Lab2 Full Adder Design and Simulation

1. Introduction

In this lab, we learned basic grammar and structure of VHDL (Very-high-speed Hardware Description Language), and the basic flow path of hardware design. Also we get to know three important types of file in hardware design.
Our goal in this lab is to design a full adder and use a proper test bench to test our design. And more importantly, find difference of the results among different level of simulation.
The most important concern in this lab is the different types of delay, this is well discussed in the later of the report.
In this lab, we only focus on the simulation, thus there is no need for hardware test. The constrain file is not used.

2. Brief Introduction to Hardware Design

2.1. Hardware design flow

There are three important step in the hardware design, which are RTL (register transistor level) design, synthesis, and implement.
RTL design is the first step of hardware design. In this step, we write source files using hardware description language (in this lab, specifically, we use VHDL). RTL design is actually a step that design the overall logic function of the design, which is the core of the whole design. In this step, the basic logic unit is gate.
In synthesis step we (actually the computer can automatically do this) use LUT (look up table), buffer and other real hardware to replace the gate in the RTL design. But the whole function of these real hardware is the same to the gate in the previous step. Synthesis step is one transition step between the gate circuit to the FPGA circuit. In out FPGA, the basic logic unit is LUT rather than gate. It uses a more convenient way to implement the function of gates. LUT is something like ROM. Inputs indicates the address of the ROM while the output is the value stored in the corresponding address. In this case we avoid the complex gate circuit, because we use the truth table stored in the LUT to implement the same function as gate circuit. In general, what synthesis do is to find the truth table of the gate circuit in the RTL design and generate LUT circuit to realize the same truth table.
In implement step we use the LUT circuit generated in the previous step to connect the real hardware LUT in the FPGA. This includes how to use multiplex, how to distribute input and output port, how to wire different LUTs and banks and so on. In this step we place the LUTs in the real FPGA and wire them together (or we call it routing). After this step we can actually know how to realize out design in a specific FPGA. If there is no other error or warning then the only thing is to generate the bit stream and apply the design in the implement step to the FPGA. Note that we may use constrain file in this step because we have to interact with the hardware. The constrain file constrains the port define (like voltage), associated the port between hardware and our design, and also provide timing limit to the sequential circuits.

2.2. Design file introduction

The first one is the source file, which describes the basic circuit logic. The second one is the simulation file (or we called test bench), which describes what are the stimulus, and when and how to stimulate signals to the inputs. And the last one is the constrain file, which describes the port connection relation between the package of the device as designed and the real world hardware pins, and also, constrain the timing of each logic unit in the hardware. Note that there is no order relation between these three files, the order describing them is just to separate them apart.**(@!)**

3. Prelab Preparation

We design a full adder in this lab. Now we define the input and output of the full adder we design here. There are three input in total. They are A, B, Cin, which represent the addend A and addend B and the carry from a lower bit respectively. There are two output in total. They are $Sum, Cout$, which represent the result of the adding, the carry of current bit respectively.
The logic expression of the full adder is

$$Sum = A \text{ xor } B \text{ xor } Cin$$
$$Cout = (Cin \text{ and } A \text{ xor } B) \text{ or } (A \text{ and } B)$$

According to the logic expression of the full adder, we can draw the gate circuit as shown in **figure 3.1**. Note that this is just one optional circuit that satisfies the logic expression above. There are also many other logic circuits that can replace the one we have here.

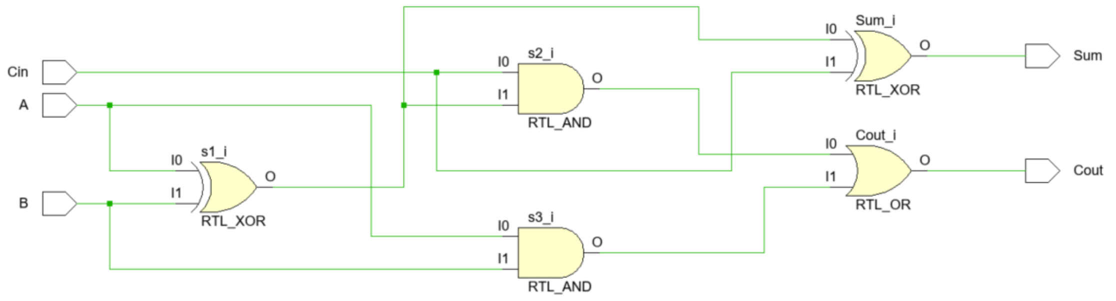
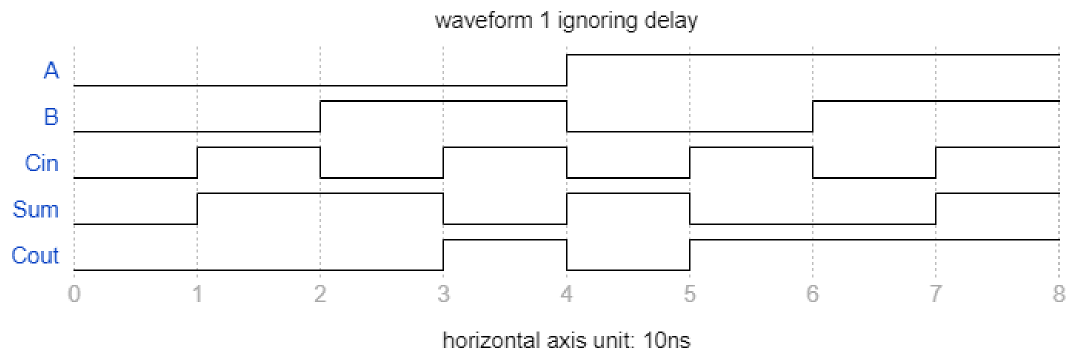


Figure 3.1

Thus we can also obtain the truth table as shown in **table 3.1**

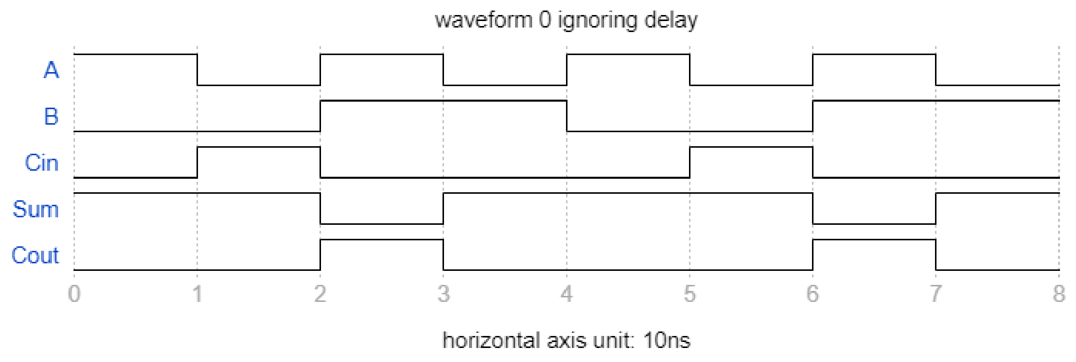
case	A	B	Cin	Sum	Cout
1	0	0	0	0	0
2	0	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0
6	1	0	1	0	1
7	1	1	0	0	1
8	1	1	1	1	1

The **waveform 1** illustrates the waveform when our inputs are given as the order in the truth table above.



Waveform 3.1

Thus similarly, we can obtain the waveform that the stimulus are given in the lecture.



Waveform 3.2

Note that here I ignore the delay. Because the result of our simulation is largely determined by how we write our programs if we consider delays.
Also the theoretical post synthesis and implement timing simulation is not considered in the prelab preparation. The reason is that we do not know how the computer design the synthesis before we really get the RTL design synthesized and implemented. The usage of LUTs and routing method also play essential role in delay.

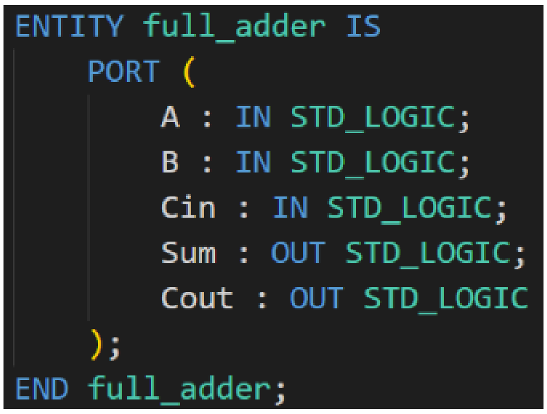
The delay analysis are well discussed in the later sections.

4. Source File Design

In this section, we go through the basic ideas about how to design a proper source file. The structure and some basic grammar are included in this section.

4.1. Port design

In a full adder there are 3 inputs and 2 outputs. We define them as shown in the **figure 4.1.1**.



```vhdl
ENTITY full_adder IS
    PORT (
        A : IN STD_LOGIC;
        B : IN STD_LOGIC;
        Cin : IN STD_LOGIC;
        Sum : OUT STD_LOGIC;
        Cout : OUT STD_LOGIC
    );
END full_adder;
```

Figure 4.1.1

Here we use STD_LOGIC as the port type, for it is more flexible in simulation and we can find our mistakes easier if there is any.

The port define is located under the entity define. Because every entity has to interact with the outside, otherwise it has no influence to our whole system. Thus once we define the entity, we define the interface, or port specifically in hardware design, of this entity.

Overall, here we define an entity named full_adder, and it has three inputs and two outputs shown in **figure 4.1.1**.

4.2. Architecture design

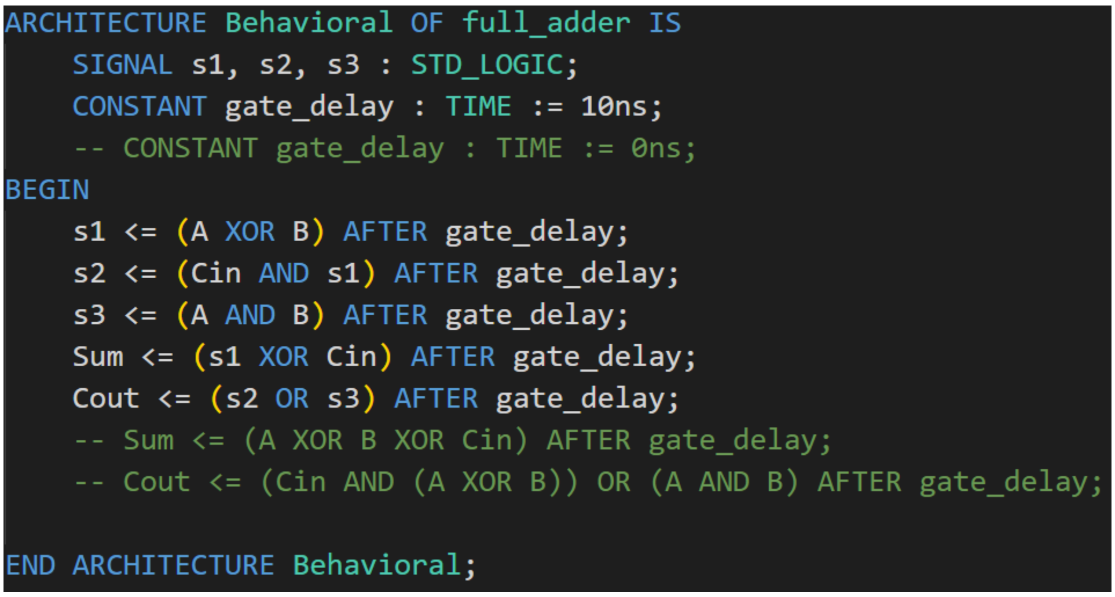
Architecture describes how the entity behaves when there is change in the inputs.

