ECE 385

Spring 2024
Experiment # 5

Lab5 : An 8-bit Multiplier in SV

Name: Jie Wang, Shitian Yang
Student ID: 3200112404, 3200112415

Prof. Chushan Li, Prof. Zuofu Cheng
ZJU-UIUC Institute
March 15, 2024, Friday D-225
TA: Jiebang Xia
Demo Point: 4/5

1.Introduction

In Lab 5, we enter the heart of digital arithmetic by designing and implementing an 8-bit multiplier using SystemVerilog. This lab exercise not only underscores the fundamental principles of multiplication algorithms but also showcases the power of hardware description languages in modeling and simulating digital circuits. By comparing the implemented algorithm to traditional multiplication methods, we gain valuable insights into the efficiencies and challenges of digital system design, preparing us for more complex engineering tasks ahead.

2.Prelab Question

1. Calculate the Switched Multiplier & Multiplicand:

Q: S*B = ? S $= -59_{10}$ $= 11000101_2$; B $= 7_{10}$ $= 00000111_2$

function	X	A	B	M	Comments for the next step
Clr_A_ld_B	0	00000000	00000111	1	M=1, add S to A
ADD	1	11000101	00000111	1	A's largest bit is 1, X=1, A_shift_in=1
SHIFT	1	11100010	10000011	1	M=1, add S to A
ADD	1	10100111	10000011	1	A's largest bit is 1, X=1, A_shift_in=1
SHIFT	1	11010011	11000001	1	M=1, add S to A
ADD	1	10011000	11000001	1	A's largest bit is 1, X=1, A_shift_in=1
SHIFT	1	11001100	01100000	0	M=0, just shift. X=1, A_shift_in=1
SHIFT	1	11100110	00110000	0	M=0, just shift. X=1, A_shift_in=1
SHIFT	1	11110011	00011000	0	M=0, just shift. X=1, A_shift_in=1
SHIFT	1	11111001	10001100	0	M=0, just shift. X=1, A_shift_in=1
SHIFT	1	11111100	11000110	0	M=0, just shift. X=1, A_shift_in=1
SHIFT	1	11111110	01100011	1	M=0, just shift. X=1, A_shift_in=1

Table-1: Calculating Process

2. Design the 8-bit multiplier in SystemVerilog:

As there are no template for the Lab5, we directly utilize our code from Lab4, and we modified the SystemVerilog code for key modules including ***`Control.sv`***, ***`adder_sub.sv`*** ***`,counter.sv`*** ***and `lab5_toplevel.sv`*** `. We then integrated an existing ***`testbench_8.sv`*** file, ensuring it was fully compatible with our 8-bit multiplier. Our modifications were compiled and subjected to simulation, which we completed successfully with no errors, confirming the functional correctness of our updates.

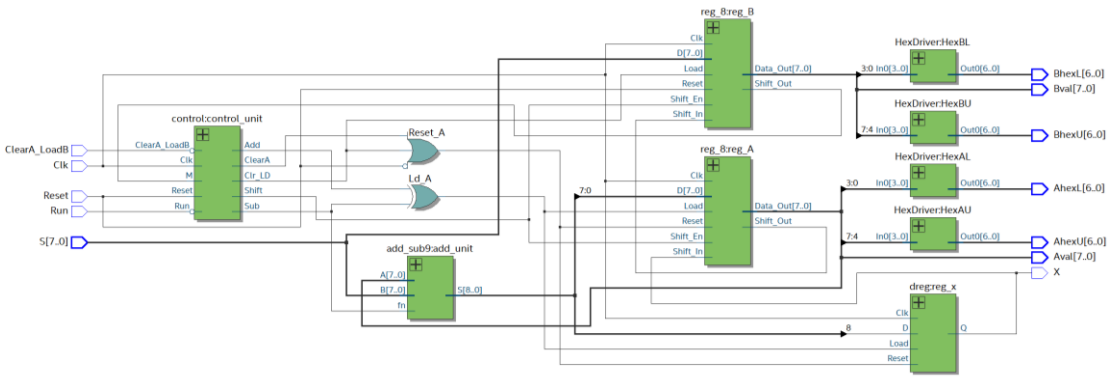


Fig-1: RTL Viewer of Our Circuit

3. Implement the 8-bit multiplier

As shown in Figure 2, here is the FSM of our system. The FSM is the backbone of the multiplier's logic, guiding it through the necessary states to compute the product of two 8-bit numbers.

