Hardware Design

**Lab 5 Report**

Keyboard and Audio Modules

**Team 01**

112062122 謝佳晉 112062144 范升維

Table of Contents:

[Basic Q3. Many-to-One Linear-Feedback Shift Register (LFSR) 1](#_Toc180783866)

[Basic Q4. One-to-Many Linear-Feedback Shift Register (LFSR) 1](#_Toc180783867)

[Advanced Q1. Content-Addressable Memory (CAM) Design 2](#_Toc180783868)

[Advanced Q2. Scan Chain Design 5](#_Toc180783869)

[Advanced Q3. Built-In Self Test (BIST) 7](#_Toc180783870)

[Advanced Q4. Mealy Machine Sequence Detector 9](#_Toc180783871)

[FPGA: Built-In Self Test (BIST) 11](#_Toc180783872)

[Testbenches 13](#_Toc180783873)

[A. Content-Addressable Memory (CAM) Design 13](#_Toc180783874)

[B. Scan Chain Design 13](#_Toc180783875)

[C. Built-In Self Test (BIST) 13](#_Toc180783876)

[D. Mealy Machine Sequence Detector 14](#_Toc180783877)

[What have we learned from Lab 4? 14](#_Toc180783878)

[Contributions 15](#_Toc180783879)

# Advanced Q1. Sliding Window Sequence Detector

In advanced Q1, we were instructed to create a sliding window sequence detector. The detector takes in the input value and update the state and dec according to the value.

一張含有 圓形, 螢幕擷取畫面 的圖片

自動產生的描述

▲ Figure 1.1: state transition diagram

* 3 inputs: *clk, rst\_n, in*
* 1 outputs: *dec*
* Internal signals: *state[3:0], next\_state[3:0]*
* 9 States: *S0, S1, S2, S3, S4, S5, S6, S7*

First, we design the state transition mechanism as Figure 1.1. In *S6*, we divide the next states into *S7* and *S8* to deal with the repeating *2’b01* cases. If the state is *S7* and the input value is *1’b1*, state will go back to *S6*, detecting another *2’b01*. On the other hand, If the state is *S8* and the input value is *1’b1*, state will go back to *S3* and make *dec* output *1’b1*.

一張含有 文字, 螢幕擷取畫面, 設計 的圖片

自動產生的描述  
▲ Figure 1.2:state signal

In Figure 1.2, we show the sequential circuit design of *state[3:0]*. If *rst\_n* is *1’b0*, state will go back to initial state *S0*. Otherwise, it will take in *next\_state[3:0]*.

一張含有 文字, 螢幕擷取畫面, 設計, 黑與白 的圖片

自動產生的描述一張含有 文字, 螢幕擷取畫面, 靜止的, Rectangle 的圖片

自動產生的描述

▲ Figure 1.3:next\_state signal ▲ Figure 1.4:dec signal

Figure 1.3 and Figure 1.4 shows our combinational circuit designs for *next\_state[3:0]* and *dec*. We set *state[3:0]* as the rightmost MUXs’ selection bits. We only have nine states, but the rightmost MUXs have 16 input ports, so default cases have to be considered. However, because states only update within the nine states, we can set the vacant input ports to whatever 4-bit values we want. For *next\_state[3:0]*, we set them to the input value same as the case when state *S8*, while in *dec*, we set them to *1’b0*.  
For *next\_state[3:0]*, we have to decide which state should the next state be based on *in* and the state at the moment, so we add 2x1 MUXs to 0th to 8th input ports.   
In *dec*’s circuit, because we only have to take output value branch into consideration when state is *S8*, we add a 2x1 MUX to the 8th input port, and set *1’b0* to other input ports.

# Advanced Q2. Traffic Light Controller

In advanced Q2, we were instructed to create a traffic light controller. The controller mechanism is taking in the input value and updating the state according to the value and clock cycles. We use a counter to keep track of the clock cycles.