To makes it easier to write the architecture, we can introduce inner signals, which can be considered as wires, in the architecture. Obviously, they can only be used in this particular architecture. Also we can define constant in the architecture for further usage.

The core function of architecture is to describe the relations between the signals/wires, or in other words, how one signal is determined by the other ones.

Note that there are lots of circuits that can realize the same logic expression, we also have plenty of ways to write the architecture. Here we use two different ways to express the logic expression. The one with internal signals and the other one do not.

The architecture is shown in **figure 4.2.1**



```vhdl
ARCHITECTURE Behavioral OF full_adder IS
    SIGNAL s1, s2, s3 : STD_LOGIC;
    CONSTANT gate_delay : TIME := 10ns;
    -- CONSTANT gate_delay : TIME := 0ns;
BEGIN
    s1 <= (A XOR B) AFTER gate_delay;
    s2 <= (Cin AND s1) AFTER gate_delay;
    s3 <= (A AND B) AFTER gate_delay;
    Sum <= (s1 XOR Cin) AFTER gate_delay;
    Cout <= (s2 OR s3) AFTER gate_delay;
    -- Sum <= (A XOR B XOR Cin) AFTER gate_delay;
    -- Cout <= (Cin AND (A XOR B)) OR (A AND B) AFTER gate_delay;

END ARCHITECTURE Behavioral;
```

Figure 4.2.1

The comment part is another optional way to write the architecture, we will discuss this in detail in the later section.

Note that we use a delay assignment in this architecture. This delay would not influence synthesis and implement, this would only work when we are doing behavior simulation. This will be discussed in detail in later delay analysis.

These are all in our source file of this lab.

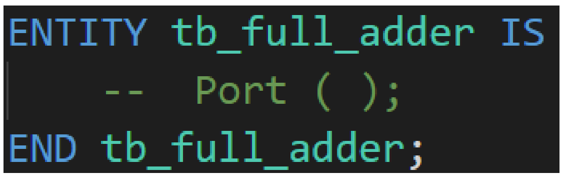
5. Simulation File Design (Test Bench Design)

Simulation is essential in digital circuits design, this is because before we proceed to the next step, we can check if there is noticeable mistakes. The earlier the mistake found, the easier we correct it.

But the simulation needs a file to describe its behavior. This is something like the simulator must know when it should assign what value to a signal. And the description is described in simulation file we are discussing now.

In VHDL simulation file, the whole system can be seen as an entity without input and outputs. Because all the signal is produced inner the simulation system. If there is input and output, there is no other stimulus and receiver to the inputs and outputs respectively.

The define of our top simulation is shown as **figure 5.1**



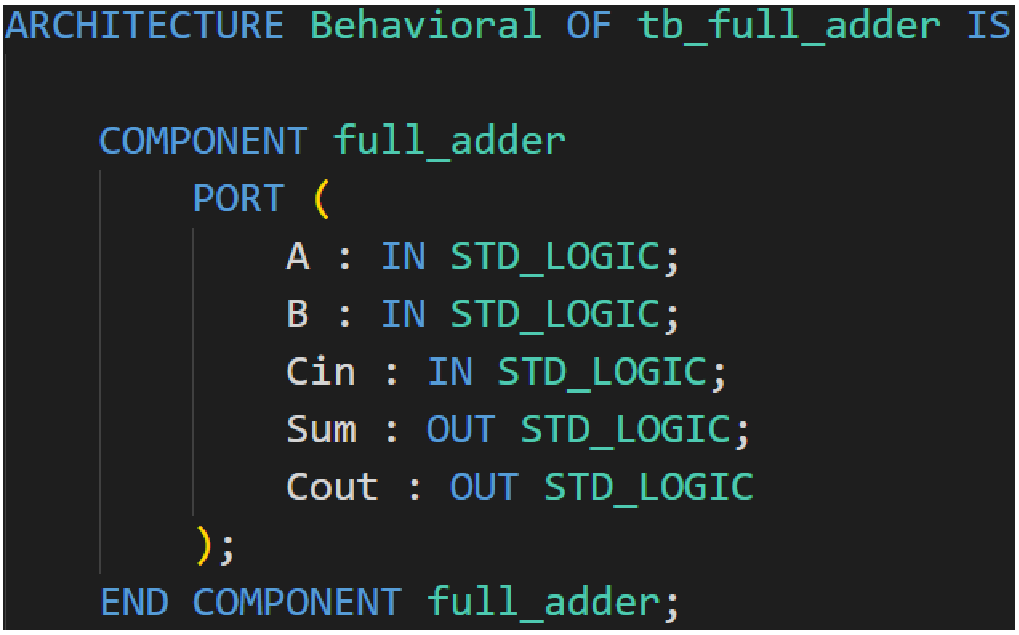
```
ENTITY tb_full_adder IS
    --  Port ( );
END tb_full_adder;
```

Figure 5.1

The core function of simulation is defined in the architecture. The simulation files need to interact with the entity we define in the source file. It should tell the entity defined in the source file what value should be assigned to which inputs, and the entity should tell the simulator what the output is.

To be exact, entity is a type of module. We can have lots of modules that has the same inner logic function, or even has totally the same logic expression. The module we call them component here. The component is actually the implementation of entity. In other word, entity is a general description of logic circuits, while the component is the one that can be referred in upper file level. However, the component is also an abstract concept, the real concrete module in the upper file level is instance. We define instances by referring to the components. Additionally, a components can be referred many times in a single file to instantiate many instances. And we

The define of component full_adder in the architecture of our test bench is shown in **figure 5.2**



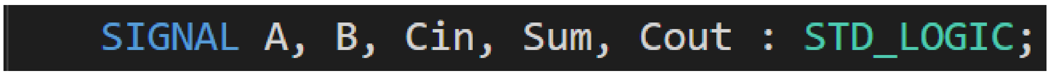
```
ARCHITECTURE Behavioral OF tb_full_adder IS

    COMPONENT full_adder
        PORT (
            A : IN STD_LOGIC;
            B : IN STD_LOGIC;
            Cin : IN STD_LOGIC;
            Sum : OUT STD_LOGIC;
            Cout : OUT STD_LOGIC
        );
    END COMPONENT full_adder;
```

Figure 5.2

Note that the input and output define should be same as the one in entity define, this is the mapping of input and output usage. If there is no definition of the input or output that is defined in the entity, the signal would be automatically attached to ground in default.

Here, in this architecture, we need drive the input signal of the instance (this will be introduced later). But we can not directly assign value to the port of the instance. We need a signal to drive the input port of the instance. The inner signals defined in this architecture are shown in **figure 5.3**

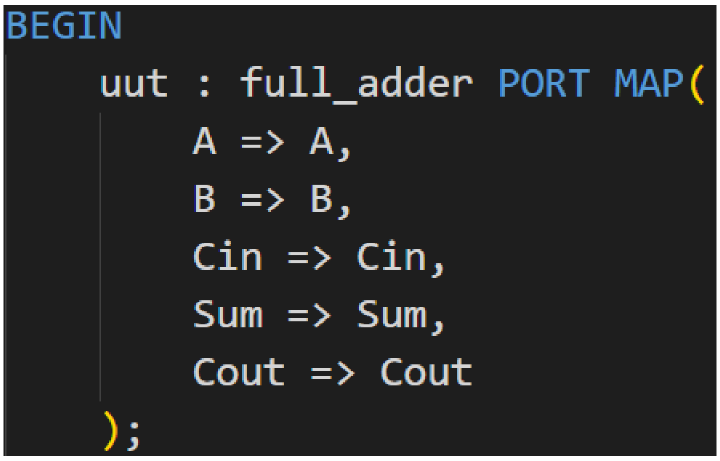


```
    SIGNAL A, B, Cin, Sum, Cout : STD_LOGIC;
```

Figure 5.3

Note that the name of these signal can be the same as the input port. In VHDL, the signal and the ports has different attributes to distinguish them, thus it can be the same name. Just do get confused in programing.

Next step is to create an instance of the component. Also, an instance without any input and output can not interacts with other instance in the same level, which is meaningless. Thus during the instantiation we should also associate the inner signal with the port of the instance. This step is called mapping, and the mapping is shown in the **figure 5.4**

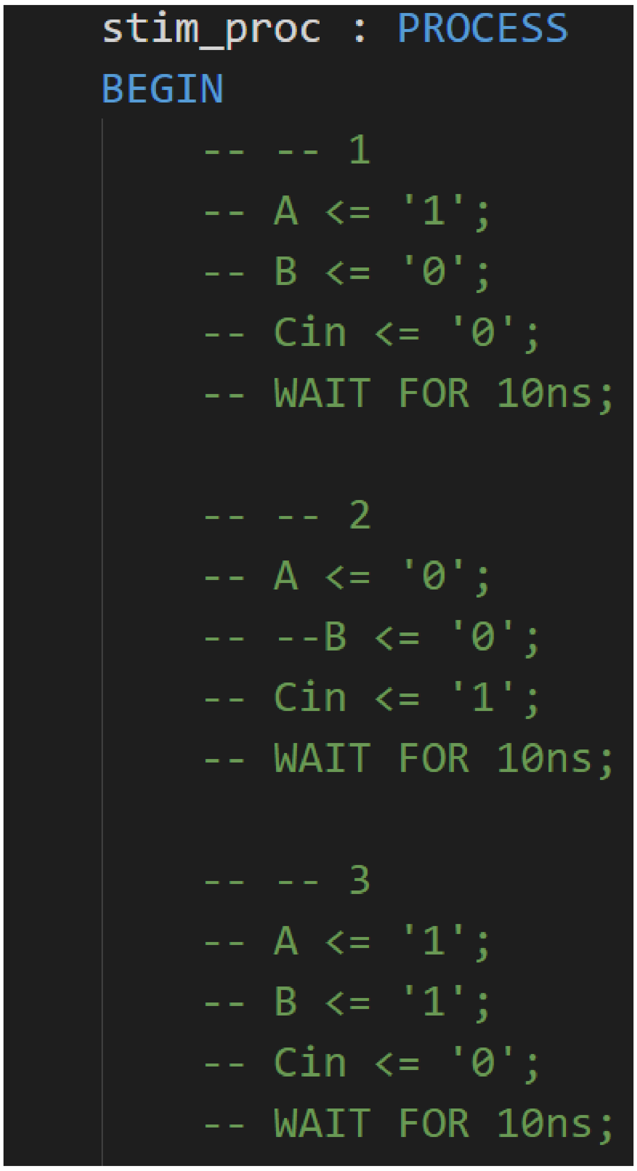


```
BEGIN
    uut : full_adder PORT MAP(
        A => A,
        B => B,
        Cin => Cin,
        Sum => Sum,
        Cout => Cout
    );
```

Figure 5.4

The configuration of the simulation file is completed from now. We now has wired all lines and instances. The next step is to tell the simulator what to do.