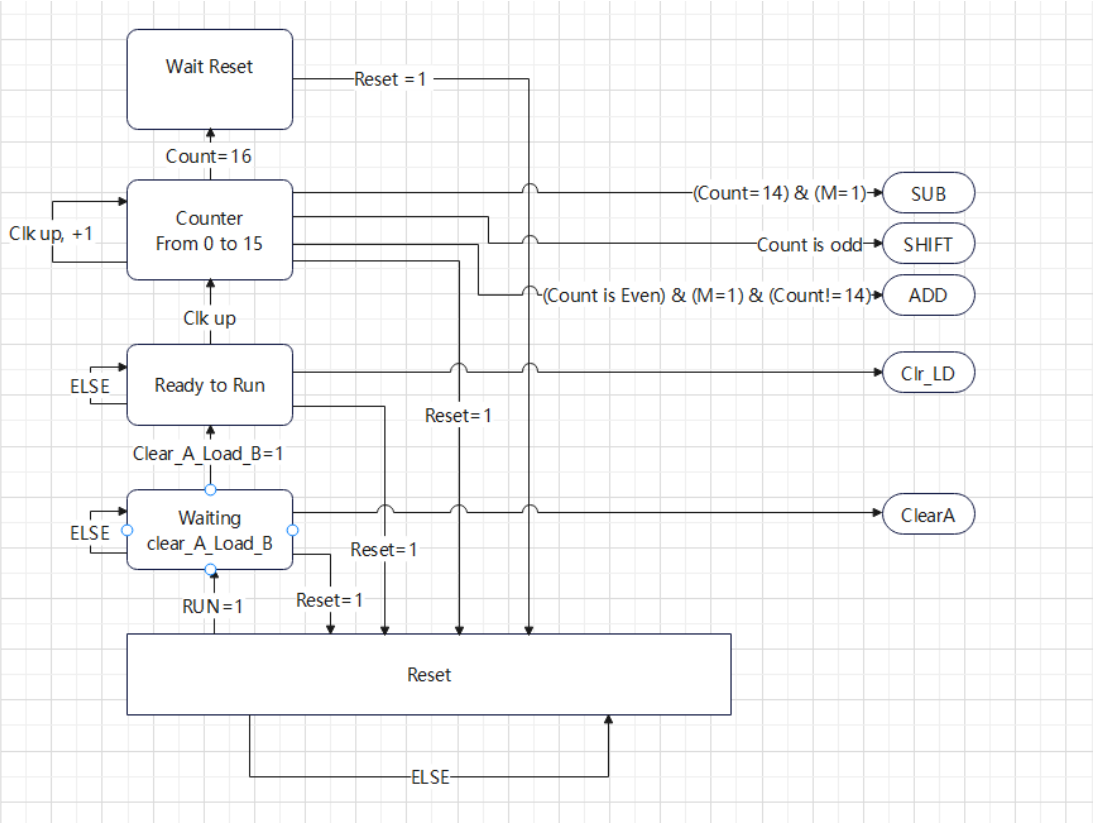


Fig-2: FSM of our 8-bit multiplier

Here is a detailed explanation of how the multiplier operates:

1. **Initialization (Reset State):** The multiplier begins its operation in the Reset state. This is the initial condition where all registers are cleared, and the system is prepared for a new multiplication operation. The Reset state ensures that the multiplier starts from a known state, free from any residual values from previous operations.
2. **Waiting State:** After the Reset, the FSM transitions to the Waiting state. In this state, the system awaits the input operands. The multiplicand is loaded from the switch settings (S), and the multiplier is loaded into Register B. The ClearA_LoadB button is used to facilitate this process, which not only loads the multiplier into Register B but also clears the X and A registers, setting the stage for the multiplication.
3. **Ready State:** Once the operands are loaded, the FSM moves the system into the Ready state. Here, the multiplier is prepared for the computation. The control unit is ready to initiate the series of addition and shift operations that are fundamental to the add-shift algorithm used for multiplication.
4. **Counter State (Add-On and Left Shift):** The heart of the multiplier's operation lies in the Counter state. This state uses a 5-bit counter to manage the add-on and left shift operations. The counter represents the intermediate steps of the add-shift algorithm, where the multiplicand is added to the accumulator (A) when the least significant bit of

the multiplier (M) is 1, and then the accumulator and multiplicand (XAB) are shifted left by one bit. This process is repeated for each bit of the multiplier, effectively multiplying the multiplicand by each bit of the multiplier and accumulating the partial products.

