ECE 385

Spring 2024

Experiment # 4

Lab4: Introduction to SystemVerilog, FPGA, CAD, and 16-bit Adders

Name: Jie Wang, Shitian Yang

Student ID: 3200112404, 3200112415

Prof. Chushan Li, Prof. Zuofu Cheng
ZJU-UIUC Institute
Jan. 26, 2024, Friday D-225
TA: Jiebang Xia
Demo Point: 5/5

1.Introduction

In lab 4, we transitioned from working with TTL physical logic circuits to RTL design on FPGA using SystemVerilog, a new programming language for us. Our main goal was to get familiar with SystemVerilog's basic syntax and learn how to use Quartus Prime for FPGA synthesis and simulation. The task can be divided into the following:

1.Extending a 4-bit serial processor to an 8-bit version.
2.Implementing three types of adders: the carry-ripple, carry-lookahead, and carry-select.
3.Analyze the adders' difference and evaluate the performance in theory and practice.

Each of these had its quirks in terms of area, power, and speed. Understanding these differences and seeing how they play out in Quartus Prime's performance analysis was a major part of lab 5. It's one thing to study these concepts in theory. This lab solidifies our understanding and prepares us for more complex challenges ahead.

2.Prelab Question

1. Extension from 4-bit to 8-bit:

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.

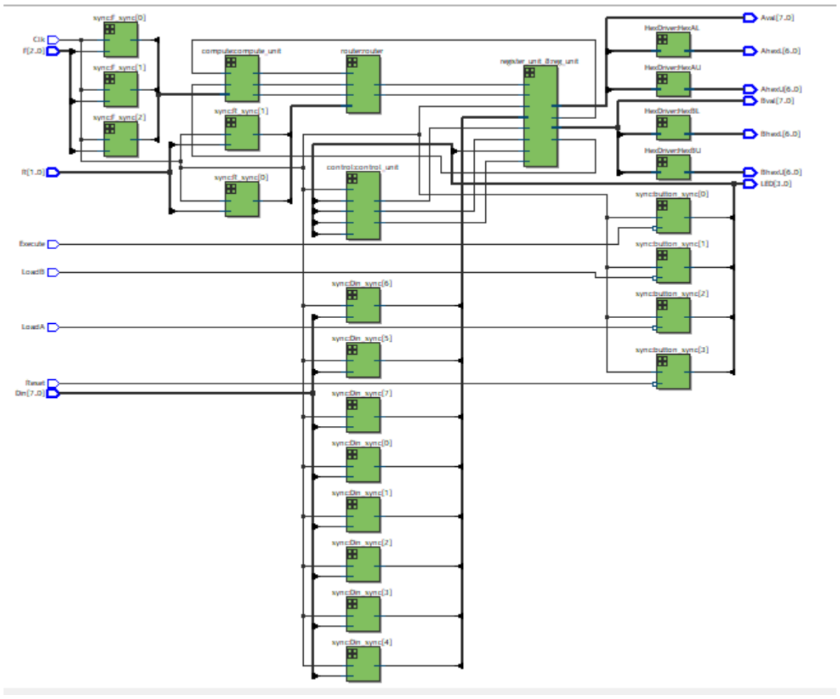


Fig-1: RTL Viewer of Our Circuit

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.

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

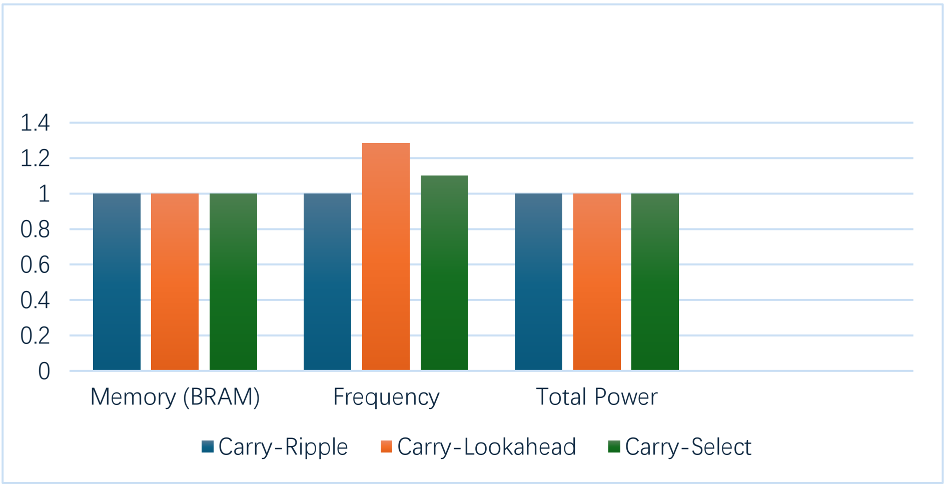


Table-2 : Memory, Frequency and Power Comparison

3. Bit-Serial Logic Processor

We successfully expand the IQT Serial Logic Processor from 4 to 8 bits.

