EE216 Re-configurable Computing Homework5 Report

Zhaojun Ni, 2022231102

December 4, 2023

1 FSM Design

In this lab, due to the need to match 5 bytes of content within each cycle while receiving only 4 bytes of content per cycle, I initially designed a shift register buffer_word as a static variable to store 8 bytes. Upon the arrival of each cycle, the content of the low 4 bytes is sent to the high 4 bits as output, and the newly received data is stored in the low 4 bytes. If the received data is empty or “#” (hash symbol), replace the data with the ASCII code for a space.

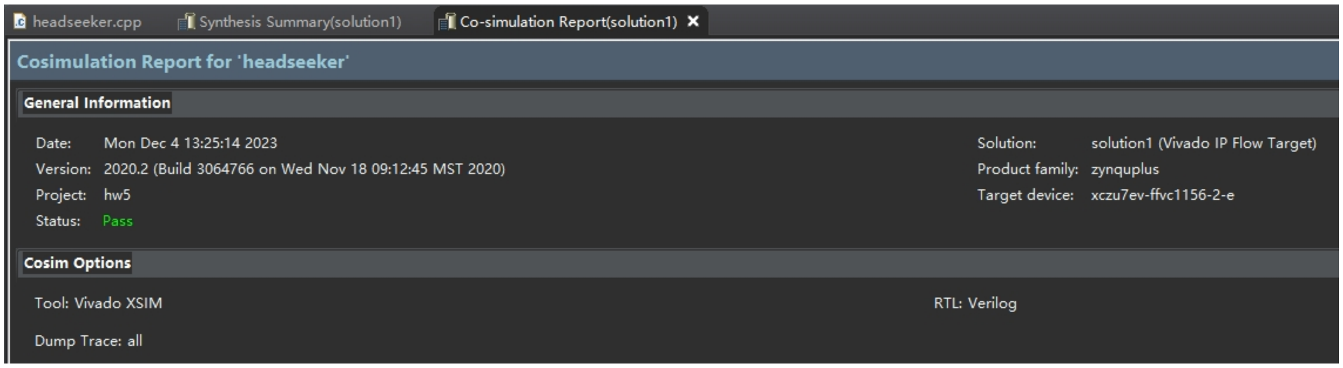
Following that, I designed a three-state state machine, namely IDLE, SEARCH, and FOUND, representing to-be-launched, searching for characters, and finding characters states, respectively. The IDLE state is the state jumped to after processing every 32 bytes, and to handle new 32 bytes of data, it jumps to the SEARCH state if the received data is not empty.

The SEARCH state is used to locate “EE216”, and since there are only three possible positions for the first ”E” when a match is found, namely in the high 7 bits, high 6 bits, or high 5 bits of the shift register, this state uses if-else branching to determine whether the subsequent 5 bytes in each position are “EE216”. A flag is used to indicate which of these three cases is true, and the count of valid output bytes is updated accordingly. To align with the start of “EE216”, the data in the shift register needs to be shifted right, and the overflow part is stored in tmp_word. Since there is at most one occurrence of “EE216” in every 32 bytes of data, once it is found, the next state of the state machine must jump to FOUND. However, the case where the state machine continues to stay in the SEARCH state without finding “EE216” needs to be considered as well. If the count of valid output bits reaches 32 without finding “EE216” within the entire data packet, it jumps to IDLE to prepare for processing the next 32 new bytes.

In the FOUND state, it indicates that “EE216” has been found before, and therefore alignment has already been achieved in this state. The state first checks whether the count of valid output bytes has reached 32; if it has, it needs to jump to IDLE. Otherwise, different handling is performed based on the three different positions where matches were found, using the static variable flag to distinguish. Essentially, the overflow part of the data in tmp_word is moved to the highest bit of the low 4 bytes of the buffer_word register, and the original low 4 bits of buffer_word are shifted to the right, with the new overflow part stored in tmp_word. It’s important to note that during the swap, a tmp_reg variable is used as a data exchange buffer.