5. **Result Generation:** After the final shift operation, the FSM has constructed the partial products within the XAB register. The final result is generated based on the ADD and SUB signals, which incorporate the 2's complement logic for negative numbers. If the most significant bit of Register B (indicating a negative number) is 1 after the seventh shift, a subtract operation is performed to account for the negative weight in the 2's complement representation.
6. **Output Storage and Display:** The computed 16-bit product is stored in Registers A and B. The result is then displayed on the hex displays, with the most significant byte (MSB) on the upper display (AhexU) and the least significant byte (LSB) on the lower display (AhexL and BhexL). The output can be observed directly on the DE2 board, providing a visual confirmation of the multiplication's correctness.

The FSM's efficient management of these states ensures that the multiplier computes the product accurately and efficiently. The use of a counter to represent the numerous states in the add-shift algorithm is a practical approach that simplifies the FSM's design while maintaining the necessary functionality. The implementation details of the control logic, the add-shift algorithm, and the register interactions are outlined in the **Code and Appendix section**.

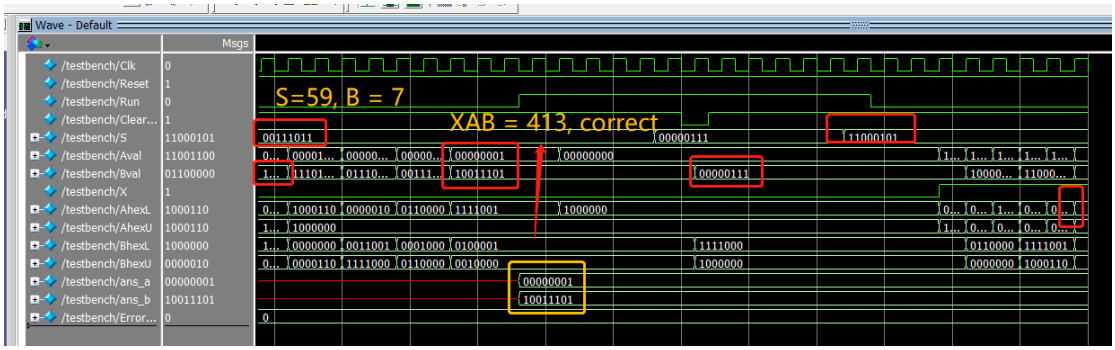


Fig-3: 0-1000ns Waveform, ErrorCnt == 0

4.Post-lab Questions

(A) Power Analysis

1. Refer to the Design Resources and Statistics in IQT.30-32 and complete the following design statistics table.

Feature	LUT	DSP	Memory (BRAM)	Flip-Flop	Frequency	Static power	Dynamic Power	Total Power
Data	94	0	0	36	85.04	96.33	2.31	158.23

(B) Question

1. What is the purpose of the X register. When does the X register get set/cleared?

X is used for the first bit of A while shifting. It maintains the positive or negative result while shifting. When Clear_A_load_B is pressed, X will be cleared. And every time after add or sub function, it needs to detect the first bit of A and load this first bit to X if it is changed.

2.What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?

Overflow: When multiplying two numbers, the result can have twice as many bits as each operand. If the system does not accommodate for this increased bit-width, it could lead to overflow.
Speed: For larger numbers or higher bit-widths, the add-shift method can become increasingly slow and complex, as it requires multiple iterations to complete a single multiplication.

3.What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method?

Advantages:

Automation: The algorithm can be fully automated and executed at a speed that manual methods cannot match.
Accuracy: High accuracy without the human error.

Disadvantages:

Overhead for Small Calculations: For small, one-off calculations, the setup and execution of the algorithm might introduce more overhead than simply doing the calculation by hand.
Complexity in Implementation: Implementing the algorithm in hardware or software requires a detailed understanding of digital systems and can be more complex than simply performing the calculation manually for a single instance.

