Indian Institute of Technology Delhi

Department of Electrical Engineering

Assignment-2

ALU Design for the SimpleRISC Core

ELL7282: Application Specific Computer Architectures

Submitted to: Prof. Kaushik Saha

Student Names:	Raj Fauzdar, Himanshu Suryawanshi
Entry No.:	2025EET2733, 2025EET2484
Submission Date:	25 October 2025

Contents

1 Introduction

The Arithmetic Logic Unit (ALU) is the core computational component of any processor, responsible for executing arithmetic and logical operations on binary data. In the SimpleRISC architecture, the ALU plays a critical role in performing operations such as addition, subtraction, logical operations, comparison, multiplication, and division within a single processor cycle.

This project focuses on designing and implementing a synthesizable, high-performance ALU for the single-cycle SimpleRISC processor core. The primary objective is to replace the placeholder ALU provided in the existing RTL with an optimized version that meets the 250 MHz performance target under standard synthesis conditions. The design must maintain correctness across all defined instruction operations while balancing speed, area, and power efficiency.

The implemented ALU supports 14 operations as defined in the SimpleRISC ISA — including arithmetic (ADD, SUB, MUL, DIV, MOD), logical (AND, OR, XOR, NOT), comparison (SLT), data transfer (PASS), and various shift operations (SLL, SRL, SRA). Each operation is triggered through a 4-bit control signal (alu_op) and produces a 32-bit result (y) along with a zero flag for conditional branching support.

To ensure accuracy and robustness, each operation was implemented using separate Verilog modules (adder, subtractor, multiplier, divider, shifters, etc.) and integrated within the top-level ALU. The final design was verified using an extensive testbench and assembler-based test programs, including custom test cases covering corner scenarios such as signed arithmetic, overflow behavior, and divide-by-zero handling.

2 Design and Algorithms

The ALU is designed as a single-cycle, fully combinational circuit, ensuring that all arithmetic and logical operations are completed within one clock cycle. Each operation including addition, subtraction, logical functions, shifts, multiplication, and division is implemented as a separate combinational module. The top-level ALU uses a 4-bit control signal (op) to select the required operation and generate the corresponding 32-bit output.

This single-cycle design eliminates the need for pipelining or sequential control, simplifying integration with the SimpleRISC processor core while maintaining high performance. The combinational structure enables parallel computation paths for different operation.

2.1 Addition

The adder module is implemented as a 32-bit Kogge–Stone parallel prefix adder, designed for high-speed operation in a single-cycle combinational ALU. It takes two N-bit operands (a, b) and a carry-in (cin) to produce an N-bit sum output.

The Kogge–Stone structure computes generate (G) and propagate (P) signals in parallel, using a logarithmic carry computation tree. This enables all carry bits to be determined in approximately O(logN) time, compared to the O(N) delay of a ripple-carry adder. The module uses five hierarchical prefix stages, each doubling the carry computation span, which drastically reduces propagation delay and supports high-frequency operation.

Compared to the Brent–Kung adder, which also uses a parallel prefix approach but has fewer interconnects and lower area, the Kogge–Stone adder achieves faster carry resolution at the cost of slightly higher wiring complexity. Brent–Kung is more area-efficient but slower, as it has greater logic depth. Since our ALU is designed for a single-cycle SimpleRISC core operating at 250 MHz, speed is the dominant factor — making the Kogge–Stone adder the preferred choice.

This adder's fully combinational and parallel nature ensures minimal critical path delay, allowing arithmetic operations to complete within a single processor cycle. Its modular, parameterized Verilog implementation also allows scalability for other word sizes without architectural changes.

In summary, the Kogge–Stone adder provides a superior trade-off of speed over area, perfectly aligning with the performance-centric requirements of the SimpleRISC ALU design.

2.2 Substraction

The subtraction module performs a 32-bit subtraction (a - b) operation using two's complement arithmetic. Instead of implementing a separate subtractor, the design reuses the existing Kogge–Stone adder to achieve efficient subtraction in a single combinational cycle.

