit
- Enable Signal - 1 bit
- Reset Signal - 1 bit

2.3.0.2 Outputs

- Quotient - 32 bit
- Remainder - 32 bit
- Done Signal - 1 bit

2.3.0.3 Functionality

The multiplier module uses a simple division algorithm, optimized to use the minimum number of gates. As a consequence, the divsion module takes multiple clock cycles to complete an operation.

On the positive edge of the reset signal, the values of the dividend and divisor are stored by the multiplier and the division is reset, but no actual division is performed.

If the enable signal is high, then on the positive edge of the clock signal the module will complete one cycle of the division operation. During this time the quotient output and remainder outputs will change, but it will not be the correct value.

Once the division is complete, the output done signal will be driven high and the quotient and remainder will be accurate.

This module supports signed division.

2.3.0.4 Diagrams

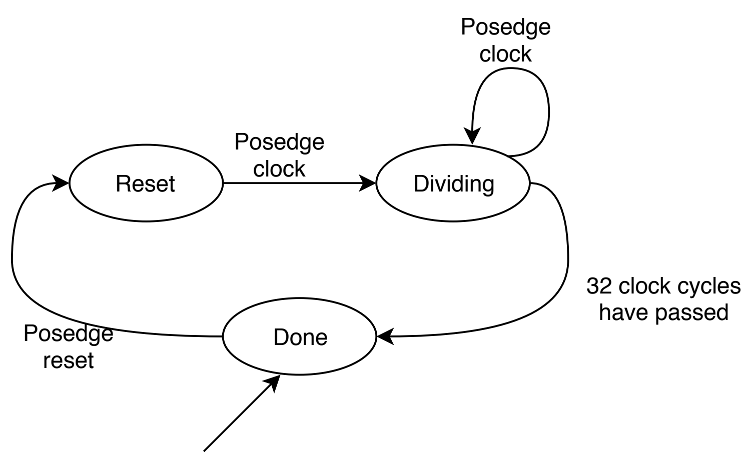

Figure 14: State change diagram of the 32 bit multiplier

2.3.0.5 Testing

A few base cases are tested with positive and negative number division inputs and largest negative inputs are tested. Division by 0 is not handled. The inputs and expected outputs are as follows:

A	B	Quotient	Remainder
100	25	4	0
175	100	1	75
10	10000	0	10
0	1	0	0
1	1	1	0
-1	1	-1	0
-1	-1	1	0