This process we call stimuli process. In this part, we assign value to the inner signal at exact time. The part of the stimuli process is shown in **figure 5.5**



```
stim_proc : PROCESS
BEGIN
    -- -- 1
    -- A <= '1';
    -- B <= '0';
    -- Cin <= '0';
    -- WAIT FOR 10ns;

    -- -- 2
    -- A <= '0';
    -- --B <= '0';
    -- Cin <= '1';
    -- WAIT FOR 10ns;

    -- -- 3
    -- A <= '1';
    -- B <= '1';
    -- Cin <= '0';
    -- WAIT FOR 10ns;
```

Figure 5.5

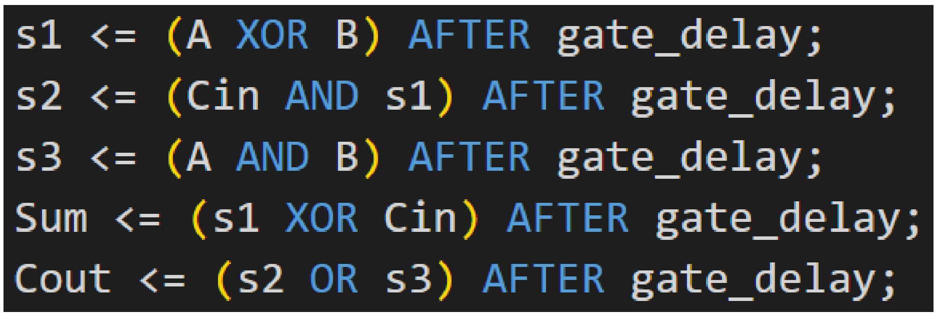
This is a process and thus the code is executed line by line. The WAIT FOR 10ns means the simulation will wait at this line for 10ns and then proceeds. This waiting is necessary because if there is no waiting all the assignment will occur at the same time and this what we do not want happen.

6. Behavioral Simulation and RTL Analysis

In this section, we use the code in the lecture note as our source file and simulation file.

6.1. Simulation result

The logic expression in the source file in this section is shown in **figure 6.1.1**.



```
s1 <= (A XOR B) AFTER gate_delay;
s2 <= (Cin AND s1) AFTER gate_delay;
s3 <= (A AND B) AFTER gate_delay;
Sum <= (s1 XOR Cin) AFTER gate_delay;
Cout <= (s2 OR s3) AFTER gate_delay;
```

Figure 6.1.1

Where gate_delay=10ns in type of TIME.

The simulation result is shown in **figure 6.1.2**

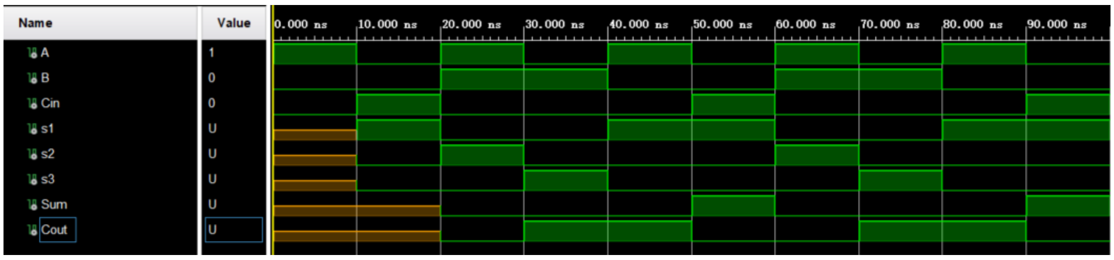


Figure 6.1.2

The result is the same as the one in the lecture note

We can clearly see that there is 10ns delay for the inner signals s1 to s3, and there are 20ns delay for the output signals. The 10ns delay is occurred by our attribution in assignment statement, the AFTER gate_delay. This means the value of the expression in right side of the assignment symbol will be calculated at each execution, but it will be stored in a buffer rather than be assigned to the left side signal. In the next execution, the value in the buffer will be assigned to the left side and the buffer may update a new value if the value of the expression in the right side of the assignment symbol changes.

Thus, in this case, if we shift the waveform of s1 to s3 left by 10ns, we will see that the waveforms of s1 and s3 are perfectly matched. However, waveform of s2 has something wrong, for example, at time=20ns and the 10ns behind it. It is the value of (Cin at 10ns ‘AND’ s1 at 10ns). But the value of s1 at 10ns is actually the value of (A at 0ns ‘XOR’ B at 0ns), thus the value of s2 is incorrect due to the delay of s1 and non-delay of Cin.

This is similar in Sum and Cout, but for their values are all determined by s1 to s3, they will be assigned after s1 to s3 have been assigned and wait for another 10ns. Thus in total they have 20ns delay. But due to the different delay in their assignment, their waveforms are also incorrect.

Briefly, if there is any difference in delay among the assignment signal of a signal, it will not get correct answer.

In the other side, if for any signal, all its assignment signal are correct and they all have synchronized delay, the signal will be correct and have another delay.

For example, if there is a signal only related to signal s1 and s3, then it would be correct and have 20ns delay.

# 我觉得这里没有解释清楚，这里用中文补充说明一下：
# 就是说，对于任意一个信号，如果它的赋值信号没有错误并且它们的延迟都统一的话，那
# 么这个信号的结果也就没有问题，否则就回出错。
# 比如说如果有一个信号是仅仅关于 s1 和 s3 的，因为 s1 和 s3 正确，并且都同步延时了
# 10ns，那么这个信号实际上也是正确的，并且延时 20ns。

Another note in this simulation. We notice that s1 and s3 are delayed for 10ns because they are determined directly by the input. Sum, Cout and s2 are delayed for 20ns because they are related to s1 to s3 somehow. But it seems that only Sum and Cout are delay for 20ns for there are 20ns unknown at the beginning. There is only 10ns known at the beginning of s2.
The truth is s2 still has 20ns delay, however, for s2 is determined by (Cin at 10ns 'AND' s1 at 10ns). And Cin at 10ns is 0, thus whatever s1 at 10ns is, the result of 'AND' is still 0. Thus s2 can be determined at 10ns.

Another note is that delay in these incorrect signal is actually meaningless, if they are incorrect, there is nothing to do with delay.

6.2. RTL analysis

This is the same figure as **figure 3.1**. For easier explanation I repeat it here and use different color to highlight the wires. The color means the level of the

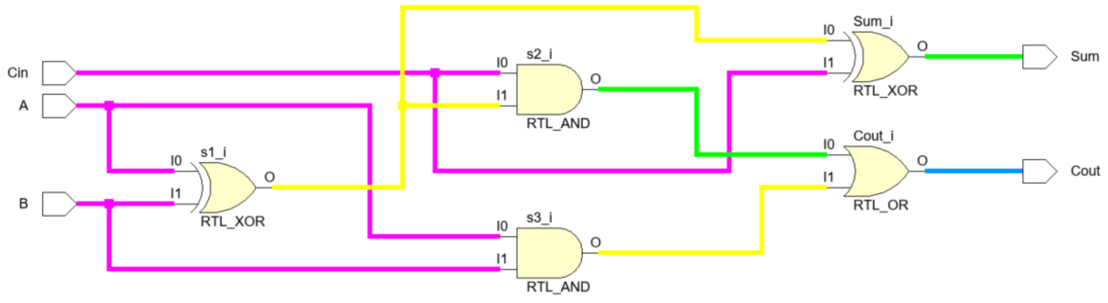


Figure 6.2.1

The level of the colors are pink, yellow, green, blue. It means after pink signals be assigned, the yellow signal can be assigned in next delay period. And this is the same for the other two colors.
Thus in this figure we can clearly see the delays of each line and each gate. The maximum delay in this figure can be regarded as 30ns which occur at the output Cout, because its value is determined by s2 and s2 is determined by s1.
Note that if the inputs of a gate are in different color, the output of the gate is definitely incorrect.

7. Post-synthesis timing Simulation and Synthesis Analysis

In this section, we also use the code in the lecture note as our source file and simulation file.

7.1. Simulation result

The simulation result is shown in the **figure 7.1.1**.

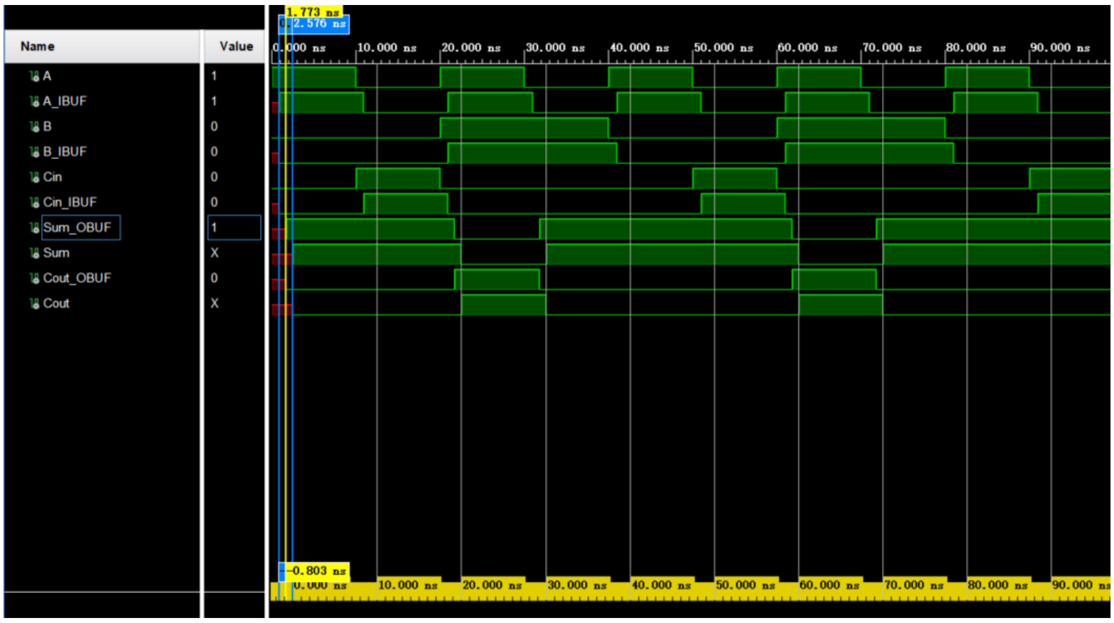


Figure 7.1.1

We can also see that there is delay when signal is passing through the system. But this one much smaller than the one we have in the behavior simulation (because we set the delay for too long). Despite of the initiate delay of approximately 0.971ns, the remaining delay is stabled at 0.803ns due to the property of the hardware LUT and buffer.

In this case, the result of the waveform do not violate with the common sense that a full-adder should behave.

The problem here is that there may be waveform shift due to the delay. And the shift will pass on from signal to next. But because it is in synthesis, the delay has been synchronized, there is no error.

7.2. Synthesis analysis

The **figure** 7.2.1 below illustrates the synthesis schematic. Also I use different color to demonstrate the different level of delay.