The subtraction is based on the equation:

A - B = A + (not(B) + 1)

Here, the operand b is first inverted using the Not module, generating its one's complement. The resulting value is then added to operand a along with a carry-in of 1 through the adder module.

This approach ensures that subtraction is implemented purely through addition logic, reducing hardware complexity while maintaining consistent delay and structure across arithmetic operations. Since both modules are fully combinational, the result is obtained within a single clock cycle, meeting the timing requirements of the SimpleRISC single-cycle ALU.

2.3 Set Less Than

The Set Less Than (SLT) module is designed to perform signed comparison between two 32-bit operands a and b. It outputs 1 if a is less than b, and 0 otherwise. The result is stored in the least significant bit (LSB) of the 32-bit output z, with all other bits implicitly zero.

The module internally reuses the subtraction module to compute the difference a - b. Based on the sign of the result and the sign bits of the operands, it determines which operand is smaller.

To handle both positive and negative operands correctly, the module first checks whether the signs of a and b differ using an XOR operation (diffsign = a[31] xor b[31]).

If the signs differ, the sign of a directly determines the result — a negative a means a is less thann b.

If the signs are the same, the sign of the subtraction result (diff[31]) indicates whether a is less than b.

This logic allows the SLT module to accurately compare signed integers using purely combinational logic, ensuring single-cycle operation. The reuse of the subtraction module makes the design hardware-efficient and consistent with other ALU operations.

2.4 Logical Operations

For these logical operations—XOR, AND, OR, NOT—we don't need any iterative or sequential algorithm because:

Direct Hardware Mapping: Each operation maps directly to simple logic gates:

XOR $\rightarrow$ XOR gates

AND $\rightarrow$ AND gates

OR $\rightarrow$ OR gates

NOT $\rightarrow$ Inverters

Parallel Execution: Modern FPGA synthesis tool automatically implements these bitwise operations in parallel across all 32 bits.

So z = a xor b is effectively 32 XOR gates working simultaneously.

There's no need for loops, shifts, or complex computation.

Efficiency: Using these operators ensures minimal delay and optimal resource usage on FPGA

2.5 Shifting Operations

Arithmetic Shift Right (ASR) shifts the bits of a number to the right while preserving the sign by replicating the most significant bit. The 32-bit input is shifted using a five-stage barrel shifter, where each stage shifts by powers of two (1, 2, 4, 8, 16) based on the corresponding bit of the 5-bit shift amount. This allows any shift from 0 to 31 positions in a single combinational cycle. Using a barrel shifter ensures high-speed parallel execution without iterative computation.

Logical Shift Right (LSR) shifts the bits of a number to the right while filling the vacated most significant bits with zeros, making it suitable for unsigned numbers. The LSR module also uses a five-stage barrel shifter, allowing all shifts from 0 to 31 bits in a single cycle. Logical shift right is commonly used in unsigned division, bit-field extraction, and masking. The barrel shifter implementation ensures efficient, parallel hardware execution.

Logical Shift Left (LSL) shifts the bits of a number to the left, filling the vacated least significant bits with zeros, effectively multiplying the number by powers of two. It uses a five-stage barrel shifter similar to ASR and LSR, allowing any shift from 0 to 31 positions in a single combinational operation.

In summary, all three shift operations are implemented using barrel shifters, providing single-cycle, fully parallel shifting for any number of positions between 0 and 31.

2.6 Multiplication

The multiplication module implements a 32-bit signed multiplication using radix-4 Booth encoding combined with a carry-save adder (CSA) tree for efficient summation of partial products. The module accepts two 32-bit inputs, a and b, and produces a 32-bit product z. Internally, all computations are performed using 64-bit signed arithmetic to accommodate intermediate results without overflow.