- Firstly, we define a new register structure to accommodate 8-bit processing.
- Secondly, we modified the Finite State Machine for control over the register and computation unit as figure 2 shows.
- Our new 8-bit register module and its I/O functions were validated through simulations.

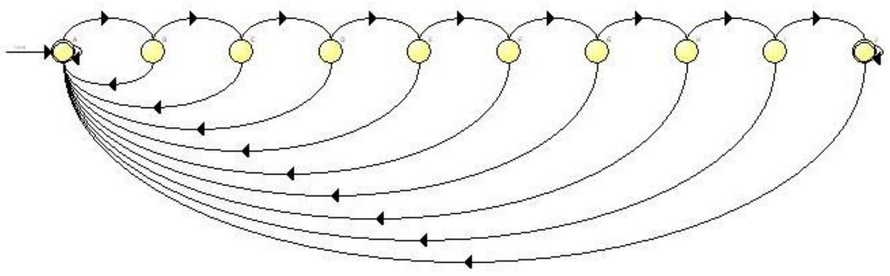
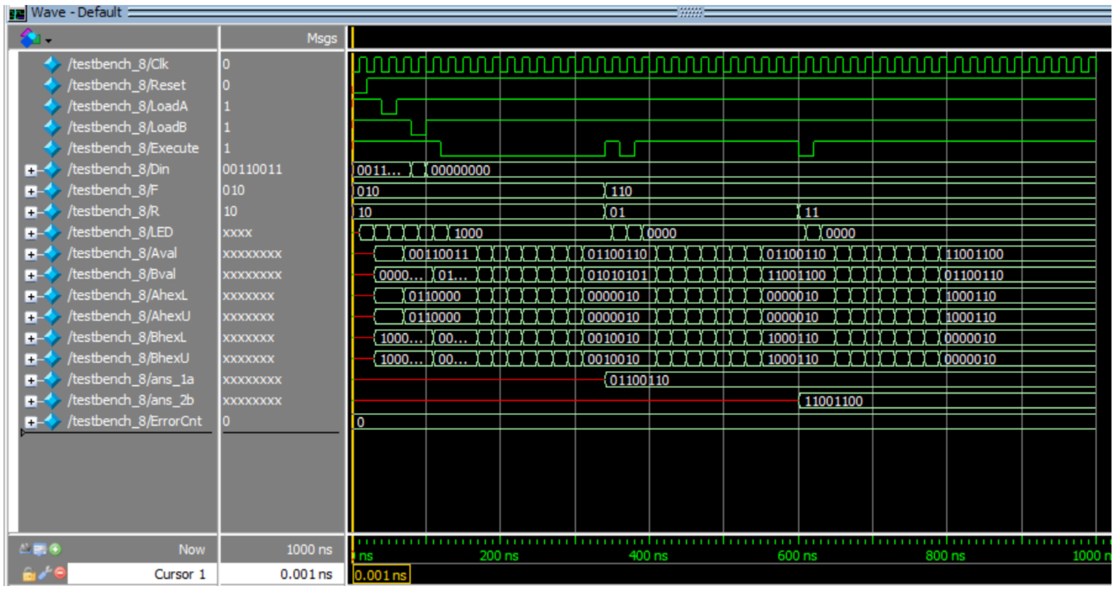


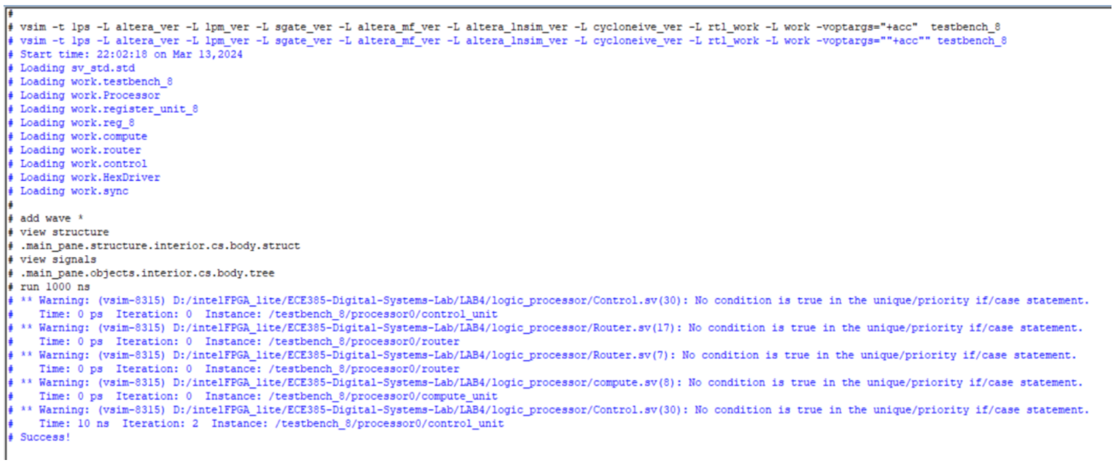
Fig 2: FSM Viewer Output, including 4 more state G, H, I, J

