Term Project 2 Report

ID: 2017120486 Name:서천함

## 1. Project Summary (2%)

In this project, I reviewed what we learnt about sequential circuits and learnt a lot of new things about verilog language. For instance, the difference between blocking(=) and nonblocking(<=) assignments; usage of wire and reg; always statement and case construct and so on. During implementation, I didn't pay much attention to the repeated states so it reported error. Next time I should be more careful about such details.

## 2. Module Description (12%)

Step 1: 2 input variables and 3 output variables are declared in this module.

```
module missionary_cannibal(clk, reset, missionary_next, cannibal_next, finish);
        input clk;
        input reset;
        output reg [1:0] missionary_next;
        output reg [1:0] cannibal_next;
        output reg finish;
```

Step2: Based on the description, we could get the complete table describing the current state, next state and finish sign as follows. According to this table, 10 different states could be declared.
( Since s4 (=4'b1100) has the same value as s2 and s11(=4'b0010) has the same valu as s8, I only defined s2 and s8 once and used the direction signal to decide which was the next state.)

| Clock(step) | Current State | Direction | Next State | Finish |
|---|---|---|---|---|
| 1 | 1111 | 1 | 1101 | 0 |
| 2 | 1101 | 0 | 1110 | 0 |
| 3 | 1110 | 1 | 1100 | 0 |
| 4 | 1100 | 0 | 1101 | 0 |
| 5 | 1101 | 1 | 0101 | 0 |
| 6 | 0101 | 0 | 1010 | 0 |
| 7 | 1010 | 1 | 0010 | 0 |
| 8 | 0010 | 0 | 0011 | 0 |
| 9 | 0011 | 1 | 0001 | 0 |
| 10 | 0001 | 0 | 0010 | 0 |
| 11 | 0010 | 1 | 0000 | 1 |
| 12 | 0000 | 0 | 1111 | 0 |

```
parameter s0 = 4'bllll,
s1 = 4'bll01,
s2 = 4'bllll0,
s3 = 4'bll100,
s5 = 4'b0101,
s6 = 4'bl010,
s7 = 4'b0010,
s8 = 4'b0011,
s9 = 4'b0001,
s11 = 4'b0000;
```

Step 3: Internal registers are designed as follows.
If reset is 1, current state is set to be the intial state(s0) and direction will be changed to initial state. If reset is not 1, current state will shifted to be the next state and direction will be inverted.
(According to the specification provided, I used D flip-flop with asychronous reset, so the transfers of input into output is synchronized by the positive-edge transition of clk or the reset. )

```
reg direction;
reg[3:0] current_state,next_state;

always @ (posedge clk or posedge reset)
begin
if(reset) direction <= 1'b1;
else direction <= ~direction;
end

always @(posedge clk or posedge reset)
begin
        if(reset) begin
                current_state <= s0;
        end
        else begin
                current_state <= next_state;
        end
end
```

Step 4: Then I had to decide what the next state is and the output of next state.
Here I used two case constructs.

For each current_state, it will change the next_state accordingly. For each next_state, it will give a value for the 3 outputs: missionary_next, cannibal_next and finish.