The first stage of the design is Booth encoding, which reduces the number of partial products by examining three bits of the multiplier at a time. The multiplier b is extended by one extra bit to simplify edge cases. For each 3-bit window, a selection signal trip is generated, which determines whether the corresponding partial product should be +A, +2A, -A, -2A, or zero. The module computes all necessary multiples of the multiplicand a in 64-bit representation. Specifically, m1 is A, m2 is 2A, mneg1 is -A, and mneg2 is -2A. Two's complement operations are implemented using a 64-bit NOT and adder unit. This approach allows efficient handling of both positive and negative numbers.

After the Booth encoding step, the partial products are generated. There are sixteen 3-bit groups, producing sixteen signed 64-bit partial products. Each selected multiple (sel[i]) is shifted left by 2*i bits to align it according to its weight in the multiplier. This alignment is performed using a 64-bit logical left shift module for each partial product. At the end of this stage, all sixteen partial products are correctly positioned to be summed in the next stage.

The summation of partial products is performed using a multi-stage carry-save adder (CSA) tree, which reduces latency by keeping sums and carries separate until the final addition. Six CSA stages systematically reduce the sixteen initial partial products: the first stage converts sixteen operands into eleven, the second stage reduces eleven to eight, the third reduces eight to four, and the fourth reduces six to four. The fifth stage reduces three of the remaining four operands to one sum and one carry, and the sixth and final stage combines the last three operands into a final sum and carry ready for the final 64-bit addition, ensuring efficient and fast combination of all partial products.

Finally, a 64-bit adder adds the final sum and final carry outputs from the last CSA stage to produce the final 64-bit product. Since the module is designed to return a 32-bit result, the most significant 32 bits are discarded, and the lower 32 bits are assigned to the output z. This step ensures that the product is correctly truncated to the desired width while maintaining proper signed behavior.

2.7 Division And Modulus

The division module implements a 32-bit signed integer division using a bitwise restoring division algorithm. It accepts a dividend a and a divisor b as inputs and produces a signed quotient and remainder. The module carefully handles both positive and negative numbers according to two's complement rules.

Initially, the module checks for division by zero. If the divisor is zero, the quotient is saturated to 0x7FFFFFFF and the remainder is assigned the dividend. This ensures safe handling of illegal operations.

For signed inputs, the quotient sign is determined by XOR-ing the signs of the dividend and divisor, while the remainder sign matches the dividend. Both dividend and divisor are converted to absolute values to simplify the unsigned division process.

The division is performed iteratively using a bitwise restoring algorithm. Starting from the most significant bit (MSB), the remainder is shifted left, and the next dividend bit is appended. If the remainder is greater than or equal to the divisor, the divisor is subtracted, and the corresponding quotient bit is set to 1. Otherwise, the quotient bit remains 0. This continues for all 32 bits of the dividend.

After computation, the module applies the original signs using two's complement where needed, ensuring correct signed outputs. The module is fully combinational in structure but iterates across bits sequentially.

This design is particularly suitable for modulus operations, as the remainder is computed directly and can be used immediately in modular arithmetic or other applications.

While not optimized for high-speed general-purpose division, it provides a robust, reliable, and synthesizable solution for signed 32-bit division and modulus calculations.

3 Simulation Results

The Arithmetic Logic Unit (ALU) was rigorously tested using a comprehensive assembly program consisting of 13 instructions covering all major operation types. The test program was successfully assembled into machine code and executed on the SimpleRISC processor core integrated with the custom ALU.

The waveform generated during simulation clearly shows the ALU ports, including the input signals a, b, and op, along with the output y. These signals help visualize how different operations are performed based on the opcode value. The register file signals are also visible in the waveform, allowing observation of data being read from and written to various registers. In addition, the TCL console displays the register values after execution of each instruction, confirming the correct operation of the ALU and proper data flow between the register file and the ALU during the simulation.