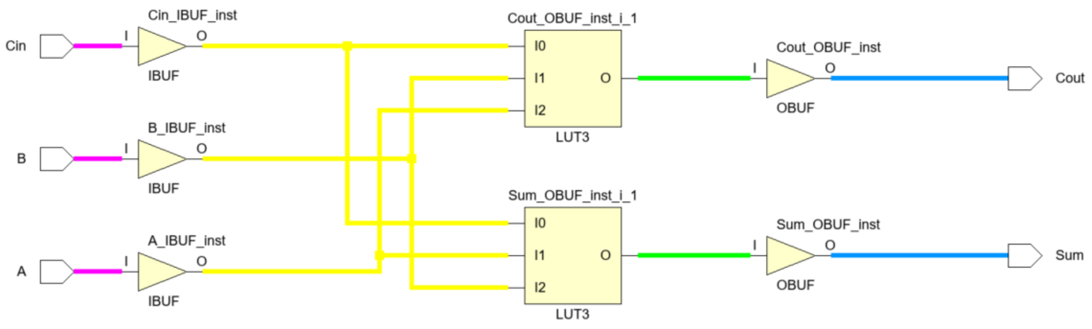


Figure 7.2.1

We can see that the LUT completely replace the complex gate. And thus it make it possible to get signal synchronized because there is no temporary inner signal to pass the signals down. Remind that in the previous section we know that if the input of an instance is all in the same delay, and the inputs are all correct, the output of the instance is also correct. This figure illustrates this well. Note that for LUT is an electric component, there must be buffer at the input and output side of it. The buffer will also cause a delay here.

From the simulation result in the previous part, we know the average delay is 0.803ns. Therefore in synthesis, the LUT and buffer has the same delay value though LUT may contain more transistors inside.

8. Post-implementation timing Simulation and Implementation Analysis

In this section, we also use the code in the lecture note as our source file and simulation file.

8.1. Simulation result

The simulation result is shown in the **figure 8.1.1**.

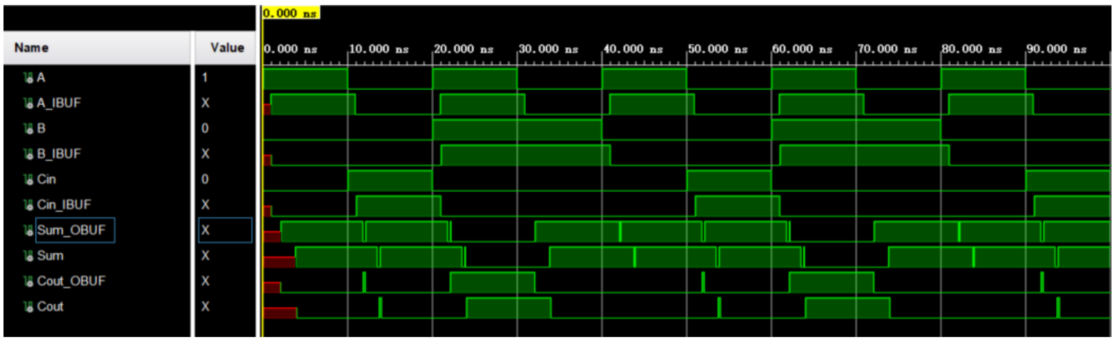


Figure 8.1.1

We can see that the overall graph of the result of this simulation is quite similar to the one in the post-synthesis timing simulation. But there is some different. The first one is that the delay of each delay level as we defined in the previous section is no longer the same. The rising edge of the A_IBUF, B_IBUF and C_IBUF are different slightly. **Figure 8.1.2** illustrates this. There are only several ps difference.

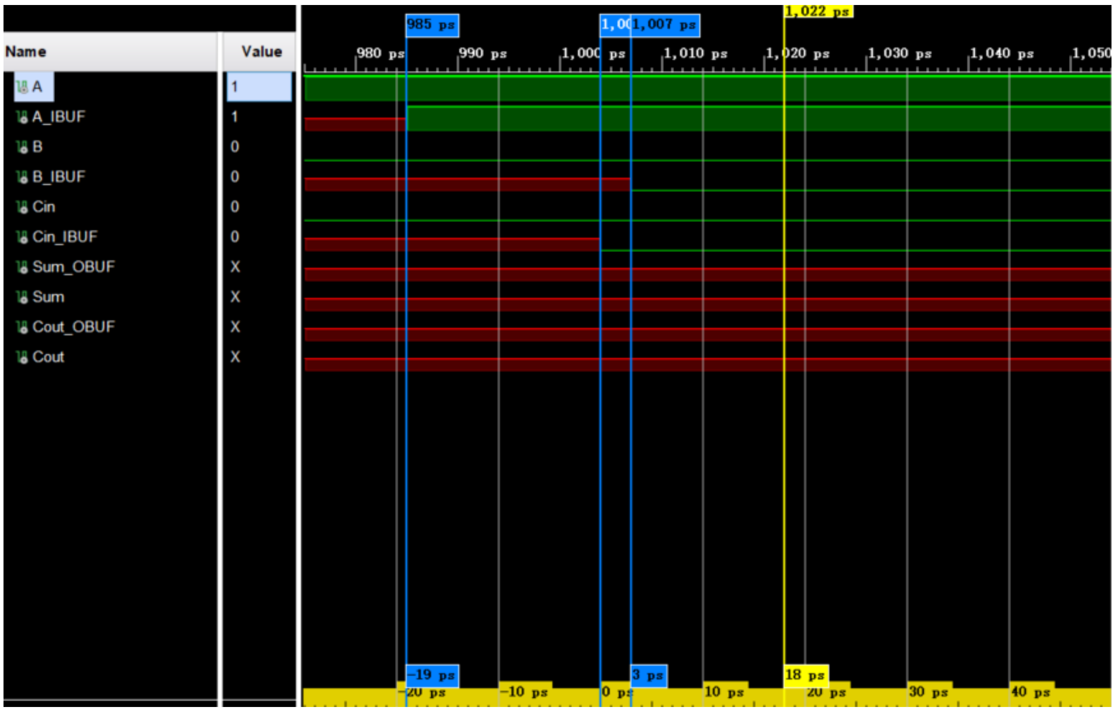


Figure 8.1.2

This kind of delay is caused by the moving of the electric field. And this will be discussed in detail in the later part, **implementation analysis**.

There is another delay in the post-implementation timing simulation. This kind of delay is caused by the latency of the unit like buffer and LUT. This kind of delay is also shown in the post-synthesis timing simulation. But there is something different. Here the delays between buffer and LUT are different. Even the delay of different LUT and different buffer is different. The **figure 8.1.3** illustrates this

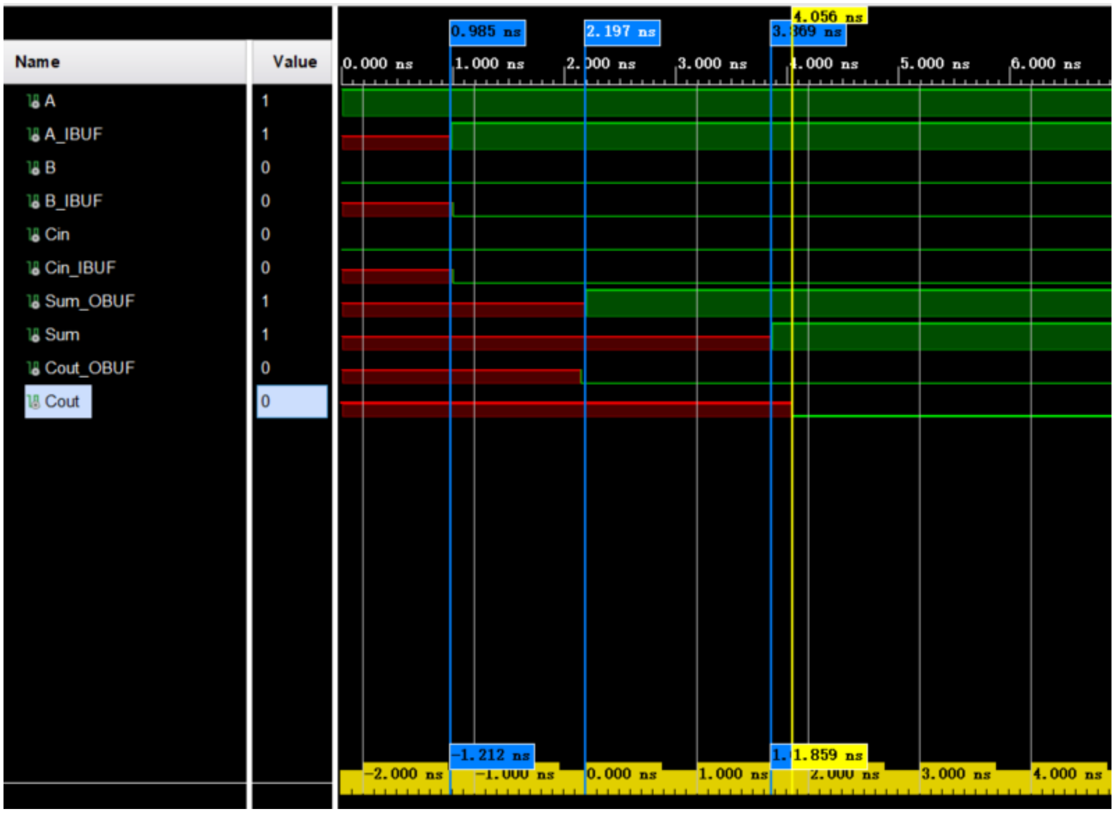


Figure 8.1.3

We can find that the delay of an input buffer is approximately 0.985ns, the delay of the LUT of Sum is approximately 1.212ns, the delay of the LUT of Cout is approximately 2.884ns, and the delay of an output buffer is approximately 1.672ns (output buffer for Sum) and 1.902ns (output buffer for Cout) (these two values are obtained in the later of the waveform). To obtain a more specific value of delay, we can use the delay when the whole system is stable, rather than the initial situation. Sometimes the initial value are correct, but other times there are some uncertain signal in the circuits, thus there might be some difference between the delay we get in the initial than the real delay. Also, we use the 'approximately' because sometimes it is hard to eliminate the influence of wire delay. There may be several ps difference in different buffer.

There is another note for the delay of LUT. LUT is actually a ROM. And in real hardware, LUT is actually a combined LUT, thus the inputs value will also influence the response time of it. Different input signal pattern may cause different latency. This latency is the cause of the spikes in the waveform. In digital circuits, there is a more general name for the cause of these spikes, which is the race and competition phenomenon (竞争冒险现象， 不确定翻译是否准确 qwq).

Here is an example on how to analysis these spikes.

Figure 8.1.4 illustrates the part of the waveform. We focus on the four marked blue cursors. They are divided into two groups, and the delay of each group is 0.839ns. The delay here is the delay between Cin_IBUF and Sum_OBUF. Cin_IBUF is the output of the input buffer of input Cin, and this signal is passed to the input of LUT Sum. Sum_OBUF is the output of LUT Sum, and this signal is passed to input of the output buffer of the output Sum. This delay tells us the how many time (in this case 0.839ns) the LUT will takes if input Cin_IBUF has changed.

