

Bowling score keeper design
ECE 742 Digital Systems-on-Chip Design
Qiao Gao
CWID:A20282211
11/25/2012

Abstraction

This paper provides an FPGA solution to a bowling score keeper. Specifically, this design achieves the functions of receiving the score of every throw in a bowling game, judging the performance, deciding the total score at any time depending on the bowling game rules and displaying the score on an LED screen. The key points of this project lie on changing the bowling score counting method to hardware description, which can be complicated as there are many branches in every state according to different conditions.

Features of this design include: get the score in at every throw, save it to judge the performance and implement the score calculation method depending on state machine structure.

The design is built on the Spartan 6 development board as hardware platform, described through VHDL code and been tested in Modelsim simulation platform. Moreover, synthesis, place and route process is accomplished on ISE platform.

Key words: Bowling score keeper, state machine

1 Introduction

Bowling game is a kind of popular leisure time sports which judge the performance by the number of bowling pins a player knock down using bowling balls. This largest challenge for this project is how to let the circuit work exactly as the flow of bowling game rules which are complex on both judging and counting score. More specifically, tough point lies on: how to judge that all the pins are down since sometimes we need to associate the judgment with the score of last throw or sometimes not; how to determine the increase on total score for every throw and how many frames a player has for the frame number is not constant value due to the game rules.

To solve these problems, I sketched specific modules in this design to handle them accordingly and finally implemented the function I want. The design of each module and how they are connected with each other will be described in part 2. Above all, the rules of bowling game will be clarified.

1.1 Bowling rules

In one time bowling game, each player has ten frames in general (this may change depending on some conditions, but ten is the basic frame number). In each frame, each player has two opportunities to knock down all the ten bowling pins. There are three possibilities at this time: knock all the pins down during first try which is called a strike, fail to finish all of them but succeed to knock all the pins left in the second try which is called a spare, or fail to knock all the pins down even after the second try. No matter what happens, this frame is over and the number of pins will be refunded to ten which means next frame is started. This is how this game goes.

Then how to calculate the score? Obviously, we cannot just add all the knocked down pin numbers together because players who achieve making a strike should be awarded more than those who make a spare or nothing. Here are the rules on score counting:

- (1) If a strike is done, the scores of next two throws (notice that the word is "throw", not "frame") could be added one more time;
- (2) If a spare is done, the score of next one throw could be added one more time;
- (3) If none of them is done, the following throws will not be influenced.

These rules may cause some interesting conditions. It is easy to notice that the time of adding on one throw may accumulate up to three if we keep playing strikes. Another interesting phenomenon is that the score of current throw have nothing to do with how many times it will be added to the total score; even a very low score can be added more than once if we performed strike or spare before that.

However, these rules bring a new question: What about the final throw in frame

ten? There would be no point on strike and spare if we have no throw left. To fix this, we add two throws for strike and on throw for spare. In other words, we have a chance to clear all the bonus score chances. Moreover, these bonus throws are not counted into formal throws which means we do not judge strike or spare for these throws and we do not add the scores of them to the final score but just add the additional ones instead.

2 Overall design

2.1 Design strategy

To implement these functions, we have to find the answer to these questions:

- (1) How to control the design in order to let it work in the correct state at any time.
- (2) How to judge a strike or a spare.
- (3) How to determine the times adding for current score.

Firstly, to answer the first question, we have to make the state flow of a bowling score keeper clear. To achieve this, a state chart is necessary (Figure 1). This chart indicate how a bowling score keeper work, but it is not the actual SM chart for this design, since the actual circuit should have state like initial or wait, which is impossible to determine before module design.

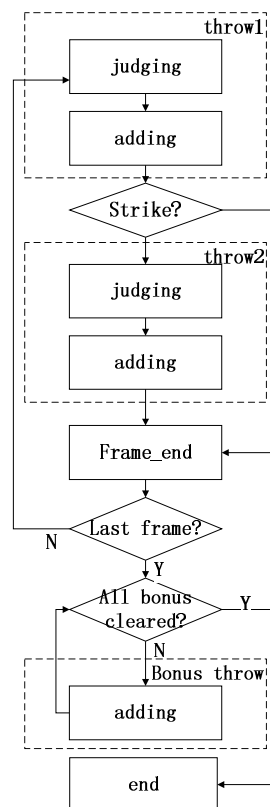


Figure 1

In this chart we can see that every frame is divided into two parts: throw1 and throw2. In addition, every throw contains a judging part and an adding part. If strike appears in throw1, then skip throw2 and go directly to end of the frame. In the frame end state, the circuit will check if it is the last frame. If not, another frame will start from very beginning. However, if it is the final frame, then next state would be clearing the bonus throw. To make principle of every state clear, the module design will be introduced in part 3.