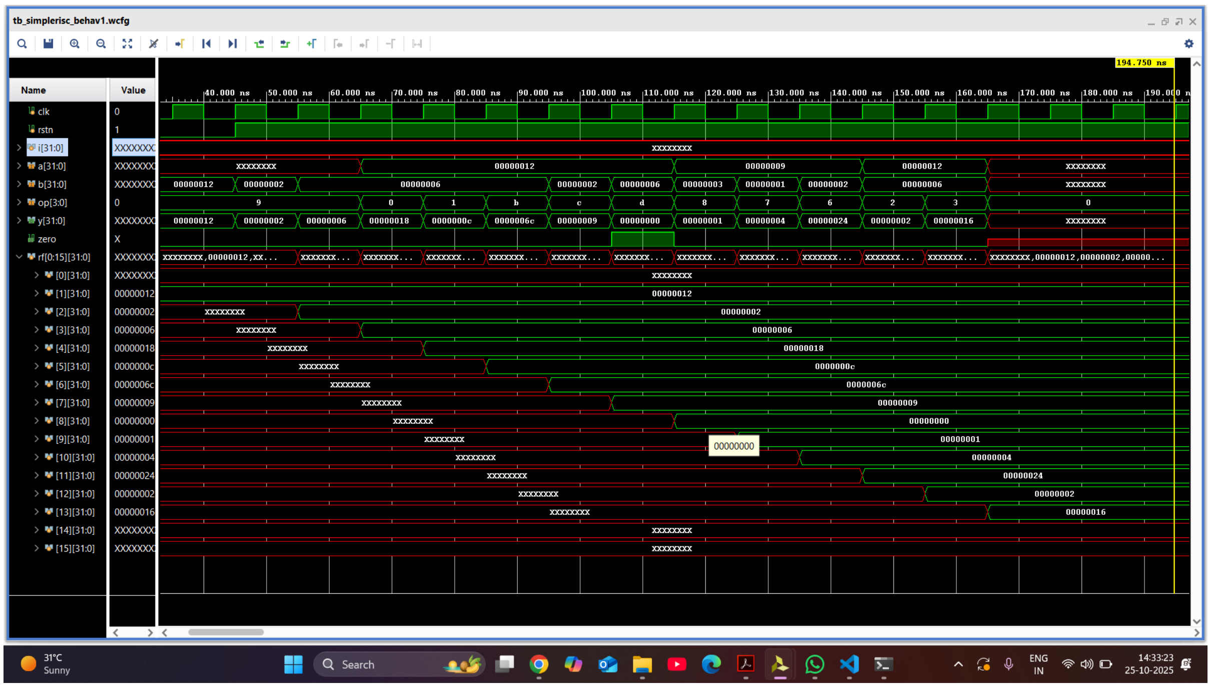
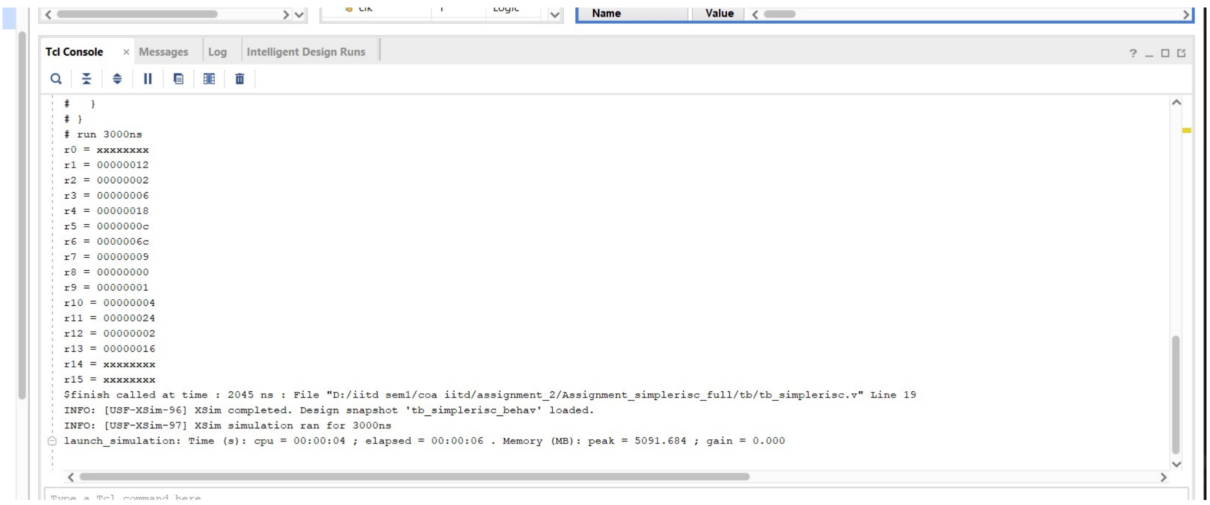