Although we encountered warnings during simulation that suggested the presence of untested conditions or non-fatal issues, our testbench effectively verified the processor's operations, as demonstrated by a zero-error count in our final simulation results.



```
Wave - Default
                        Msgs
/testbench_8/Clk        0
/testbench_8/Reset      0
/testbench_8/LoadA      1
/testbench_8/LoadB      1
/testbench_8/Execute    1
/testbench_8/Din        00110011
/testbench_8/F          010
/testbench_8/R          10
/testbench_8/LED        xxxx
/testbench_8/Aval       xxxxxxxx
/testbench_8/Bval       xxxxxxxx
/testbench_8/AhexL      xxxxxxx
/testbench_8/AhexU      xxxxxxx
/testbench_8/BhexL      xxxxxxx
/testbench_8/BhexU      xxxxxxx
/testbench_8/ans_1a     xxxxxxxx
/testbench_8/ans_2b     xxxxxxxx
/testbench_8/ErrorCnt   0
Now      1000 ns
Cursor 1 0.001 ns
```

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



```
#
# vsim -t 1ps -L altera_ver -L lpm_ver -L sgate_ver -L altera_mf_ver -L altera_lnsim_ver -L cycloneive_ver -L rtl_work -L work -voptargs="+acc"  testbench_8
# vsim -t 1ps -L altera_ver -L lpm_ver -L sgate_ver -L altera_mf_ver -L altera_lnsim_ver -L cycloneive_ver -L rtl_work -L work -voptargs=""+acc"" testbench_8
# Start time: 22:02:18 on Mar 13,2024
# Loading sv_std.std
# Loading work.testbench_8
# Loading work.Processor
# Loading work.register_unit_8
# Loading work.reg_8
# Loading work.compute
# Loading work.router
# Loading work.control
# Loading work.HexDriver
# Loading work.sync
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run 1000 ns
# ** Warning: (vsim-8315) D:/intelFPGA_lite/ECE385-Digital-Systems-Lab/LAB4/logic_processor/Control.sv(30): No condition is true in the unique/priority if/case statement.
#    Time: 0 ps  Iteration: 0  Instance: /testbench_8/processor0/control_unit
# ** Warning: (vsim-8315) D:/intelFPGA_lite/ECE385-Digital-Systems-Lab/LAB4/logic_processor/Router.sv(17): No condition is true in the unique/priority if/case statement.
#    Time: 0 ps  Iteration: 0  Instance: /testbench_8/processor0/router
# ** Warning: (vsim-8315) D:/intelFPGA_lite/ECE385-Digital-Systems-Lab/LAB4/logic_processor/Router.sv(7): No condition is true in the unique/priority if/case statement.
#    Time: 0 ps  Iteration: 0  Instance: /testbench_8/processor0/router
# ** Warning: (vsim-8315) D:/intelFPGA_lite/ECE385-Digital-Systems-Lab/LAB4/logic_processor/compute.sv(8): No condition is true in the unique/priority if/case statement.
#    Time: 0 ps  Iteration: 0  Instance: /testbench_8/processor0/compute_unit
# ** Warning: (vsim-8315) D:/intelFPGA_lite/ECE385-Digital-Systems-Lab/LAB4/logic_processor/Control.sv(30): No condition is true in the unique/priority if/case statement.
#    Time: 10 ns  Iteration: 2  Instance: /testbench_8/processor0/control_unit
# Success!
```

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:

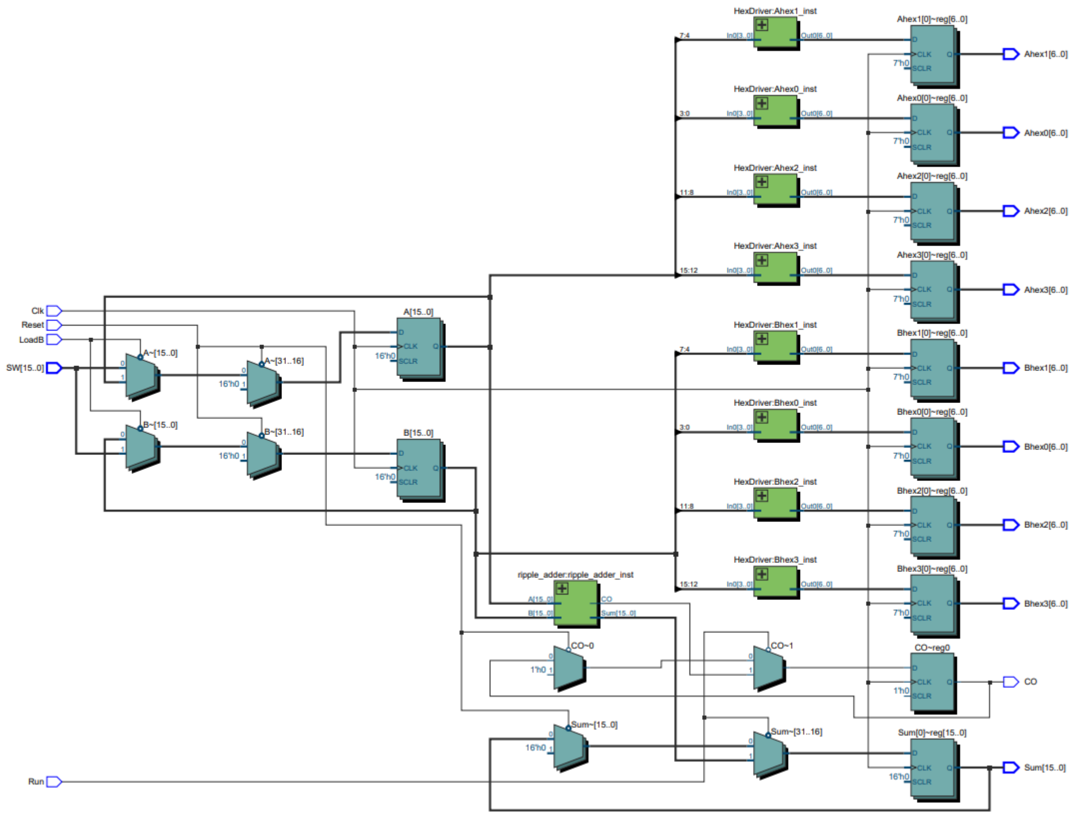


Fig-5: RTL Viewer of Adder Circuit

A. 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

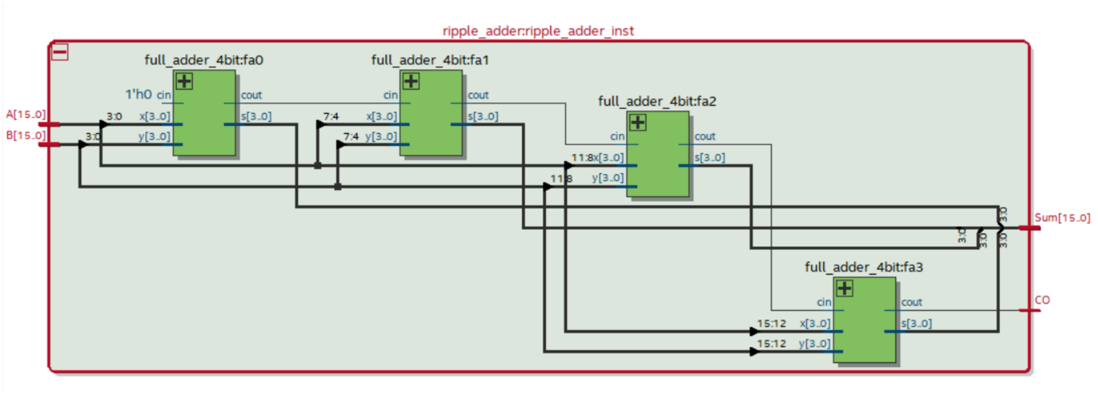
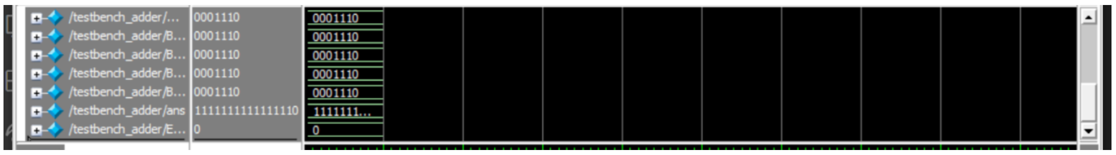