2 COSIM Result

There are four cases in the testbench, representing scenarios where there is no “EE216” in the 32 bytes, “EE216” appears in the third byte, it appears in the first byte, and it appears in the second byte. It’s important to note that if “EE216” appears at the beginning of a byte, in such cases, alignment is not required. My design has successfully passed the cosimulation, and the simulation waveforms for the four cases are as follows.



Cosimulation Report for 'headseeker'

General Information

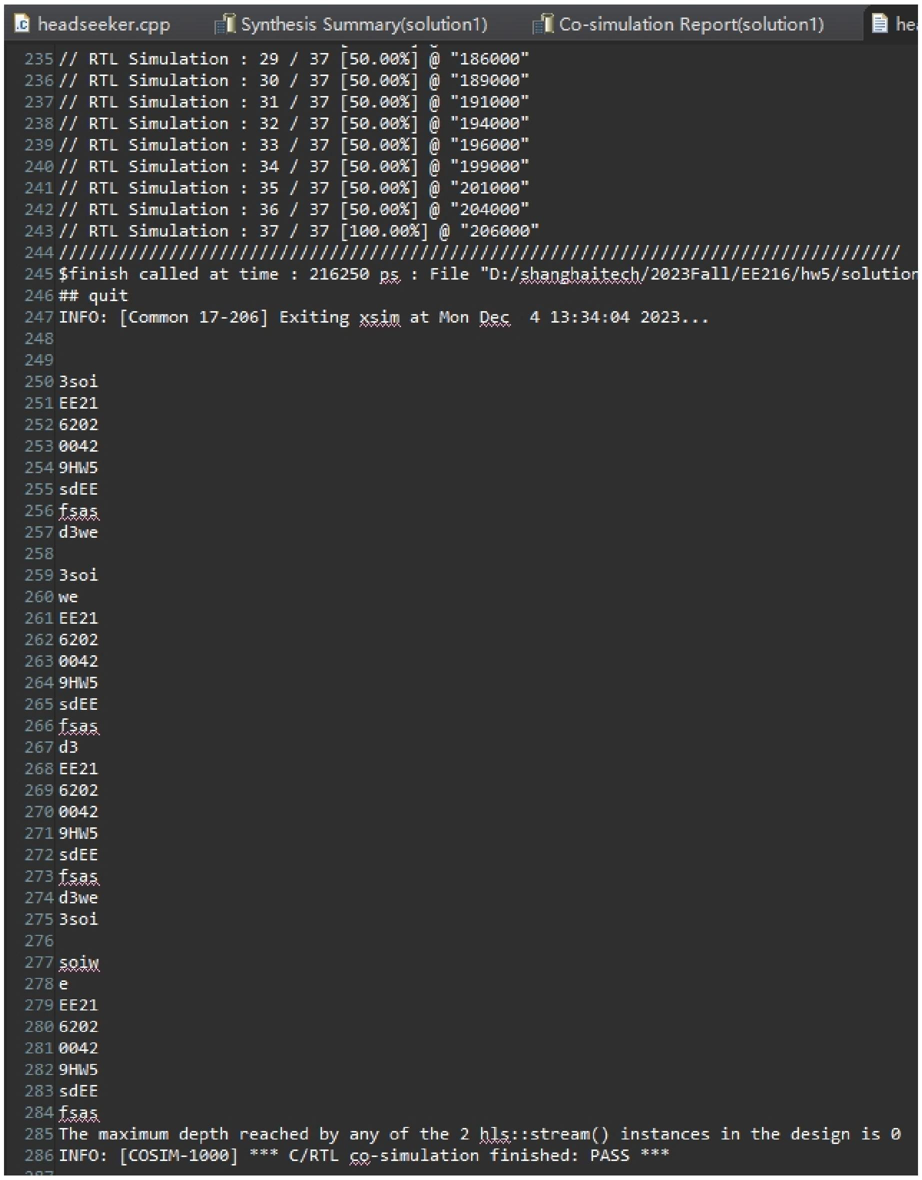
```
Date:      Mon Dec 4 13:25:14 2023                                   Solution:        solution1 (Vivado IP Flow Target)
Version:   2020.2 (Build 3064766 on Wed Nov 18 09:12:45 MST 2020)    Product family:  zynquplus
Project:   hw5                                                       Target device:   xczu7ev-ffvc1156-2-e
Status:    Pass
```

Cosim Options

```
Tool: Vivado XSIM                                                    RTL: Verilog

Dump Trace: all
```

Figure 1: COSIM PASSED.