For the second question, we can make a module with the state flow shown below (Figure 2):

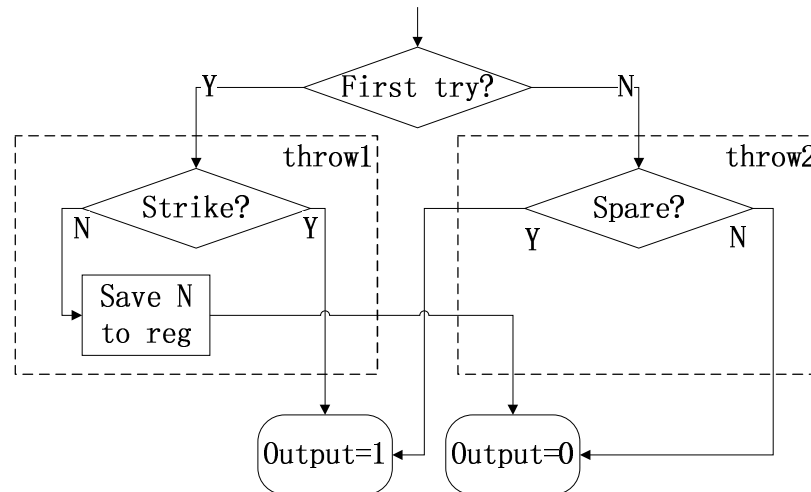


Figure 2

In this figure, we can see that firstly we check which condition this part of circuit is working on, throw1 or throw 2, which is determined by first throw signal. This signal will be '1' if the outer state is throw1. Secondly, in throw1, we compare the input score with 10 in order to see it is a strike or not. If it is ten, then make the output '1'. If not, set the output to '0' and store the input into a register, which would be used to judge a spare. Moreover, in throw2, we should add the score to the number in the register firstly. Then, compare it with 10 to see whether it is a spare. If it is, output would be set to '1'. Otherwise we set it to '0'.

The third question, which is the most significant one for score calculation, can be solved by adjusting some registers. The purpose of adding registers is to store the strike and spare information of the throws before so that the times of adding can be generated in current adding state. More specifically, the structure of these registers is shown below (Figure 3):

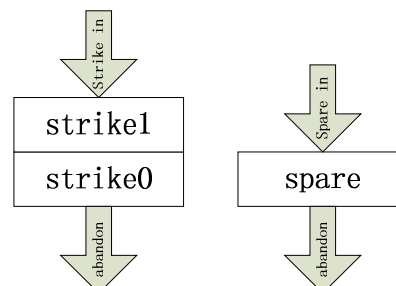


Figure 3

There are three registers: strike0, strike1 and spare. The strike part works more like a queue: New strike information come to strike1 and the strike1 will come into strike0 at the same time. Moreover, the information in strike0 will be abandoned during this process. By doing this, every strike information can be save for two throw. The principle for “spare” is just the same. Finally, just add them all together then we get the times of additional addition for our score. All of the registers should be refreshed during every throw. More specifically, if this is the first throw, output the judging result to strike1 and ‘0’ to spare since no way to make a “spare” in this part. If this is the second throw, then output the judging information to “spare” and ‘0’ to strike1 since no way to make a strike this time. Additionally, in bonus throw, output ‘0’ to both of them because there is no need to judge the score. What’s more, we can clear the registers by doing this. As a result, whether these registers have all been cleared could be used as the condition of the end of a game, which make it simple that we needn’t to care about how many bonus throw we should have.

2.2 System overview

After the discussion above, we can make our module connections according to Figure 4.

