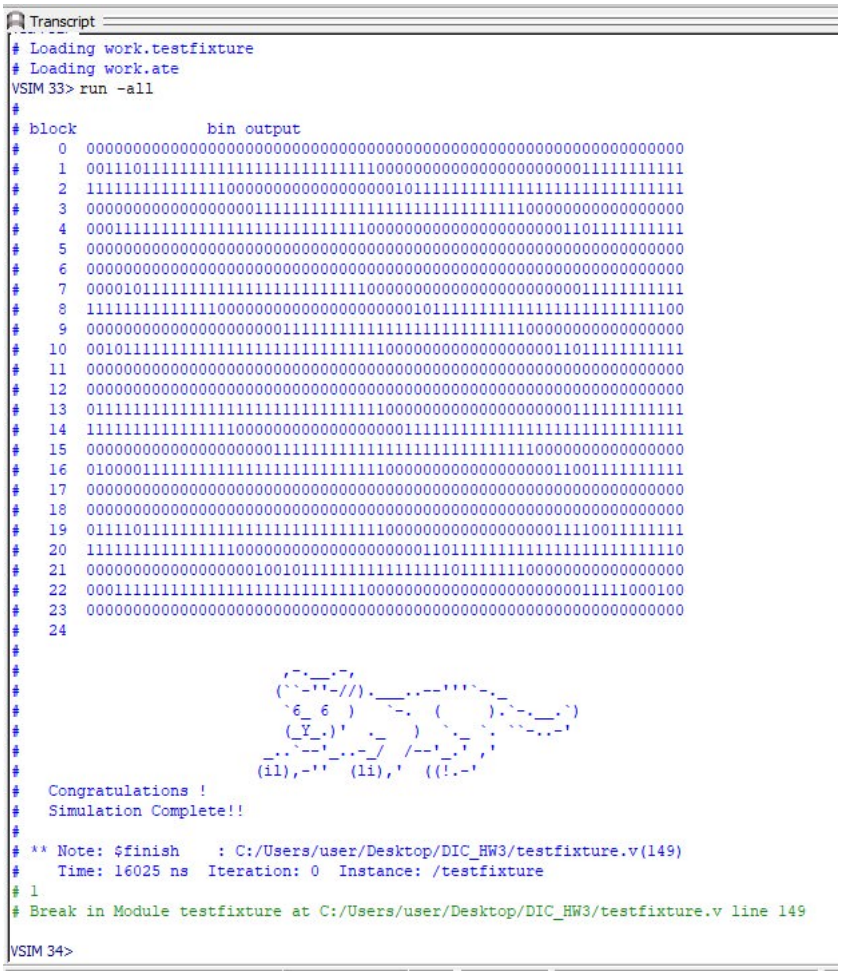
HOMEWORK 3

資工所 碩一 田少谷 **P76071268**

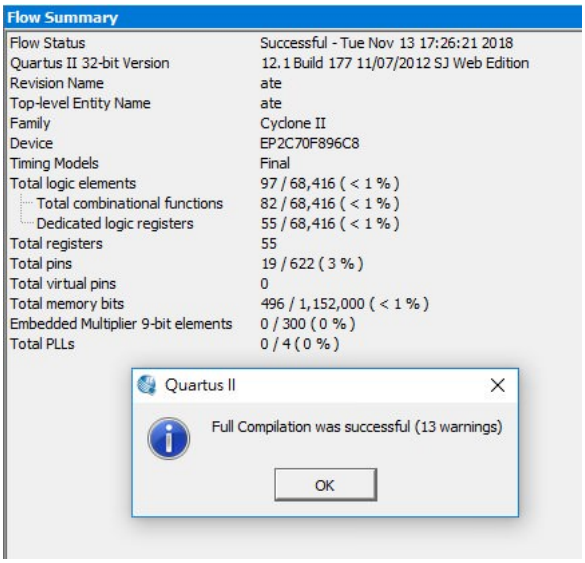
Functional Simulation:



```
Transcript
# Loading work.testfixture
# Loading work.ate
VSIM 33> run -all
#
# block              bin output
#     0  00000000000000000000000000000000000000000000000000000000000000000000
#     1  00111011111111111111111111111111100000000000000000000000011111111111
#     2  11111111111111100000000000000000010111111111111111111111111111111111
#     3  00000000000000000011111111111111111111111111111110000000000000000000
#     4  00011111111111111111111111111000000000000000000000001101111111111
#     5  00000000000000000000000000000000000000000000000000000000000000000000
#     6  00000000000000000000000000000000000000000000000000000000000000000000
#     7  00001011111111111111111111111000000000000000000000011111111111
#     8  11111111111111000000000000000000010111111111111111111111111100
#     9  00000000000000000000111111111111111111111111110000000000000000000
#    10  00101111111111111111111111111110000000000000000001101111111111
#    11  00000000000000000000000000000000000000000000000000000000000000000000
#    12  00000000000000000000000000000000000000000000000000000000000000000000
#    13  01111111111111111111111111111110000000000000000000111111111111
#    14  11111111111111000000000000000001111111111111111111111111111111
#    15  00000000000000000001111111111111111111111111110000000000000000000
#    16  01000011111111111111111111111111100000000000000000110011111111111
#    17  00000000000000000000000000000000000000000000000000000000000000000000
#    18  00000000000000000000000000000000000000000000000000000000000000000000
#    19  01111011111111111111111111111110000000000000000001111001111111
#    20  11111111111111100000000000000000001101111111111111111111111110
#    21  00000000000000000010010111111111111111101111111000000000000000000
#    22  00011111111111111111111111111000000000000000000000011111000100
#    23  00000000000000000000000000000000000000000000000000000000000000000000
#    24
#
#                          ,-.__.-,
#                         (``-''-//).___..--'''`-._
#                          `6_ 6  )   `-.  (     ).`-.__.`)
#                          (_Y_.)'  ._   )  `._ `. ``-..-'
#                        _..`--'_..-_/  /--'_.' ,'
#                       (il),-''  (li),'  ((!.-'
#   Congratulations !
#   Simulation Complete!!
#
# ** Note: $finish    : C:/Users/user/Desktop/DIC_HW3/testfixture.v(149)
#    Time: 16025 ns  Iteration: 0  Instance: /testfixture
# 1
# Break in Module testfixture at C:/Users/user/Desktop/DIC_HW3/testfixture.v line 149

VSIM 34>
```

Synthesis by Quartus:



Flow Summary

Flow Status	Successful - Tue Nov 13 17:26:21 2018