Figure 1: Waveform after simulation

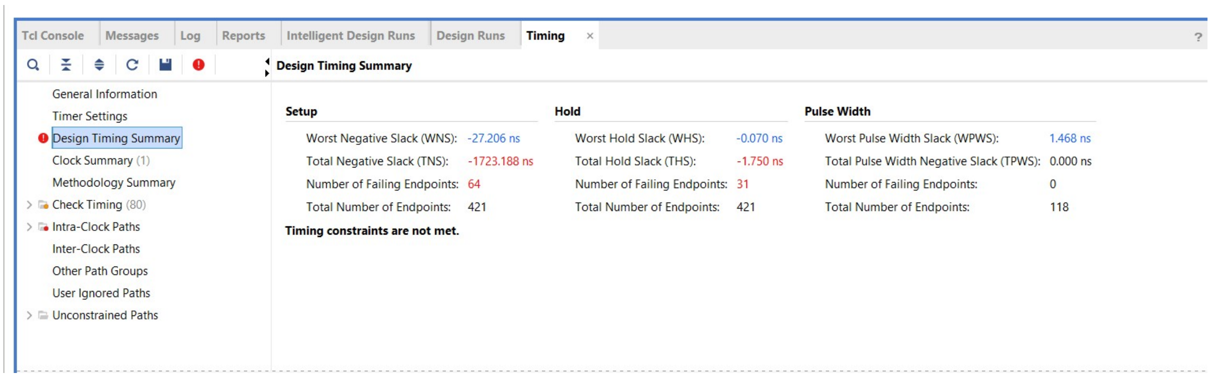


```
#    }
# }
# run 3000ns
r0 = xxxxxxxx
r1 = 00000012
r2 = 00000002
r3 = 00000006
r4 = 00000018
r5 = 0000000c
r6 = 0000006c
r7 = 00000009
r8 = 00000000
r9 = 00000001
r10 = 00000004
r11 = 00000024
r12 = 00000002
r13 = 00000016
r14 = xxxxxxxx
r15 = xxxxxxxx
$finish called at time : 2045 ns : File "D:/iitd sem1/coa iitd/assignment_2/Assignment_simplerisc_full/tb/tb_simplerisc.v" Line 19
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_simplerisc_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 3000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:06 . Memory (MB): peak = 5091.684 ; gain = 0.000
```

Figure 2: TCL console Output

4 Synthesis Results

The ALU design underwent synthesis targeting a 250 MHz operating frequency, corresponding to a 4 ns clock period. The synthesis results indicate that the current implementation does not meet the target timing constraints. The design achieved a maximum operating frequency of approximately 74 MHz, significantly below the 250 MHz target.