5. Bug Log

- Description of all bugs encountered, and corrective measures taken:

1. Can not install the USB Blaster Driver

One of Jie's Laptop is accidentally broken, and he was unable to install the USB Blaster Driver from the DE-2 board on a new device with Quartus Prime 20.1. After several attempts and discussions within the ECE385 WeChat Group, we found a solution by referring to installation guides on Terasic and Intel's websites. This resource guided us through the correct installation process, eventually resolving our driver installation issue.

2. LED Digit not Functioning

Although our code passed all the test cases in ***'test_bench.sv'***, it didn't display correctly on the DE-2 board. After reinvestigating the specific LED pin assignment, we fixed the wrongly assigned LED, and the digit can be displayed correctly.

3. DE Board Cannot Load Data

We found that our toplevel file fall into an infinite loop, which disabled it from entering new data. After we fix it according to the routine in Lab Manual, it works correctly.

6.Conclusion

Through lab5, we practiced on the overall deployment of System Verilog in a new way.

The design and implementation of the 8-bit multiplier in SystemVerilog have been a comprehensive learning experience that has deepened the understanding of digital design and FPGA applications. The functionality of the multiplier was successfully demonstrated on the DE2 board, showcasing the effectiveness of the add-shift algorithm in performing multiplication for 2's complement.

Overall, this lab was a practical application of what we've learned, giving us confidence in our skills as we prepare to solve more advanced projects. We're looking forward to applying this knowledge in our future work.

7. References

[1] KTTECH. (2017, January 31). ECE 385 Lab 5: An 8-bit Multiplier in SV. Retrieved from https://kttechnology.wordpress.com/2017/02/10/ece-385lab5-an-8-bit-multiplier-in-sv/ Teaching Assistant Blog
[2] ECE385 Faculty. (n.d.). Lab 5 description
[3] ECE385 Faculty. (n.d.). Introduction to SystemVerilog (pdf)
[4] ECE385 Faculty. (n.d.). Introduction to Quartus Prime in the lab manual.

8. Appendix: Written Description of .sv Modules

Adder_8bits_with_Subtract

1.Description:

This module implements an 8-bit adder that can also perform subtraction through the addition of the two's complement of the second operand. It leverages the full_adder_8bits module for the core addition operation and adjusts the second operand and carry-in based on the subtraction flag.

2.Description of the Operation:

The module adjusts the second operand, b, by XORing it with a mask generated by replicating the subtract flag 8 times. This effectively negates b if subtraction is required. It then sets the carry-in (b_cin) to the value of the subtract flag, effectively adding 1 in case of subtraction, thus completing the two's complement operation. The full_adder_8bits module is used to add a to the adjusted b (b_adjusted), with b_cin as the initial carry-in. This results in either addition or subtraction based on the subtract flag.

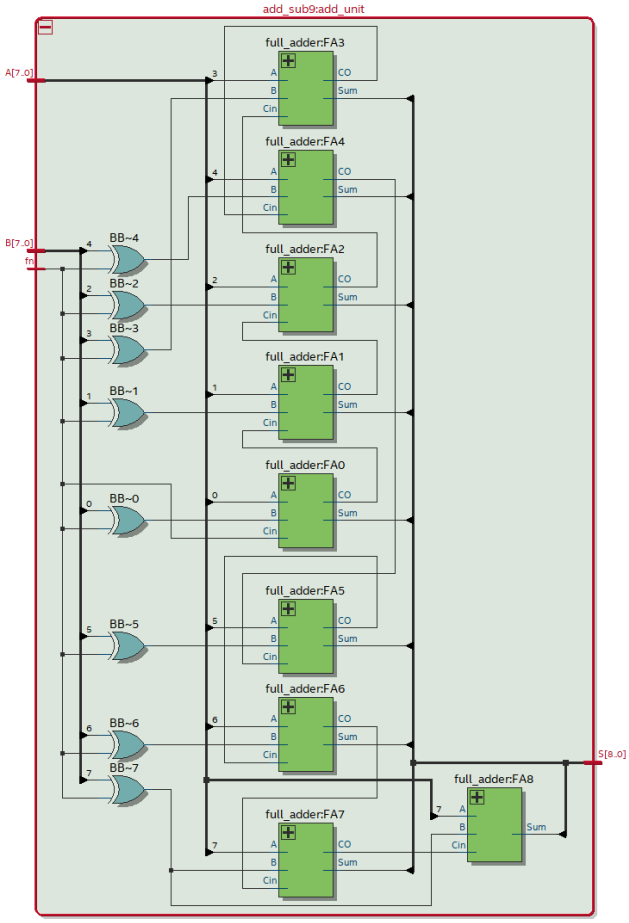
3.Features:

The module supports both addition and subtraction operations on 8-bit operands. This dual functionality is achieved through a simple yet effective method of operand adjustment, enabling the reuse of the addition logic for subtraction by applying the two's complement principle.