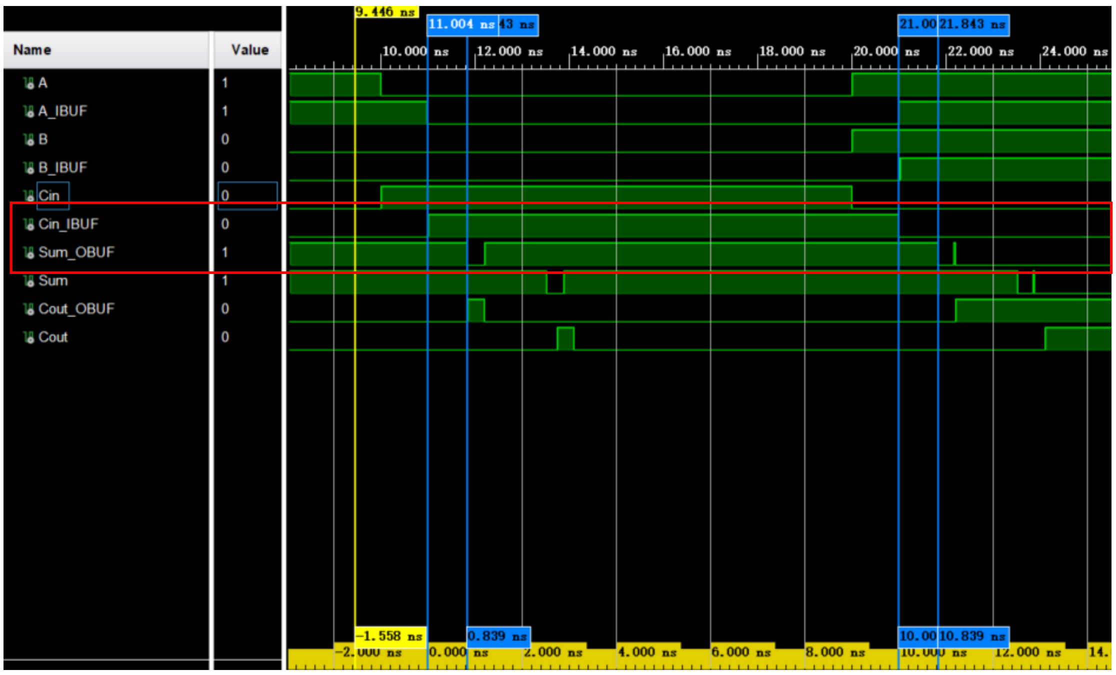


Figure 8.1.4

Similarly, we can get the time the LUT Sum takes when A_IBUF and B_IBUF has changed. These analysis are illustrated in **figure 8.1.5 and figure 8.1.6** respectively. Thus the delay of LUT Sum in change of A_IBUF is 1.212ns, and the delay of LUT Sum in change of B_IBUF is 1.147ns.

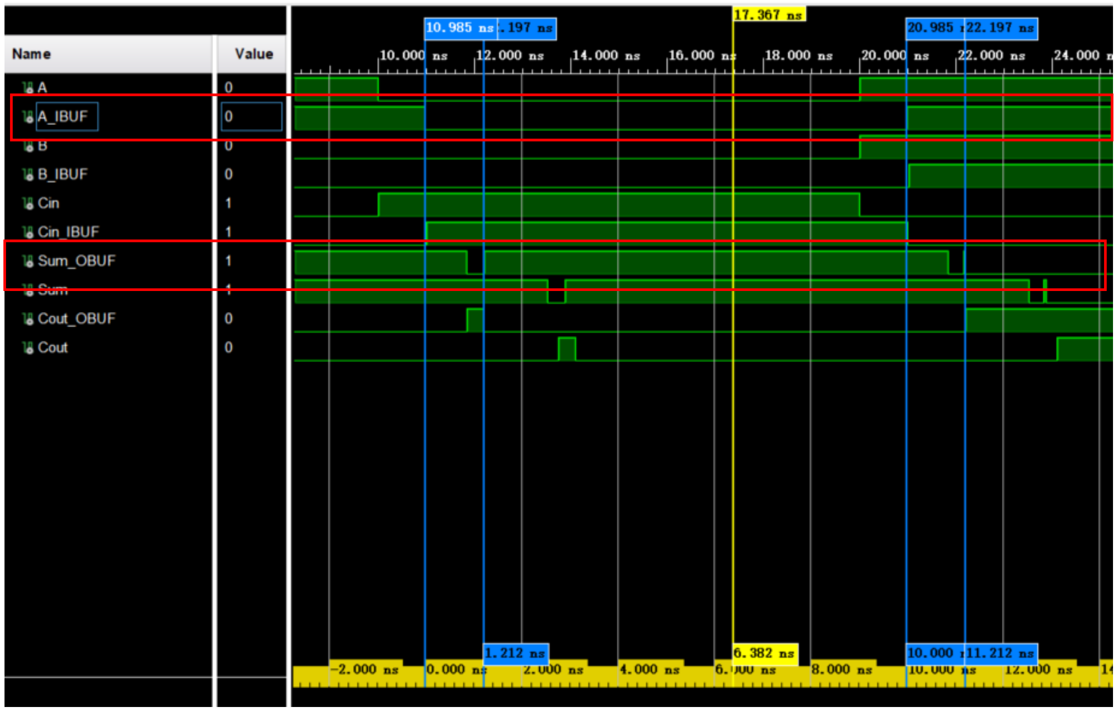


Figure 8.1.5

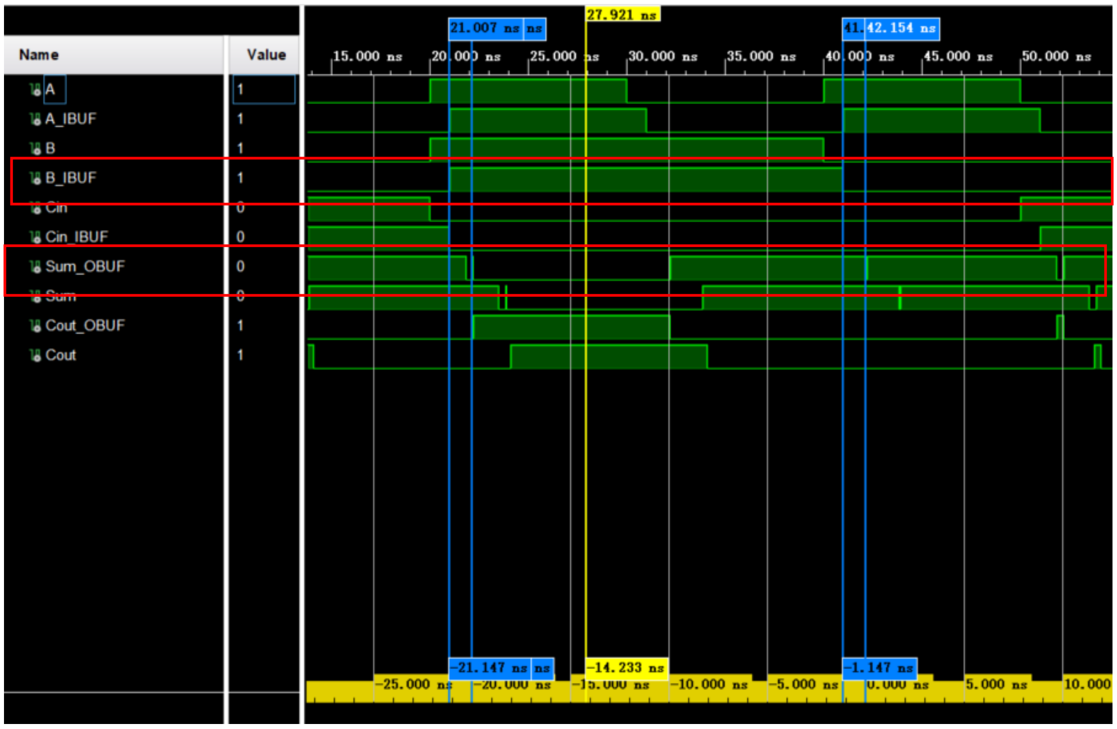


Figure 8.1.6

In summary, the three delays are 0.839ns (Cin_IBUF), 1.147ns (B_IBUF) and 1.212ns (A_IBUF). If there is a value change in A_IBUF, B_IBUF and Cin_IBUF simultaneously, the Sum_OBUF will first response to the change of Cin_IBUF because the delay is the least (0.839ns), then it will response to the change of B_IBUF (1.147ns), the last response is done after the longest delay 1.212ns, when it response to the change of A_IBUF.

Now we turn back to see these spikes. Take the spike at approximately 22.2ns for example. The **figure 8.1.7 to figure 8.1.9** illustrates the different response of Sum_OBUF to the change of A_IBUF, B_IBUF and Cin_IBUF. At time 21.004ns, Cin_IBUF drops from 1 to 0. At time 21.843ns, the LUT Sum response to this change. At this time point, the LUT has inputs A_IBUF=0, B_IBUF=0, Cin_IBUF=1->0. Thus the output value change from 1 to 0 naturally.

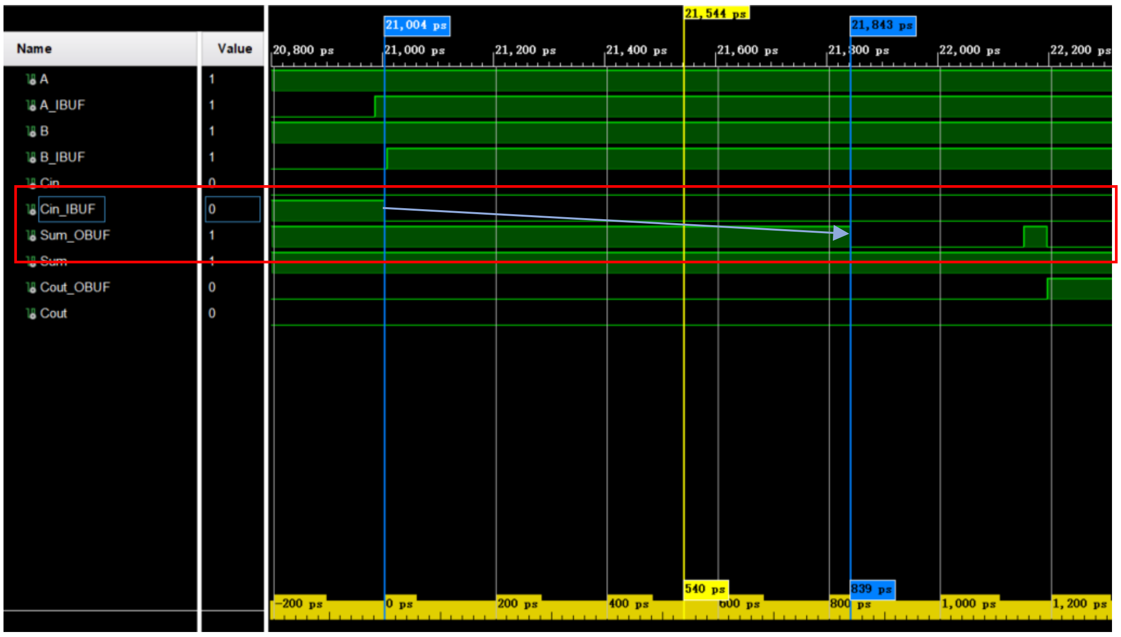


Figure 8.1.7

Similarly at time 21.004ns, B_IBUF rises from 0 to 1. At time 22.154ns, the LUT Sum response to this change. At this time point, the LUT has inputs A_IBUF=0, B_IBUF=1, Cin_IBUF=0 (Cin_IBUF is 0 because this happens at time 21.843ns). Thus the output value change from the temporary value 0 to another temporary one 1 naturally.