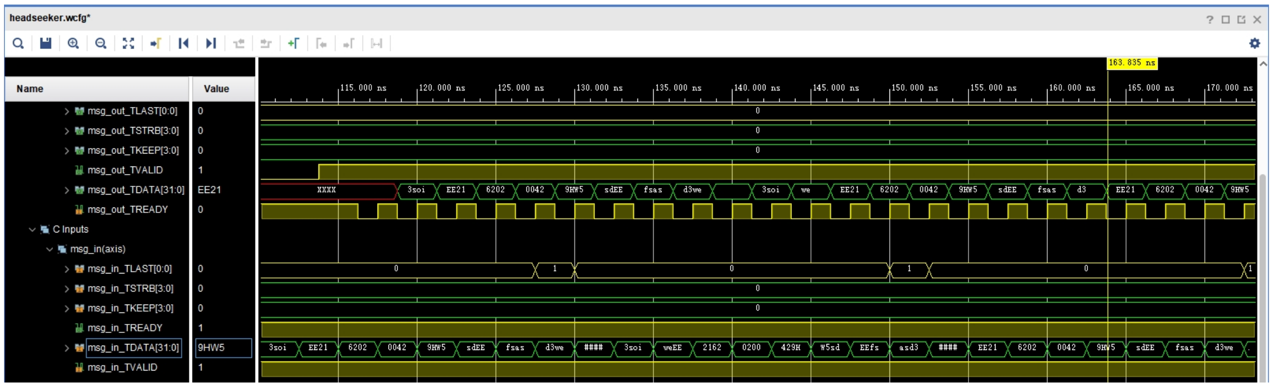
```
// RTL Simulation : 29 / 37 [50.00%] @ "186000"
// RTL Simulation : 30 / 37 [50.00%] @ "189000"
// RTL Simulation : 31 / 37 [50.00%] @ "191000"
// RTL Simulation : 32 / 37 [50.00%] @ "194000"
// RTL Simulation : 33 / 37 [50.00%] @ "196000"
// RTL Simulation : 34 / 37 [50.00%] @ "199000"
// RTL Simulation : 35 / 37 [50.00%] @ "201000"
// RTL Simulation : 36 / 37 [50.00%] @ "204000"
// RTL Simulation : 37 / 37 [100.00%] @ "206000"
////////////////////////////////////////////////////////////////////////////////////
$finish called at time : 216250 ps : File "D:/shanghaitech/2023Fall/EE216/hw5/solution
## quit
INFO: [Common 17-206] Exiting xsim at Mon Dec  4 13:34:04 2023...


3soi
EE21
6202
0042
9HW5
sdEE
fsas
d3we

3soi
we
EE21
6202
0042
9HW5
sdEE
fsas
d3
EE21
6202
0042
9HW5
sdEE
fsas
d3we
3soi

soiw
e
EE21
6202
0042
9HW5
sdEE
fsas
The maximum depth reached by any of the 2 hls::stream() instances in the design is 0
INFO: [COSIM-1000] *** C/RTL co-simulation finished: PASS ***
```