Inputs/Outputs:

Inputs: a[7:0], b[7:0], subtract
Outputs: result[8:0], carry_out

4. RTL View



5.Purpose and Operation of Each Module

1)Module: full_adder_8bits

Purpose:

Performs 8-bit binary addition between two operands, a and b, along with an input carry (cin). It constructs the sum bit by bit and computes the final carry out.

Operation:

Uses a series of full_adder modules to perform bit-wise addition across all 8 bits of a and b.
Each full_adder module computes the sum and carry out for its respective bit position, with the carry out chained to the next more significant bit's carry in.
The sum (s) and final carry out (cout) are then available as outputs, representing the 8-bit sum and the overflow/carry-out respectively.

Inputs/Outputs:

Inputs: a[7:0], b[7:0], cin
Outputs: s[7:0], cout

2)Module: full_adder

Purpose:

A fundamental building block used to add two single bits along with a carry-in, producing a sum and a carry-out. This module is instantiated multiple times within full_adder_8bits to construct a bit-wise adder.

Operation:

Calculates the sum (s) as the XOR of inputs x, y, and cin.

Determines the carry out (cout) based on whether at least two of the three inputs are 1.

Inputs/Outputs:
Inputs: x, y, cin
Outputs: s, cout

lab5_toplevel.sv adder_sub.sv

```
module adder_sub(
    input logic[7:0] a,
    input logic[7:0] b,
    input logic subtract,
    output logic[8:0] result,
    output logic carry_out
);
    logic[7:0] b_adjusted;
    logic b_cin;
    logic carry_out_8bit; // For the carry out of the 8-bit adder

    assign b_adjusted = b ^ {8{subtract}};
    assign b_cin = subtract;

    full_adder_8bits apply_add(a, b_adjusted, b_cin, result[7:0], carry_out_8bit);

    assign a7 = a[7];
    assign b7 = b[7];

    full_adder fa8(a7, b7, b_cin, result[8], carry_out);

endmodule



module full_adder_8bits(
    input logic[7:0] a,
    input logic[7:0] b,
    input logic cin,
    output logic[7:0] s,
    output logic cout
);
    logic[7:0] s_int;
    logic[7:0] cout_int;

    full_adder fa0(a[0], b[0], cin, s_int[0], cout_int[0]);
    full_adder fa1(a[1], b[1], cout_int[0], s_int[1], cout_int[1]);
    full_adder fa2(a[2], b[2], cout_int[1], s_int[2], cout_int[2]);
    full_adder fa3(a[3], b[3], cout_int[2], s_int[3], cout_int[3]);
    full_adder fa4(a[4], b[4], cout_int[3], s_int[4], cout_int[4]);
    full_adder fa5(a[5], b[5], cout_int[4], s_int[5], cout_int[5]);
    full_adder fa6(a[6], b[6], cout_int[5], s_int[6], cout_int[6]);
    full_adder fa7(a[7], b[7], cout_int[6], s_int[7], cout_int[7]);


    assign s = s_int;
    assign cout = cout_int[7];

endmodule


module full_adder
(
    input   logic       x,
    input   logic       y,
    input   logic       cin,
```

2. Control Unit

1.Description:

This SystemVerilog design comprises several interconnected modules aimed at managing arithmetic operations within a digital system. Central to the design is a state machine that interacts with control logic to determine the execution of addition, subtraction, shifting, and load/clear operations based on various input signals. This setup is particularly suited for implementing arithmetic algorithms in hardware, such as those found in digital signal processing or arithmetic logic units (ALUs).