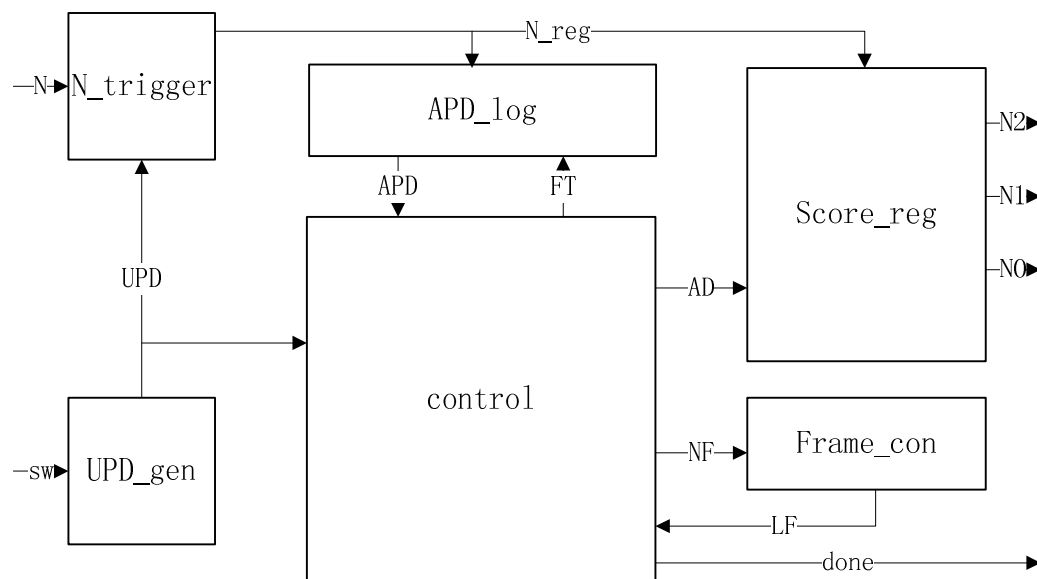


Figure 4

One thing that I want to indicate is that all these modules are associated with `clk` and `rst_n`, which represent clock and reset signal. Functions of each module are described below:

UPD_gen: generate UPD signal which is ‘1’ for one clock cycle after `sw` is changed to ‘1’. UPD is used to refresh the score and it symbols a new start of one time try.

N_trigger: Push N into this design and use UPD as a trigger. It guarantees the score data can only be changed at the beginning of every throw.

APD_log: check the result of one throw. FT='1' when the current throw is the first try. If the result is a strike or a spare, APD will be '1'.

Score_reg: Accumulate all the scores of throws and calculate the final score in this part. If AD='1', then N will be added to total score every clock cycle, which means we can control the adding times by control the length of AD. After calculation, output three 4-bit BCD numbers which is the score we want.

Frame_com: Count the frame number in this part. When NF='1', then plus one to the frame number every clock cycle. If it is the last frame, then LF='1'.

Control: use state machine to control all the signals. This is the core of the design and all the bowling rules are built inside.

Some details will be provided below.

3 Design details

3.1 Control module

The control module is the most important part of the design since it determines when and how we can use other modules to achieve the function we want. Figure 5 shows the state graph for control module.

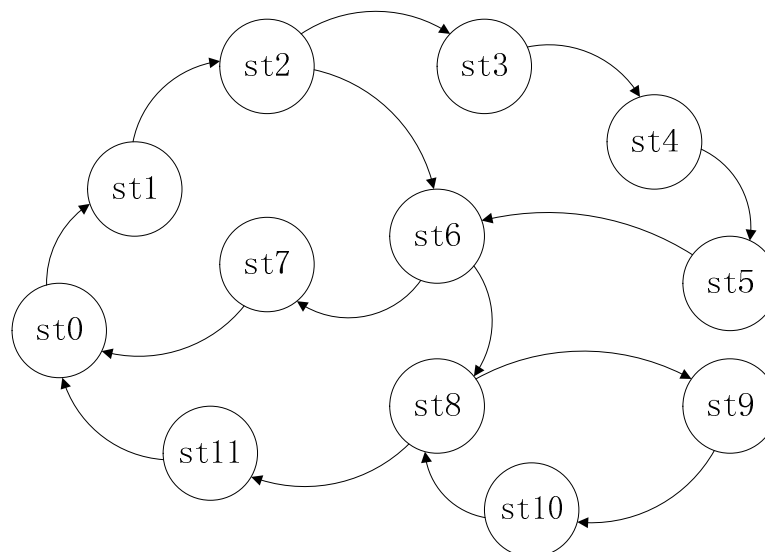


Figure 5

State transition conditions and state functions are not marked in this figure since there are too many signals to perform them in one figure. The comment of every state is clarified below:

State0: Let FT='1' and wait for UPD to start;

State1: Wait for one clock period so that the N_reg can be available from N_trigger;

State2: Set AD to '1'. Save the APD to strike and spare registers. If APD='1', go to state6, otherwise, go to state3;

State3: let FT='0' and wait for UPD to start;
 State4: Wait for one clock period so that the N_reg can be available from N_trigger;
 State5: Set AD to '1'. Save the APD to strike and spare registers;
 State6: If LF='1', go to state8, otherwise go to state7;
 State7: Set NF to '1' to add one to frame counter;
 State8: Check if the strike and spare registers are all '0', if so, go to state11. If not, go to state9;
 State9: Wait for UPD to start;
 State10: Set AD to '1'. Save '0' to strike and spare registers;
 State11: Stop the game and set done signal to '1'. Only when rst_n='0', go to st0 and clear all the signals.