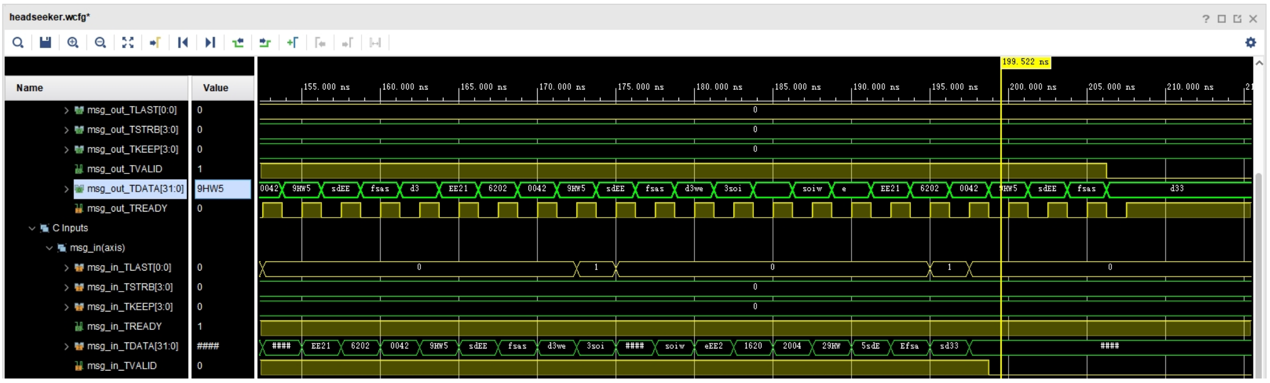
Figure 2: COSIM result.



headseeker.wcfg*

Name	Value
msg_out_TLAST[0:0]	0
msg_out_TSTRB[3:0]	0
msg_out_TKEEP[3:0]	0
msg_out_TVALID	1
msg_out_TDATA[31:0]	EE21
msg_out_TREADY	0
C Inputs	
msg_in(axis)	
msg_in_TLAST[0:0]	0
msg_in_TSTRB[3:0]	0
msg_in_TKEEP[3:0]	0
msg_in_TREADY	1
msg_in_TDATA[31:0]	9HW5
msg_in_TVALID	1

Figure 3: First two cases co-sim waveform.



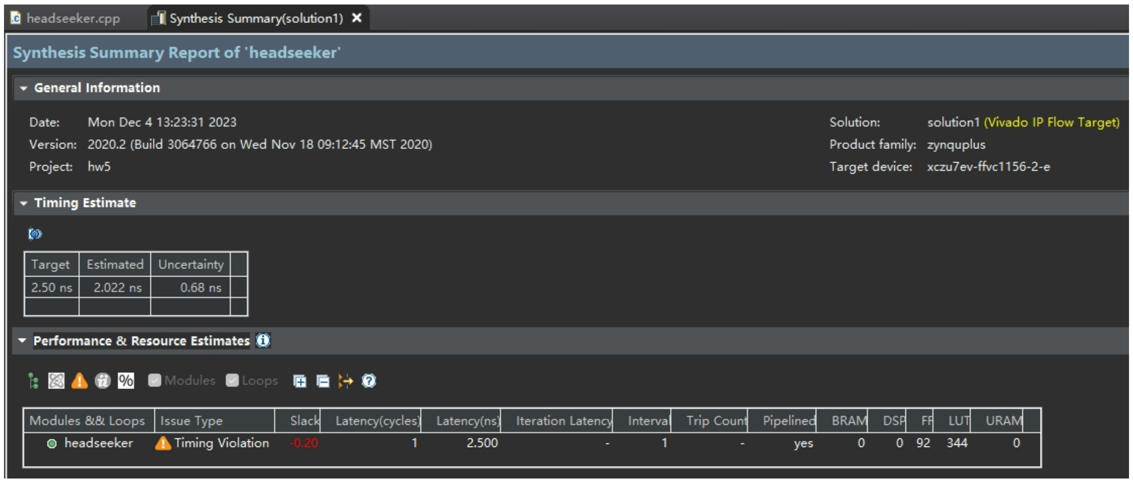
headseeker.wcfg*

Name	Value
msg_out_TLAST[0:0]	0
msg_out_TSTRB[3:0]	0
msg_out_TKEEP[3:0]	0
msg_out_TVALID	1
msg_out_TDATA[31:0]	9HW5
msg_out_TREADY	0
C Inputs	
msg_in(axis)	
msg_in_TLAST[0:0]	0
msg_in_TSTRB[3:0]	0
msg_in_TKEEP[3:0]	0
msg_in_TREADY	1
msg_in_TDATA[31:0]	####
msg_in_TVALID	0

Figure 4: Last two cases co-sim waveform.