2.Description of the Operation:

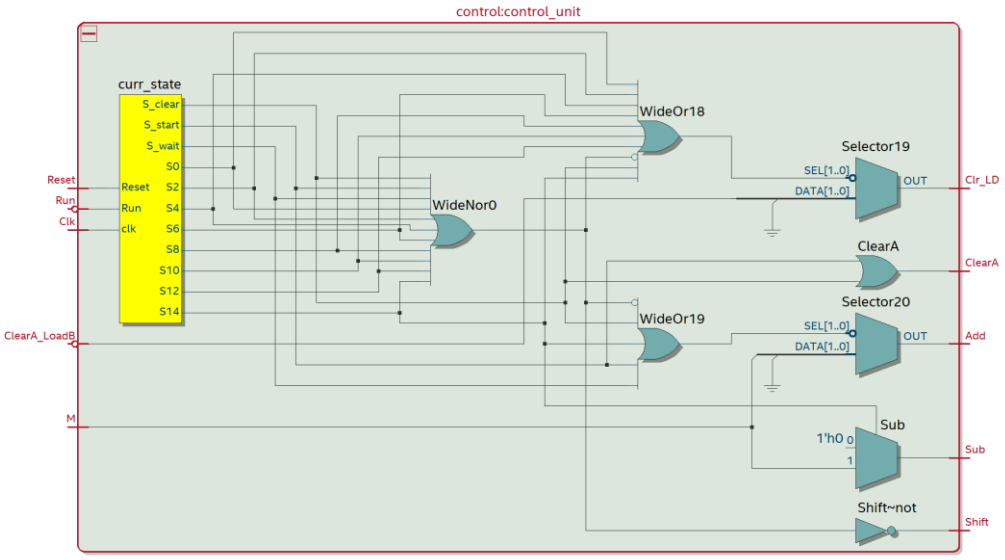
The operation begins with the control module, which manages the overall process flow based on input signals like Clk (Clock), Reset, ClearA_LoadB, Run, and M. This module uses internal state control logic to generate signals for clearing/loading, shifting, adding, and subtracting.
A 5-bit counter is employed within the control_with_counter module to execute specific actions at different counts, such as performing subtraction when a certain count is reached or deciding when to shift or add based on the counter's current state.

The dreg_control module is a simple D-type register used for holding the state of control signals, ensuring that operations like load, add, and subtract are executed only when appropriate conditions are met.

3.Features:

- **Modular Design:** The design is highly modular, with clear separations of concerns among different functionalities. This makes the design scalable and easy to debug or extend.
- **State-Controlled Execution:** It leverages state machines for controlling the execution flow, making it robust and flexible for various arithmetic operations.
- **Versatility in Operations:** The design supports a range of arithmetic operations including addition, subtraction, and shifting, making it suitable for a wide array of arithmetic processing tasks.

4. RTL View



Purpose and Operation of Each Module

(1)Module: control

Purpose:

Serves as the main control logic unit for a digital system, managing the execution of operations like load/clear, shift, add, and subtract based on the system state and input signals.

Operation:

Manages system states through a sequence of control signals derived from input conditions and internal logic. Utilizes dreg_control modules to hold and transition states based on Run and ClearA_LoadB inputs. Controls the arithmetic and data manipulation operations by generating appropriate control signals (Clr_LD, ClearA, Shift, Add, Sub) to other system components.

Inputs/Outputs:

Inputs: Clk, Reset, ClearA_LoadB, Run, M
Outputs: Clr_LD, ClearA, Shift, Add, Sub

(2)Module: dreg_control

Purpose:

A digital register (D-type flip-flop) designed to hold a single bit of data, used within control logic to maintain the state of a control signal over time.

Operation:

Holds a 1 or 0 state based on the Run input signal, allowing for simple state transitions in control logic.Resets to 0 when a high signal is received on the Reset input, ensuring the system can return to a known state.

Inputs/Outputs:

Inputs: Clk (Clock), Reset, Run
Outputs: Data_out (The current state of the control signal)

(3)Module: control_with_counter

Purpose:

Integrates counter logic with control operations, determining the execution of specific actions (like clear, shift, add, or subtract) based on the counter's value and input conditions.

Operation:

Coordinates with the counter_5_bits module to track the progression of counts and trigger actions at predetermined points. Defines default operation states and modifies them based on the count value and Run condition, effectively linking time-based events with control logic actions. Specific operations (e.g., subtraction, addition, shifting) are triggered based on the combination of count, Run, and M inputs, aligning arithmetic operations with the system's current state and requirements.