Fig-6: RTL Viewer of Ripple Adder Circuit

5.**Schematic Block Diagrams**



```
/testbench_adder/...   0001110            0001110
/testbench_adder/B...  0001110            0001110
/testbench_adder/B...  0001110            0001110
/testbench_adder/B...  0001110            0001110
/testbench_adder/B...  0001110            0001110
/testbench_adder/ans   1111111111111110   1111111...
/testbench_adder/E...  0                  0
```

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

2) 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

3) 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

B. 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.

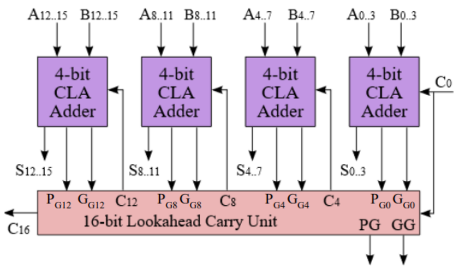


Fig-8: A 4x4-bit Hierarchical Carry-Lookahead Adder Block Diagram

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

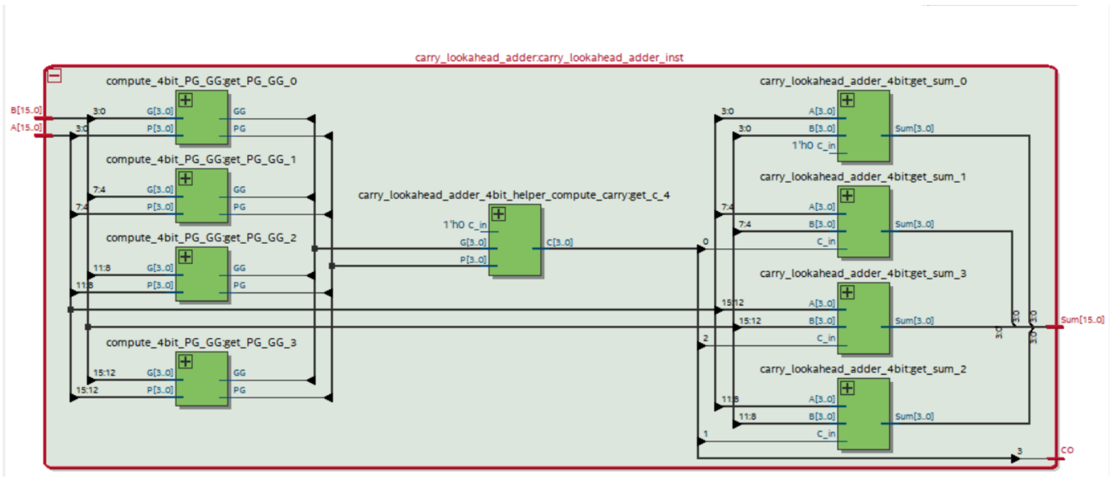

carry_lookahead_adder:carry_lookahead_adder_inst
compute_4bit_PG_GG:get_PG_GG_0
compute_4bit_PG_GG:get_PG_GG_1
compute_4bit_PG_GG:get_PG_GG_2
compute_4bit_PG_GG:get_PG_GG_3
carry_lookahead_adder_4bit_helper_compute_carry:get_c_4
carry_lookahead_adder_4bit:get_sum_0
carry_lookahead_adder_4bit:get_sum_1
carry_lookahead_adder_4bit:get_sum_3
carry_lookahead_adder_4bit:get_sum_2
B[15..0]
A[15..0]
G[3..0]
P[3..0]
GG
PG
1'h0 c_in
C[3..0]
A[3..0]
B[3..0]
Sum[3..0]
C_in
Sum[15..0]
CO

Fig-9: RTL Viewer of Lookahead Adder Circuit

5.**Schematic Block Diagrams**

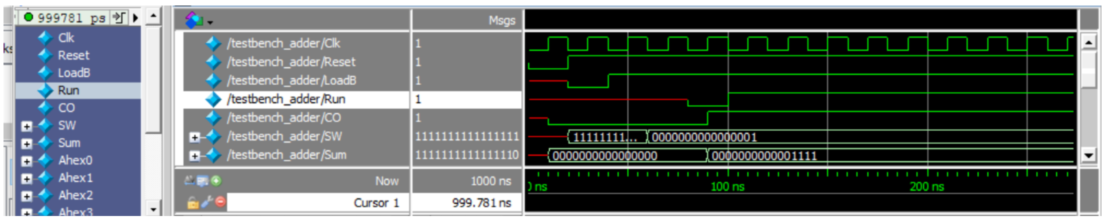


Fig-10: Schematic Block Diagrams of Lookahead Adder Circuit

6. Purpose and Operation of Each Module

1) 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.