一張含有 圓形, 文字, 螢幕擷取畫面 的圖片

自動產生的描述

▲ Figure 2.1: state transition diagram

* 3 inputs: *clk, rst\_n, lr\_has\_car*
* 2 outputs: *hw\_light[2:0], lr\_light[2:0]*
* Internal signals*: cnt[7:0], state[2:0], next\_state[2:0]*
* 6 states: *A, B, C, D, E, F*
* Other parameters: *greentime*(8’d69), *yellowtime*(8’d24)

Before designing our circuits for this problem, we complete the state transition diagram in the lab powerpoint, as shown in Figure 2.1. We add some go-back arrows and the *rst\_n* case.

一張含有 文字, 螢幕擷取畫面, 圖形, Rectangle 的圖片

自動產生的描述

▲ Figure 2.2:state signal

In Figure 2.2, we show the sequential circuit design of *state[2:0]*. If *rst\_n* is *1’b0*, state will go back to initial state *A*. Otherwise, it will take in *next\_state[2:0]*.

一張含有 文字, 螢幕擷取畫面, 圖形, 設計 的圖片

自動產生的描述

▲ Figure 2.3: cnt signal

Figure 2.3 shows our sequential circuit design for *cnt*. Because we have to set *cnt* to *8’d0* when *rst\_n* == *1’b0* or state changes, we add a 2x1 MUX before the DFF and set the 1st input port to *8’d0*. For the 0th input port, it takes in the updated value of *cnt*. When state is *A* or *D*, *cnt* will increase if *cnt* < *greentime* and stop counting if *cnt* == *greentime* to make sure that state can change at exactly next cycle and in case of overflow. On the other hand, when state is *B* or *F*, *cnt* will increase if cnt < *yellowtime* and stop counting if *cnt* == *yellowtime* for the same reason.

一張含有 文字, 螢幕擷取畫面, 黑色, Rectangle 的圖片

自動產生的描述  
▲ Figure 2.4: hw\_light signal and lr\_light signal

*hw\_light* changes only in the front 3 states, and *lr\_light* changes only in the back 3 states. Therefore, we use MUXs and plug in *state[3:0]* as selection.

一張含有 螢幕擷取畫面, 文字, Rectangle, 設計 的圖片

自動產生的描述

▲ Figure 3.4:next\_state signal

Figure 2.4 shows our combinational circuit designs for *next\_state[2:0]* (we set the initial value of *next\_state[2:0]* to *A*)*.* When state is *A*, *next\_ state[2:0]* will update according to *cnt*, *lr\_has\_car*.  
When state is *B*, *C*, *D*, *E*, *F*, *next\_state[2:0]* will update only based on *cnt* values. However, when state is *C* or *F*, updating requires only one clock cycle. Therefore, we don’t need to actually use the value of *cnt* but update *next\_state[2:0]* to next value directly. For default cases, we simply plug in *A*.

## FPGA Demonstration 1

In this demonstration, we are required to make a ascending-pitch and descending-pitch music box. We modify “PlayerCtrl.v”, “Music.v”, and “Top.v” in Music\_box\_Sample\_Code directory, and merge them with those files in the Music\_box\_Sample\_Code directory originally and “KeyboardDecoder.v” in Keyboard\_Sample\_Code directory.  
In “Music.v”, we modify the music from “Little Apple” to ascending pitches from C4 to C8 in order, which has 28 pitches in total.  
 一張含有 文字, 螢幕擷取畫面, 便利貼, 設計 的圖片

自動產生的描述

▲ Figure 4.1: modified PlayerCtrl

Figure 4.1 shows the modified version of PlayerCtrl. We add the mechanism of Reversing and Holding when reach the end of the direction. In order to change direction, we add an additional input port sel (linked with *speed*) to determine whether ascending or descending should be implemented at the moment. It can also help us to hold the note at the ends of ascending and descending without actually changing *ibeat*’s value. Note that we set the reset value of *ibeat* to *8’d255* in order to handle unwanted increment of *ibeat* because of collision of *reset* and *clk.*

一張含有 螢幕擷取畫面, 圖形, 設計 的圖片

自動產生的描述  
▲ Figure 4.2: reset signal

In Top module, we define *reset’s* value to *been\_ready && key\_down[KEY\_CODES\_ENTER].*As shown in Figure 4.2.

