# 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 = -5910 = 110001012

B = 710 = 000001112

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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

We carefully modified the SystemVerilog code for key modules including ***`Control.sv`***, ***`Register\_unit.sv `,`Reg\_4.sv` and `Processor.sv `***. We then integrated an existing ***`testbench\_8.sv`*** file, ensuring it was fully compatible with our enhanced 8-bit processor. 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

### 2.Adders Implementation:

Most of our time are spent on the document reading and environment configuration. We finished the adder code modification very soon, but it is time-consuming to understand the path setting and usage of testbench. Thankfully, we finished the lab in time with the assistance of our fellow classmates and [the detailed blog written by a formal ECE385 TA.](https://kttechnology.wordpress.com/)

Detail information please read Operation of the Adder Circuit.

### 3. Performance analysis(Design Analysis Comparison Results)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Carry-Ripple | Carry-Lookahead | Carry-Select |
| Memory (BRAM) | 1 | 1 | 1 |
| Frequency | 1 | 1.284 | 1.1 |
| Total Power | 1 | 1.02 | 1 |

Table-1 : Memory, Frequency and Power Comparison

Table-2 : Memory, Frequency and Power Comparison

## 3. Bit-Serial Logic Processor

**Fig 2:** FSM Viewer Output, including 4 more state G, H, I, J

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

Fig-4: Passed all the testcase in testbench\_8.sv

## 4.Operation of the Adder Circuit

### Overview:

As demonstrated in Fig-3, The data flow in our adders can be considered as below:

**Fig-5:** RTL Viewer of Adder Circuit

### Ripple adder:

1.Description:

The Ripple Carry Adder is a basic digital adder design that implements addition of multi-bit binary numbers. It is composed of a series of one-bit full adders connected in a chain.

2.Description of the Operation:

In an RCA, each full adder takes in two corresponding bits from the input numbers and the carry-out from the previous adder as its input. The least significant bit adder receives the initial carry-in (often set to 0). The sum output from each adder forms one bit of the final result, and the carry-out is passed to the next most significant bit's full adder. This process "ripples" from the least significant to the most significant bit, hence the name.

3.Features:

Simple and straightforward design.

Easy to implement and understand.

Scalable to any bit-size by adding more full adders.

The delay is proportional to the number of bits due to the ripple effect, leading to slower operation for large bit sizes.

4. RTL View

**Fig-6:** RTL Viewer of Ripple Adder Circuit

5.Schematic Block Diagrams

**Fig-7:** Schematic Block Diagrams of Ripple Adder Circuit

6.Purpose and Operation of Each Module

1) Module: ripple\_adder

Purpose:

The ripple\_adder module is designed to perform a 16-bit addition by employing a series of 4-bit adders, utilizing the ripple carry technique.

Operation:

The module instantiates four 4-bit full adders (full\_adder\_4bit) to compute the sum of two 16-bit inputs A and B. Each 4-bit adder receives a carry-in from the previous adder's carry-out, starting with an initial carry-in of 1'b0 for the least significant bits.

Inputs/Outputs:

Inputs:

A[15:0]: First 16-bit operand

B[15:0]: Second 16-bit operand

Outputs:

Sum[15:0]: 16-bit sum of A and B

CO: Carry-out of the most significant bit addition

Module: full\_adder\_4bit

Purpose:

The full\_adder\_4bit module is a component used in the ripple\_adder to add 4-bit slices of the input operands along with a carry-in bit.

Operation:

This module contains four instances of the full\_adder module, each performing a 1-bit addition. The carry-out of each bit addition is passed as the carry-in to the next significant bit.

Inputs/Outputs:

Inputs:

x[3:0]: First 4-bit operand slice

y[3:0]: Second 4-bit operand slice

cin: Carry-in from the previous less significant slice

Outputs:

s[3:0]: 4-bit sum of x, y, and cin

cout: Carry-out to the next more significant slice

Module: full\_adder

Purpose:

The full\_adder module is the fundamental building block that adds two single bits along with a carry-in bit to produce a sum bit and a carry-out bit.