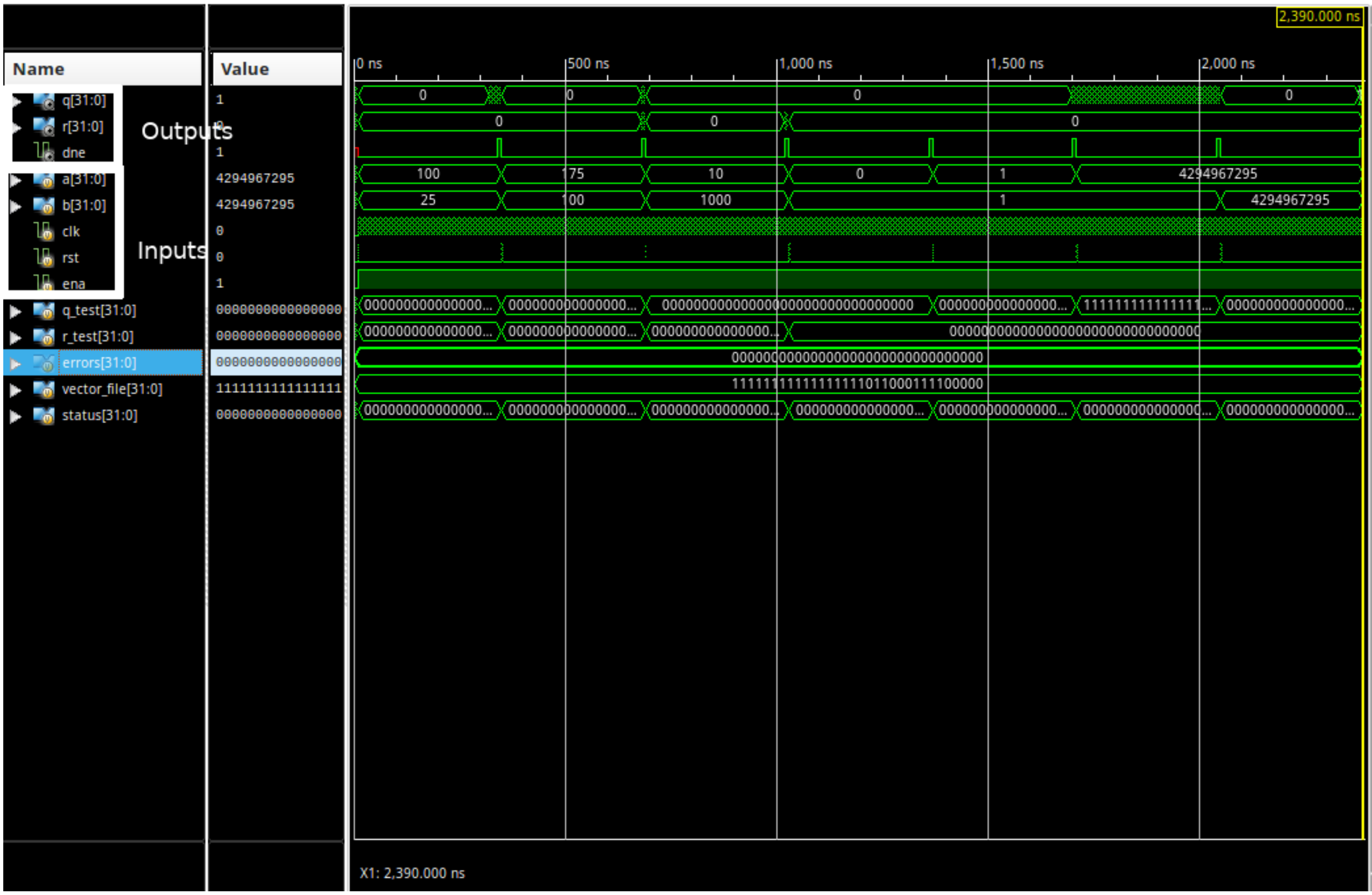

Figure 15: Simulation output of 32 bit multiplier

2.4 32 Bit ALU

2.4.0.1 Inputs

- Operand A - 32 bit
- Operand B - 32 bit
- Operation - 3 bit
- Clock Signal - 1 bit
- Enable Signal - 1 bit
- Reset Signal - 1 bit

2.4.0.2 Outputs

- Result - 32 bit
- Extra - 32 bit
- Done Signal - 1 bit

2.4.0.3 Functionality

The 32 bit ALU utilizes the 32 bit adder, multiplier, and divider modules described above. A multiplexer is used to select the appropriate output from the module, depending on the operation code used.

As this module implements a multiplier and divider that take several clock cycles to complete, this ALU requires a reset and clock signal. On the positive edge of the reset signal the values of the operands are stored and the reset for the arithmetic submodule is triggered.

Because a multiplexer is used, the operation code must remain constant throughout the duration of the operation. Like the multiplier and divider, a done signal is driven high on the completion of any operation.

The lower 32 bits of the operation result are stored in the result output, while any additional operation results are stored in the extra output. This includes the carry/borrow bit from the addition/subtraction operation, the upper 32 bits of the product, or the 32 bits of the remainder.

The subtraction operation is implemented by taking the inverse of the second operand and driving the adder's carry in high.

The different codes for operations are as follows:

Operation	Code
Add	0
Subtract	1
Multiply	2
Divide	3

2.4.0.4 Diagrams

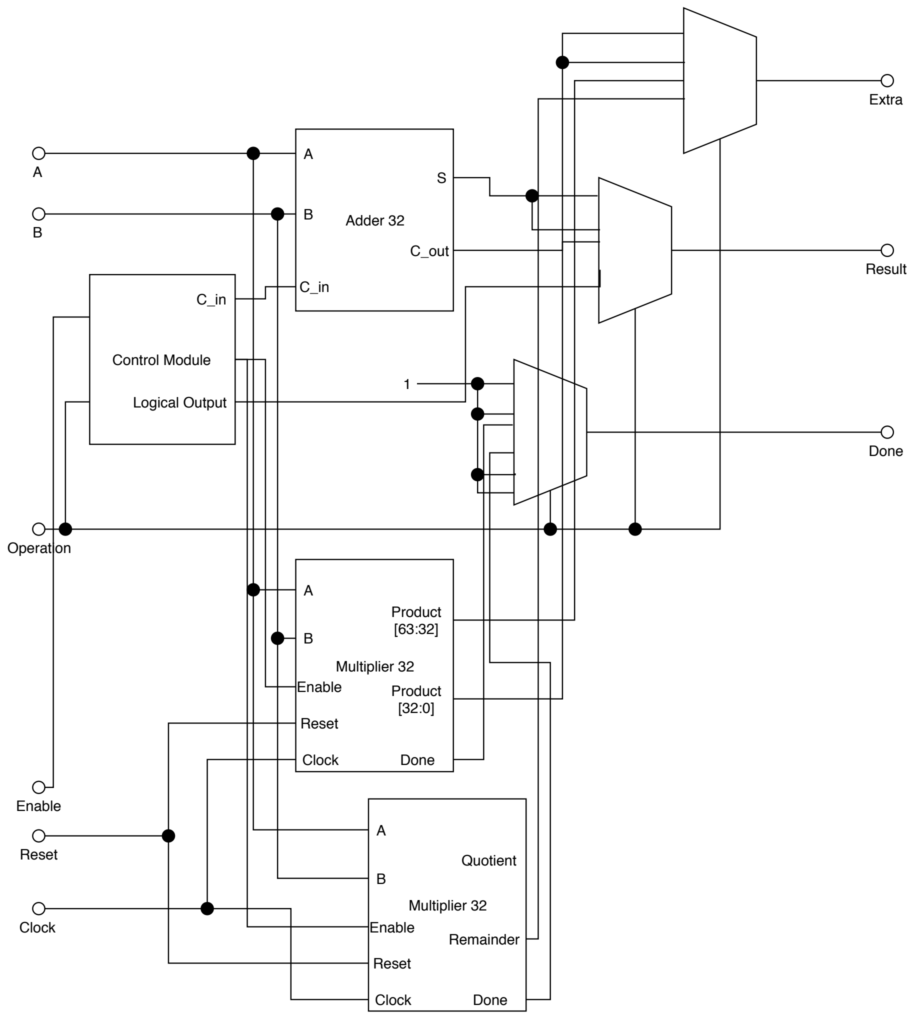

Figure 16: Logical diagram of the 32 bit ALU

2.4.0.5 Testing

As each submodule has been tested individually, only one of each type of operation is tested. The inputs and expected outputs are as follows:

A	B	Operation	Result	Extra
1	2	0	3	0
3	4	1	-1	0
5	6	2	30	0
7	8	3	0	7

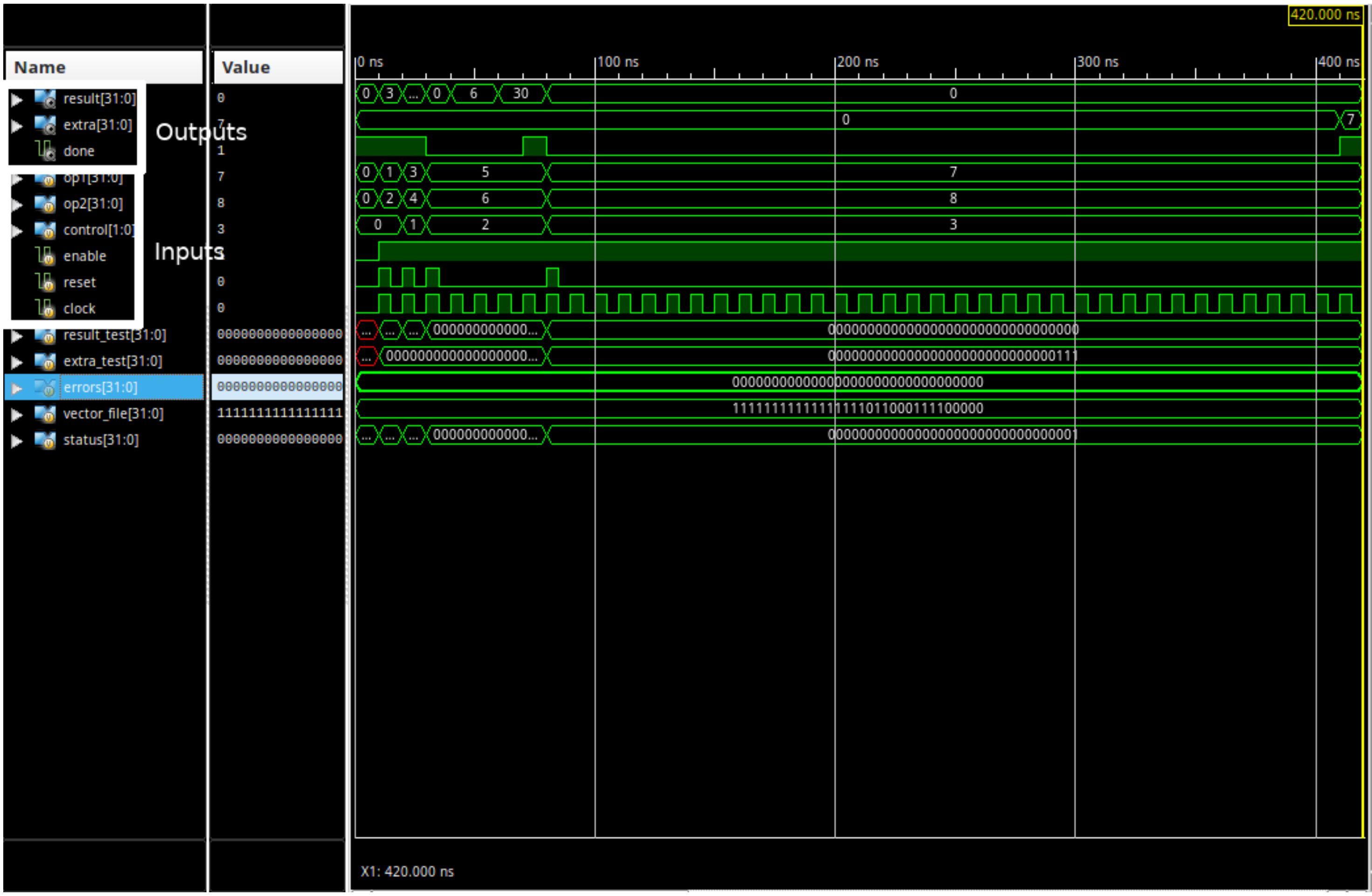

Figure 17: Simulation output of 32 bit ALU

2.5 Left Shift

2.5.0.1 Inputs

- Data In - Parameterized Width

2.5.0.2 Outputs

- Data Out - Parameterized Width

2.5.0.3 Functionality

The left shift module is a parameterized module, meaning that it has variable width inputs and outputs, as well as variable length shift. These parameters are set during instantiation of the module.

2.5.0.4 Testing

A 32 bit left shifter, with a shift amount of 2 is tested. The following cases are tested:

Input	Output
00000000000000000000000000000000	00000000000000000000000000000000
00000000000000000000000000000001	00000000000000000000000000000100
10000000000000000000000000000000	00000000000000000000000000000000
11000000000000000000000000000000	00000000000000000000000000000000