It can be seen from the figure that for every throw (including bonus throws), we have to wait for UPD to start. Once UPD is '1', one try starts and the judging part begins. All of these operations are done in state0, state3 and state9.

What's more, in state1 and state4, the circuit does nothing but wait for a clock cycle. The reason for adding these two states is that the preparation of APD will take two clock periods to finish after UPD: one for N_reg and another for the APD logic part. There is no need to add this state to bonus throw because we do not need APD in this period.

More importantly, as I have mentioned in design strategy, we have to control the length of AD to control the times of addition, which means we have to control the time of state2, state5 and state10 in this state machine. In order to do this, I use the structure below in my VHDL description:

```

if clk'event and clk='1' then
  if n="00" then
    state<=nextstate
    n<=state_length;
  else
    n<=n-"01";
  end if;
end if;

```

In this structure, state transitions occur only when n equals to zero. As a result, we can set the length of every state by configuring state_length. When generating AD, I set the state length to the sum of strike1, strike0 and spare.

3.2 Score registers

The function of this module can be summarized into three steps:

The first step is addition. Specifically, the input score data should be added to the total score in this part. One thing that has to be noticed that is we have to make the add operation as a BCD adder. The source code is shown below:

```
s0<='0'&reg(3 downto 0)+N;
nextN0<=s0(3 downto 0)+6 when s0>9 else s0(3 downto 0);
c0<='1' when s0>9 else '0';
s1<='0'&reg(7 downto 4)+unsigned'(0=>c0);
nextN1<=s1(3 downto 0)+6 when s1>9 else s1(3 downto 0);
c1<='1' when s1>9 else '0';
nextN2<=reg(9 downto 8)+unsigned'(0=>c1);
```

The nextN0 and nextN1 are all BCD numbers and nextN2 is just a two bit number because the largest score is 300 and it just take two bit on the highest digit.

Then the next step should be done, which is storing all the numbers in a 10-bit register. To achieve this, I use the source code below:

```
if clk'event and clk='1' then
    if AD='1' then
        reg <= nextN2&nextN1&nextN0;
        else reg<= reg;
        end if;
    end if;
end if;
```

All the numbers are stored in reg and the data in the register will be used for next addition. Moreover, this step is controlled by AD, which make one time addition occurs every clock cycle when it is '1', just as I mentioned in design strategy.

Finally, we should pick the information out to output to external circuit. This function can be achieve by following code:

```
N2<="00"&reg(9 downto 8);
N1<=reg(7 downto 4);
N0<=reg(3 downto 0);
```

I add zeros to the highest position number to make it a BCD number in order to make it easier to display it on a screen.

4 Simulation

In order to test the design, I built the test bench below:

```
library IEEE;
```

```

        use IEEE.numeric_bit.all;

entity bowling_test is
end bowling_test;

architecture tester of bowling_test is
    signal clk, rst_n,sw,done:bit;
    signal N,N0,N1,N2,N3:unsigned (3 downto 0);
component bowling_top
    port(clk,rst_n,sw:in bit;N:in unsigned(3 downto 0);
        done:out bit;N0,N1,N2,N3:out unsigned(3 downto 0));
end component;
begin
    clk<=not clk after 5ns;
    rst_n<='0', '1' after 20ns;
    sw<=not sw after 500ns ;
    N<=
        "0000"
        ,"1010" after 1000ns
        ,"0110" after 2000ns
        ,"0001" after 3000ns
        ,"0100" after 4000ns
        ,"0101" after 5000ns
        ,"0000" after 6000ns
        ,"0110" after 7000ns
        ,"0110" after 8000ns
        ,"0100" after 9000ns
        ,"1010" after 10000ns
        ,"0000" after 11000ns
        ,"0000" after 12000ns
        ,"1010" after 13000ns
        ,"0011" after 14000ns
        ,"0101" after 15000ns
        ,"1010" after 16000ns
        ,"1000" after 17000ns
        ,"0010" after 18000ns
        ,"1010" after 19000ns
        ,"1010" after 20000ns;

    t: bowling_top port map(clk,rst_n,sw,N,done,N0,N1,N2,N3);
end tester;

```

It can be seen from the code that the clock frequency is 100MHz and the system deigns at 20ns. Then I set 1000ns for each throw by controlling sw. Finally, I put

example scores in.