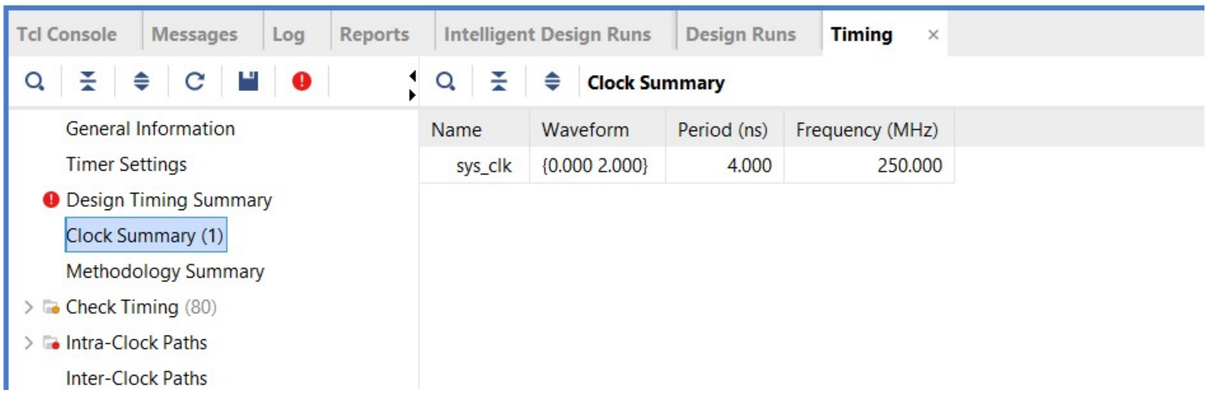


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	-27.206 ns	Worst Hold Slack (WHS):	-0.070 ns	Worst Pulse Width Slack (WPWS):	1.468 ns
Total Negative Slack (TNS):	-1723.188 ns	Total Hold Slack (THS):	-1.750 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	64	Number of Failing Endpoints:	31	Number of Failing Endpoints:	0
Total Number of Endpoints:	421	Total Number of Endpoints:	421	Total Number of Endpoints:	118

Timing constraints are not met.

Figure 3: Timing Report after Synthesis



Clock Summary

Name	Waveform	Period (ns)	Frequency (MHz)
sys_clk	{0.000 2.000}	4.000	250.000

Figure 4: Timing Constraints

4.1 250 MHz?

In a single-cycle, non-pipelined ALU architecture, every instruction must complete in a single clock cycle, including the slowest operation. The target clock frequency for the design is 250MHz, which corresponds to a clock period of 4ns. This implies that all combinational logic paths from inputs to outputs must be completed within 4ns to meet timing requirements. Any operation whose combinational delay exceeds this limit will prevent the ALU from operating at the desired frequency.

The ALU under consideration contains multiple functional modules, including arithmetic operations such as addition and subtraction, logical operations such as AND, OR, XOR, and NOT, shift operations, multiplication, and division/modulus. Each of these modules contributes differently to the overall critical path of the ALU.

In a single-cycle ALU architecture, the clock period must accommodate the slowest operation, because all instructions share the same clock. Even though addition, subtraction, and bitwise operations are fast enough, the presence of multiplication, division, modulus, and shift instructions dominates the critical path. As a result, the maximum frequency of the ALU is limited by these slow operations and cannot reach the desired 250MHz.