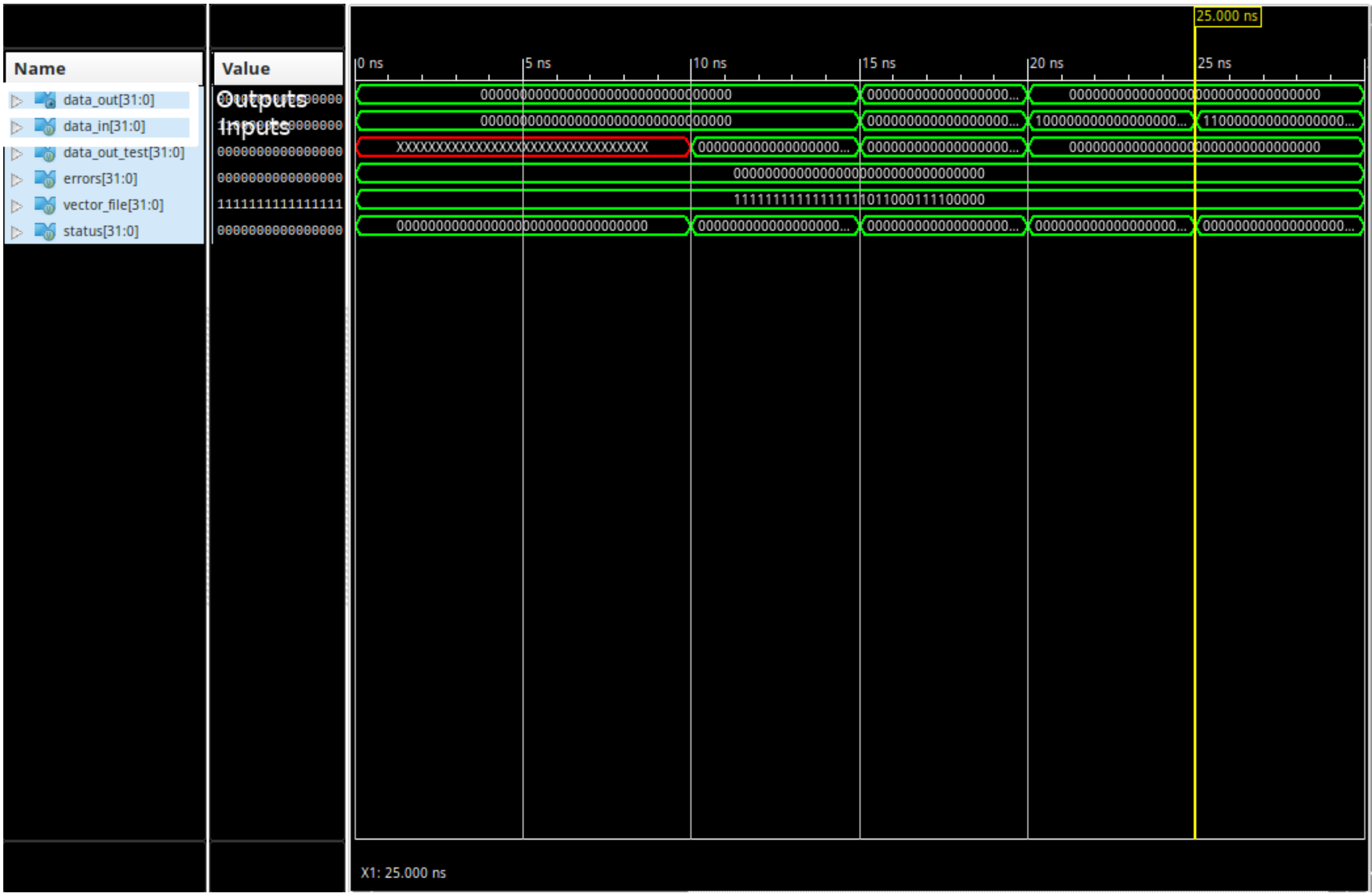

Figure 18: Simulation output of 32 bit left shifter

2.6 4 Input Multiplexer

2.6.0.1 Inputs

- Data In 1 - Parameterized Width
- Data In 2 - Parameterized Width
- Data In 3 - Parameterized Width
- Data In 4 - Parameterized Width
- Select - 2 bit

2.6.0.2 Outputs

- Data Out - Parameterized Width

2.6.0.3 Functionality

The 4 input multiplexer is a parameterized module, meaning that it has variable width inputs and outputs. These parameters are set during instantiation of the module.

This module selects one of four inputs based upon the select signal.

2.6.0.4 Testing

A 32 bit 4 input multiplexer is tested using the following cases:

Data 1	Data 2	Data 3	Data 4	Select	Output
3	4	5	6	0	3
3	4	5	6	1	4
3	4	5	6	2	5
3	4	5	6	3	6
3	4	5	7	3	7