一張含有 螢幕擷取畫面, 寫生, 輪廓, 設計 的圖片

自動產生的描述一張含有 螢幕擷取畫面, 設計 的圖片

自動產生的描述

▲ Figure 4.3: SpeedCtrl ▲ Figure 4.4: DirectionCtrl

Figure 4.3 and 4.4 shows our design of SpeedCtrl and DirectionCtrl. They are similar to the state transition mechanism in the previous problems. We use *speed*/*dir* as state and use *next\_speed*/*next\_dir* as next\_state.

一張含有 螢幕擷取畫面, 正方形, Rectangle 的圖片

自動產生的描述  
▲ Figure 4.4: Top module design

Figure 4.4 shows the overall design of our Top module.

We merge the KeyboardDecoder and the original Top module in music box’s sample and modify PlayerCtrl to calculate *ibeatNum* according to the direction. Besides, we add additional signals, dir, speed, next\_dir and next\_speed, and update them according to keyboard states. In this way, we can handle the direction and speed changing much easier and clearer.

# Testbenches

## A. Content-Addressable Memory (CAM) Design

In this testbench, we combine the waveform on the spec and some special cases:

1. same data in different stored data lines
2. data matching fails
3. ren and wen both positive simultaneously

## B. Scan Chain Design

In question B, we wrote our testbench fully complying with the input in the spec and receive the same waveform.

*a* = *4'b1111* = *4'd15*

=> *p* = *a* \* *b* = *8'd150* = *8'b10010110*

*b* = *4'b1010* = *4'd10*

## C. Built-In Self Test (BIST)

In this testbench, we modify the *rst\_n* and *scan\_en* according to the working pattern of Scan Chain Design (reset → scan in → capture → scan out). We can see that the sequence of *scan\_in* output is *8'b10111101*, which is the same as the reversed reset value of LFSR.

## D. Mealy Machine Sequence Detector

In this testbench, we reference the waveform on the spec and get the exactly waveform.

# What have we learned from Lab 4?

In this lab, we learned how to implement Mealy and Moore Machines using Verilog, and we explored several interesting concepts including CAM and BIST. Drawing from our previous lab experience, where we struggled to convert our code into circuit diagrams due to insufficient planning, we adopted a more structured approach this time. We thoroughly planned our design before implementation. Additionally, leveraging the coding style principles learned during the Lab 1-3 Review before midterm - such as proper if-else hierarchical structuring and the limit of only using system clock for module driving - we were able to complete the programming and create the circuit diagram more efficiently (though admittedly, this might be partially due to the more streamlined circuit design).

Furthermore, we encountered a valuable learning point in the first problem. Due to the absence of a reset input, the Compare Array output could potentially produce unexpected results due to residual signals. To address this, we implemented the === operator instead of == in Verilog to filter out 'z' and 'x' signals. It's worth noting that while this approach isn't replicable in physical circuits, it serves as a useful tool for simulation observation purposes.

# Contributions

謝佳晉：

wrote Verilog modules: Q1, Q2, Q3, Q4, fpga

wrote testbenches: Q1, Q2, Q3, Q4

performed simulation: Q1, Q2, Q3, Q4

drew diagram: Q3

wrote report: basic Q3, basic Q4, Q3, tb

范升維：

wrote Verilog modules: Q1, Q2, Q3, Q4, fpga

performed simulation: Q1, Q2, Q3, Q4

drew diagram: Q1, Q2, Q3, Q4

wrote report: Q1, Q2, Q3, Q4, tb, what we learned

made FPGA demonstration

organized whole report

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | b-Q3 | | b-Q4 | Q1 | Q2 | Q3 | Q4 | FPGA | What we learned |
| wrote Verilog modules |  | |  |  |  |  |  |  |  |
| wrote testbenches |  | |  |  |  |  |  |  |  |
| performed simulation |  | |  |  |  |  |  |  |  |
| drew diagram |  | |  |  |  |  |  |  |  |
| wrote report |  | |  |  |  |  |  |  |  |
| wrote report tb |  | |  |  |  |  |  |  |  |
|  | |  |
| organized whole report |  | |

|  |
| --- |
| Both |
| 謝佳晉 |
| 范升維 |