The maximum number of throw is 21 because when there are two bonus throws, the last throw must be a strike, otherwise there will be no more than one chance.

I find some examples from internet which have been shown below (Figure 6):

Kate	-	/	6	1	4	5	-	6	6	/		X	-	-		X	3	5	X	8	/	114
		16		23		32		38		58		68		68		86		94		114		
Mary		X		X		X		X		X		X		X		X		X	X	X	X	300
		30		60		90		120		150		180		210		240		270		300		
Jack		X	-	-		X	-	-		X	-	-		X	-	-		X	-	-		50
		10		10		20		20		30		30		40		40		50		50		
Mike	9	/	9	/	9	/	9	/	9	/	9	/	9	/	9	/	9	/	9	/	-	181
		19		38		57		76		95		114		133		152		171		181		
Joan	-	/	-	/	-	/	-	/	-	/	-	/	-	/	-	/	-	/	-	/	-	100
		10		20		30		40		50		60		70		80		90		100		

Figure 6

The test bench above adds the stimulation signals the same with the score of Kate (the first line). If the final score is 114, then we can assert that this works properly for this input.

N3 in this test represents the score of every throw which will make the result more clear. After adding all the signals to wave form and running it for 22000ns (guarantee all the throws are done), I get the wave below (Figure 7):

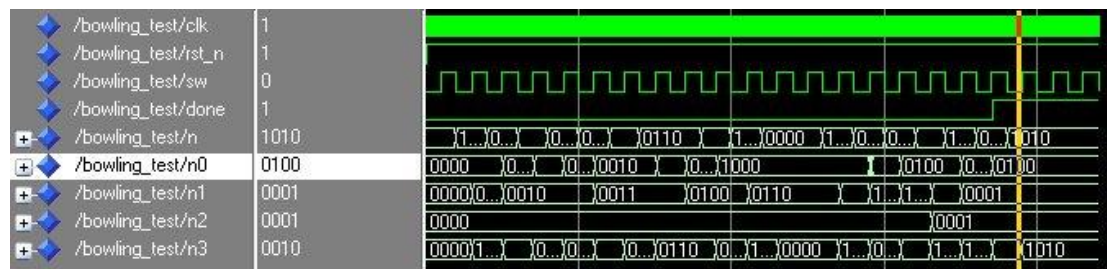


Figure 7

It shows that the final result for N2, N1 and N0 is "0001 0001 1010" when done signal is '1', which is a correct result that is the same with the result 114 in figure 6.

We can also add N3, N2, N1 and N0 to a list to make sure the design works well on any states. Here is the list of our result (Figure 7):

ns	delta	/bowling_test/n3	/bowling_test/n2	/bowling_test/n1	/bowling_test/n0
0	+0	0000	0000	0000	0000
1515	+2	1010	0000	0000	0000
1545	+2	1010	0000	0001	0000
2515	+2	0110	0000	0001	0000
2545	+2	0110	0000	0001	0110
2555	+2	0110	0000	0010	0010
3515	+2	0001	0000	0010	0010
3545	+2	0001	0000	0010	0011
4515	+2	0100	0000	0010	0011
4545	+2	0100	0000	0010	0111
5515	+2	0101	0000	0010	0111
5545	+2	0101	0000	0011	0010
6515	+2	0000	0000	0011	0010
7515	+2	0110	0000	0011	0010
7545	+2	0110	0000	0011	1000
8545	+2	0110	0000	0100	0100
9515	+2	0100	0000	0100	0100
9545	+2	0100	0000	0100	1000
10515	+2	1010	0000	0100	1000
10545	+2	1010	0000	0101	1000
10555	+2	1010	0000	0110	1000
11515	+2	0000	0000	0110	1000
13515	+2	1010	0000	0110	1000
13545	+2	1010	0000	0111	1000
14515	+2	0011	0000	0111	1000
14545	+2	0011	0000	1000	0001
14555	+2	0011	0000	1000	0100
15515	+2	0101	0000	1000	0100
15545	+2	0101	0000	1000	1001
15555	+2	0101	0000	1001	0100
16515	+2	1010	0000	1001	0100
16545	+2	1010	0001	0000	0100
17515	+2	1000	0001	0000	0100
17535	+2	1000	0001	0001	0010
18515	+2	0010	0001	0001	0010
18535	+2	0010	0001	0001	0100
19515	+2	1010	0001	0001	0100

Figure 7

I have tried all the inputs in Figure 6 and the results are all correct. As a result, the logic correctness of this design are tested and it works properly.