NCTU-EE IC LAB - Spring 2018

Lab06 Exercise

Design: Security Attack

Data Preparation

1. Extract test data from TA's directory:

   % tar xvf ~iclabta01/Lab06.tar

2. The extracted LAB directory contains:

   a. **00_TESTBED**

   b. **01_RTL**

   c. **02_SYN**

   d. **03_GATE**

Design Description

Security is an important issue to protect private information. There are many ways to encrypt. In this lab, you are asked to attack a simple security code as the following description.

Consider the following multiplication:

□□□ × □□□ = ?

You will receive six decimal numbers sequentially, and place the numbers into six blanks of above equation arbitrarily. The desired goal is to get the minimum product of the multiplication. Each input is in 4-bit BCD (binary-coded decimal) code, as described in the following table:

Decimal digit	4-bit BCD code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

To reduce the complexity of the circuit, the first and the second numbers are constrained to

the middle digits of multiplicand and multiplier, and the others can be put arbitrarily. For example, you are given 1, 2, 3, 4, 5 and 6. 1 and 2 are confined in the middle digit of multiplicand and multiplier, so the equation becomes:

□1□ × □2□ = ?

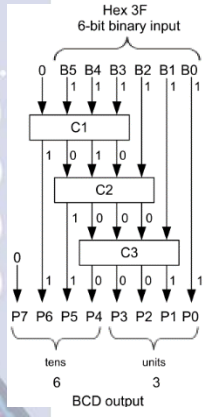
And 3, 4, 5 and 6 can be placed into blanks of above equation arbitrarily. You need to find the **minimum** product of the multiplication, and output the answer in **BCD code**.

Binary-to-BCD Conversion Algorithm (Double dabble) : Soft IP Design

Shift and **A**dd-3 algorithm is very famous method for convert binary code to BCD. This algorithm follows some basic rules,

1) Shift the binary number left one bit.
2) If 8 shifts have taken place, the BCD number is in the hundreds, tens and units column.
3) If the binary value in any of the BCD columns is greater than 4, add 3 to that value in that BCD column.
4) Iterated from step (1) until whole conversion is done.

Operation	Hundreds	Tens	Units	Binary	
HEX				F	F
Start				1 1 1 1	1 1 1 1
Shift 1			1	1 1 1 1	1 1 1
Shift 2			1 1	1 1 1 1	1 1
Shift 3			1 1 1	1 1 1 1	1
Add 3			1 0 1 0	1 1 1 1	1
Shift 4		1	0 1 0 1	1 1 1 1	
Add 3		1	1 0 0 0	1 1 1 1	
Shift 5		1 1	0 0 0 1	1 1 1	
Shift 6		1 1 0	0 0 1 1	1 1	
Add 3		1 0 0 1	0 0 1 1	1 1	
Shift 7	1	0 0 1 0	0 1 1 1	1	
Add 3	1	0 0 1 0	1 0 1 0	1	
Shift 8	1 0	0 1 0 1	0 1 0 1		
BCD	2	5	5		



Inputs and Outputs

Input signal	Bit width	Definition
clk	1	Clock
rst_n	1	Asynchronous active-low reset
in_valid	1	Enable input signal
in	4	Input number.

Output signal	Bit width	Definition
out_valid	1	Enable output check
out	4	Output number

1. The **in** is valid only when **in_valid** is high, and is delivered for 6 cycles continuously. When in_valid is low, **in** is set to 0.
2. All input signals are synchronized at **negative edge** of the clock.

3. There is **only one reset** before the first pattern.
4. **out_valid** and **out** should be low after initial reset.
5. **out_valid** should not be raised when **in_valid** is high.
6. **out** should output **continuously 6 cycles** without any interruption. e.g. Your answer is 123 and you need to output 0, 0, 0, 1, 2, 3 sequentially.
7. The TA's pattern will capture your output for checking at **clock negative edge**.

Specifications

1. Top module name: **SA** (design file name: **SA.v**)
2. It is **asynchronous** reset and **active-low** architecture.
3. The reset signal would be given only once at the beginning of simulation. All output signals should be reset after the reset signal is asserted.
4. The clock period is **6ns**.
5. The input delay is set to **0.5*(clock period)**.
6. The output delay is set to **0.5*(clock period)**, and the output loading is set to **0.05**.
7. The input delay of **clk** and **rst_n** should be **zero**.
8. The synthesis result (syn.log) of data type **cannot** include any **latches and error.**
9. After synthesis, you can check **SA.area** and **SA.timing**. The area report is valid when the slack in the end of timing report should be **non-negative**.
10. The gate level simulation **cannot** include any timing violations **without** the **notimingcheck** command.
11. The next group of inputs will come in 2 cycles after your **out_valid** is pulled down.
12. The **out_valid** should be high within **100 cycles** after **in_valid** pulls to low.
13. The performance is determined by area and latency. The lower, the better.
14. In this lab, you should write your own **syn.tcl file** and **pattern**.
15. Using **top** wire load mode and **compile ultra**.