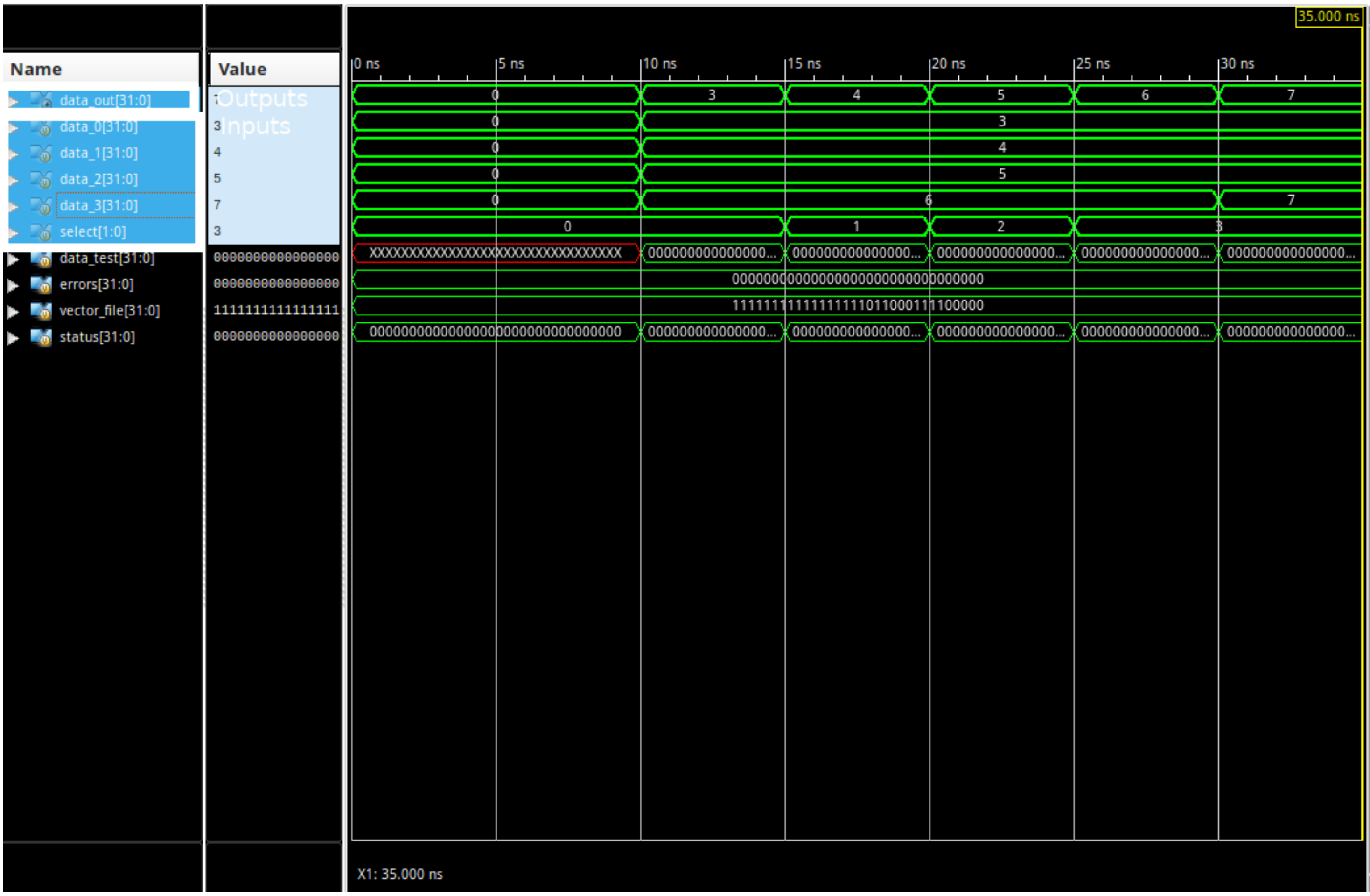

Figure 19: Simulation output of 32 bit 4 input multiplexer

2.7 Sign Extender

2.7.0.1 Inputs

- Data In - Parameterized Width

2.7.0.2 Outputs

- Data Out - Parameterized Width

2.7.0.3 Functionality

The sign extender is a parameterized module, meaning that it has variable width inputs and outputs. These parameters are set during instantiation of the module.

This module extends the width of the input data to defined output width, maintaining the two's complement sign of the data.

2.7.0.4 Testing

A 16 bit to 32 bit sign extender is tested using the following cases:

Data In [15:0]	Data Out [31:0]
15	15
-15	-15
255	255
-255	-255
0	0

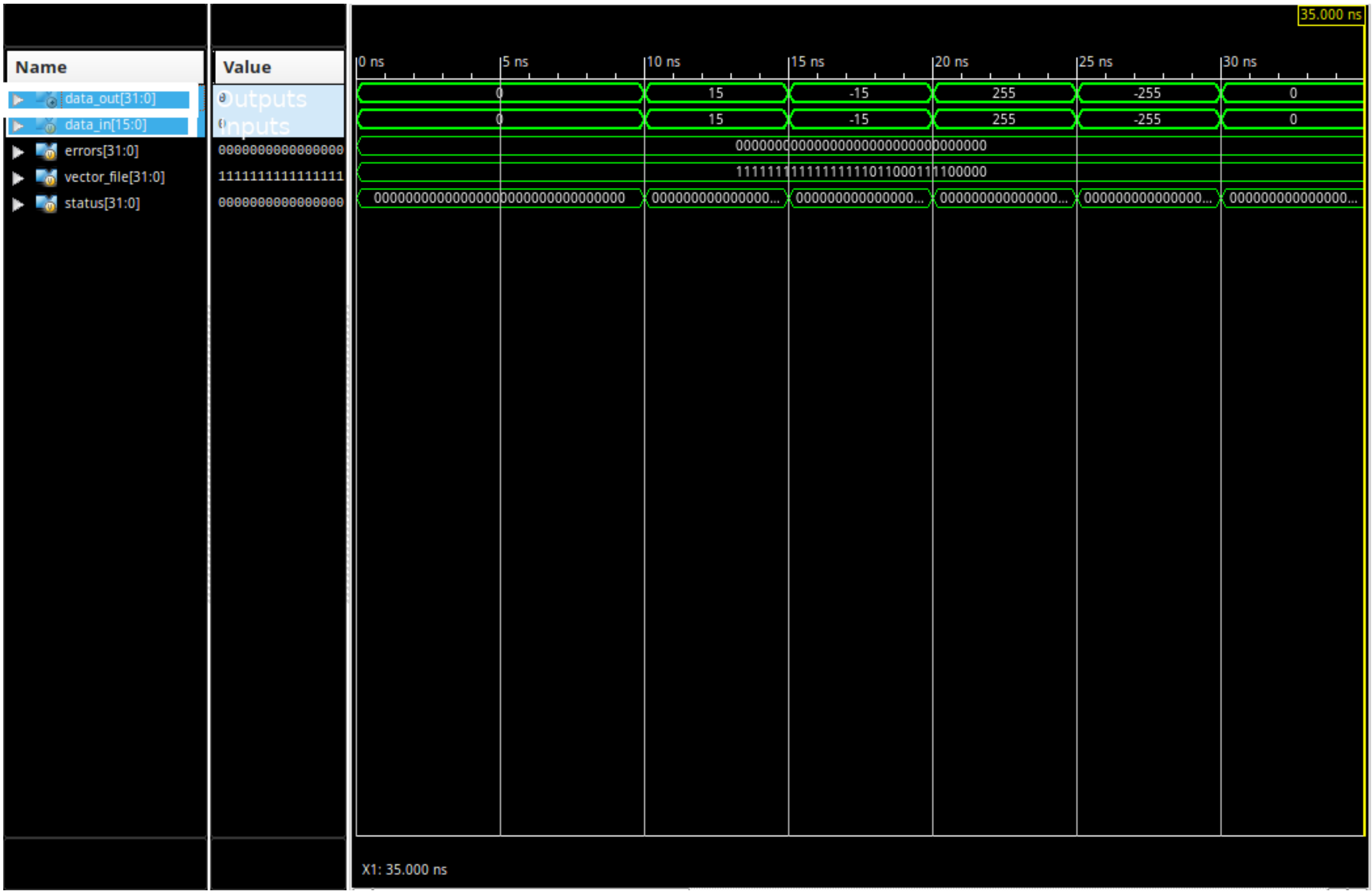

Figure 20: Simulation output of 16 bit to 32 bit sign extender

2.8 Register

2.8.0.1 Inputs

- Read 1 Address - 5 bit
- Read 2 Address - 5 bit
- Write Enable - 1 bit
- Write Address - 5 bit
- Data In - 32 bit

2.8.0.2 Outputs

- Data Out 1 - 32 bit
- Data Out 2 - 32 bit

2.8.0.3 Functionality

The register module stores 32 different 32 bit values. These values are automatically read when either read address is changed. The address corresponds to the register index.

On the positive edge of the write enable signal, the data from data in is written to the specified write address.