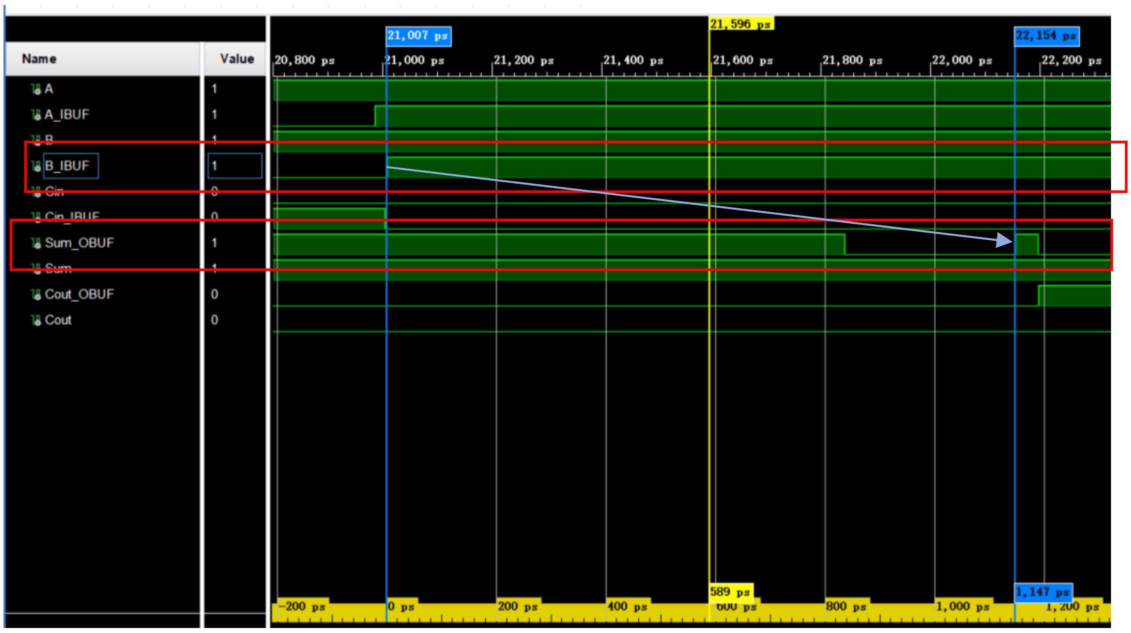


Figure 8.1.8

Similarly at time 20.985ns, A_IBUF rises from 0 to 1. At time 22.197ns, the LUT Sum response to this change. At this time point, the LUT has inputs A_IBUF=1, B_IBUF=1, Cin_IBUF=0. Thus the output value change from the temporary value 1 to the stable one 0 naturally.

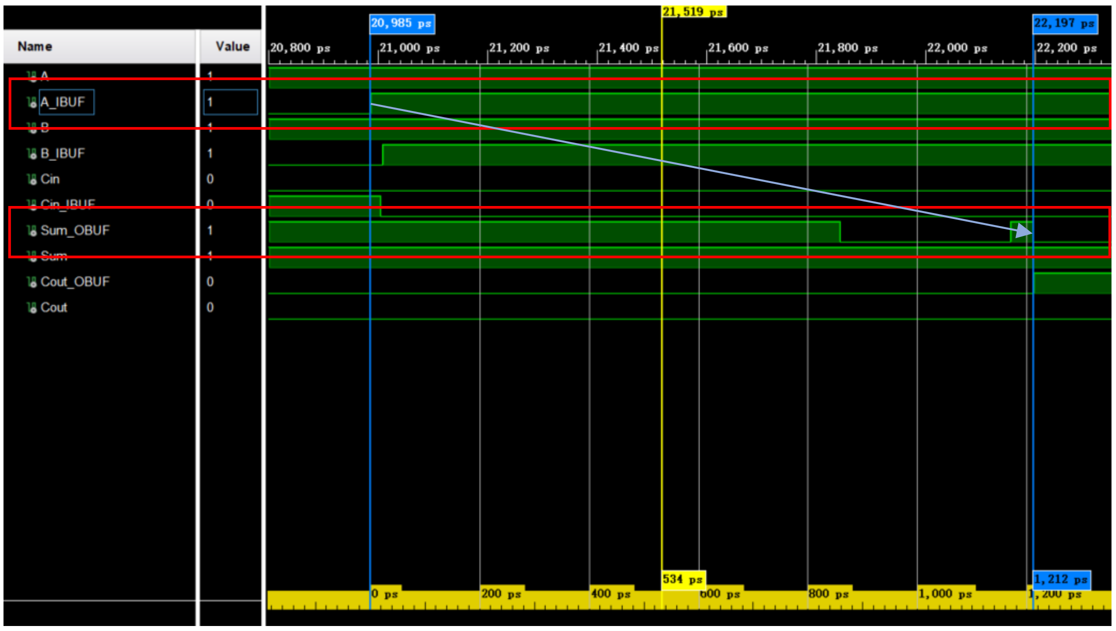


Figure 8.1.9

Therefore, the spike here is caused by the difference response delay of the LUT when the three inputs get changed. Also the dip is caused by the same reason.

8.2. Implementation analysis

The **figure 8.2.1** illustrates the overall device placing result.

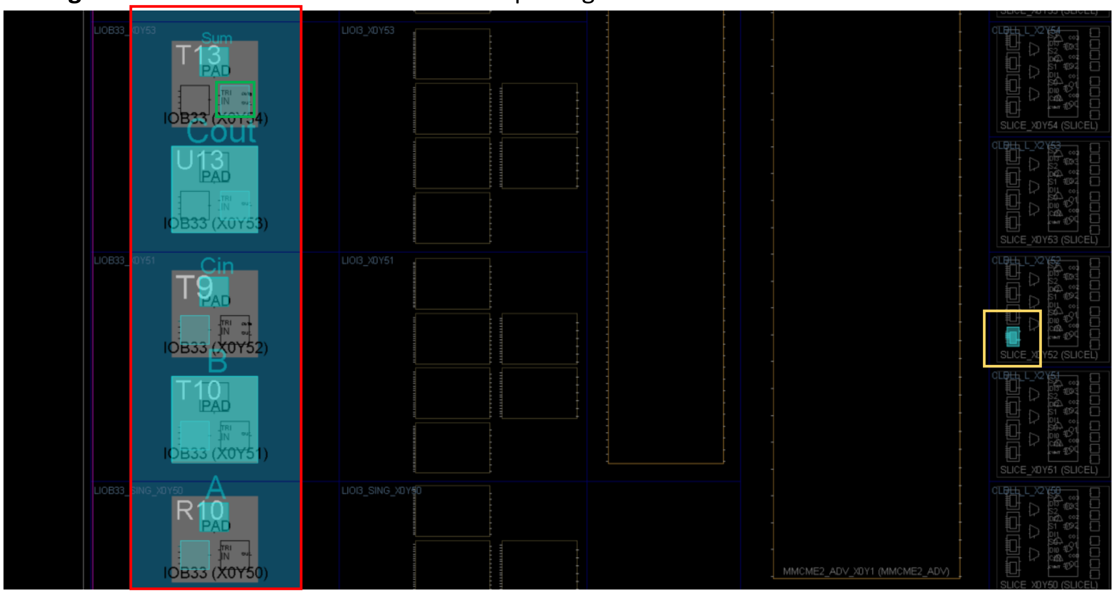


Figure 8.2.1

The right side of the figure is the logic array side of the block. The cyan highlighted module is our LUT defined in the synthesis, which is surrounded by a yellow rectangle. This is where our core logic process locates. This part will be discussed later. The left side of the figure is the port side of the block in the FPGA chip, surrounded by a red rectangle. Here we can find our designed five inputs and output of out entity. The ports of the block are actually integrated with multiplexer and buffer and other sub component essential to the port. For example, the component that surrounded by the green rectangle is the buffer defined in the synthesis. And the mid-side of this figure is the routing chips that can be set to gets different section of block connected. This can be also considered as wiring modules. Actually the routing chips are everywhere in the FPGA.

Figure 8.2.2 illustrates the outside package distribution of this implementation.

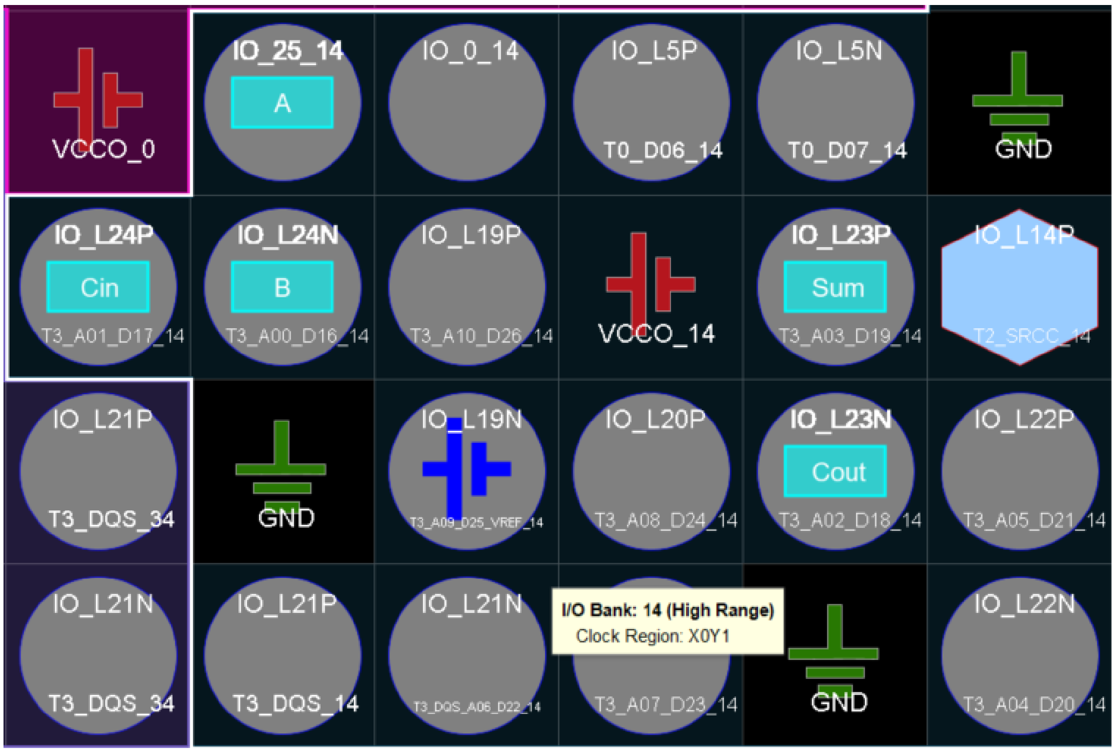


Figure 8.2.2

Our implementation is placed in the block whose clock region is X0Y1, and the interactive port to the outside is bank 14. Here we can see the mapping of the input we define and the real port of the hardware device. For example, the input port A is associated to inout port IO_25_14. But the output and high resistance is disabled because A is a pure input port in this case.