3 Optimization

Under a clock constraint of 400MHz, the Slack for my design has improved from the originally designed -8ns to -0.20ns. Below, I will elaborate on the optimization methods I employed.



Synthesis Summary Report of 'headseeker'

General Information

Date:	Mon Dec 4 13:23:31 2023	Solution:	solution1 (Vivado IP Flow Target)
Version:	2020.2 (Build 3064766 on Wed Nov 18 09:12:45 MST 2020)	Product family:	zynquplus
Project:	hw5	Target device:	xczu7ev-ffvc1156-2-e

Timing Estimate

Target	Estimated	Uncertainty
2.50 ns	2.022 ns	0.68 ns

Performance & Resource Estimates

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
headseeker	Timing Violation	-0.20	1	2.500	-	1	-	yes	0	0	92	344	0

Figure 5: Synthesis Summary.

Optimization 1: Utilizing # pragma directives for initial optimization. Initially, in the original design version of the headseeker function, I employed # pragma HLS pipeline II = 1 to optimize the design towards a smaller interval. However, initially, I found that although the synthesis report estimated a clock period of 1.821ns, the interval ranged from 3 to 54. Even after specifying the loop count using tripcount in certain loops (1 to 3, corresponding to determining the position of the first 'E' in 'EE216'), the results were not satisfactory, with an interval ranging from 4 to 18.

Optimization 2: Reducing complex loops and using if-else to manually unroll instead. In my original design, for operations like processing buffer_word, I initially followed a byte-by-byte processing approach using loops (e.g., 4 times) for shifting operations within buffer_word. Later, I unrolled the loop with unroll=4. However, I realized this was unnecessary. Referring to the code in Lab6, utilizing bus data types like ap_ axiu <64, 0, 0, 0>, I could perform bit selection assignments using buffer_word.data.range(). This operation is similar to bit selection assignments in Verilog and is likely executed in parallel. After synthesizing, I observed that the interval had reduced to 1, but the slack significantly increased, even reaching -8. Upon examining the highlighted portions in the Analysis section, I found issues during the matching of the 'EE216' characters in the SEARCH state. Originally, I used two nested loops for this purpose. The first loop iterated over the three possible positions where the first 'E' in 'EE216' might appear, storing the subsequent five bytes as the name variable for comparison with the ASCII codes of 'EE216'. Later, I realized that assigning values to the name variable was unnecessary, and I could directly compare these 40 bits with 'EE216' using bit selection from buffer_word. Additionally, loop comparisons are time-consuming, so I rewrote them using if-else statements. Originally, the second loop was responsible for right-shifting characters in three different cases. Due to the loop count being determined by variables j and flag, this uncertainty in loop count could be fatal in hardware. Therefore, I fixed the loop counts and bit selections for the three cases, incorporating them into an if-else framework. Similarly, in the FOUND state, I enumerated the cases for loops with uncertain counts and expanded them using if-else statements. This resulted in a slack of around -2ns for my design!

Optimization 3: Using a smaller data width. In the post-synthesis Analysis, I also found that some critical paths were due to integer additions, and upon inspecting the source, I discovered that it was related to certain static variables, such as flag and count. Originally, these were defined as int, but later, I changed them to the ap_ uint <x>data format. For flag, I used 2 bits to represent the three cases. As for count, it is used to count the number of valid output bytes, and since it inevitably reaches 32, a uint5 might not be sufficient. Therefore, I designed its bit width to be 6.

Additionally, with optimizations such as reordering some if-else branches, my final design achieved a Slack of -0.2ns. Moreover, the cosimulation was successful, and the simulated waveforms were correct.