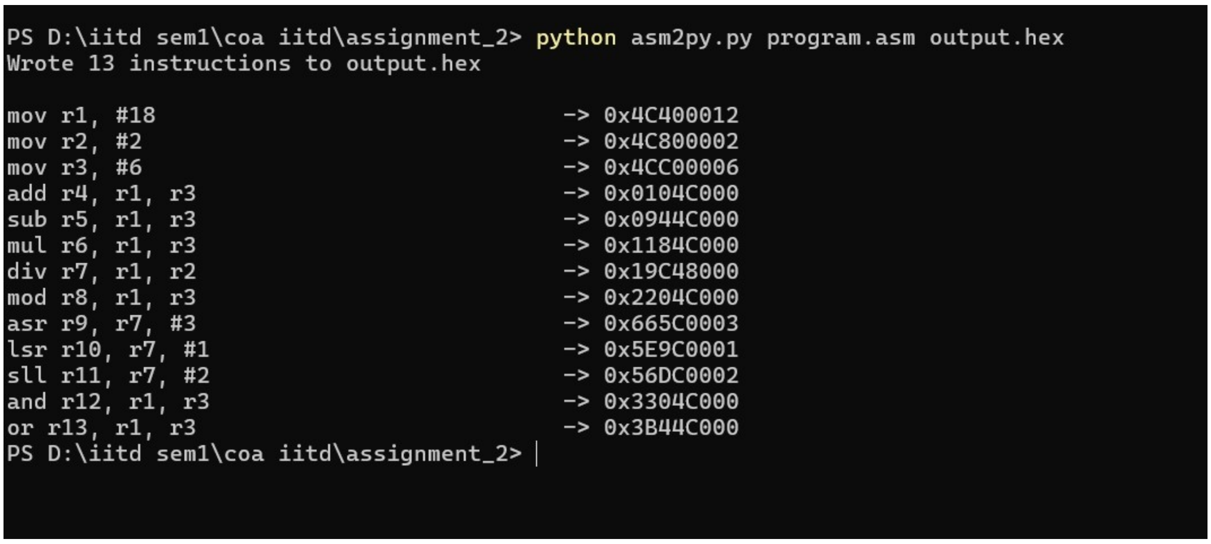
The inherent limitation of single-cycle, non-pipelined architectures is that all instructions must complete in the same cycle, meaning the clock cannot be faster than the longest combinational delay. This is why, in the current ALU design, single-cycle operation at 250MHz is not feasible. Only simpler operations, such as logic and addition/subtraction, could theoretically meet the 4ns requirement, but the presence of slow operations forces the clock to run much slower.

To achieve 250MHz, alternative architectural strategies must be adopted. One approach is pipelining, where slow operations such as multiplication, division, and shifts are split into multiple stages, allowing each stage to complete in less than 4ns. Another approach is a multi-cycle ALU, where slow instructions take multiple clock cycles to execute, allowing the clock period to be determined by faster operations. Additionally, using FPGA-specific DSP blocks for multiplication and dedicated shift primitives can reduce combinational delay.

In conclusion, the current single-cycle, non-pipelined ALU cannot achieve 250MHz due to the long combinational delays of multiplication, division, modulus, and shift operations. While simple operations are fast enough, the single-cycle architecture requires that all instructions complete within the same 4ns period, making it impossible to meet the target frequency without pipelining, multi-cycle execution, or hardware optimizations.

5 Testcase Description

In this experiment, a set of testcases were written in SimpleRISC assembly to verify the functionality of the ALU. The test program includes a variety of arithmetic, logical, and shift operations such as ADD, SUB, MUL, DIV, MOD, AND, OR, XOR, LSL, LSR, ASR, and SLT. Each instruction operates on different registers to ensure that all ALU modules are properly tested. The corresponding machine code (HEX file) was generated from this assembly program using a Python-based assembler script provided with the assignment. The generated .hex file is then loaded into the instruction memory for simulation. A screenshot of the command-line execution of this script is shown to demonstrate the assembly-to-hex conversion process.



```
PS D:\iitd sem1\coa iitd\assignment_2> python asm2py.py program.asm output.hex
Wrote 13 instructions to output.hex

mov r1, #18                          -> 0x4C400012
mov r2, #2                           -> 0x4C800002
mov r3, #6                           -> 0x4CC00006
add r4, r1, r3                       -> 0x0104C000
sub r5, r1, r3                       -> 0x0944C000
mul r6, r1, r3                       -> 0x1184C000
div r7, r1, r2                       -> 0x19C48000
mod r8, r1, r3                       -> 0x2204C000
asr r9, r7, #3                       -> 0x665C0003
lsr r10, r7, #1                      -> 0x5E9C0001
sll r11, r7, #2                      -> 0x56DC0002
and r12, r1, r3                      -> 0x3304C000
or r13, r1, r3                       -> 0x3B44C000
PS D:\iitd sem1\coa iitd\assignment_2>
```

Figure 5: Assembly to hex Conversion