The **figure 8.2.3** illustrates the overall routing results.

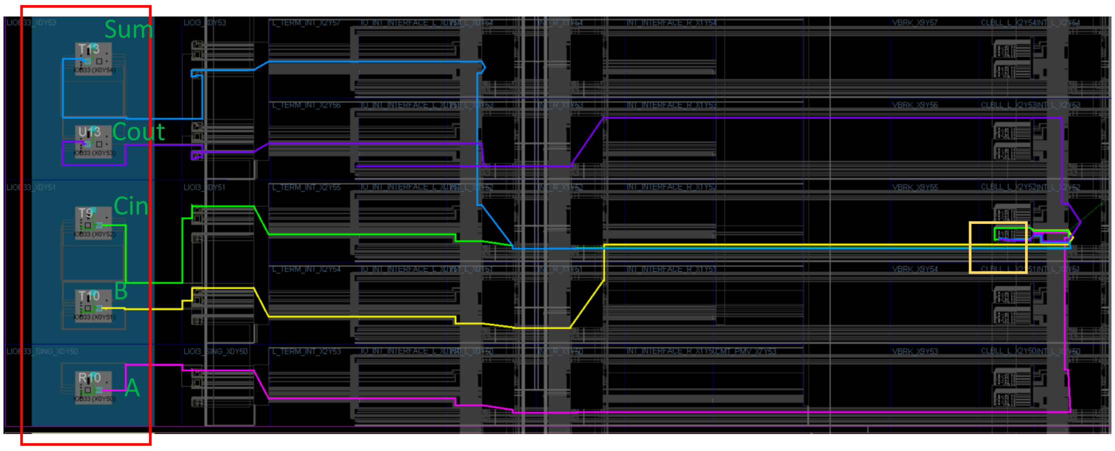


Figure 8.2.3

Similarly, the red rectangle shows the input and output side of the block. And the yellow rectangle shows the location of core logic process.

Figure 8.2.4 illustrates the inner schematic of the integrated port A, in type of IOB33.

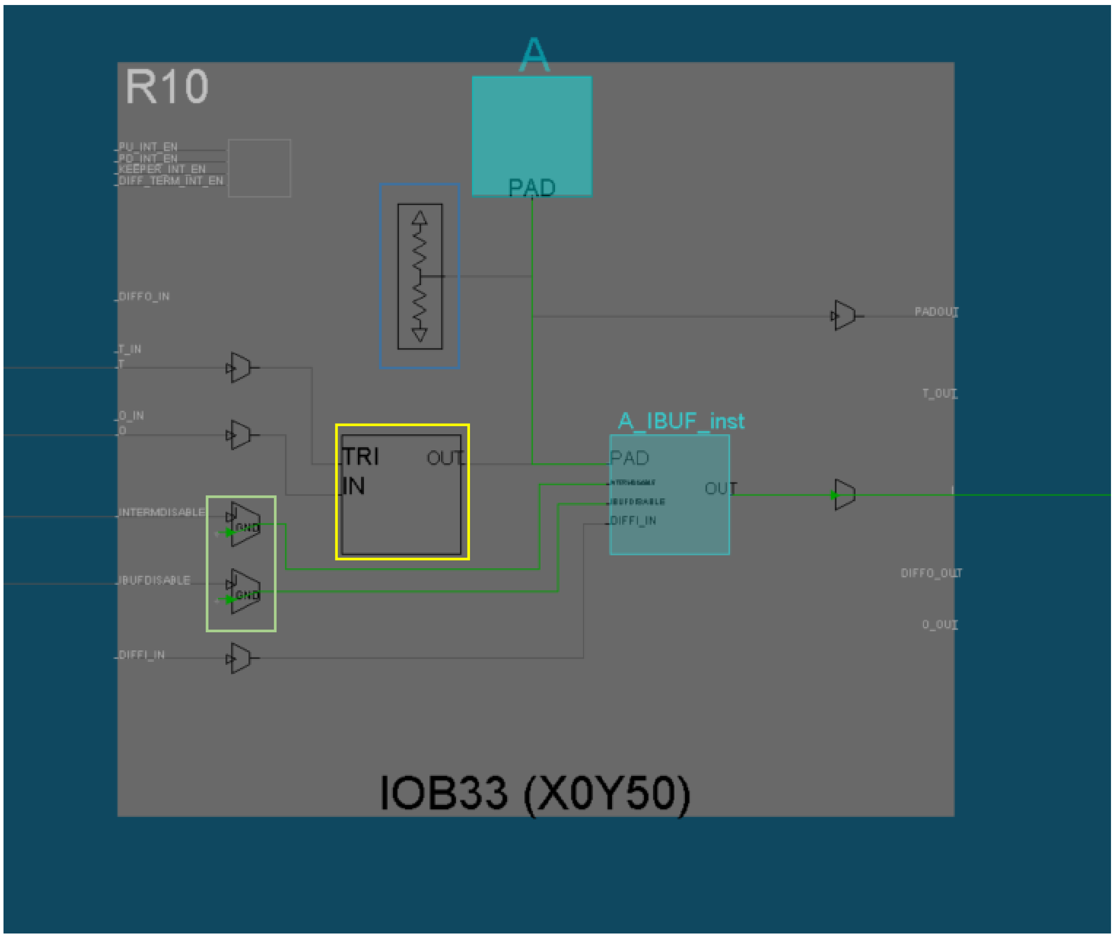


Figure 8.2.4

We take this port as example to explain the function of each sub module. They are all real hardware in the chip.

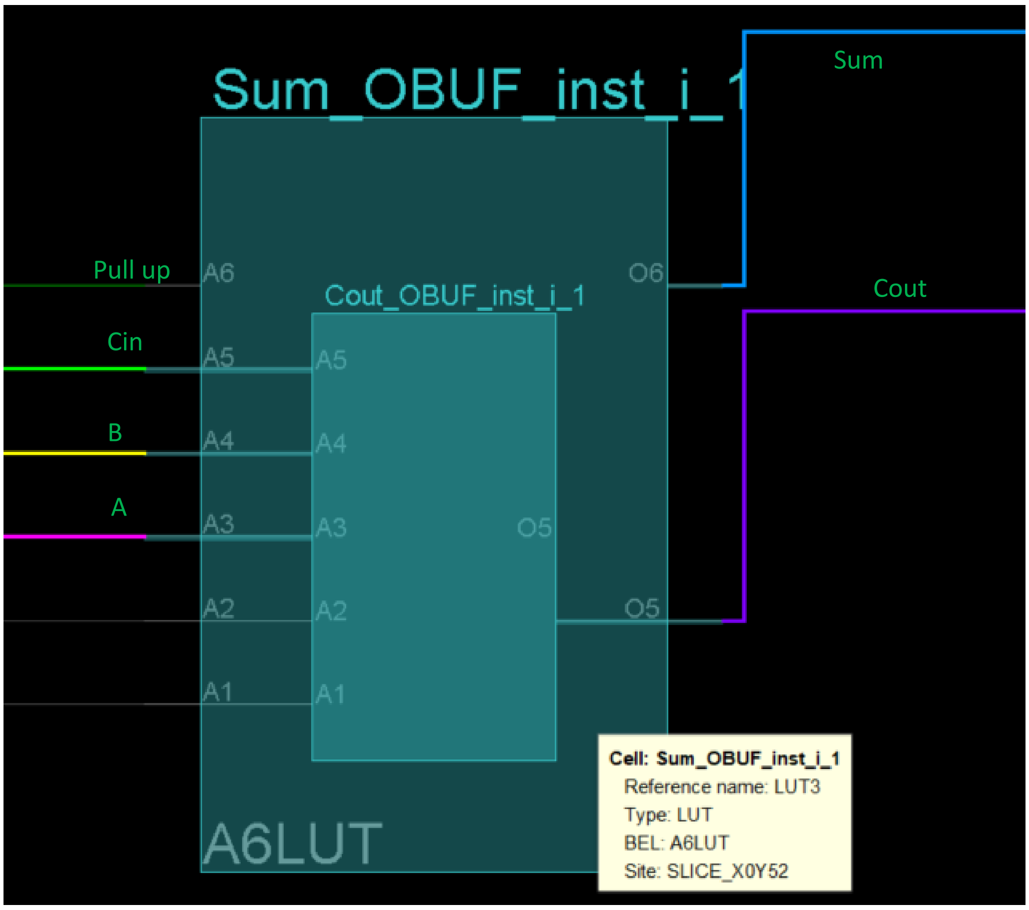
PAD (passivation opening) is the real port to the package or bank port. The pad is inside the chip. The NET inside IC needs to be led to the outside of IC for packaging, but because the width of the wire is too thin to withstand the welding pressure, it needs to be connected to a large metal block as a support, the large metal block under pressure is pad.

The module surrounded by the blue rectangle is pull up or keep selector. This is used when this is an output port, to implement a higher current and voltage driving ability. For this is an input one, thus it is disabled.

The module surrounded by the yellow rectangle is the output buffer, while the input of the output buffer is the output trigger and output value (signal). The real output value will be stored in the buffer, when the trigger signal occurs (or dis-occurs), the buffer will drive its output to the value it is stored. If the port is connected to a bus, the pull and keep will remain the output high resistance to prevent violence. Because this is an input port thus this is also unused.

The cyan highlighted module named A_IBUF_inst is the input buffer, its name is defined because we use it here. The other three inputs of this buffer is interrupt disable, input buffer disable, and differential input. The first two signals determine whether the input can be interrupted and buffered, respectively. In our implementation, we allow interrupt and allow buffer, thus all the disable are set to ground (the multiplexer is surrounded by a bright green rectangle). For we are single input, the differential input are also disabled.

Figure 8.2.5 illustrates the LUT we used in our design.



Cell: Sum_OBUF_inst_i_1
Reference name: LUT3
Type: LUT
BEL: A6LUT
Site: SLICE_X0Y52

Figure 8.2.5

In our FPGA, the basic logic unit is LUT6m, thus this is actually a LUT6, which means it can have maximum 6 inputs and 1 output, or 5 inputs and 2 outputs. This will be discussed later. Now we focus on the signal we define here.

The input side has six inputs but only 4 of them are used. A5 to A3 are the three inputs corresponding to our designed three inputs A, B and Cin. A6 will be discussed later.

The output side has two inputs and all of them are used. Obviously they are Sum and Cout.

Though this LUT is a 6 inputs and 2 output LUT, it can not have 6 inputs and 2 outputs work simultaneously. Because it is not a 6-bit address, 2-bit value ROM. The LUT6 is actually a combination of two LUT5, which is a 5-bit address, 1-bit value ROM. **Figure 8.2.6** illustrates the inner schematic of a LUT6.

When used as a LUT6, all the inputs are used and only O6 is used. In this case, I5 will choose which result of LUT5 will be delivered to the real output O6.