Operation:

The module performs bit-wise addition using XOR gates to compute the sum and AND, OR gates to determine the carry-out.

Inputs/Outputs:

Inputs:

x: First operand bit

y: Second operand bit

cin: Carry-in bit

Outputs:

s: Sum bit of x, y, and cin

cout: Carry-out bit indicating if there is an overflow out of the bit addition

### Lookahead adder:

1.Description:

The Carry Lookahead Adder is an advanced type of adder that improves upon the RCA by reducing the carry propagation delay.

Fig-8: A 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

2.Description of the Operation:

The CLA uses the concepts of 'generate' and 'propagate' to predict the carry-out of each bit without waiting for the previous bit's carry-out. The generate function determines if a carry will be generated by a pair of bit additions, and the propagate function determines if a carry will be passed through. These functions are used to quickly calculate carries for each bit, allowing for simultaneous sum calculation across all bits.

3.Features:

Faster than the RCA due to reduced carry propagation delay.

More complex design as it involves additional logic for generate and propagate functions.

Better suited for high-speed operations and large bit-size additions.

Typically consumes more area on a chip due to the additional logic required.

4. RTL View

**Fig-9:** RTL Viewer of Lookahead Adder Circuit

5.Schematic Block Diagrams

**Fig-10:** Schematic Block Diagrams of Lookahead Adder Circuit

Purpose and Operation of Each Module

Module: carry\_lookahead\_adder

Description:

The carry\_lookahead\_adder module implements a 16-bit adder using the carry-lookahead logic to improve the speed of binary addition by reducing the carry propagation delay between consecutive full adders.

Operation:

The module consists of four instances of compute\_4bit\_PG\_GG that compute the propagate and generate signals for each 4-bit block of inputs A and B. The carry\_lookahead\_adder\_4bit\_helper\_compute\_carry computes the carry-out signals for each block, which are then used by instances of carry\_lookahead\_adder\_4bit to calculate the final sum.

Inputs/Outputs:

Inputs:

A[15:0]: First 16-bit operand.

B[15:0]: Second 16-bit operand.

Outputs:

Sum[15:0]: 16-bit sum of A and B.

CO: Carry-out signal representing an overflow out of the most significant bit.

Module: compute\_4bit\_PG\_GG

Purpose:

The compute\_4bit\_PG\_GG module computes the propagate and generate signals for a 4-bit block of the adder. These signals are essential for the carry-lookahead logic to determine the carry for each bit without waiting for the previous bits.

Operation:

It calculates the block-level propagate signal PG by ANDing all the propagate signals, and the block-level generate signal GG by ORing all the individual generate signals conditioned on the propagate signals.

Inputs/Outputs:

Inputs:

P[3:0]: Propagate signals for a 4-bit block.

G[3:0]: Generate signals for a 4-bit block.

Outputs:

PG: Block-level propagate signal.

GG: Block-level generate signal.

Module: carry\_lookahead\_adder\_4bit

Purpose:

This module performs 4-bit binary addition using carry-lookahead logic to compute the sum and carry-out signals for the given inputs and carry-in.

Operation:

The module generates individual generate G and propagate P signals, which are used by carry\_lookahead\_adder\_4bit\_helper\_compute\_carry to compute the carry signals C. The sum is then computed using the propagate signals and carry signals.

Inputs/Outputs:

Inputs:

A[3:0]: First 4-bit operand.

B[3:0]: Second 4-bit operand.

C\_in: Carry-in signal.

Outputs:

Sum[3:0]: 4-bit sum of A and B.

CO: Carry-out signal.

Module: carry\_lookahead\_adder\_4bit\_helper\_compute\_carry

Purpose:

The purpose of this module is to compute the carry signals for a 4-bit block of the carry-lookahead adder.