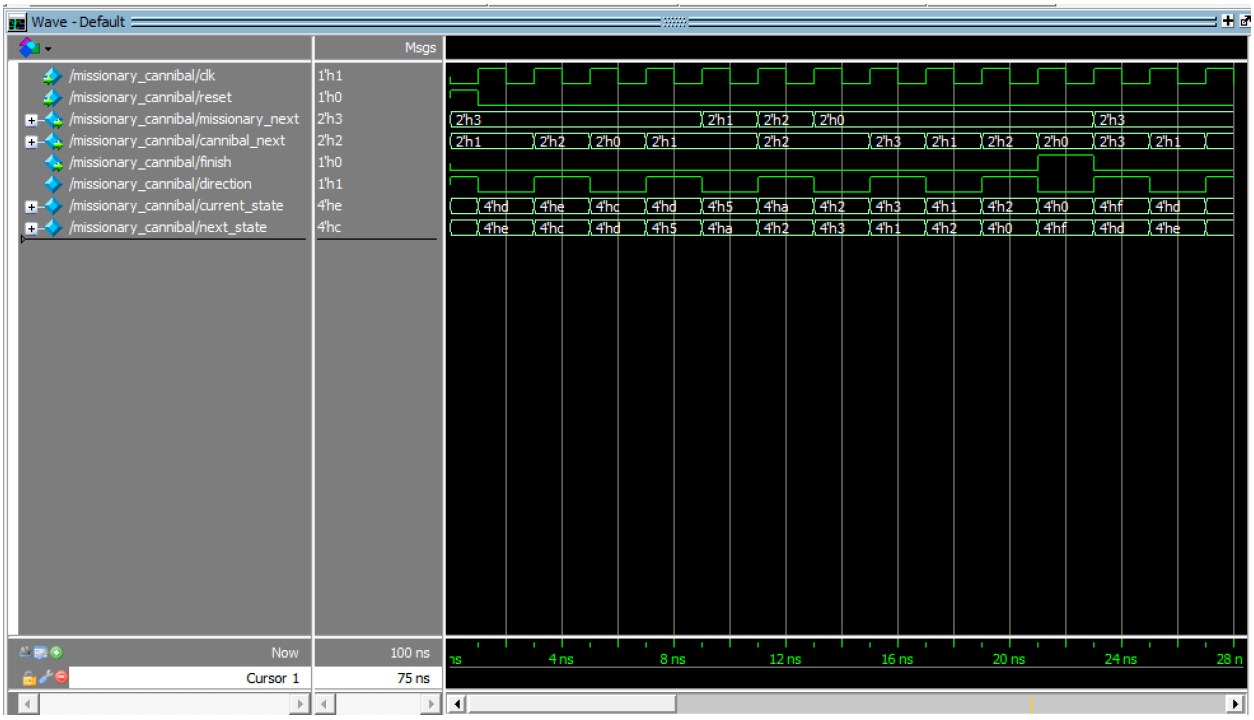
s4 and s10 should be omitted here because the case struct cannot differentiate next states having the same value. Instead, I use direction sign to differentiate them and decide what's the next state.

Default cases are set to be the initial state.

```
always @( * )
begin
        case(current_state)
                s0: begin next_state = s1; end
                s1: begin if(direction) next_state = s2; else next_state = s5; end
                s2: begin next_state = s3; end
                s3: begin next_state = s1; end
                s5: begin next_state = s6; end
                s6: begin next_state = s7; end
                s7: begin if(direction) next_state = s8; else next_state = s11; end
                s8: begin next_state = s9; end
                s9: begin next_state = s7; end
                s11: begin next_state = s0; end
                default : begin next_state = s0; end
        endcase
end

always @(posedge clk or posedge reset)
begin
        if(reset) begin
                missionary_next <= 2'b11;
                cannibal_next <= 2'b01;
                finish <= 0;
        end
        else begin
                case(next_state)
                        s0: begin missionary_next <= 2'b11; cannibal_next <= 2'b11; finish <= 0; end
                        s1: begin missionary_next <= 2'b11; cannibal_next <= 2'b01; finish <= 0; end
                        s2: begin missionary_next <= 2'b11; cannibal_next <= 2'b10; finish <= 0; end
                        s3: begin missionary_next <= 2'b11; cannibal_next <= 2'b00; finish <= 0; end
                        s5: begin missionary_next <= 2'b01; cannibal_next <= 2'b01; finish <= 0; end
                        s6: begin missionary_next <= 2'b10; cannibal_next <= 2'b10; finish <= 0; end
                        s7: begin missionary_next <= 2'b00; cannibal_next <= 2'b10; finish <= 0; end
                        s8: begin missionary_next <= 2'b00; cannibal_next <= 2'b11; finish <= 0; end
                        s9: begin missionary_next <= 2'b00; cannibal_next <= 2'b01; finish <= 0; end
                        s11: begin missionary_next <= 2'b00; cannibal_next <= 2'b00; finish <= 1; end
                        default : begin missionary_next <= 2'b11; cannibal_next <= 2'b11; finish <= 0; end
                endcase
        end
end
```

## 3. Simulation screenshot (6%)




(Script:
quit -sim
vsim -gui work.missionary_cannibal

add wave -position insertpoint sim:/missionary_cannibal/*

force -deposit clk 0 0ns, 1 1ns -repeat 2ns
force reset 1 0ns, 0 2ns

run 100ns)