Inputs/Outputs:

Inputs: Clk, Reset, Run, M
Outputs: ClearA, Shift, Add, Sub

(4)Module: counter_5_bits

Purpose:

This module acts as a 5-bit binary counter, incrementing its count with each clock cycle. It is designed to output specific flags (time_15 and wait_flag) that signal particular count values for controlling subsequent logic in a larger system.

Operation:

Utilizes a series of full_adder modules to increment the count value on each clock pulse. Generates a time_15 flag when the count reaches 14 (binary 01110), indicating a specific timed event. The wait_flag is set when the count exceeds 15 (binary 10000), used to initiate a wait or pause in the system operation. The count is reset to 0 upon a high signal on the Reset input.

Inputs/Outputs:

Inputs: Clk (Clock), Reset

Outputs: count[4:0] (5-bit count value), time_15 (flag), wait_flag (flag)

3. Register Unit

1.Description:

The SystemVerilog modules provided, register_8bits and dreg, are designed for digital storage and manipulation within a hardware design context. The register_8bits module functions as an 8-bit register capable of loading data, maintaining current state, and shifting data left under control of external signals.

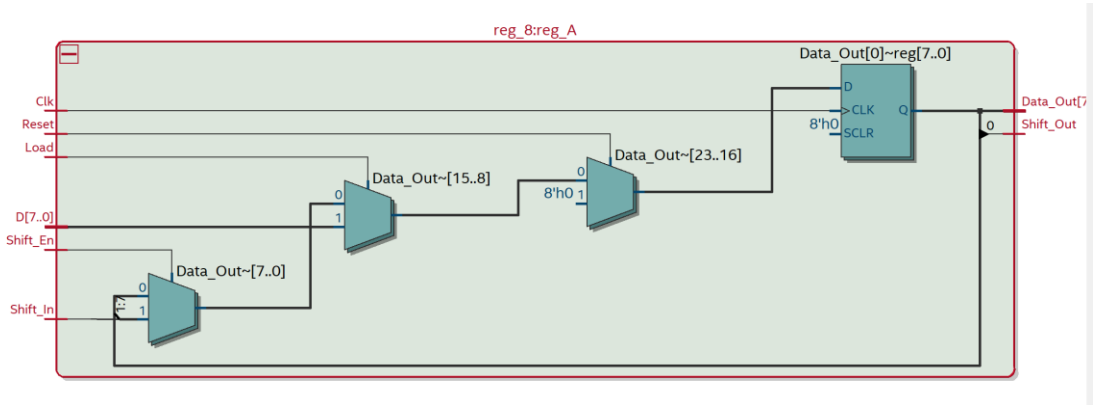
2.Description of the Operation:

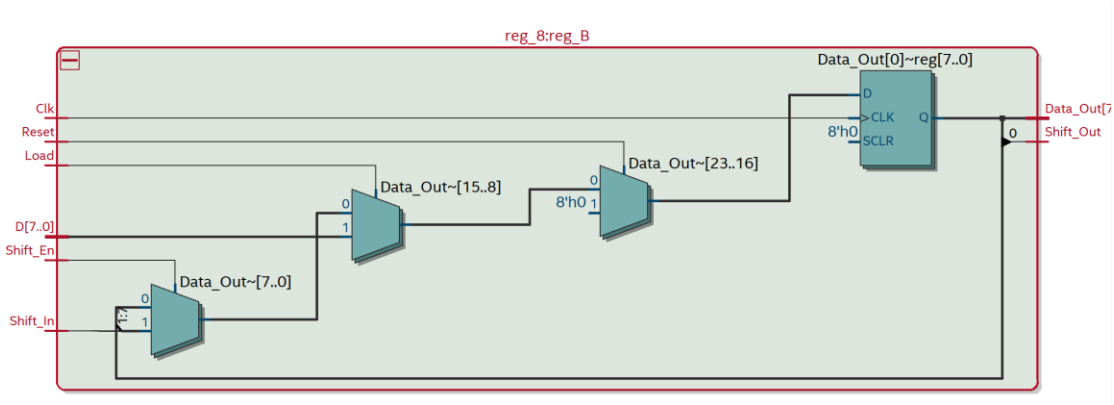
Reset: Clears the register's contents to zero.
Load Data (Load_d): Loads an 8-bit value into the register from Data_in.
Shift Function (Shift_function): Performs a left shift operation on the register's contents, introducing a new bit on the right from Left_shift_In.
Output: The module outputs the current stored data through Data_out and provides the bit shifted out on the left through Left_shift_Out.