Operation:

It calculates carry signals based on individual generate G, propagate P, and the input carry-in C\_in.

Inputs/Outputs:

Inputs:

P[3:0]: Propagate signals for the 4-bit block.

G[3:0]: Generate signals for the 4-bit block.

C\_in: Input carry signal.

Outputs:

C[3:0]: Carry signals for the 4-bit block.

### C. Select adder:

1.Description:

The Carry Select Adder is designed to improve the speed of addition by speculatively calculating two possible results for each bit addition, based on the assumption of the carry-in being either 0 or

Fig-11: 16-bit Carry-Select Adder Block Diagram

2.Description of the Operation:

The CSA divides the input numbers into blocks and computes two sums for each block, one assuming the carry-in is 0, and the other assuming it is 1. Once the actual carry-in is known, the correct sum is selected using multiplexers. This allows the CSA to begin computing sums for the next block without waiting for the carry-out of the previous block.

3.Features:

Provides a good trade-off between speed and complexity.

Utilizes extra hardware to speculate and select correct sums, leading to an increase in power consumption and chip area.

Well-suited for medium-sized bit-width operations where speed is a concern but area and power are not critical constraints.

4. RTL View

**Fig-12:** RTL Viewer of Select Adder Circuit

5.Schematic Block Diagrams

**Fig-13:** Schematic Block Diagrams of Select Adder Circuit

Purpose and Operation of Each Module

Module: carry\_select\_adder

Description:

The carry\_select\_adder module is an optimized adder design that improves upon the traditional ripple carry adder's performance by speculatively computing sums and carry-outs for both possible incoming carry values.

Operation:

The module is made of a series of 4-bit carry-select adder units (carry\_select\_adder\_4bit\_unit). The first 4-bit adder is a regular full adder that starts with an initial carry-in of 1'b0. For subsequent 4-bit blocks, carry-select adder units are used which compute two possible results for each block based on the assumption that the carry-in is either 1'b0 or 1'b1. The correct result is selected once the carry-in is known.

Inputs/Outputs:

Inputs:

A[15:0]: First 16-bit operand.

B[15:0]: Second 16-bit operand.

Outputs:

Sum[15:0]: 16-bit sum of A and B.

CO: Carry-out of the most significant bit addition.

Module: carry\_select\_adder\_4bit\_unit

Purpose:

Each carry\_select\_adder\_4bit\_unit serves as a building block of the carry\_select\_adder and computes two sets of sums and carry-outs based on both possible carry-in values.

Operation:

The module instantiates two 4-bit full adders (full\_adder\_4bit), one assuming carry-in is 1'b0 and another assuming 1'b1. After both adders compute their respective sums and carry-outs, a multiplexer selects the correct sum and carry-out based on the actual carry-in, C\_in.

Inputs/Outputs:

Inputs:

A[3:0]: First 4-bit operand slice.

B[3:0]: Second 4-bit operand slice.

C\_in: Carry-in from the previous less significant slice.

Outputs:

Sum[3:0]: 4-bit sum of A and B.

CO: Carry-out to the next more significant slice.

## 5.Post-lab Questions

### (A)

1.

### (B)

**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.

## 6. Bug Log

Description of all bugs encountered, and corrective measures taken:

## 7.Conclusion

Through lab5, we practiced on the overall deployment of System Verilog together with the

performance analysis as below:

# 8. 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](https://learn.intl.zju.edu.cn/bbcswebdav/pid-101276-dt-content-rid-1361038_1/xid-1361038_1)

[3] ECE385 Faculty. (n.d.). [Introduction to SystemVerilog (pdf)](https://learn.intl.zju.edu.cn/bbcswebdav/pid-101280-dt-content-rid-1361044_1/xid-1361044_1)

[4] ECE385 Faculty. (n.d.). [Introduction to Quartus Prime in the lab manual.](https://learn.intl.zju.edu.cn/bbcswebdav/pid-101280-dt-content-rid-1361046_1/xid-1361046_1)