2) 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.

3) 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.

4) 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

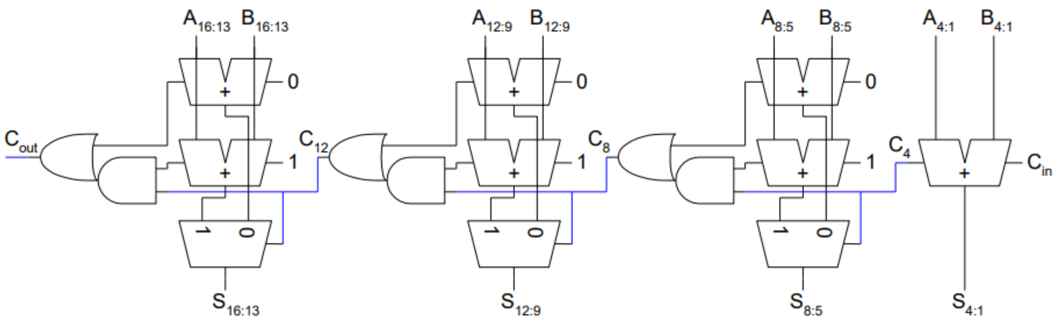


Fig-11: 16-bit Carry-Select Adder Block Diagram

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.

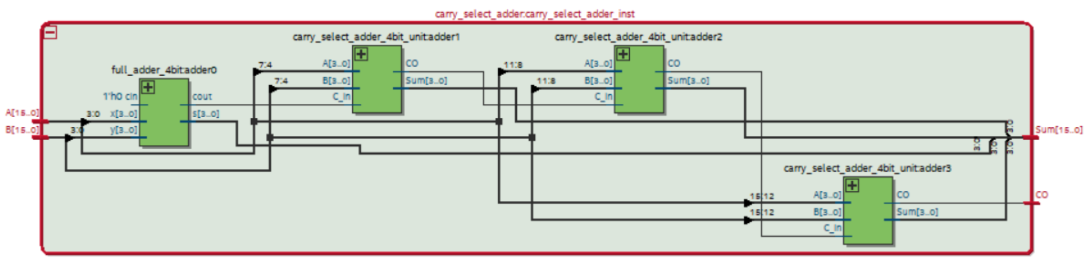
3.**Features:**

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.

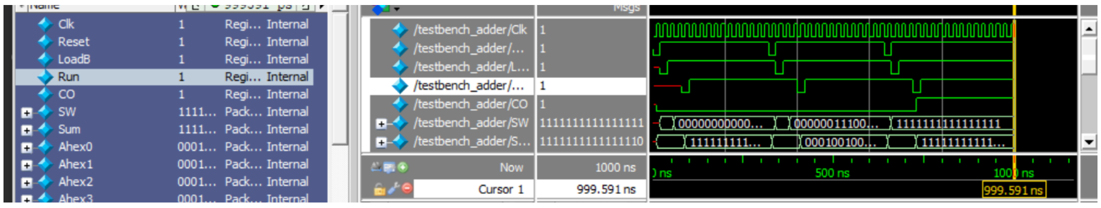
4. RTL View



carry_select_adder:carry_select_adder_inst
full_adder_4bit:adder0
carry_select_adder_4bit_unit:adder1
carry_select_adder_4bit_unit:adder2
carry_select_adder_4bit_unit:adder3
A[15..0]
B[15..0]
Sum[15..0]
CO
1'h0 cin
x[3..0]
y[3..0]
cout
s[3..0]
A[3..0]
B[3..0]
C_in
CO
Sum[3..0]

Fig-12: RTL Viewer of Select Adder Circuit

5.**Schematic Block Diagrams**



```
Clk      1        Regi... Internal
Reset    1        Regi... Internal
LoadB    1        Regi... Internal
Run      1        Regi... Internal
CO       1        Regi... Internal
SW       1111...  Pack... Internal
Sum      1111...  Pack... Internal
Ahex0    0001...  Pack... Internal
Ahex1    0001...  Pack... Internal
Ahex2    0001...  Pack... Internal
Ahex3    0001...  Pack... Internal

/testbench_adder/Clk     1
/testbench_adder/...     1
/testbench_adder/L...    1
/testbench_adder/...     1
/testbench_adder/CO      1
/testbench_adder/SW      1111111111111111   00000000000...   00000011100...   1111111111111111
/testbench_adder/S...    1111111111111110   111111111...     000100100...     111111111111...
Now       1000 ns        0 ns   500 ns   1000 ns
Cursor 1  999.591 ns     999.591 ns
```

Fig-13: Schematic Block Diagrams of Select Adder Circuit

6. Purpose and Operation of Each Module

1) 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.