Quartus II 32-bit Version	12.1 Build 177 11/07/2012 SJ Web Edition
Revision Name	ate
Top-level Entity Name	ate
Family	Cyclone II
Device	EP2C70F896C8
Timing Models	Final
Total logic elements	97 / 68,416 (< 1 %)
Total combinational functions	82 / 68,416 (< 1 %)
Dedicated logic registers	55 / 68,416 (< 1 %)
Total registers	55
Total pins	19 / 622 (3 %)
Total virtual pins	0
Total memory bits	496 / 1,152,000 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 300 (0 %)
Total PLLs	0 / 4 (0 %)

Gate-Level Simulation (cycle= 25):

Explanation:

1. When reset==1, initialize the value of all regs as 0.
2. I add an extra buffer[64] so that it can be compared to the value of the threshold.
3. With the advise of others, I set the range of the value of block_count between 0 to 5, so that the code will be easier(namely, no need to write if block_count==6, 12 bla bla bla)
4. Also, the value of bin has to be set up in "assign", as it can assign value **instantly** when the value of block_count changes. This is the adjustment I made after encountering errors.

Extra thoughts during completing the homework:

1. Code Simplification

I tried to improve my codes after it succeeded to function, with the guidance of masters in our lab.
Firstly, express numbers in their complete bit length, so that the program will not fill the empty bits with wrong numbers.

```
before: if (block_count== 5)
after: if (block_count== 5'd5)
```

Then, I was told to use **bit comparison** to substitute number comparison.

```
before: if (block_count== 5'd5)
after: if (block_count[2] & block_count[0])
```

As in my code, the value range of block_count is between 0 and 5, so it's impossible to have 7(111 in binary) but only 5(101 in binary). Therefore, use **and** comparing the [0] and [2] position of block_count can achieve the same idea as comparing ==5.

2. Division of Combinational and Sequential circuit

At first, I was told that I could improve my coding style by **dividing combinational and sequential circuit**. However, I wasn't sure if it's about coding style or the performance of circuits.
Therefore, after some further explanation, I came to understand that the circuit produced by the following two codes are actually the same:

```
always @(*)
    c= a+b

always @(posedge clk)
    out<= c


always @(posedge clk)
    out<= a+b
```

Of course the first one will be easier to debug; however, under the situation that this homework doesn't require thousands of lines of code, it's totally fine to code in the second instance's way.
Eventually, I stick to the second coding style, as I feel comfortable coding this way, and I can write a few lines less. Since it does no harm, why not?

tags: IC Design Adaptive Threshold Engine ATE image segmentation