When used as two LUT5. All the inputs and outputs are used, but I5 is set to high forever. Because O5 is the output of the upper LUT5, and O6 must be the output of lower LUT5, thus the input of the multiplexer must be 1.

This is why the maximum usage of this LUT6 is 6 inputs with 1 output or 5 inputs with 2 outputs.

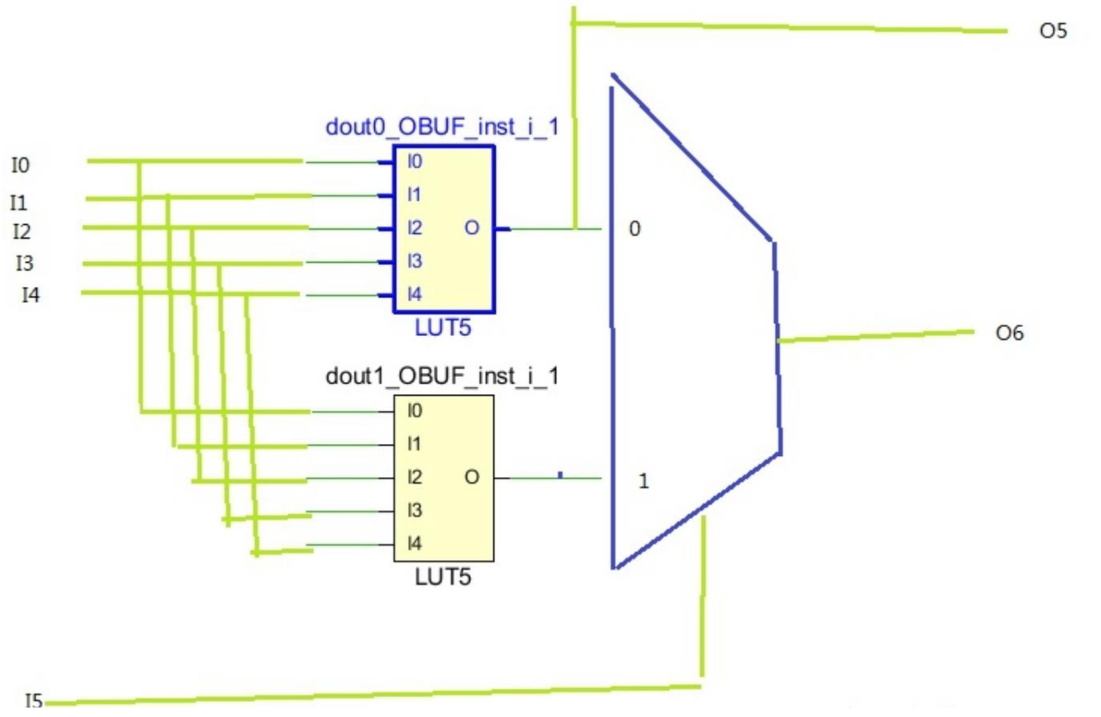
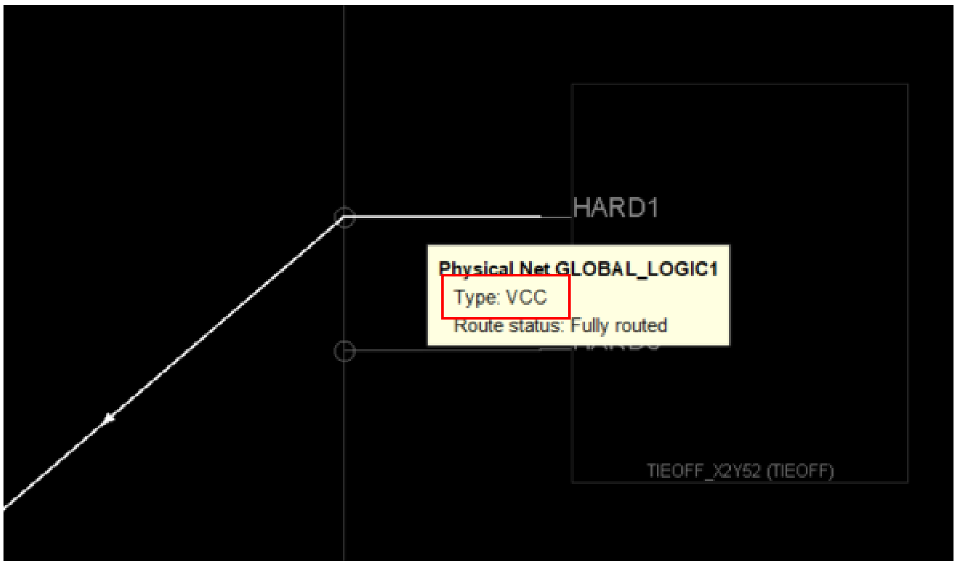


Figure 8.2.6

Our case is quite similar to the latter one mentioned above. The mapping from this figure to our figure is I0-I5 map to A1-A6. The notation of the output in the two figures are the same.

We find that A3-A5 are all given to the two LUT5. One of the LUT5 (in our schematic, this is the lower one) gets the result of Sum, while the other one gets the result of Cout. The result of the upper LUT5, which is Cout, is given to the output O5 directly, and the other one is passed through the multiplexer and given to the output O6.

Thus the input A6 must be 1 permanently. In the implementation, A6 is actually connected an inside digital high voltage port to realize this. **Figure 8.2.7** illustrates this. Note the type of the wire.



Physical Net GLOBAL_LOGIC1
Type: VCC
Route status: Fully routed

Figure 8.2.7

Now back to the wire delay. We find that there is several ps wire delay difference.
This kind of delay is caused by the length of the wire. When there is a voltage change in the one side of the wire, the voltage of the other side will not change immediately because the electric field needs time to reconstruct, or in other word, to move from one side to the other. This moving speed of the electric field is nearly the light speed. But in silicon based circuits it is much slower because the influence of the media. We can have a simple overview of this kind of delay. Here is an example of this.
Suppose there is a wire based on silicon with length $l = 100000\text{nm}$. The relative di-electric constant of silicon is approximately $\varepsilon r \approx 11.4$.

SILICON	硅		11.0 - 12.0

Thus the delay of the voltage change in the two side of the wire due to the moving of electric field is

$$t_d = \frac{l}{\frac{c}{\varepsilon r}} \approx \frac{100000 \times 10^{-9}}{\frac{3 \times 10^8}{11.4}} = 3.8 \times 10^{-12}\text{s} = 3.8\text{ps}$$

This is quite similar to the wire delay shown in the **section 8.1, figure 8.1.2**. They are in the same order of magnitude.
However, due to the dominated gate delay, the pure wire delay is hard to detect. But the relative wire delay can be measure easier.
In the **figure 8.1.2**, the wire length difference of B_IBUF and Cin_IBUF can be approximately $100\mu\text{m}$, and this cause the 3ps delay between the two signal in the waveform.

9. Additional Simulations with Different Gate Delay

The delay analysis makes me think more about the AFTER statement in the source file after the assignment statement.
The different result between the behavior simulation and timing simulation is caused by the different type of delay. The significant different of these delay is that the duration is different. If we

change the duration to the same scale, the simulation would probably give a similar result.
Here I change the gate delay in the source file to 1ns, which means the value to the right of the assignment will be assigned to the left after 1ns.
Figure 9.1 illustrates the behavior simulation after this little modification.

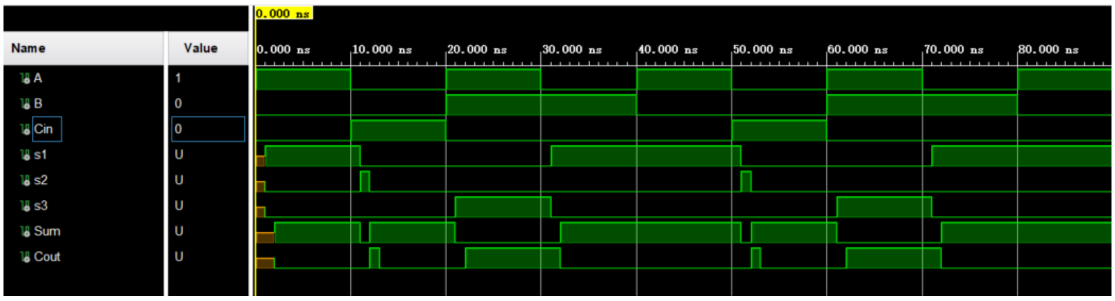


Figure 9.1

There is spikes and the overall waveform of Sum and Cout are correct now.

Figure 9.2 illustrates the post-synthesis timing simulation after change 'WAIT FOR 10ns' to 'WAIT FOR 1ns' in the test bench file.

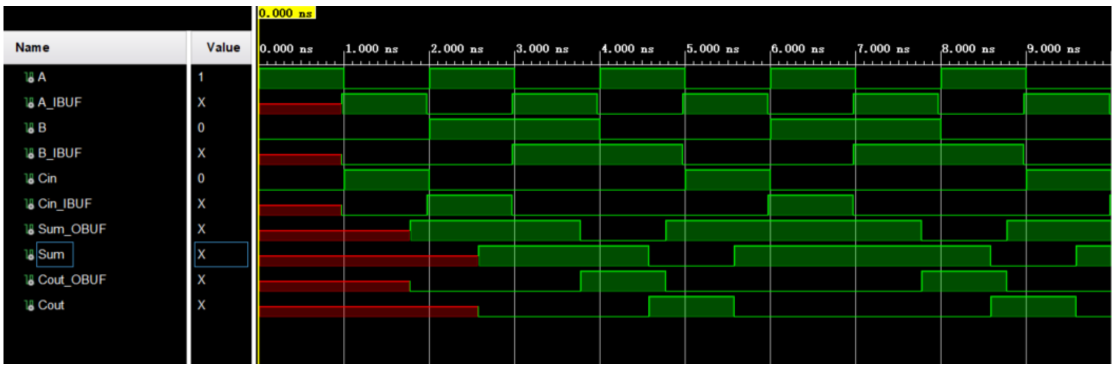


Figure 9.2

And we can find that it is still correct even the delay scale is almost the same as the wait time between different stimulus.
This is because the re-wire during synthesis. This part was explained in **section 7.2, figure 7.2.1**.
All the signal has the same delay level. But in the RTL level, we do not have such property.
This also explains why we choose LUT to implement FPGA rather than use plenty of real gates.

10. Conclusion and Experience

In this lab, we go through the basic steps of digital circuit design and designed a full adder. With different simulation, we analysis the difference in results and have a further understanding of what the delay really is and how it is caused. The delay is caused by the wiring length and the latency of each electrical components. We also get a brief view of the inner side of the hardware in FPGA, and learned how it works inside.
Though this is the first lab of our digital system design course. But I really learned a lot about FPGA in hardware level and how the synthesis and implementation do while designing. Although I have learned how to use FPGA before, it just remains in how to use it, rather than understand it. Even now I can not say that I have comprehend this totally. There are also lots of things that I do not get the idea of, like the routing planning and AXI bus.

Though it is a hard time to do so many experiments and search result of what happens, even missing the ddl, but I appreciate this experience of exploring/learning the unknown for me.