3.Features:

- **Versatility in Data Handling:** Capable of both storing and manipulating data, including shifting operations that are essential in many digital processing tasks. Serial Data Processing: Facilitates serial-to-parallel and parallel-to-serial conversion, useful in communication interfaces. State
- **Retention:** Maintains its state across clock cycles, allowing for persistent storage of data within a cycle-based processing environment.

4. RTL View





4. Purpose and Operation of Each Module

1)Module: register_8bits

Purpose:

This module is an 8-bit register designed for storing and optionally shifting its contents. It supports loading new data, holding the current data, or performing a left shift operation based on control signals. It's useful in digital designs that require temporary storage and manipulation of 8-bit wide data, such as in digital signal processing, data path units of microprocessors, or shift registers in communication interfaces.

Operation:

On a reset (Reset high), the module clears its contents to 0. If the Load_d signal is high during a clock edge, the module loads the 8-bit Data_in into Data_out. If Shift_function is high, the module performs a left shift operation on the next clock edge, taking Left_shift_In as the new least significant bit (LSB). This allows for serial data processing or manipulation. The leftmost bit (MSB) shifted out is available at Left_shift_Out, enabling chaining of registers or extraction of shifted-out data.

Inputs/Outputs:

Inputs: Clk (Clock), Reset, Left_shift_In, Load_d, Shift_function, Data_in[7:0]
Outputs: Left_shift_Out, Data_out[7:0]
Module: dreg

Purpose:

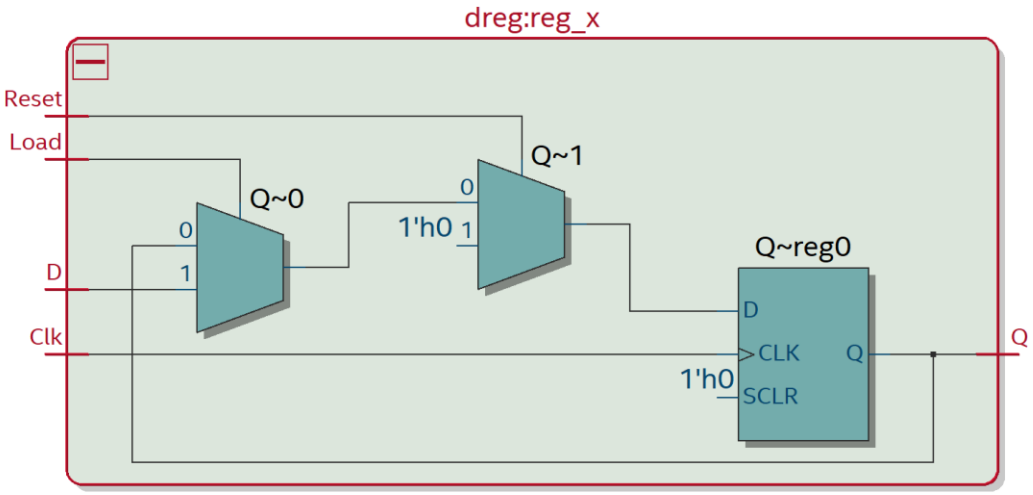
A simple D-type flip-flop (DFF) that stores a single bit of data. This module can be used as a building block for larger storage elements or for simple state storage in control logic. It is fundamental in digital systems for holding state, debouncing, synchronization, or simple flag/status storage.

Operation:

Upon a reset (Reset high), the stored bit is cleared to 0.
If Load_d is high on a clock edge, the module loads the Data_in bit into Data_out.
If neither reset nor load conditions are met, the output retains its current state, ensuring stable storage of a single bit across clock cycles.

Inputs/Outputs:

Inputs: Clk (Clock), Reset, Load_d, Data_in
Outputs: Data_out


dreg:reg_x
Reset
Load
D
Clk
Q~0
0
1
Q~1
0
1'h0
1
Q~reg0
D
CLK
Q
1'h0
SCLR
Q