2) 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

1. Compare the usage of LUT, Memory, and Flip-Flop of your bit-serial logic processor exercise in the IQT with your TTL design in Lab 3. Make an educated guess of the usage of these resources for TTL assuming the processor is extended to 8-bit. Which design is better, and why?

Bit-Serial Logic Processor (SystemVerilog):

LUT Usage: Bit-serial processors typically require a moderate number of LUTs because each operation is performed serially.

Memory Usage: Memory requirements are generally lower for bit-serial processors as they often use registers to store intermediate values.

Flip-Flop Usage: Flip-flops are used extensively in a bit-serial processor to hold the state of each bit being processed, as well as control signals for each cycle of the serial operation.

TTL Design:

LUT Usage: In a TTL implementation, LUTs are not a resource because TTL technology does not use configurable logic blocks.

Memory Usage: TTL designs might use separate memory chips if needed, but typically they would be built using discrete logic without large-scale on-chip memory.

Flip-Flop Usage: Flip-flops would still be used in a TTL design, but they would be discrete components rather than integrated parts of configurable logic blocks

The SystemVerilog implementation would be better. It would take up less physical space, consume less power, and likely operate at higher speeds due to the reduced propagation delay of integrated circuits compared to discrete TTL components.

2. For the adders, refer to the Design Resources and Statistics in IQT.16-18 and complete the following design statistics table for each adder. This is more comprehensive than the above design analysis and is required for every SystemVerilog circuit.

	Ripple Adder	**Lookahead Adder**	**Select Adder**
LUT	117	123	129
DSP	0	0	0
Memory	0	0	0
Flip-Flop	105	105	105
Frequency / MHz	184.23	236.57	204.16
Static Power / mW	98.5	98.5	98.5
Dynamic Power / mW	0	3.1	0
Total Power / mW	139.11	142.2	139.11

3) **Observe the data plot and provide explanation to the data, i.e., does each resource breakdown comparison from the plot makes sense? Are they complying with the theoretical design expectations, e.g., the maximum operating frequency of the carry-lookahead adder is higher than the carry-ripple adder? Which design consumes more power than the other as**

you expected, why?

Expected: The power consumption and operating frequency are correlated with the complexity of the design and the efficiency of logic implementation. The CLA should have the highest frequency but also the highest power consumption, while the CRA should show the opposite. The CSA's performance and power usage should fall between the two. If the plot reflects these relationships, it aligns with theoretical expectations. Any deviations would warrant a closer examination of the design implementation and potential optimizations.

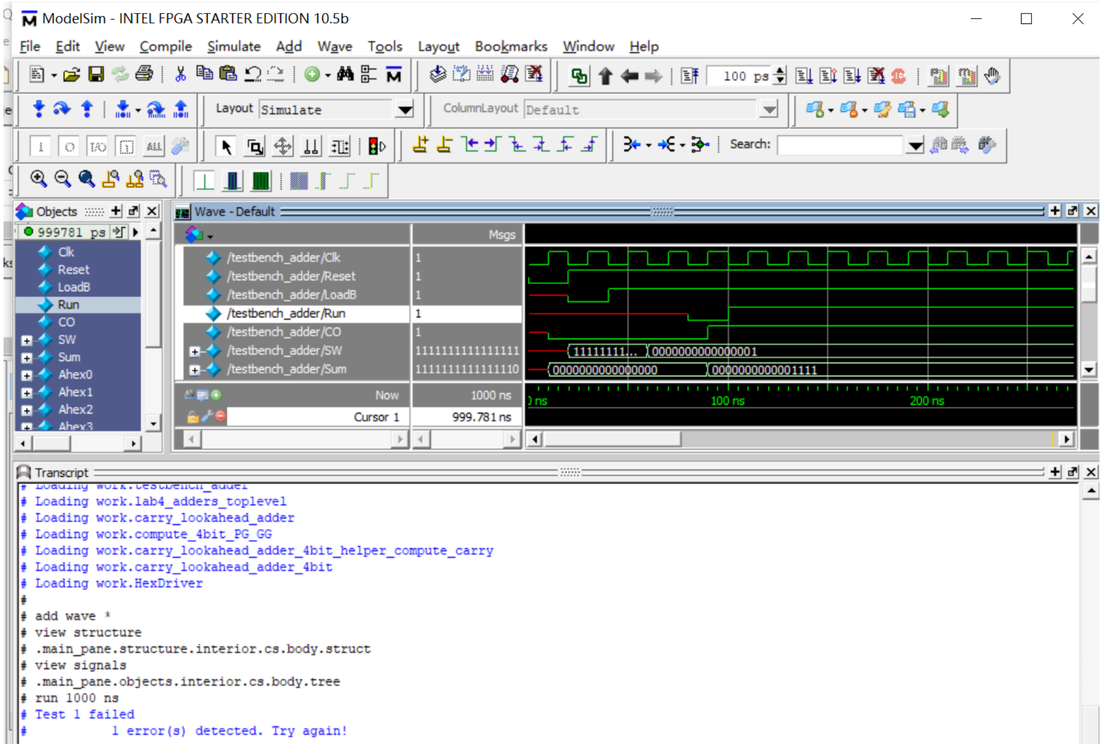
Real: The Lookahead Adder seems to offer the best performance in terms of speed, with a slight increase in total power consumption that can be justified by its significant improvement in operating frequency. The Ripple Adder, while being the least resource-intensive and the most power-efficient, offers the slowest performance. The Select Adder provides a middle ground, with better performance than the Ripple Adder without a significant power penalty. This analysis demonstrates that the trade-offs between speed, resource usage, and power consumption are consistent with the theoretical expectations for these adder designs.