Specifications (Soft IP)

1. Top module name: **B2BCD_unit** (design file name: **B2BCD_unit.v**)
2. Input signals：**binary_code**
3. Output signals：**BCD_code**
4. Output loading is set to **0.05**.
5. Using **top** wire load mode and **compile ultra**.
6. Two parameter: one used to declare the length of input binary bits, the other used to present the BCD digit number.

   Ex. #(4,2) : 4 means input binary bits which the largest number is 15 and 2 means output BCD digital number which are "1" and "5". Of course, the truly output bits should be 8 bits, you can change your output bit by using the second parameter. There won't

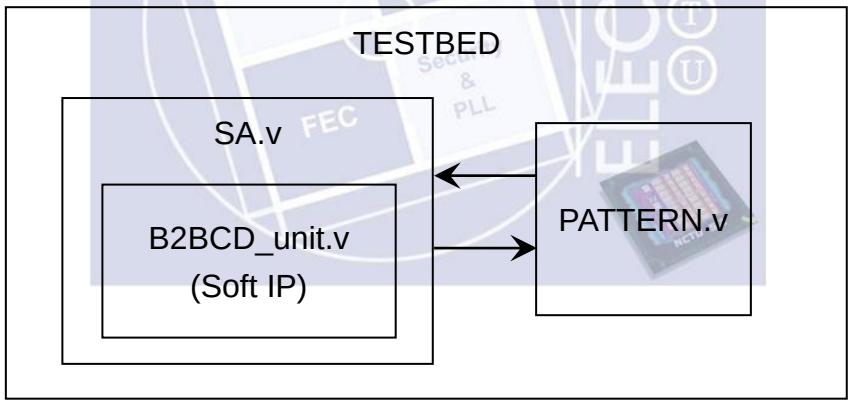
be the illegal case #(4,1) or #(4,3).

Soft IP Testing environment

```
//synopsys translate_off
`include "B2BCD_unit.v"
//synopsys translate_on

module SA(
// input port
binary_code,
// output port
BCD_code
);
….
// 4-bit binary to BCD design check
B2BCD_unit #(4,2) BCD_U1(.binary_code (binary_code[3:0]),.BCD_code(bcd_code_unitout[7:0]));
…
endmodule
```

Block diagram



Note

1. **Grading policy:**

   RTL and Gate-level simulation correctness: 45%

   Performance: 30%

   - Latency 10%
   - Area 20%

   Soft IP function correctness: 25% (No second demo)

   - 1-bits binary code to BCD 1%

- 2-bits binary code to BCD 1%
- 3-bits binary code to BCD 1%
- 4-bits binary code to BCD 2%
- 5-bits binary code to BCD 1%
- 6-bits binary code to BCD 1%
- 7-bits binary code to BCD 1%
- 8-bits binary code to BCD 2%
- 9-bits binary code to BCD 1%
- 10-bits binary code to BCD 1%
- 11-bits binary code to BCD 1%
- 12-bits binary code to BCD 2%
- 13-bits binary code to BCD 1%
- 14-bits binary code to BCD 1%
- 15-bits binary code to BCD 1%
- 16-bits binary code to BCD 2%
- 17-bits binary code to BCD 1%
- 18-bits binary code to BCD 1%
- 19-bits binary code to BCD 1%
- 20-bits binary code to BCD 2%

2. **Please upload the following file on e3 platform before noon (12:00 p.m.) on Apr. 30:**
   - SA_iclabXX.v、B2BCD_unit_iclabXX.v (XX = Your account)
   - Ex. SA_iclab99.v、B2BCD_unit_iclab99.v
3. **Template folders and reference commands:**

   01_RTL/ (RTL simulation) **./01_run**

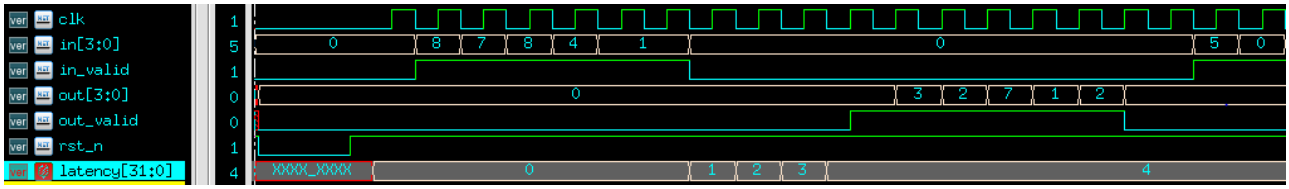
   02_SYN/ (Synthesis) **./01_run_dc**

   (Check the design if there's latch or not in ***syn.log*****)**

   (Check the design's timing in /Report/***SA.timing***)

   03_GATE / (Gate-level simulation) **./01_run**

Sample Waveform



Hint for building up IP

B7	B6	B5	B4	B3	B2	B1	B0
	0	0	0	1	1	1	1
	0	0	0	1	1	1	1
	0	0	0	1	1	1	1
0	0	0	1	0	1	0	1
	1			5			

Hint for tcl file (due to Soft IP)

```
#======================================================
#  Read RTL Code
#======================================================
set hdlin_auto_save_templates TRUE
read_verilog B2BCD_unit.v
read_verilog { $DESIGN\.v}
current_design $DESIGN
```