2.8.0.4 Testing

The register module is tested by first reading the first register to ensure that all registers are correctly initialized. Various values are written to different registers and then read again to ensure that the values were correctly written. The test cases are sequentially as follows:

Read 1 Address	Read 2 Address	Write Enable	Write Address	Data In
00	00	0	00	00000000
00	00	1	01	ffffffff
00	01	0	00	00000000
01	00	0	02	dddddddd
02	01	0	00	00000000
01	01	1	02	bbbbbbbb
02	01	0	00	00000000

Data 1 Out	Data 2 Out
00000000	00000000
00000000	00000000
00000000	ffffffff
ffffffff	00000000
00000000	ffffffff
ffffffff	ffffffff
bbbbbbbb	ffffffff

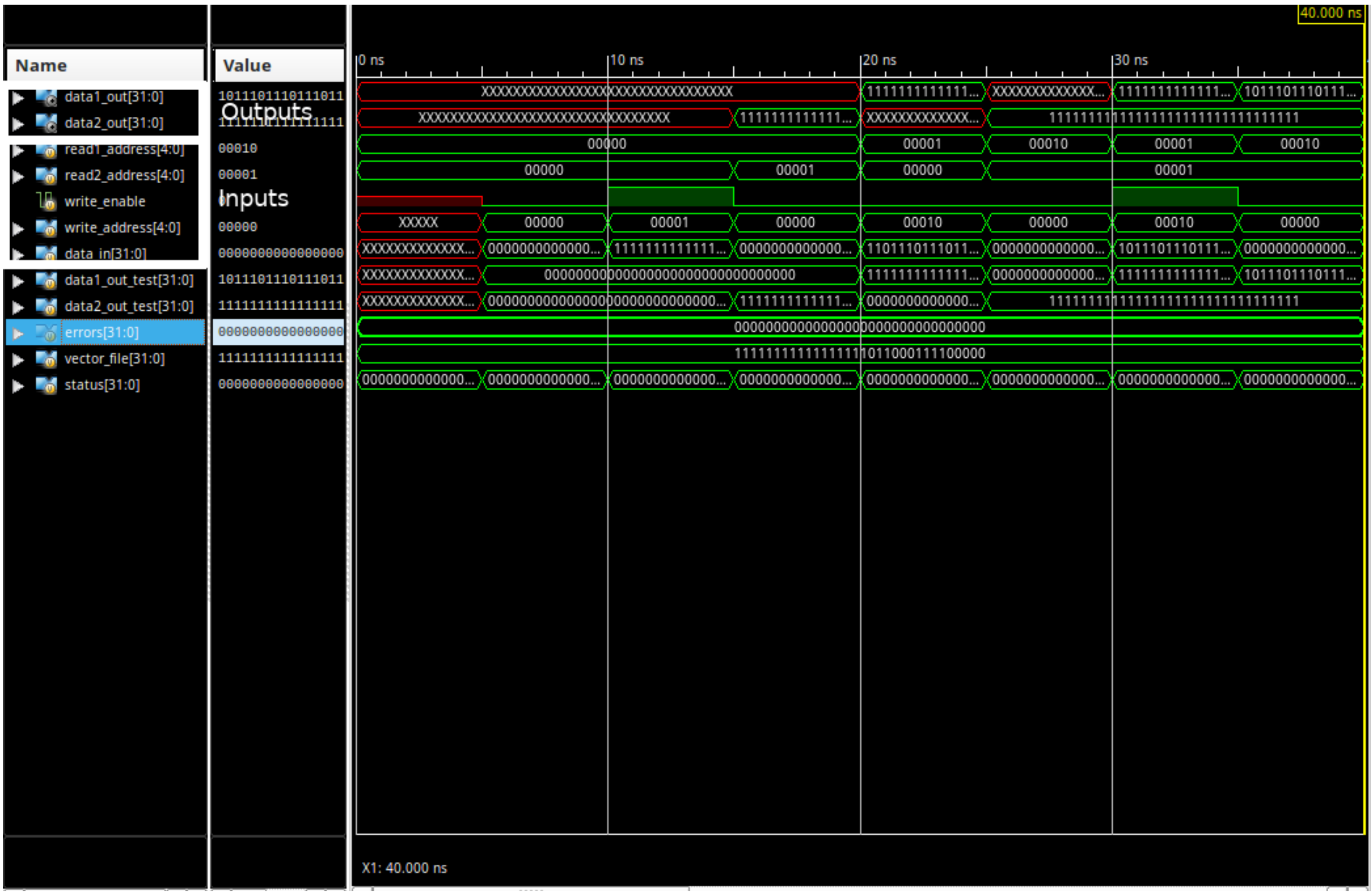

Figure 21: Simulation output of the register module