6. Bug Log

➢ **<u>Description of all bugs encountered, and corrective measures taken:</u>**

1. Lookup Adder cannot pass the test case 1, but case 2, 3

It is strange that our code can run perfectly on the DE2 board, while it fails on the



```
# Loading work.testbench_adder
# Loading work.lab4_adders_toplevel
# Loading work.carry_lookahead_adder
# Loading work.compute_4bit_PG_GG
# Loading work.carry_lookahead_adder_4bit_helper_compute_carry
# Loading work.carry_lookahead_adder_4bit
# Loading work.HexDriver
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run 1000 ns
# Test 1 failed
#           1 error(s) detected. Try again!
```

It is because we picked up the PG4s and GG4s to be a module, which will increase the computing time and fail the test case 1. The simulation is not perfectly align with the logic circuit deployment, after we expand the expression, the bug is fixed.

2. Logic processor can not compile due to the LED setting

When we modified the code, we wrongly edited the LED[3:0] to LED[7:0], but the testbench is unchanged, so there was compile error. We realized that it is our misunderstanding of the LED functionality, in fact, 4 bit is enough to display all the information needed by the 8-bit logic processor.

3. Fail to load the Adder module into the DE2 board due to lack of USB_Blaster_Driver.

Reading the lab4 manual, we were confused to find the lack of USB Blaster. Thanks to the Google and classmate, we found a useful CSDN blog regarding the configuration of driver, which is interestingly operated by the DE2 board! After downloading it, everything works smoothly.

7.Conclusion

Through lab4, we practiced on the overall deployment of System Verilog together with the performance analysis as below:
Resource Usage: The Lookahead Adder required slightly more LUTs due to its complex carry computation logic, while the Ripple Adder was the most resource-efficient. The Select Adder's resource usage was the highest, but it offered a balance between speed and complexity.

Operating Frequency: The Lookahead Adder achieved the highest operating frequency, demonstrating the efficiency of parallel carry computation. The Select Adder had a moderate frequency, and the Ripple Adder had the lowest, reaffirming the impact of the carry propagation delay on the operating speed.

Power Consumption: All adders had identical static power usage, but the Lookahead Adder had a marginal increase in dynamic power, which translated into a slightly higher total power consumption. However, this was justified by its superior speed.

Understanding the trade-offs between different adder designs is crucial in real-world scenarios where specific constraints dictate the choice of one design over another. For example, battery-operated devices may favor power efficiency over speed, while high-performance computing systems may prioritize speed.

8. References

[1] Terasic Technologies Inc. (n.d.). Altera USB-Blaster Driver Installation Instructions. Retrieved from https://www.terasic.com.tw/wiki/Altera_USB_Blaster_Driver_Installation_Instructions
[2] Quanqueen. (2020, November 5). [学习日记——USB-Blaster 的驱动安装]. Retrieved from https://blog.csdn.net/quanqueen/article/details/109266720.
[3] ECE385 Faculty. (n.d.). Introduction to SystemVerilog (pdf)
[4] ECE385 Faculty. (n.d.). Introduction to Quartus Prime in the lab manual.
[5] KTTECH. (2017, January 31). ECE 385 Lab 4: Introduction to SystemVerilog and FPGA. Retrieved from https://kttechnology.wordpress.com/2017/01/31/ece-385-lab-4-introduction-to-system-verilog-and-fpga/. Teaching Assistant Blog